DRAFT (26 February 2003) — Do Not Distribute

PHYSICALLY BASED IMAGE SYNTHESIS: DESIGN AND IMPLEMENTATION OF A RENDERING SYSTEM

Matt Pharr and Greg Humphreys

[Just as] other information should be available to those who want to learn and understand, program source code is the only means for programmers to learn the art from their predecessors. It would be unthinkable for playwrights not to allow other playwrights to read their plays [and] only be present at theater performances where they would be barred even from taking notes. Likewise, any good author is well read, as every child who learns to write will read hundreds of times more than it writes. Programmers, however, are expected to invent the alphabet and learn to write long novels all on their own. Programming cannot grow and learn unless the next generation of programmers have access to the knowledge and information gathered by other programmers before them.

— Erik Naggum

Preface

Rendering is a crucial component of most computer graphics work. Without rendering, the results of algorithms for animation, geometric modeling, and texturing would have no visual representation. At the very highest level of abstraction, rendering converts a description of a three-dimensional scene into an image for display. By its very nature, rendering incorporates ideas from a broad range of disciplines, including physics, astronomy, biology, psychology, and pure mathematics. The interdispiplinary nature is one of the reasons rendering is such a fascinating area to study.

Rendering is sufficiently important that almost all modern computers use dedicated hardware to accelerate interactive rendering. Huge computational resources are available in specialized graphics processing units (GPUs) to improve the visual quality of interactive graphics. In this book, however, we will focus on non-interactive rendering on CPUs. GPUs are not yet sufficiently flexible to be programmed to implement many of the algorithms in this book, though work in this area is progressing rapidly.

This book presents a variety of modern rendering algorithms through the documented source code for a complete rendering system. This system, called `lrt`, is written using a programming methodology called *literate programming* that mixes prose describing the system with the source code that implements it. Note that this book and the system it describes are by no means complete; many interesting topics in rendering will not be covered either because they didn't fit well with

the architecture of the software system (e.g. finite element radiosity algorithms), or because we believed that the pedagogical value of the algorithm was outweighed by the complexity of the implementation. In most cases, we will provide pointers to further reading so the reader can continue his studies.

We believe that the literate programming approach is a valuable way to teach ideas in computer science. Not only does the implementation help clarify how an algorithm is implemented in practice, but by showing these algorithms in the context of a complete and non-trivial software system we are also able to address issues in the design and implementation of medium-sized rendering systems. Often, all of the subtleties of an algorithm can be missed until it is implemented; seeing someone else's implementation is (sometimes) nearly as good.

Intended Audience

Our primary audience is students in upper-level undergraduate or in graduate-level computer graphics classes. This book is not self-contained; it assumes existing knowledge of computer graphics at the level of an introductory college-level course. Certain key concepts from such a course will be presented again here, such as basic vector geometry and transformations.

Our secondary, but equally important audiences are advanced graduate students, researchers, and software developers in industry. Though many of the ideas in this manuscript will likely be familiar to these readers, reading explanations of some algorithms in the literate style should be of interest. We have also also included implmentations and descriptions of more recently-developed algorithms and techniques, including subdivision surfaces, Monte Carlo light transport, and volumetric scattering models; these should be of particular interest even to experienced researchers. We also hope that it will be useful for this audience to see one way to organize a complete non-trivial rendering system.

Overview and Goals

`lrt` was designed and implemented with two main goals in mind: it should be *complete*, and it should be *illustrative*. Completeness implies that the system should not lack important features found in high-quality rendering systems. In particular, it means that important practical issues, such as anti-aliasing, robustness in the face of numerical error, and the use of physical units to describe light and reflection should be addressed thoroughly.

It is often quite difficult to retrofit such functionality to a rendering system after it has been designed, as these features can have subtle implications for all components of the system, and even the overall architectural design.

Our second goal means that we tried to choose algorithms, data structures, and rendering techniques with care. Since their implementations will be examined by more readers than those in most rendering systems, we tried to select the cleanest and most elegant algorithms that we were aware of. The second goal also implies that the system should be small enough for a single person to understand completely.

Note that there is an obvious tension between these two goals. Implementing and describing

every useful technique that would be found in a production rendering system would not only make this book extremely long, but it would make the system too big to fully understand. In cases where `lrt` lacks a generally useful feature, we have attempted to design the architecture so that feature could be easily added without altering the overall system design. Exercises at the end of each chapter suggest programming projects that involve new features.

Efficiency was a tertiary goal. Since rendering systems often run for many minutes or hours in the course of generating an image, efficiency is clearly important. However, we have mostly confined ourselves to *algorithmic* efficiency rather than low-level code optimization. In many cases, obvious micro-optimizations take a back seat to clear, well-organized code. For this reason as well as portability, `lrt` is not presented as a parallel or multi-threaded application, although parallelizing `lrt` would not be very difficult.

In the course of presenting `lrt`, we hope to convey some hard-learned lessons from some years of rendering research and development. Writing a good renderer is much more complex than stringing together a set of fast algorithms; making the system both flexible and robust is the hard part. The system's performance must degrade gracefully as more geometry is added to it, as more light sources are added, or as any of the other axes of complexity are pushed. Numeric stability must be handled carefully; stable algorithms that don't waste floating-point precision are critical. In the end, a renderer is something like an operating system: managing large amounts of data and computation without crashing, while always returning correct (or in some cases, reasonable) results.

Coding Conventions

We have chosen to write `lrt` in C++. However, we use a subset of the language, both to make the code easier to understand for the non C++ expert, as well as to improve portability between compilers. In particular, we have avoided multiple inheritance and the use of exceptions. We have also used only a small subset of C++'s extensive standard library for similar reasons. In particular, we use the `iostream` input/output facilities and the `vector`, `set`, and `map` container classes.

Types, objects, and variables are named to indicate their scope; classes and functions that have global scope all start with capital letters. The system also uses no global variables. Small utility classes, module-local `static` variables, and functions that are used in just one part of the system start with lower-case letters.

Finally, we will omit various pieces of `lrt`'s entire collection of source code from this document. For example, when there are a number of cases to be handled, all with nearly identical code, we will present one case and note that the code for the remaining cases is omitted (of course, it's not omitted from the final program source code!). Furthermore, when we declare a new class (`Foo`, for example), we won't include the code for the class declaration in this document. Instead of having many fragments like:

⟨*Classes*⟩≡
```
class Foo {
public:
    ⟨Foo Public Methods⟩
private:
    ⟨Foo Private Methods⟩
    ⟨Foo Private Data⟩
};
```

for every new class, each class declaration fragment will be omitted and we will immediately start adding methods to ⟨*Foo Public Methods*⟩ and so forth.

Code Optimization

As mentioned above, we have tried to make `lrt` efficient by using well-chosen algorithms rather than by having many low-level optimizations. However, we have used a profiler to find which parts of it account for most of the execution time and have performed some local optimization of those parts. There are a handful of techniques that are particularly useful to keep in mind when trying to write efficient code for modern processors:

- Use your profiler well! Optimization should be driven by statistics about the performance of the system on typical scenes. It doesn't do any good to optimize based on scenes that aren't interesting (e.g. a single sphere), and it is hopeless to try to speed up the program without understanding which parts of it are truly the bottlenecks.

- On current CPU architectures, the slowest mathematical operations are divides, square-roots, and trigonometric functions. Addition, subtraction, and multiplication are generally ten to fifty times faster than those operations. Code changes that reduce the number of slow mathematical operations can help performance substantially; for example, replacing a series of divides by a value v with the computing the value $1/v$ and then multiplying by that value can help a lot. Contrary to a popular misconception, compilers will generally not manipulate mathematical expressions to faster, semantically equivalent ones because of concerns over numerical precision.

- Declaring short functions as `inline` can speed up code substantially, both by removing the run-time overhead of performing a function call (which may involve saving values in registers to memory) as well as by giving the compiler larger basic blocks to optimize.

- As CPUs are continuing to become faster while RAM access rates are growing at a much slower pace, waiting for values to be loaded from memory

is becoming a major performance barrier. While in the past it may have been advantageous to precomupte the values of expensive functions and store them in tables, the modern trend is towards re-computation of simple values rather than accessing memory.

Additional Reading

Knuth's article *Literate Programming*(Knu84) describes the main ideas behind literate programming as well as his `web` programming environment. The entire TEX typesetting system was written with this system and has been published as a book(Knu93a). More recently, Knuth has published a collection of graph algorithms in *The Stanford Graphbase*(Knu93b). Both of these are enjoyable to read and are respectively excellent presentations of modern automatic typesetting and graph algorithms. The website `http://www.literateprogramming.com` has pointers to many articles about literate programming as well as a variety of literate programming systems; many refinements have been made since Knuth's original development of the idea.

The implementation of the `lcc` C compiler is described in a literate program written by Fraser and Hansen and published as *A Retargetable C Compiler: Design and Implementation* (FH95).

A good introduction to the C++ programming language and C++ standard library is the third edition of Stroustroup's *The C++ Programming Language*(Str97).

Some notable books on rendering and image synthesis include *Radiosity and Realistic Image Synthesis* (CW93), which primarily describes the finite-element radiosity method; *Principles of Digital Image Synthesis* (Gla95); an encyclopediac two-volume summary of theoretical foundations for realistic rendering; and *Illumination and Color in Computer Generated Imagery* (Hal89), one of the first books to present rendering in a physically-based framework.

A number of papers have been written that describe the design and implementation of other rendering systems. The REYES architecture, which forms the basis for Pixar's RenderMan renderer, was first described by Cook et al (CCC87); a number of improvements to the original algorithm are described in (AG00). Ward describes *Radiance*, which is focused on accurate lighting simulation in a paper and a book (War94b; LS98). Gritz and Hahn describe the BMRT ray-tracer (GH96), and the Maya renderer is described by Sung et al (SCW+98). *Introduction to Ray Tracing*, which describes the state-of-the-art in ray-tracing in 1989 (Gla89a), Heckbert's chapter sketches the design of a ray-tracer. Finally, Shirley's recent book XXXX.

The complete source code to a number of ray-tracers and renderers is available on the web. Notable ones include Mark VandeWettering's *MTV*, which was the first widely-distributed freely-available ray-tracer; it was posted to the `comp.sources.unix` newsgroup in 1988. Craig Kolb's *rayshade* had a number of releases during the 1990s; its current homepage is `http://graphics.stanford.edu/ cek` The *radiance* system is available from `http://radsite.lbl.gov/radiance/HOME.html`.

Contents

1 Introduction 1

1.1 Literate Programming 1
1.2 Overview of Rendering 4
1.3 Overview of the Program 4
1.4 Defining the Scene 4
1.5 Simulating the Camera 7
1.6 Ray Intersections 9
1.7 Shading and Lighting 11
1.8 How to Progress Through this Book 14

2 Geometry and Transformations 15

2.1 Vectors 16
2.2 Points 21
2.3 Normals 23
2.4 Rays 24
2.5 Axis-Aligned Bounding Boxes 27
2.6 Transformations 30

2.7 Differential Geometry 46

3 Shapes 51

3.1 Basic Shape Interface 52
3.2 Spheres 55
3.3 Cylinders 65
3.4 Disks 69
3.5 Other Quadrics 72
3.6 Triangles and Meshes 74
3.7 Subdivision Surfaces 83

4 Intersection Acceleration 113

4.1 Approaches To Reducing Intersections 114
4.2 Primitive Lists 121
4.3 Regular Grid Accelerator 122
4.4 Kd Tree 133

5 Color and Radiometry 151

5.1 Spectral Representation 152
5.2 Basic Radiometry 157
5.3 Working with Radiometric Integrals 161
5.4 Surface Reflection 166

6 Camera Models and Film 171

6.1 Camera Model 171
6.2 Projective Camera Models 175
6.3 Environment Camera 184
6.4 Film 187

7 Sampling and Reconstruction 191

7.1 Signal Processing and Sampling Theory 192
7.2 Image Sampling Interface 197
7.3 Stratified Sampling 200
7.4 Low-Discrepancy Sequences 205
7.5 Best-Candidate Sampling Patterns 214
7.6 Image Reconstruction 227

8 Image Pipeline 237

8.1 Processing Camera Samples 237
8.2 Spectral Image Storage 239
8.3 Image Display Pipeline 240
8.4 Tone Mapping 244
8.5 Device RGB Conversion and Output 258

9 Reflection Models 263

9.1 Basic Interface 265
9.2 Specular Reflection and Transmission 269
9.3 Lambertian Reflection 280
9.4 Microfacet Models 281
9.5 Lafortune Model 291
9.6 Fresnel Incidence Effects 294

10 Materials 297

10.1 BSDFs 297
10.2 Material Interface and Bump Mapping 302
10.3 Matte 311
10.4 Plastic 312
10.5 Translucent 313
10.6 Glass 314
10.7 Shiny Metal 315
10.8 Diffuse Substrate 316
10.9 Measured Data 318

11 Texture 321

11.1 Sampling and Anti-Aliasing 323
11.2 Texture Interface 323
11.3 Basic Textures 324
11.4 2D Mappings 326
11.5 Interpolated Textures 330
11.6 Image Maps 333
11.7 Solid and Procedural Texturing 343
11.8 Noise 347

12 Light Sources **357**

12.1 Light Interface 357
12.2 Point Lights 360
12.3 Infinite Point Lights 367
12.4 Area Lights 368
12.5 Infinite Area Lights 371

13 Volume Scattering **375**

13.1 Volume Scattering Processes 376
13.2 Phase Functions 380
13.3 Volume Description 383
13.4 Subsurface Scattering 391

14 Monte Carlo Integration **395**

14.1 Background 396
14.2 Sample Patterns 410
14.3 Sampling Reflection Functions 411
14.4 Sampling Light Sources 429
14.5 Sampling Volume Scattering 440

15 Light Transport **443**

15.1 The Light Transport Equation 444
15.2 Whitted Integrator 446
15.3 Direct Lighting Integrator 447
15.4 Integral Over Paths 456
15.5 Path Tracing 458
15.6 Bidirectional Path Tracing 464
15.7 Photon Mapping 469
15.8 Irradiance Caching 477
15.9 Volume Integration 483

16 Summary and Conclusion **491**

16.1 Major Projects 491

A Utilities **493**

A.1 The C++ Standard Library 494

A.2 Error Reporting 495
A.3 Statistics 499
A.4 Memory Management 504
A.5 Matrix Routines 510
A.6 Mathematical Utility Functions 512
A.7 Random Numbers 515
A.8 Hash Tables 515
A.9 Octrees 516
A.10 Kd Trees 522
A.11 Main Include File 528

B TIFF Input and Output 531

B.1 Input 532
B.2 Output 536

C Dynamic Object Creation 541

C.1 Parameter Sets 542
C.2 Reading Dynamic Libraries 548

D Rendering Interface 557

D.1 Tokens and Parameter Management 559
D.2 Global Settings and Graphics Options 562
D.3 Graphics State 571
D.4 Transformations 575
D.5 Geometric Primitives 577
D.6 Light Sources 581
D.7 Volumes 582

1. Introduction

This chapter provides a high-level description of `lrt` and the entire rendering system from the top down. We describe what happens during rendering by tracing a single ray through the system. Along the way we introduce the major classes in the system. Subsequent chapters will describe the various classes and their methods in detail. Because this is a top-down exploration of the system, some concepts will be referred to before they are defined. Therefore, the reader will benefit from reading this section more than once.

1.1 Literate Programming

In the course of the development of the TEX typesetting system, Donald Knuth developed a new programming methodology based on the simple idea that *programs should be written more for people's consumption than for computers' consumption.* He named this methodology *literate programming*. This book (including the chapter you're reading now) is a long literate program. Literate programs are written in a meta-language that mixes a document formatting language (e.g. LATEX or HTML) and a programming language (e.g. C++). The meta-language compiler then can transform the literate program into either a document suitable for typesetting (this process is generally called *weaving*), or into source code suitable for

compilation (*tangling*).

The literate programming meta-language provides two important features. The first is a set of mechanisms for mixing English text with source code. This makes the description of the program just as important as its actual source code, encouraging careful design and documentation on the part of the programmer. Second, the language provides mechanisms for presenting the program code to the reader in an entirely different order than it is supplied to the compiler. This feature makes it possible to describe the operation of the program in a very logical manner. Knuth named his literate programming system `web` since literate programs tend to have the form of a web: various pieces are defined and inter-related in a variety of ways and programs are written in a structure that is neither top-down nor bottom-up.

As a simple example, consider a function `InitGlobals()` that is responsible for initializing all of the program's global variables. If all of the variable initializations are presented to the reader at once, `InitGlobals()` might be a large collection of variable assignments the meanings of which are unclear because they do not appear near the definition or use of the variables. A reader would need to search through the rest of the entire program to see where each particular variable was declared in order to understand the function and the meanings of the values it assigned to the variables. As far as the human reader is concerned, it would be better to present the initialization code near the code that actually declares and uses the global.

In a literate program, then, one can instead write `InitGlobals` like this:

⟨*Function Definitions*⟩≡
```
void InitGlobals() {
    ⟨Initialize Global Variables⟩
}
```

Here we have added text to a *fragment* called ⟨*Function Definitions*⟩. (This fragment will be included in a C++ source code file when the literate program is woven for the compiler.) The fragment contains the definition of the `InitGlobals` function. The `InitGlobals` function itself includes another fragment, ⟨*Initialize Global Variables*⟩. At this point, no text has been added to the initialization fragment. However, when we introduce a new global variable `ErrorCount` somewhere later in the program, we can now write:

⟨*Initialize Global Variables*⟩≡
```
ErrorCount = 0;
```

Here we have started to define the contents of ⟨*Initialize Global Variables*⟩. When our literate program is turned into source code suitable for compiling, the literate programming system will substitute the code `ErrorCount = 0;` inside the definition of the `InitGlobals` function. Later on, we may introduce another global `FragmentsProcessed`, and we can append it to the fragment:

⟨*Initialize Global Variables*⟩+≡
```
FragmentsProcessed = 0;
```

The +≡ symbol after the fragment name shows that we have added to a previously defined fragment. When tangled, the result of the above fragment definitions is the code:

```
void InitGlobals() {
```

```
  ErrorCount = 0;
  FragmentsProcessed = 0;
}
```

By making use of the text substitution that is made easy by fragments, we can decompose complex functions into logically-distinct parts. This can make their operation substantially easier to understand. We can write a function as a series of fragments:

⟨*Function Definitions*⟩+≡
```
  void func(int x, int y, double *data) {
      ⟨Check validity of arguments⟩
      if (x < y) {
          ⟨Swap parameter values⟩
      }
      ⟨Do precomputation before loop⟩
      ⟨Loop through and update data array⟩
  }
```

The text of each fragment is then expanded inline in `func` for the compiler. In the document, we can introduce each fragment and its implementation in turn–these fragments may of course include additional fragments, etc. This style of decomposition lets us write code in collections of just a handful of lines at a time, making it easier to understand in detail. Another advantage of this style of programming is that by separating the function into logical fragments, each one can be written and verified independently–in general, we will try to make each fragment less than ten lines or so of code, making it easier to understand the operation of the function as a whole.

Of course, inline functions could be used similarly in a traditional programming environment, but using fragments to decompose functions has three advantages. The first is that all of the fragments can immediately refer to all of the parameters of the original function as well as any function-local variables that are declared in preceeding fragments; it's not necessary to pass them all as parameters, as would need to be done with inline functions. The second advantage is that one generally names fragments with more descriptive names than one gives to functions; this improves program readability and understandability. Finally, since it's so easy to use fragments to decompose complex functions, one does more decomposition in practice.

In some sense, the literate programming language is just an enhanced macro substitution language tuned to the task of rearranging program source code provided by the user. The simplicity of the task of this program can belie the mental shift in programming methodology that literate programming leads to.

1.2 Overview of Rendering

Computer graphics is often divided into three main sub-problems: modeling, animation, and rendering. *Modeling* generally refers to geometric modeling, and involves specifying the precise, digital description of the shape of an object. Topics in modeling include curved surface representations such as subdivision surfaces, quadric surfaces and splines, as well as methods for representing dense polygon meshes. *Animation* deals with the specification of motion. Motion may involve the laws of real physics, or cartoon physics. This book is concerned with the problem of *rendering*. Rendering is the process of producing an image from a description of a scene. For this reason, rendering is sometimes referred to by the more precise name of *image synthesis*.

The scene description input to the rendering system must specify all the different aspects of objects and the environment that determined their appearance when viewed with a camera. Appearance in turn depends on shape, motion, light and color, texture, reflection and illumination. For the purpose of creating a single frame of an animation or a still picture, we can ignore motion (except, as we will see, when we model motion blur, the bluring of moving objects over the time range the camera's shutter is open). Texture and reflection will typically be combined into a model of the material. Scenes also consist of light sources and a camera.

In recent years, rendering algorithms have advanced from ad-hoc methods chosen mainly for computational efficiency to physically-based algorithms that try to model the physics of light propagation and scattering at a more detailed and accurate level of abstraction. In conjunction with more sophisticated techniques for solving the mathematics of these new problems, the field of rendering continues to improve the accuracy and realism of rendered images.

1.3 Overview of the Program

`lrt` has four main phases:

1. Defining the scene
2. Simulating the camera
3. Tracing rays to compute visible objects
4. Shading and lighting, which may trace more rays

1.4 Defining the Scene

An important part of a renderer is the interface that it provides for specifying the scene to be rendered. Scene descriptions are communicated to `lrt` via text scene description files; statements in the file set up rendering options and describe the geometry, materials, and lights in the scene.

The `main` function is pretty simple; it parses scene input from these input files, specified on the command line. `main` surrounds the scene parser with the API calls `RiBegin` and `RiEnd`, which perform general system initialization and cleanup, respectively.

⟨*main program*⟩≡
```
int main(int argc, char *argv[]) {
    fprintf(stderr, "lrt version %1.3f of %s at %s\n", LRT_VERSION,
                __DATE__, __TIME__);
    ⟨Set debugging environment⟩
    RiBegin();
    ⟨Process scene representation⟩
    RiEnd();
    return 0;
}
```

If the user ran `lrt` with no command-line arguments, then the scene description is read from standard input. Otherwise we loop through the command line arguments, processing each input filename in turn.

⟨*Process scene representation*⟩≡
```
if (argc == 1) {
    ⟨Parse scene from standard input⟩
} else {
    ⟨Parse scene from input files⟩
}
```

The `ParseFile` function parses a text scene description file, either from standard input or from a file on disk. The mechanics of parsing scene description files will not be described here.

563 curGfxOptions
498 Error
569 MakeScene
7 Render
562 RiBegin
562 RiEnd

⟨*Parse scene from standard input*⟩≡
```
ParseFile("-");
```

⟨*Parse scene from input files*⟩≡
```
for (int i = 1 ; i < argc ; i++) {
    if (!ParseFile(argv[i]))
        Error("Couldn't open input file \"%s\"\n", argv[i]);
}
```

As the scene file is parsed, objects are created that represent the camera, lights, and the geometric primitives in the scene. Each primitive has attributes, such as a transformation that positions it in the scene, and material properties that describe its texture and reflection properties.

At the end of the description of the scene for a particular frame, code in the fragment ⟨*Create scene and render*⟩ will be executed. Information about the different types of objects and their parameters is stored in the *graphics options* object, `curGfxOptions`. The graphics options object has a method that bundles all of the information together into a `Scene`.

⟨*Create scene and render*⟩≡
```
Scene *scene = curGfxOptions->MakeScene();
scene->Render();
delete scene;
```

The `Scene` holds a number of key objects. All of the light sources in the scene are there, and all of the geometric primitives are managed by a `Primitive` that the scene holds.

The camera object controls the viewing and lens parameters such as field of view and aperture. The film is considered part of the camera object; it handles image storage. After the image has been comupted, a sequence of image operations is applied to make adjustments to the image before finally writing it to disk. The `Camera` and `Film` classes are described in Chapter 6 and the imaging process and `DisplayInfo` class are described in Chapter 8.

The `Sampler` object controls how the image plane is sampled in order to compute pixel values. The sample values are then mapped to rays by the camera. Generating good distributions of samples is an important part of the rendering process and is discussed in Chapter 7.

The `Integrator` object controls the overall technique used to simulate light transport in the scene. Example integrators include ray casting and recursive ray tracing. See Chapter 15 for these and other, more sophisticated integrators.

⟨*Scene Data*⟩≡
```
vector<Light *> lights;
Primitive *prims;
Camera *camera;
Sampler *sampler;
SurfaceIntegrator *surfaceIntegrator;
VolumeIntegrator *volumeIntegrator;
vector<VolumeRegion *> volumeRegions;
DisplayInfo *displayInfo;
```

DisplayInfo 241
Light 358
lights 579
Point 21
Primitive 9
Sampler 198
Scene 5
VolumeIntegrator 484
VolumeRegion 383
volumeRegions 579

⟨*Scene Data*⟩+≡
```
BBox bound;
Point center;
Float radius;
```

⟨*Scene Method Declarations*⟩+≡
```
BBox Scene::WorldBound() const {
    return bound;
}
```

⟨*Scene Method Declarations*⟩+≡
```
void BoundingSphere(Point *c, Float *rad) const {
    *c = center;
    *rad = radius;
}
```

1.5 Simulating the Camera

To compute an image, the `scene`'s `Render` method is invoked. For each of a series of positions on the image plane, this method shoots a ray from the camera out into the scene. By computing the color along that ray for all of those image positions, the image of the scene is generated.

⟨*Scene Methods*⟩+≡
```
void Scene::Render() {
    ⟨Declare variables for progress reporting⟩
    Sample *sample = surfaceIntegrator->AllocateSample(this);
    while (sampler->GetNextSample(sample)) {
        ⟨Report rendering progress⟩
        ⟨Compute camera ray⟩
        ⟨Evaluate radiance along ray⟩
        ⟨Add sample to image⟩
    }
    delete sample;
    cerr << endl;
    camera->film->WriteDisplayImage(*displayInfo);
}
```

⟨*Declare variables for progress reporting*⟩≡
```
ProgressReporter progress(sampler->TotalSamples(), "Rendering");
```

⟨*Report rendering progress*⟩≡
```
static StatsCounter cameraRaysTraced("Camera", "Camera Rays Traced");
++cameraRaysTraced;
progress(stderr);
```

`lrt` provides a camera model that simulates the process of image formation. A `Sampler` generates a series of multi-dimensional sample values to be used by the `Camera` to construct the position and direction of each of the rays traced into the scene and the integrators for Monte Carlo integration. The main function of the `Camera` class is to provide a `GenerateRay` method, which does the mapping from sample values to rays.

⟨*Camera Interface Declarations*⟩≡
```
virtual void GenerateRay(const Sample *sample,
    Ray &ray) const = 0;
```

⟨*Compute camera ray*⟩≡
```
RayDifferential ray;
camera->GenerateRay(sample, ray);
⟨Generate ray differentials for camera ray⟩
```

We also generate the rays for offsets of one pixel in the x and y direction on the image and store them in the `RayDifferential` declared in the previous fragment. These will be useful for anti-aliasing computations in Chapter 11.

444 AllocateSample
173 film
198 GetNextSample
498 ProgressReporter
26 Ray
26 RayDifferential
5 Scene
501 StatsCounter
199 TotalSamples
240 WriteDisplayImage

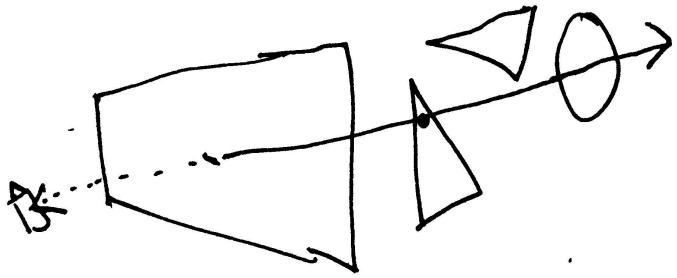


Figure 1.1: The basic ray-tracing algorithm. For each of a series of positions on the plane, the camera constructs a ray through that position out into the scene. The closest visible object along that ray is found by performing intersection tests with objects in the scene and recording which hit, if any, is the closest one to the camera. Shading computations are performed at the intersection point to comupte a color to store in the image.

Spectrum 155
SurfaceIntegrator::L 444

⟨*Generate ray differentials for camera ray*⟩≡
```
++sample->imagex;
camera->GenerateRay(sample, ray.rx);
--sample->imagex;
++sample->imagey;
camera->GenerateRay(sample, ray.ry);
ray.hasDifferentials = true;
```

Given a ray, the `SurfaceIntegrator` traces that ray through the scene and computes the light or radiance `L` that returns to the camera along the ray–see Figure 1.1. Details of an integrator will be shown in the next section. Scene radiance values are represented with the `Spectrum` class, the abstraction that defines the representation of general energy distributions by wavelength.

⟨*Evaluate radiance along ray*⟩≡
```
Float alpha;
Spectrum L = surfaceIntegrator->L(this, ray, sample, &alpha);
L += volumeIntegrator->L(this, ray, sample, &alpha);
```

In addition to returning the ray's radiance, the integrator sets the `alpha` variable to the *alpha value* at the hit point. Alpha can be thought as an extra component in an image, encoding the opacity of each pixel. If the ray hits an opaque object, `alpha` will be one. If the object is partially transparent, `alpha` will be between zero and one, and if no object is intersected, `alpha` is zero. Storing an alpha value with each pixel can be useful for a variety of post-processing effects; for example, we can composite a rendered object on top of a photograph, using the pixels in the image of the photograph wherever the rendered image's alpha channel is zero, using the rendered image where its alpha channel is one, and using a mix of the two for the remaining pixels.

After we have the ray's radiance, we can add its value to the image. We first compute the raster-space depth value of the hit point (using the position of the first

intersection found by the SurfaceIntegrator) and initialize Praster. We make sure that the hit found wasn't farther away than the far clip plane (which is at depth 1, after the projection has been applied). We then call the Sampler::AddSample() method, which updates the pixels in the image given the results from this sample. The details of this process are given in section 7.6.

⟨*Add sample to image*⟩≡
```
Float screenz = camera->ScreenDepth(ray(ray.maxt));
if (screenz > 1.) {
    L = 0.;
    alpha = 0.;
}
Point Praster(sample->imagex, sample->imagey, screenz);
sampler->AddSample(camera->film, Praster, L, alpha);
```

1.6 Ray Intersections

⟨*Primitive Declarations*⟩≡
```
class Primitive : public ReferenceCounted<Primitive> {
public:
    ⟨Primitive Interface⟩
    virtual ~Primitive();
};
```

237 AddSample
309 Bump
10 dgShading
47 DifferentialGeometry
173 film
21 Point
26 Ray
509 Reference
508 ReferenceCounted
175 ScreenDepth
10 Surf

Individual objects in the scene as well as collections of objects are represented in lrt as Primitives. Primitives include both the geometric description of the shapes of objects as well as information about their materials. There are five key methods that Primitives implement:

⟨*Primitive Interface*⟩≡
```
virtual BBox WorldBound() const = 0;
virtual bool CanIntersect() const;
virtual bool Intersect(const Ray &r, Surf *s) const = 0;
virtual bool IntersectP(const Ray &r) const = 0;
virtual void Refine(vector<Reference<Primitive> > &refined) const;
```

The first method, WorldBound, returns a box that encompasses the extent of the object in the scene. The BBox class is defined in Section 2.5. In addition to being able to bound themselves, all primitives either must either be able to determine if a given ray passes through them or else refine themselves into a new set of primitives. Repeated refinement must eventually lead to intersectable primitives.

⟨*GeometricPrimitive Methods*⟩+≡
```
void GeometricPrimitive::Bump(const DifferentialGeometry &dg,
        DifferentialGeometry *dgShading) const {
    material->Bump(dg, Ns, Ss, dgShading);
}
```

We will also define a GeometricPrimitive, which represents a single primitive shape (e.g. a sphere) in the scene. GeometricPrimitives are constructed during

the processing of the scene description file. Each is an encapsulation of a `Shape`, a `Material` and possibly an `AreaLight` (if the object is emissive).

As we will see, `GeometricPrimitives` have similar methods to `Shapes` and `Materials`. The geometry of each primitive in `lrt` is represented by an abstract `Shape` class. `lrt` implements many shapes, including triangle meshes, quadric surfaces (spheres, cylinders, cones, paraboloids, hyperboloids), non-uniform rational b-splines (NURBS), and subdivision surfaces. Shapes are discussed in Chapter 3.

The material properties of each primitive are represented by an abstract `Material` class, as described in Chapter 10. `lrt` is capable of modeling many different materials such as glass, mirrors, plastics, and metals. Materials encapsulate the color, texture, and reflective properties of the surface.

⟨*GeometricPrimitive Data*⟩≡
```
Reference<Shape> shape;
Reference<Material> material;
AreaLight *areaLight;
Texture<Normal> *Ns;
Texture<Vector> *Ss;
```

The `GeometricPrimitive`'s `Intersect` method tests a ray against the object to find an intersection. It uses methods of the `Shape` contained in the `GeometricPrimitive` to do the actual test and then updates information in the `Surf` about the hit found, if any.

areaLight 581
AreaLight 368
DifferentialGeometry 47
Material 303
Normal 23
Reference 509
Texture 323
Vector 16

The details of how different shapes perform these intersection test will be discussed in Chapter 3. `Intersect` returns a boolean indicating whether the shape was hit, and information about the hit if an intersection occurs. The hit itself is not represented as a point, but rather as a small patch on the surface. This information is stored in a structure called `DifferentialGeometry` that includes position, normal, tangents to the surface, and surface parameters. Section 2.7 describes the differential geometry abstraction in detail.

⟨*Primitive Declarations*⟩+≡
```
struct Surf {
    ⟨Surf Method Declarations⟩
    ⟨Surf Data Members⟩
};
```

⟨*Surf Data Members*⟩≡
```
DifferentialGeometry dgGeom;
mutable DifferentialGeometry dgShading;
const GeometricPrimitive *primitive;
```

The `Surf` returned from `Intersect` includes both information about the differential geometry of the point on the surface as well as information about its material properties. `dgGeom` is the differential geometry computed by the shape's intersection routine. Another differential geometry object, `dgShading`, holds a possibly-perturbed version of `dgGeom`; this is computed by the shading system and enables effects like bump-mapping, which perturbs the surface normal to simulate bumpy surfaces on smooth base geometry. The primitive is also stored in the `Surf` so that other primitive attributes may be fetched by the shading system if needed.

⟨*GeometricPrimitive Methods*⟩+≡
```
bool GeometricPrimitive::Intersect(const Ray &r, Surf *surf) const {
    Float thit;
    if (shape->Intersect(r, &thit, &surf->dgGeom)) {
        surf->primitive = this;
        r.maxt = thit;
        return true;
    }
    return false;
}
```

⟨*Surf Method Declarations*⟩≡
```
Surf() { primitive = NULL; }
```

1.7 Shading and Lighting

lrt provides a number of different integrators for achieving differing levels of realism or providing different functionality. Here we will present the classic Whitted-style ray-tracing integration method.

⟨*WhittedIntegrator Method Definitions*⟩≡
```
Spectrum WhittedIntegrator::L(const Scene *scene, const RayDifferential &ray,
        const Sample *sample, Float *alpha) const {
    Surf surf;
    Spectrum L(0.);
    if (scene->Intersect(ray, &surf)) {
        if (alpha) *alpha = 1.;
        ⟨Compute emitted and reflected light⟩
    }
    else {
        ⟨Handle ray with no intersection⟩
    }
    return L;
}
```

10	dgGeom
6	prims
26	Ray
26	RayDifferential
5	Scene
55	Shape::Intersect
155	Spectrum
10	Surf
447	WhittedIntegrator

For the integrator to determine what primitive is hit by a ray, it calls the Intersect method of the Scene class. Intersect returns information about the closest hit in the Surf structure, as discussed in the previous section. Because scenes usually contain many distinct geometric primitives, we pass the intersection-test request on to the Scene's lone Primitive. These sets, described in section 4.2, will typically accelerate the ray tracing computation by only testing rays against those objects that the ray is likely to intersect.

⟨*Scene Method Declarations*⟩+≡
```
bool Intersect(const Ray &ray, Surf *surf) const {
    return prims->Intersect(ray, surf);
}
```

These geometric calculations provide half of the functionality of lrt. The other half lies in the shading process. Recall that the integrator returns a power spectrum

along a ray. In the case when a ray intersects a geometric primitive, the reflected and emitted light is returned. The total amount of light returned is represented by a power spectrum in the outgoing ray direction, `-ray.D`.

Because Whitted-style ray tracing works by recursively evaluating radiance along reflected and refracted ray directions, we keep track of the depth of recursion in the variable `rayDepth`. After a predetermined recursion depth, we stop tracing reflected and refracted rays. By default the maximum recursion depth is five. More advanced integrators might use other techniques to terminate computation early. One such technique is *Russian Roulette* sampling, described in section **??**.

⟨*Compute emitted and reflected light*⟩≡
```
⟨Compute emitted light if an area light source⟩
⟨Evaluate BSDF at hit point⟩
⟨Compute reflection by integrating over the lights⟩
if (rayDepth++ < maxDepth) {
    ⟨Trace rays for specular reflection and refraction⟩
}
--rayDepth;
⟨Clean up from integration⟩
```

BSDF 298

If the ray happened to hit geometry that is itself emissive, we compute its emitted radiance by calling the `Surf`'s `Le` method. If the object is not a light source, this method will return 0. Light sources are discussed bn Chapter 12.

⟨*Compute emitted light if an area light source*⟩≡
```
L += surf.Le(-ray.D);
```

To compute reflected light, the integrator must have access to material properties of the surface at the intersection point as well as illumination arriving at that point.

In order to describe the reflection of light at a point, `lrt` uses a class called `BSDF`, which stands for "Bidirectional Scattering-Distribution Function". These functions take an incoming direction and an outgoing direction and return a value that indicates the amount of light that is reflected from the incoming direction to the outgoing direction (actually, BSDF's use a fraction per-wavelength, so they really return a `Spectrum`). `lrt` provides built-in `BSDF` classes for several standard scattering functions used in computer graphics. Examples of BSDFs include Lambertian reflection, and the Torrance-Sparrow microfacet model; these are defined in Chapter 9.

The `BSDF` at a surface point provides all information needed to shade that point, but BSDFs may vary across a surface. Surfaces with complex material properties, such as wood or marble, have a different BSDF at each point. Even if wood is modelled as perfectly diffuse, the diffuse color at each point will depend on the wood's grain. These spatial variations of shading parameters are described with `Textures`, which in turn may be either described procedurally or stored in texture maps; see Chapter 11.

Here we won't go into further detail about BSDFs and texturing; a method in `Surf` returns a pointer to the `BSDF` at the intersection point on the object.

⟨*Evaluate BSDF at hit point*⟩≡
```
BSDF *bsdf = surf.GetBSDF(ray);
```

The class `Light` implements light sources. There are a number of different types of light sources in `lrt`, including point lights, directional lights, area lights, and ambient lights.

For each light, we compute the light energy, or differential irradiance, falling on the surface at the point being shaded by calling the light's `dE()` method. (Radiometric concepts such as energy and differential irradiance are discussed in Chapter 5.) This method also returns the direction vector from the point being shaded to the light source. The `dE()` methods will themselves generally trace rays as well to make sure no other objects are between the light source and the point being shaded, casting a shadow on the point.

To evaluate the contribution to the reflection light, we multiply `dE` by the BSDF. The BSDF is a function of the incoming and outgoing direction. We add the contribution from this light source to a running total of reflected radiance, stored in `L`.

After this step, we have computed reflection due to *direct lighting*: light that arrives at the surface directly from emissive objects (as opposed to light that has reflected off other objects in the scene before arriving at the point.)

⟨*Compute reflection by integrating over the lights*⟩≡

```
Vector wi;
for (u_int i = 0; i < scene->lights.size(); ++i) {
    VisibilityTester visibility;
    Spectrum dE = scene->lights[i]->dE(surf.dgShading.P,
        surf.dgShading.Nn, &wi, &visibility);
    if (!dE.Black() && visibility.Unoccluded(scene))
        L += bsdf->f(-ray.D, wi) * dE *
            visibility.Transmittance(scene);
}
```

300 BSDF::NumSpecular
10 dgShading
579 lights
26 Ray
494 size
155 Spectrum
16 Vector
359 VisibilityTester

In 1979, Turner Whitted developed a new rendering algorithm based on the fact that light scattered by perfectly specular surfaces (like mirrors or glass objects) could be modeled with ray-tracing. When a specularly reflective or transmissive object is hit by a ray, new rays are also traced in the reflected and refracted directions. The radiance along these rays is computed in the same way we compute radiance along camera rays. It is then scaled appropriately and added to the radiance scattered from the original point. For each of the specular components of the BSDF, we have the BSDF generate a ray; if the value of the BSDF in that direction is non-zero, we call the `Scene`'s radiance estimation function, which will call back to the `WhittedIntegrator`'s `L` function. By continuing this process recursively, realistic images of multiple reflection and refraction can be generated.

⟨*Trace rays for specular reflection and refraction*⟩≡

```
for (int i = 0; i < bsdf->NumSpecular(); ++i) {
    Spectrum fr = bsdf->f_delta(i, -ray.D, &wi);
    if (!fr.Black())
        L += scene->L(Ray(surf.dgShading.P, wi)) * fr *
            fabsf(Dot(wi, surf.dgShading.Nn));
}
```

When we're done, we need to free the `BSDF` that the shader returned.

⟨*Clean up from integration*⟩≡
```
delete bsdf;
```

Finally, rays that don't hit any geometry in the scene may still carry radiance back to the image;

XXXX do this via volume rendering stuff?? XXX

⟨*Handle ray with no intersection*⟩≡
```
if (alpha) *alpha = 0.;
for (u_int i = 0; i < scene->lights.size(); ++i)
    L += scene->lights[i]->Le(ray);
if (alpha && !L.Black()) *alpha = 1.;
return L;
```

That's all there is to it folks!

1.8 How to Progress Through this Book

DAG of possibilities, chapter dependencies.

Additional Reading

lights 579
size 494

Knuth's article *Literate Programming*(Knu84) describes the main ideas behind literate programming as well as his `web` programming environment. The entire TEX typesetting system was written with this system and has been published as a book(Knu93a). More recently, Knuth has published a collection of graph algorithms in *The Stanford Graphbase*(Knu93b). Both of these are enjoyable to read and are respectively excellent presentations of modern automatic typesetting and graph algorithms. The website `http://www.literateprogramming.com` has pointers to many articles about literate programming as well as a variety of literate programming systems; many refinements have been made since Knuth's original development of the idea.

The implementation of the `lcc` C compiler is described in a literate program written by Fraser and Hansen and published as *A Regartegable C Compiler: Design and Implementation* (FH95).

A good introduction to the C++ programming language is XXX. Stroustroup XXX is the definitive reference and extensively describes the use of the standard library.

General books on rendering: Radiosity and Realistic Image Synthesis, Principles of Digital Image Synthesis, Introduction to Ray Tracing, Roy Hall.

Appel ray tracing shadows, etc (App68)

Kay and Greenberg on transparency (KG79)

Whitted original paper (Whi80)

2. Geometry and Transformations

⟨*geometry.h**⟩≡
  ⟨*Source Code Copyright*⟩
  `#ifndef GEOMETRY_H`
  `#define GEOMETRY_H`
  `#include "lrt.h"`
  `#include <float.h>`
  ⟨*Geometry Classes*⟩
  ⟨*Geometry Inline Functions*⟩
  `#endif // GEOMETRY_H`

⟨*geometry.cc**⟩≡
  ⟨*Source Code Copyright*⟩
  `#include "lrt.h"`
  `#include "geometry.h"`
  ⟨*BBox Method Definitions*⟩

We now present the fundamental geometric primitives around which `lrt` is built. Our representation of actual scene geometry (triangles, etc.) is presented in Chapter 3; here we will discuss fundamental building blocks of 3D graphics, such as points, vectors, rays, and transformations. We assume that the reader is familiar with the basics of vector geometry and linear algebra.

Affine Spaces

In order to compute numeric coordinates for points and vectors, we need also to have a *coordinate system* that their coordinates are in relation to. An *affine space* is defined by a *frame* given by a point $\mathbf{p}_o$ (the *origin* of the space), and a set of *basis vectors*. In an n-dimensional space, the basis vectors are a set of n

linearly independent vectors. All vectors $\vec{\mathbf{v}}$ in the space can be expressed as a linear combination of the basis vectors. Given a vector $\vec{\mathbf{v}}$ and the basis vectors $\vec{\mathbf{v}}_i$, we can compute scalar values s_i such that

$$\vec{\mathbf{v}} = s_1\vec{\mathbf{v}}_1 + \cdots + s_n\vec{\mathbf{v}}_n$$

The scalars s_i are the *representation* of $\vec{\mathbf{v}}$ with respect to the basis. Similarly, for all points $\mathbf{p}$, we can compute scalars s_i such that

$$\mathbf{p} = \mathbf{p}_o + s_1\vec{\mathbf{v}}_1 + \cdots + s_n\vec{\mathbf{v}}_n$$

This brings us to an ambiguity, however: to define a frame we need a point and a set of vectors. But we can only meaningfully talk about points and vectors with respect to a particular frame. Therefore, we will define a *standard frame* with origin $(0,0,0)$ and basis vectors $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$ that other frames will be defined with respect to. We will call this coordinate system *world space*; all other coordinate systems are defined in terms of it.

2.1 Vectors

⟨*Geometry Classes*⟩≡
```
class Vector {
public:
    ⟨Vector Constructors⟩
    ⟨Vector Methods⟩
    ⟨Vector Public Data⟩
};
```

A *vector* is a direction in 3D space. The most convenient representation of a vector is a three-tuple of components that give its magnitude in terms of the *x*, *y*, and *z* axes of the space it is defined in. The individual components of a vector $\vec{\mathbf{v}}$ will be written $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$.

⟨*Vector Public Data*⟩≡
```
Float x, y, z;
```

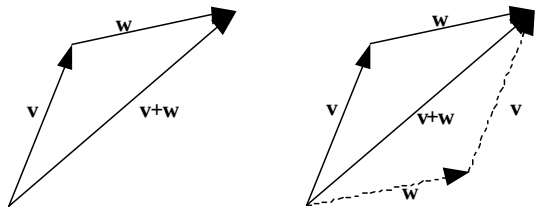
The `Vector` constructor allows values for `x`, `y`, and `z` to be passed in. The default for all these values is 0.0.

⟨*Vector Constructors*⟩≡
```
Vector()
    : x(0.), y(0.), z(0.) {
}
```

⟨*Vector Constructors*⟩+≡
```
Vector(Float xx, Float yy, Float zz)
    : x(xx), y(yy), z(zz) {
}
```

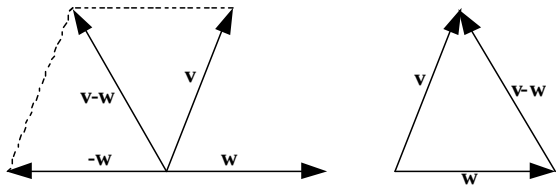
Arithmetic

Adding and subtracting vectors is done component-wise. The usual geometric interpretation of vector addition and subtraction is shown in Figures 2.1 and 2.2.



(a) $\vec{\mathbf{v}}+\vec{\mathbf{w}}$ (b) $\vec{\mathbf{v}}+\vec{\mathbf{w}}=\vec{\mathbf{w}}+\vec{\mathbf{v}}$

Figure 2.1: Vector addition. Notice that the sum $\vec{\mathbf{v}}+\vec{\mathbf{w}}$ forms the diagonal of the parallelogram formed by $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$. Also, the figure on the right shows the commutativity of vector addition.



(a) (b)

Figure 2.2: Vector subtraction. The difference $\vec{\mathbf{v}}-\vec{\mathbf{w}}$ is the other diagonal of the parallelogram formed by $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$.

⟨*Vector Methods*⟩≡
```
Vector operator+(const Vector &v) const {
    return Vector(x + v.x, y + v.y, z + v.z);
}

Vector& operator+=(const Vector &v) {
    x += v.x; y += v.y; z += v.z;
    return *this;
}
```

The code for subtracting two vectors is similar, and not shown here.

Scaling

We can also multiply a vector component-wise by a scalar, effectively changing its length. We need three functions to do this in order to cover all of the different ways that this operation may be written in source code (e.g. `v*s`, `s*v`, and `v *= s`.)

⟨*Vector Methods*⟩+≡
```
Vector operator*(Float f) const {
    return Vector(f*x, f*y, f*z);
}

Vector &operator*=(Float f) {
    x *= f; y *= f; z *= f;
    return *this;
}
```

Vector 16

⟨*Geometry Inline Functions*⟩+≡
```
inline Vector operator*(Float f, const Vector &v) { return v*f; }
```

Similarly, a vector can be divided component-wise by a scalar. The code for scalar division is similar to scalar multiplication, though division of a scalar by a vector is not well-defined, so is not included. Here we will use the optimization of turning three divides into one divide to compute the reciprocal and then three multiplications.

⟨*Vector Methods*⟩+≡
```
Vector operator/(Float f) const {
    Float inv = 1.f/f;
    return Vector(x * inv, y * inv, z * inv);
}

Vector &operator/=(Float f) {
    Float inv = 1.f/f;
    x *= inv; y *= inv; z *= inv;
    return *this;
}
```

We also provide the unary negation operator for `Vectors`. This returns a new vector pointing in the opposite direction of the original one.

⟨*Vector Methods*⟩+≡
```
Vector operator-() const {
    return Vector(-x, -y, -z);
}
```

Normalization

It is often necessary to *normalize* a vector; that is, to compute a new vector pointing in the same direction but with length of one. To do this, we divide each component by the length of the vector, denoted in text by $\|\vec{\mathbf{v}}\|$. The method to do this is called `Hat`, which is a common mathematical notation for a normalized vector.

⟨*Vector Methods*⟩+≡
```
Float LengthSquared() const { return x*x + y*y + z*z; }
Float Length() const { return sqrtf( LengthSquared() ); }
Vector Hat() const { return (*this)/Length(); }
```

Dot and Cross Product

Two further useful operations on vectors are the dot product (also known as the scalar or inner product) and the cross product. For two vectors $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$, their *dot product* $(\vec{\mathbf{v}} \cdot \vec{\mathbf{w}})$ is defined as

16 Vector

$$\mathbf{v}_x\mathbf{w}_x + \mathbf{v}_y\mathbf{w}_y + \mathbf{v}_z\mathbf{w}_z$$

⟨*Geometry Inline Functions*⟩+≡
```
inline Float Dot(const Vector &v1, const Vector &v2) {
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}
```

The dot product has a simple relationship to the angle between the two vectors:

$$(\vec{\mathbf{v}} \cdot \vec{\mathbf{w}}) = \|\vec{\mathbf{v}}\| \, \|\vec{\mathbf{w}}\| \cos\theta$$

where θ is the angle between $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$. It follows from this that $(\vec{\mathbf{v}} \cdot \vec{\mathbf{w}})$ is zero if and only if $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$ are perpendicular (provided that neither $\vec{\mathbf{v}}$ nor $\vec{\mathbf{w}}$ is *degenerate*– equal to $(0,0,0)$). Forthermore, if $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$ are both of unit length, we can easily compute the cosine of the angle between them with their dot product. As the cosine of the angle between two vectors often needs to be computed in computer graphics, we will frequently make use of this property.

A few basic properties directly follow from the definition. If $\vec{\mathbf{u}}$, $\vec{\mathbf{v}}$, and $\vec{\mathbf{w}}$ are vectors and s is a scalar value, then

$$\begin{aligned} (\vec{\mathbf{u}} \cdot \vec{\mathbf{v}}) &= (\vec{\mathbf{v}} \cdot \vec{\mathbf{u}}) \\ (s\vec{\mathbf{u}} \cdot \vec{\mathbf{v}}) &= s(\vec{\mathbf{v}} \cdot \vec{\mathbf{u}}) \\ (\vec{\mathbf{u}} \cdot (\vec{\mathbf{v}} + \vec{\mathbf{w}})) &= (\vec{\mathbf{u}} \cdot \vec{\mathbf{v}}) + (\vec{\mathbf{u}} \cdot \vec{\mathbf{w}}) \end{aligned}$$

The *cross product* is another useful vector operation. Given two vectors in 3D, the cross product $\vec{\mathbf{v}} \times \vec{\mathbf{w}}$ is a vector that is perpendicular to both of them. It is defined as:

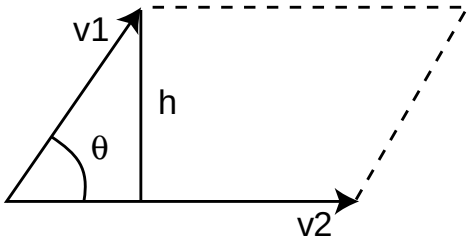


Figure 2.3: The area of a parallelogram with edges given by vectors $\vec{\mathbf{v}}_1$ and $\vec{\mathbf{v}}_2$ is equal to $\vec{\mathbf{v}}_2 h$. The cross product can easily compute this value as $\vec{\mathbf{v}}_1 \times \vec{\mathbf{v}}_2$.

$$\begin{aligned}(\tilde{\mathbf{v}}\times\tilde{\mathbf{w}})_x &= (\mathbf{v}_y\mathbf{w}_z) - (\mathbf{v}_z\mathbf{w}_y)\\ (\tilde{\mathbf{v}}\times\tilde{\mathbf{w}})_y &= (\mathbf{v}_z\mathbf{w}_x) - (\mathbf{v}_x\mathbf{w}_z)\\ (\tilde{\mathbf{v}}\times\tilde{\mathbf{w}})_z &= (\mathbf{v}_x\mathbf{w}_y) - (\mathbf{v}_y\mathbf{w}_x)\end{aligned}$$

XXX left-handed vs. right handed coordinate systems, etc... XXX

An easy way to remember this is to compute the "determinant" of the matrix:

Vector 16

$$\vec{\mathbf{v}}\times\vec{\mathbf{w}} = \begin{vmatrix} i & \mathbf{v}_x & \mathbf{w}_x \\ j & \mathbf{v}_y & \mathbf{w}_y \\ k & \mathbf{v}_z & \mathbf{w}_z \end{vmatrix}$$

where i, j, and k represent the axes $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$, respectively.

⟨*Geometry Inline Functions*⟩+≡

```
inline Vector Cross(const Vector &v1, const Vector &v2) {
    return Vector((v1.y * v2.z) - (v1.z * v2.y),
                  (v1.z * v2.x) - (v1.x * v2.z),
                  (v1.x * v2.y) - (v1.y * v2.x));
}
```

Using the basic properties of the dot product, it can be shown that if $\vec{\mathbf{u}} = \vec{\mathbf{v}}\times\vec{\mathbf{w}}$, then

$$\|\vec{\mathbf{u}}\| = \|\vec{\mathbf{v}}\|\,\|\vec{\mathbf{w}}\|\sin\theta, \tag{2.1.1}$$

where θ is the angle between $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$. An important implication of this is that the cross product of two perpendicular unit vectors is itself a unit length vector. Note also that the result of the cross product is a degenerate vector if $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$ are parallel.

This definition also shows a convenient way to compute the area of a parallelogram–see Figure 2.3. If the two edges of the parallelogram are given by vectors $\vec{\mathbf{v}}_1$ and $\vec{\mathbf{v}}_2$, and has height h, the area is given by $\|\vec{\mathbf{v}}_2\|\,h$. Since $h = \sin\theta\,\|\vec{\mathbf{v}}_1\|$, we can use Equation 2.1.1 to see that the area is $\vec{\mathbf{v}}_1 \times \vec{\mathbf{v}}_2$.

Coordinate system from a vector

We can use the fact that the cross product gives a vector orthogonal to the two vectors to write a function that takes one vector and returns two new vectors so that the three of them form an orthonormal coordiante system. Specifically, all three of the vectors will be perpendicular to each other. Note that the other two vectors

returned are only unique up to a rotation about the given vector. This function assumes that the vector passed in, v1, has already been normalized.

We first construct a perpendicular vector by zeroing one of the two components of the original vector and permuting the remaining two. Inspection of the two cases should make clear that v2 will be normalized and that the dot product $(v_1 \cdot v_2)$ will be equal to zero. Given these two perpendicular vectors, one more cross product wraps things up to give us the third, which by definition of the cross product will be be perpendicular to the first two.

⟨*Geometry Inline Functions*⟩+≡
```
inline void CoordinateSystem(const Vector &v1,  Vector *v2, Vector *v3) {
    if (fabsf(v1.x) > fabsf(v1.y)) {
            Float invLen = 1.f / sqrtf(v1.x*v1.x + v1.z*v1.z);
            *v2 = Vector(-v1.z * invLen, 0.f, v1.x * invLen);
    }
    else {
        Float invLen = 1.f / sqrtf(v1.y*v1.y + v1.z*v1.z);
        *v2 = Vector(0.f, v1.z * invLen, -v1.y * invLen);
    }
    *v3 = Cross(v1, *v2);
}
```
20 Cross
16 Vector

2.2 Points

⟨*Geometry Classes*⟩+≡
```
class Point {
public:
    ⟨Point Constructors⟩
    ⟨Point Methods⟩
    ⟨Point Public Data⟩
};
```

A point is a zero-dimensional quantity that represents a location in 3D space. To represent a Point, we simply need to know its *x*, *y*, and *z* coordinates with respect to its coordinate system. Although the same (x,y,z) representation is used as is used for vectors, the fact that a point represents a position and a vector represents a direction leads to a number of important differences in how they are treated.

⟨*Point Public Data*⟩≡
```
Float x,y,z;
```

⟨*Point Constructors*⟩≡
```
Point()
    : x(0.), y(0.), z(0.) {
}
```

⟨*Point Constructors*⟩+≡
```
Point(Float xx, Float yy, Float zz)
    : x(xx), y(yy), z(zz) {
}
```

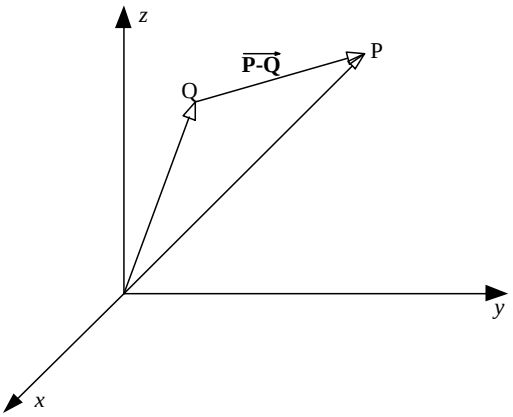


Figure 2.4: Obtaining the vector between two points. The vector $\mathbf{P} \vec{-} \mathbf{Q}$ is the component-wise subtraction of the points **P** and **Q**.

There are certain `Point` methods which either return or take a `Vector`. For instance, you can add a vector to a point, offsetting it in the given direction, obtaining a new point. Alternately, you can subtract one point from another, obtaining the vector between them, as shown in Figure 2.4.

Point 21
Vector 16

⟨*Point Methods*⟩≡

```
Point operator+(const Vector &v) const {
    return Point(x + v.x, y + v.y, z + v.z);
}

Point &operator+=(const Vector &v) {
    x += v.x; y += v.y; z += v.z;
    return *this;
}

Vector operator-(const Point &p) const {
    return Vector(x - p.x, y - p.y, z - p.z);
}

Point operator-(const Vector &v) const {
    return Point(x - v.x, y - v.y, z - v.z);
}

Point &operator-=(const Vector &v) {
    x -= v.x; y -= v.y; z -= v.z;
    return *this;
}
```

The distance between two points is easily computed by subtracting the two of them to compute a vector and then finding the length of that vector.

⟨*Geometry Inline Functions*⟩+≡
```
inline Float Distance(const Point &p1, const Point &p2) {
    return (p1 - p2).Length();
}
inline Float DistanceSquared(const Point &p1, const Point &p2) {
    return (p1 - p2).LengthSquared();
}
```

Although it doesn't make sense mathematically to weight points by a scalar or add two points together, we will still allow these operations in order to be able to compute weighted sums of points, which does make sense so long as the weights used sum to one.

⟨*Point Methods*⟩+≡
```
Point &operator+=(const Point &p) {
    x += p.x; y += p.y; z += p.z;
    return *this;
}
```

⟨*Point Methods*⟩+≡
```
Point operator+(const Point &p) {
    return Point(x + p.x, y + p.y, z + p.z);
}
```

19 Length
19 LengthSquared
21 Point

⟨*Point Methods*⟩+≡
```
Point operator* (Float f) const {
    return Point(f*x, f*y, f*z);
}
```

⟨*Point Methods*⟩+≡
```
Point &operator*=(Float f) {
    x *= f; y *= f; z *= f;
    return *this;
}
```

⟨*Geometry Inline Functions*⟩+≡
```
inline Point operator*(Float f, const Point &p) { return p*f; }
```

2.3 Normals

⟨*Geometry Classes*⟩+≡
```
class Normal {
public:
    ⟨Normal Constructors⟩
    ⟨Normal Methods⟩
    ⟨Normal Public Data⟩
};
```

A *surface normal* is a vector that is perpendicular to a surface at a particular position. It can be defined as the cross product of any two non-parallel vectors that are tangent to the surface at a point. Although normals have some similarities with

vectors, it is important to distinguish between the two of them; because normals are defined in terms of their relationship to a particular surface. For example they behave differently with respect to transformations; this difference is discussed in Section 2.6.

The implementations of `Normal`s and `Vector`s are very similar: like vectors, normals are represented by three `Float`s `x`, `y`, and `z`, they can be added and subtracted to compute new normals and they can be scaled and normalized. However, a normal cannot be added to a point and we cannot take the cross product of two normals. Note that in an unfortunate turn of terminology normals are *not* necessarily normalized.

We provide an extra `Normal` constructor that constructs a `Normal` from a `Vector`. In order to ensure that this conversion only happens when specifically intended, the C++ `explicit` keyword is added. We will also add a `Vector` constructor that goes the other way.

⟨*Normal Constructors*⟩+≡
```
explicit Normal(const Vector &v)
  : x(v.x), y(v.y), z(v.z) {}
```

⟨*Vector Constructors*⟩+≡
```
explicit Vector(const Normal &n);
```

Normal 23
Vector 16

⟨*Geometry Inline Functions*⟩+≡
```
inline Vector::Vector(const Normal &n)
  : x(n.x), y(n.y), z(n.z) { }
```

Thus, given the declarations `Vector v; Normal n;`, the assignment `n = v` is illegal, so we must explicitly convert the vector, as in `n = Normal(v)`.

We also overload the `Dot` function to compute dot products between the various possible combinations of normals and vectors.

⟨*Geometry Inline Functions*⟩+≡
```
inline Float Dot(const Normal &n1, const Vector &v2) {
    return n1.x * v2.x + n1.y * v2.y + n1.z * v2.z;
}
inline Float Dot(const Vector &v1, const Normal &n2) {
    return v1.x * n2.x + v1.y * n2.y + v1.z * n2.z;
}
inline Float Dot(const Normal &n1, const Normal &n2) {
    return n1.x * n2.x + n1.y * n2.y + n1.z * n2.z;
}
```

2.4 Rays

⟨*Geometry Classes*⟩+≡
```
class Ray {
public:
    ⟨Ray Constructor Declarations⟩
    ⟨Ray Method Declarations⟩
    ⟨Ray Public Data⟩
};
```

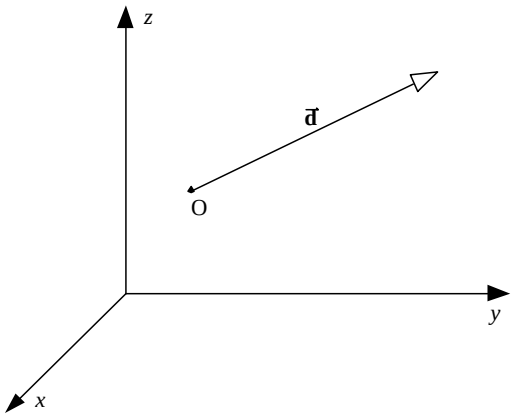


Figure 2.5: A ray is a semi-infinite line defined by its *origin* and *direction*.

A *ray* is a semi-infinite line specified by its origin and direction. We represent a `Ray` with a `Point` for the origin, and a `Vector` for the direction. A ray is denoted as $\mathbf{r}$; it has origin $o(\mathbf{r})$ and direction $\vec{\mathbf{d}}(\mathbf{r})$, as shown in Figure 2.5.

The *parametric form* of a ray gives the set of points that the ray passes through:

$$\mathbf{r}(t) = o(\mathbf{r}) + t\vec{\mathbf{d}}(\mathbf{r}) \tag{2.4.2}$$

21 `Point`
26 `Ray`
16 `Vector`

Because we will be referring to these variables often throughout the code, the origin and direction members of a `Ray` are named simply `O` and `D`.

⟨*Ray Public Data*⟩≡
```
Point O;
Vector D;
```

In addition, we include fields to restrict the ray to a particular segment. These fields, called `mint` and `maxt`, allow us to restrict the ray to a potentially finite segment of points $[\mathbf{r}(\text{mint}), \mathbf{r}(\text{maxt}))$. Notice that these fields are declared as `mutable`, meaning that they can be changed even if the `Ray` structure that contains them is `const`. Because we need to update these fields all the time, we elect to keep the code simpler rather than adding mutator methods.

⟨*Ray Public Data*⟩+≡
```
mutable Float mint, maxt;
```

For simulating motion blur, each ray may have a unique time value associated with it. The rest of the renderer is responsible for constructing a representation of the scene at the appropriate time for each ray.

⟨*Ray Public Data*⟩+≡
```
Float time;
```

Constructing `Ray`s is straightforward. A default constructor is provided, which lets the default constructors of `Point`s and `Vector`s set the origin and direction to $(0,0,0)$. Alternatively, a particular point and direction can be provided. Also note that `mint` is initialized to a small constant rather than 0. This is a classic ray-tracing hack to avoid false self-intersections due to floating point precision problems.

⟨*Ray Constructor Declarations*⟩≡
```
Ray(): mint(RAY_EPSILON), maxt(FLT_MAX), time(0.f) {}
Ray(const Point &origin, const Vector &direction,
    Float start = RAY_EPSILON, Float end = FLT_MAX, Float t = 0.f)
    : O(origin), D(direction), mint(start), maxt(end), time(t) {
}
```

⟨*Global Constants*⟩≡
```
#define RAY_EPSILON 1e-3f
```

Because a ray can be thought of as a function of a single parameter t, we will overload the function application operator for rays. This way, when we need to find the point at a particular distance along a ray, we can write code like:

```
Ray r(Point(0,0,0), Vector(1,2,3));
Point p = r(1.7);
```

⟨*Ray Method Declarations*⟩≡
```
Point operator()(Float t) const { return O + D * t; }
```

Ray differentials

Point 21
Vector 16

In order to be able perform better anti-aliasing with the texture functions defined in Chapter 11, we will keep track of some additional information with each camera ray that we trace. In Section 11.1, we will use this information to estimate the area on the image plane that a part of the scene being shaded projects to. From this, we can compute the texture's average value over that area, leading to a better final image.

With each ray, we store information about two auxiliary rays in the `RayDifferential` class. These two rays are represent camera rays offset one pixel in the x and y directions. By determining at the area that these three rays project to on the object being shaded, we can estimate the filter extent necessary for proper anti-aliasing.

Because the `RayDifferential` class inherits from `Ray`, geometric interfaces in the system are written to take `const Ray &` values, so that either a `Ray` or `RayDifferential` can be passed in and the routines can just treat either as a `Ray`. Only the routines related to anti-aliasing and texturing need to take `RayDifferential` parameters.

⟨*Geometry Classes*⟩+≡
```
class RayDifferential : public Ray {
public:
    ⟨RayDifferential Constructors⟩
    ⟨RayDifferential Public Data⟩
};
```

⟨*RayDifferential Constructors*⟩≡
```
RayDifferential() { hasDifferentials = false; }
RayDifferential(const Ray &ray) : Ray(ray) { hasDifferentials = false; }
```

⟨*RayDifferential Public Data*⟩≡
```
bool hasDifferentials;
Ray rx, ry;
```

2.5 Axis-Aligned Bounding Boxes

Two-dimensional extents

⟨*Geometry Classes*⟩+≡
```
struct Extent2D {
    ⟨Extent2D Constructors⟩
    ⟨Extent2D Data⟩
};
```

It's useful to have a structure that holds a representation of an axis-aligned region of space in two-dimensions; `Extent2D` takes care of that here. This will be useful later, for example in Chapters 6 and 7, where it will simplify a number of functions by saving us from needing to pass four individual `Float`s when we are providing them with a 2D region on the image plane.

⟨*Extent2D Constructors*⟩≡
```
Extent2D() { x0 = x1 = y0 = y1 = 0.; }
Extent2D(Float xx0, Float xx1, Float yy0, Float yy1) {
    x0 = min(xx0, xx1);
    x1 = max(xx0, xx1);
    y0 = min(yy0, yy1);
    y1 = max(yy0, yy1);
}
```

513 max
513 min

⟨*Extent2D Data*⟩≡
```
Float x0, x1, y0, y1;
```

Three-dimensional bounding boxes

⟨*Geometry Classes*⟩+≡
```
class BBox {
public:
    ⟨BBox Constructors⟩
    ⟨BBox Method Declarations⟩
    ⟨BBox Public Data⟩
};
```

The scenes that we will render will often contain objects that are computationally expensive to process. For many operations, it is often useful to have a three-dimensional *bounding volume* that encloses an object. If, for example, we know that we cannot see the bounding volume, we can avoid processing all of the objects inside of it.

The measurable benefit of this technique is related to two factors: the expense of processing the bounding volume compared to the expense of processing the objects inside of it, and the tightness of the fit. If we have a very loose bound around an object, we will often incorrectly determine that its contents need to be examined further. However, in order to make the bounding volume a closer fit, it may be necessary to make the volume a complex object itself, and the expense of processing it increases.

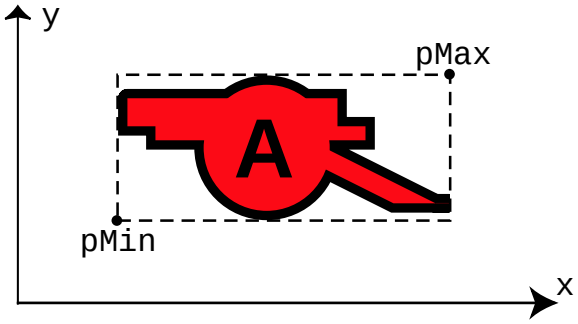


Figure 2.6: An example axis-aligned bounding box. We store only the coordinates of the minimum and maximum points of this box; all other box corners are implicit in this representation.

There are many choices for bounding volumes; we will be using *axis-aligned bounding boxes* (AABBs). (Other popular choices are spheres and *oriented bounding boxes* (OBBs)). An AABB can be described by one of its vertices and three lengths, each representing the distance spanned along the *x*, *y*, and *z* coordinate axes. Alternatively, two opposite vertices of the box describe it. We will store the positions of the vertex with minimum *x*, *y*, and *z* values, and the one with maximum *x*, *y*, and *z*. A 2D illustration of a bounding box and its representation is shown in Figure 2.6.

INFINITY 514
Point 21

The default `BBox` constructor sets the extent to be degenerate; by violating the invariant that `pMin.x <= pMax.x`, etc., we ensure than any operations done with this box will have the correct result for a completely empty box.

⟨*BBox Constructors*⟩≡
```
BBox() {
    pMin = Point( INFINITY,  INFINITY,  INFINITY);
    pMax = Point(-INFINITY, -INFINITY, -INFINITY);
}
```

⟨*BBox Public Data*⟩≡
```
Point pMin, pMax;
```

It is also useful to be able to initialize a `BBox` to enclose a single point.

⟨*BBox Constructors*⟩+≡
```
BBox(const Point &p) : pMin(p), pMax(p) { }
```

If the user passes two corner points, `p1` and `p2` to define the box, since `p1` and `p2` are not necessarily ordered so that [[p1.x ¡= p2.x]] etc, we need to find their minimum and maximum component-wise values.

⟨*BBox Constructors*⟩+≡
```
BBox(const Point &p1, const Point &p2) {
    pMin = Point(min(p1.x, p2.x),
                 min(p1.y, p2.y),
                 min(p1.z, p2.z));
    pMax = Point(max(p1.x, p2.x),
                 max(p1.y, p2.y),
                 max(p1.z, p2.z));
}
```

Given a bounding box and a point, we can compute a new bounding box that encompasses that point as well as the space that the original box encompassed.

⟨*BBox Method Definitions*⟩≡
```
BBox Union(const BBox &b, const Point &p) {
    BBox ret = b;
    ret.pMin.x = min(b.pMin.x, p.x);
    ret.pMin.y = min(b.pMin.y, p.y);
    ret.pMin.z = min(b.pMin.z, p.z);
    ret.pMax.x = max(b.pMax.x, p.x);
    ret.pMax.y = max(b.pMax.y, p.y);
    ret.pMax.z = max(b.pMax.z, p.z);
    return ret;
}
```

```
513 max
513 min
 28 pMax
 28 pMin
 21 Point
```

And similarly, we can construct a new bounding box that also encompasses the space encompassed by another bounding box. The definition of this function is similar to the `Union` method above that takes a `Point`; the difference is the `pMin` and `pMax` of the other box are used for the `min()` and `max()` tests, respectively.

⟨*BBox Method Declarations*⟩+≡
```
friend BBox Union(const BBox &b, const BBox &b2);
```

We can also take two bounding boxes and compute their intersection: the bounding box that encloses the parts of them that overlap.

⟨*BBox Method Definitions*⟩+≡
```
BBox Intersection(const BBox &b1, const BBox &b2) {
    BBox ret;
    ret.pMin.x = max(b1.pMin.x, b2.pMin.x);
    ret.pMin.y = max(b1.pMin.y, b2.pMin.y);
    ret.pMin.z = max(b1.pMin.z, b2.pMin.z);
    ret.pMax.x = min(b1.pMax.x, b2.pMax.x);
    ret.pMax.y = min(b1.pMax.y, b2.pMax.y);
    ret.pMax.z = min(b1.pMax.z, b2.pMax.z);
    return ret;
}
```

We can also easily determine if two `BBox`es overlap seeing if their extents overlap in *x*, *y*, and *z*.

⟨*BBox Method Declarations*⟩+≡
```
bool Overlaps(const BBox &b) const {
    bool x = (pMax.x >= b.pMin.x) && (pMin.x <= b.pMax.x);
    bool y = (pMax.y >= b.pMin.y) && (pMin.y <= b.pMax.y);
    bool z = (pMax.z >= b.pMin.z) && (pMin.z <= b.pMax.z);
    return (x && y && z);
}
```

We have a quick test that tells us if a given point is inside the bounding box.

⟨*BBox Method Declarations*⟩+≡
```
bool Inside(const Point &pt) const {
    return (pt.x >= pMin.x && pt.x <= pMax.x &&
            pt.y >= pMin.y && pt.y <= pMax.y &&
            pt.z >= pMin.z && pt.z <= pMax.z);
}
```

And finally, the `Expand` method pads out the bounding box by a user-supplied constant factor.

⟨*BBox Method Declarations*⟩+≡
```
void Expand(Float delta) {
    pMin -= Vector(delta, delta, delta);
    pMax += Vector(delta, delta, delta);
}
```

pMax 28
pMin 28
Point 21
Vector 16

⟨*BBox Method Declarations*⟩+≡
```
Float Volume() const {
    Vector d = pMax - pMin;
    return d.x * d.y * d.z;
}
```

2.6 Transformations

⟨*transform.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef TRANSFORM_H
#define TRANSFORM_H
#include "lrt.h"
#include "geometry.h"
⟨Transform Declarations⟩
⟨Transform Inline Functions⟩
#endif // TRANSFORM_H
```

⟨*transform.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "transform.h"
#include "shapes.h"
⟨Transform Methods⟩
```

In general, a *transformation* $\mathbf{T}$ can be described as a mapping from points to points and from vectors to vectors:

$$\mathbf{p}' = \mathbf{T}(\mathbf{p}) \qquad \vec{\mathbf{v}}' = \mathbf{T}(\vec{\mathbf{v}})$$

The transformation $\mathbf{T}$ may be an arbitrary procedure. However, we will consider a subset of all of the possible transformations in this chapter. In particular, they will be:

- Linear: If $\mathbf{T}$ is an arbitrary linear transformation and s is an arbitrary scalar, then $\mathbf{T}(s\vec{\mathbf{v}}) = s\mathbf{T}(\vec{\mathbf{v}})$ and $\mathbf{T}(\vec{\mathbf{v}}_1 + \vec{\mathbf{v}}_2) = \mathbf{T}(\vec{\mathbf{v}}_1) + \mathbf{T}(\vec{\mathbf{v}}_2)$. These two properties can greatly simplify reasoning about transformations.
- Continuous: roughly speaking, $\mathbf{T}$ leaves the neighborhoods around $\mathbf{p}$ and $\vec{\mathbf{v}}$ around $\mathbf{p}'$ and $\vec{\mathbf{v}}'$.
- One-to-one and invertible: for each $\mathbf{p}$, $\mathbf{T}$ maps $\mathbf{p}$ to a single $\mathbf{p}'$. Furthermore, for each $\mathbf{p}'$, we can find an inverse transform such that $\mathbf{T}^{-1}(\mathbf{p}') = \mathbf{p}$.

We will often want to take a point, vector, or normal defined with respect to one coordinate frame and find its coordinate values with respect to another frame. Using basic properties of linear algebra, it can be shown that in three dimensions, a 4*x*4 matrix can express the linear transformation of a point or vector from one frame to another. Furthermore, such a 4*x*4 matrix suffices to express all linear transformations of points and vectors within a fixed frame, such as translation in space or rotation around a point. As such, there are two different (and incompatible!) ways that a matrix can be interpreted:

1. *Transformation of the frame*: given a point, the matrix could express how to compute a *new* point in the same frame that represents the transformation of the original point (e.g. by translating it in some direction.)
2. *Transformation from one frame to another*: a matrix can express how a new point in a new frame is computed given a point in an original frame.

In general, transformations like these make it possible to work in the most convenient coordinate space. For example, we can write routines that define a virtual camera that looks at a scene to be rendered assuming that the camera is located at the origin, is looking down the *z* axis, and where the *y* axis points in the up direction. These assumptions may greatly simplify the camera implementation. However, so that we can place the camera at any point in the scene looking in any direction, we can construct a transformation that maps points in the scene's coordinate space to the camera's coordinate space.

⟨*Transform Declarations*⟩≡
```
class Transform {
public:
    ⟨Transform Constructor Declarations⟩
    ⟨Transform Method Declarations⟩
private:
    ⟨Transform Private Data⟩
};
```

A transformation is represented by the elements of the matrix `m[4][4]`, represented by a reference to a `Matrix4x4` object. The low-level `Matrix4x4` class is defined in Appendix A.5. `m` is stored in *row-order* form; to reference the matrix element $m_{i,j}$, where i and j range from zero to three, and where i is the row number and j is the column number, we access element `m[i][j]`. For convenience, we also store the inverse of the matrix `m` in the `m_inv` member; it will be handy to have the inverse easily available for a number of situations.

⟨*Transform Private Data*⟩≡
```
Reference<Matrix4x4> m, m_inv;
```

Basic operations

When a new `Transform` is created, it will default to the *identity transformation*: the transformation that maps each point and each vector to itself. This is represented by the *identity matrix*:

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Inverse 512
Matrix4x4 510
Reference 509

⟨*Transform Constructor Declarations*⟩≡
```
Transform() {
    m = m_inv = new Matrix4x4;
}
```

⟨*Transform Constructor Declarations*⟩+≡
```
Transform(Float mat[4][4]) {
    m = new Matrix4x4(mat[0][0], mat[0][1], mat[0][2], mat[0][3],
        mat[1][0], mat[1][1], mat[1][2], mat[1][3],
        mat[2][0], mat[2][1], mat[2][2], mat[2][3],
        mat[3][0], mat[3][1], mat[3][2], mat[3][3]);
    m_inv = m->Inverse();
}
```

⟨*Transform Constructor Declarations*⟩+≡
```
Transform(const Reference<Matrix4x4> &mat) {
    m = mat;
    m_inv = m->Inverse();
}
```

⟨*Transform Constructor Declarations*⟩+≡
```
Transform(const Reference<Matrix4x4> &mat,
        const Reference<Matrix4x4> &minv) {
    m = mat;
    m_inv = minv;
}
```

Homogeneous coordinates

Given a frame defined by $(\mathbf{p}, \vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2, \vec{\mathbf{v}}_3)$, there is ambiguity between the representation of a point (p_x, p_y, p_z) and a vector $(\mathbf{v}_x, \mathbf{v}_y, \mathbf{v}_z)$ with equivalent coordinates. However, taking the definition of the representations of points and vectors, we can write the point as $[s_1\, s_2\, s_3\, 1][\vec{\mathbf{v}}_1\, \vec{\mathbf{v}}_2\, \vec{\mathbf{v}}_3\, \mathbf{p}]^T$ and the vector as $[s'_1\, s'_2\, s'_3\, 0][\vec{\mathbf{v}}_1\, \vec{\mathbf{v}}_2\, \vec{\mathbf{v}}_3\, \mathbf{p}]^T$ These four-vectors of three s_i values and a zero or one are *homogeneous* representations of the point and the vector. The fourth coordinate of the homogeneous representation is sometimes called the weight. For a point, its value can be any scalar other than zero: the homogeneous points $[1, 3, -2, 1]$ and $[-2, -6, 4, -2]$ describe the same Cartesian point$(1, 3, -2)$.

Given

$$\mathbf{M} = \begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{22} & m_{33} \end{pmatrix}$$

Then

$$\mathbf{M}[1,0,0,0]^T = [m_{00}, m_{10}, m_{20}, m_{30}]^T.$$

So directly reading the columns of the matrix shows how the basis vectors

$$\begin{aligned} \vec{\mathbf{x}} &= [1,0,0,0]^T \\ \vec{\mathbf{y}} &= [0,1,0,0]^T \\ \vec{\mathbf{z}} &= [0,0,1,0]^T \\ \mathbf{p} &= [0,0,0,1]^T \end{aligned}$$

Are transformed by the matrix. And by characterizing how the basis is transformed, the transformation thus characterizes how any point or vector specified in terms of that basis is transformed.

So, for example, if we know how the basis vectors are changed by a linear transform, we can determine what that transformation is from the coordinates of the transformed basis vectors.

Specifically, the coordinates of the basis vectors in the matrix must be defined with respect to some particular frame. Then, the matrix describes how stuff in that frame is transformed...

We will not use homogeneous coordinates explicitly in our code; there is no `Homogeneous` class. However, the various transformation routines in the next section will implicitly convert points, vectors, and normals to homogeneous form, transform the homogeneous points, and then convert them back before returning the result. We will explain this further as it happens.

Translations

One of the simplest transformations is the translation $\mathbf{T}(\Delta x, \Delta y, \Delta z)$. When applied to a point $\mathbf{p}$, it translates $\mathbf{p}$'s coordinates by Δx, Δy, and Δz, as shown in Figure 2.7.

The translation has some simple properties:

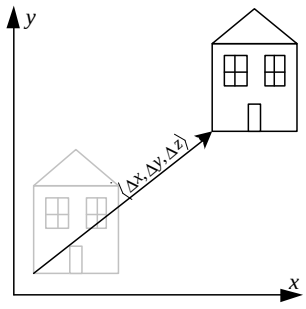


Figure 2.7: Translation in 2D.

$$\begin{aligned}
\mathbf{T}(0,0,0) &= \mathbf{I} \\
\mathbf{T}(x_1,y_1,z_1) \times \mathbf{T}(x_2,y_2,z_2) &= \mathbf{T}(x_1+x_2, y_1+y_2, z_1+z_2) \\
\mathbf{T}(x_1,y_1,z_1) \times \mathbf{T}(x_2,y_2,z_2) &= \mathbf{T}(x_2,y_2,z_2) \times \mathbf{T}(x_1,y_1,z_1) \\
\mathbf{T}^{-1}(x,y,z) &= \mathbf{T}(-x,-y,-z)
\end{aligned}$$

Translation should only affect points, leaving vectors unchaged.

In matrix form, the translation transformation is:

$$\mathbf{T}(\Delta x, \Delta y, \Delta z) = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

When we consider the operation of a translation matrix on a point, we see the value of homogeneous coordinates. Consider the product of the matrix for $\mathbf{T}(\Delta x, \Delta y, \Delta z)$ with a point $\mathbf{p}$ in homogeneous coordinates $[x\,y\,z\,1]$:

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+\Delta x \\ y+\Delta y \\ z+\Delta z \\ 1 \end{pmatrix}$$

As expected, we have computed a new point with its coordinates offset by $(\Delta x, \Delta y, \Delta z)$. However, if we apply $\mathbf{T}$ to a vector $\vec{\mathbf{v}}$, we have:

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}$$

The result is the same vector $\vec{\mathbf{v}}$. This makes sense, since translations shouldn't have any effect on vectors; because vectors represent directions, a translation leaves them unchanged.

We will define a routine that creates a new `Transform` matrix that represents a given translation. We define this as a plain global function: rather than operating

on a `Transform` that already exists, this returns a new `Transform` with the given translation. Though the possible extra creation of temporary `Transform`s could have a negative performance impact if called frequently, this doesn't have a significant impact on `lrt` since new `Transform`s aren't computed during the main rendering loop after the scene has been specified.

⟨*Transform Methods*⟩+≡
```
Transform Translate(const Vector &delta) {
    Matrix4x4 *m, *minv;
    m = new Matrix4x4(1, 0, 0, delta.x,
                      0, 1, 0, delta.y,
                      0, 0, 1, delta.z,
                      0, 0, 0,       1);
    minv = new Matrix4x4(1, 0, 0, -delta.x,
                         0, 1, 0, -delta.y,
                         0, 0, 1, -delta.z,
                         0, 0, 0,       1);
    return Transform(m, minv);
}
```

Scaling

510 Matrix4x4
32 Transform
16 Vector

Another basic transformation is the *scale transform*. This has the effect of taking a point or vector and multiplying its components by scale factors in x, y, and z: $\mathbf{S}(2,2,1)(x,y,z) = (2x,2y,z)$. It has the following basic properties:

$$\begin{aligned}
\mathbf{S}(1,1,1) &= \mathbf{I} \\
\mathbf{S}(x_1,y_1,z_1) \times \mathbf{S}(x_2,y_2,z_2) &= \mathbf{S}(x_1x_2,y_1y_2,z_1,z_2) \\
\mathbf{S}^{-1}(x,y,z) &= \mathbf{S}\left(\frac{1}{x},\frac{1}{y},\frac{1}{z}\right)
\end{aligned}$$

We can differentiate between *uniform scaling*, where all three scale factors have the same value and *non-uniform scaling*, where they may have different values. The general scale matrix is

$$\mathbf{S}(x,y,z) = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

⟨*Transform Methods*⟩+≡
```
Transform Scale(Float x, Float y, Float z) {
    Matrix4x4 *m, *minv;
    m = new Matrix4x4(x, 0, 0, 0,
                      0, y, 0, 0,
                      0, 0, z, 0,
                      0, 0, 0, 1);
    minv = new Matrix4x4(1.f/x, 0, 0, 0,
                         0, 1.f/y, 0, 0,
                         0, 0, 1.f/z, 0,
                         0, 0, 0, 1);
    return Transform(m, minv);
}
```

X, Y, and Z axis rotations

Another useful type of transformation is the rotation. In general, we can define an arbitrary axis from the origin in any direction and can then rotate around that axis by a given angle. The most common rotations of this type are around the *x*, *y*, and *z* coordinate axes. We will write these rotations as $\mathbf{R}_x(\theta)$, etc.. The rotation around an arbitrary axis (x,y,z) is denoted by $\mathbf{R}_{(x,y,z)}(\theta)$.

Matrix4x4 510
Transform 32

Rotations also have some basic properties:

$$\begin{aligned}
\mathbf{R}_a(0) &= \mathbf{I} \\
\mathbf{R}_a(\theta_1) \times \mathbf{R}_a(\theta_2) &= \mathbf{R}_a(\theta_1 + \theta_2) \\
\mathbf{R}_a(\theta_1) \times \mathbf{R}_a(\theta_2) &= \mathbf{R}_a(\theta_2) \times \mathbf{R}_a(\theta_1) \\
\mathbf{R}_a^{-1}(\theta) &= \mathbf{R}_a(-\theta) = \mathbf{R}_a^T(\theta)
\end{aligned}$$

where $\mathbf{R}^T$ is the matrix transpose of $\mathbf{R}$. This property, that the inverse of $\mathbf{R}$ is equal to its transpose (a quantity that is much easier to compute than a full matrix inverse!), stems from the fact that we know that $\mathbf{R}$ is an *orthonormal matrix*; its upper 3*x*3 components are all normalized and orthogonal to each other.

The matrix for rotation around the *x* axis is

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 2.8 gives an intuition for how this matrix works. It's easy to see that

$$\mathbf{R}_x(\theta) \cdot [1,0,0,0]^T = [1,0,0,0];$$

it leaves the *x* axis unchanged. In maps the *y* axis $(0,1,0)$ to $(0,\cos\theta,\sin\theta)$ and the *z* axis to $(0,-\sin\theta,\cos\theta)$. More specifically, reading the columns of $\mathbf{R}_x(\theta)$, we can directly see what vectors the original coordinate axes are transformed to. The *y* and *z* axes remain in the same plane, perpendicular to the *x* axis, but are rotated around the circle by the given angle. An arbitrary point in space is similarly rotated about *x* while staying in the same *yz* plane as it was originally.

The implementation of the `RotateX` creation function is straightforward.

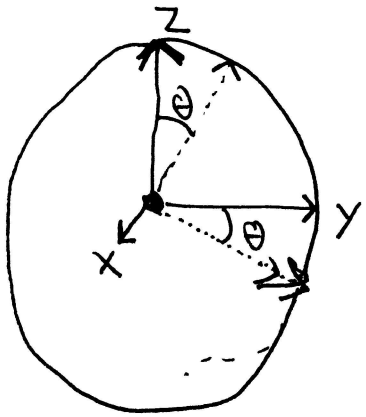


Figure 2.8: Rotation by an angle θ about the *x* axis leaves the *x* coordinate unchanged. The *y* and *z* axes are mapped to the vertices given by the dashed lines; *y* and *z* coordinates move accordingly.

⟨*Transform Methods*⟩+≡

```
Transform RotateX(Float angle) {
    Float sin_t = sinf(Radians(angle));
    Float cos_t = cosf(Radians(angle));
    Matrix4x4 *m = new Matrix4x4(1,   0,      0,   0,
                                 0, cos_t, -sin_t, 0,
                                 0, sin_t,  cos_t, 0,
                                 0,   0,      0,   1);
    return Transform(m, m->Transpose());
}
```

510 Matrix4x4
514 Radians
32 Transform
511 Transpose

Similarly, for rotation around *y* and *z*, we have

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The implementations of `RotateY` and `RotateZ` follow directly and will not be included here.

Rotation around an arbitrary axis

Finally, we provide rotation around an arbitrary axis. The usual derivation of this is based on computing rotations that map the given axis to a fixed axis (e.g. *z*), performing the rotation there, and then rotating the fixed axis back to the original axis. A more elegant derivation can be constructed with vector algebra.

Consider a normalized direction vector *a* that gives the axis to rotate around by angle θ and a vector *v* to be rotated (see Figure 2.9). First, we can compute the point *p* along the axis *a*: this point is in the plane perpendicular to *a* that also goes through the end-point of *v*.

$$p = \vec{\mathbf{a}} \cos\alpha = \vec{\mathbf{a}}(\vec{\mathbf{v}} \cdot \vec{\mathbf{a}}).$$

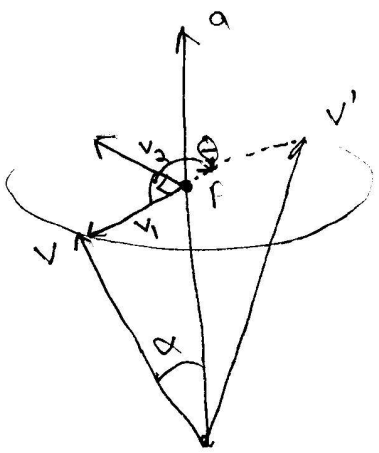


Figure 2.9: Rotation about an arbitrary axis *a*: ...

We now compute a pair of basis vectors $\vec{\mathbf{v}}_1$ and $\vec{\mathbf{v}}_2$ in this plane. Trivially, one of them is

$$\vec{\mathbf{v}}_1 = \vec{\mathbf{v}} - \vec{\mathbf{p}}$$

and the other can be computed with a cross product

$$\vec{\mathbf{v}}_2 = (\vec{\mathbf{v}}_1 \times \vec{\mathbf{a}}).$$

Because $\vec{\mathbf{a}}$ is normalized, $\vec{\mathbf{v}}_1$ and $\vec{\mathbf{v}}_2$ have the smae length, equal to the distance from $\vec{\mathbf{v}}$ to p. To now compute the rotation by θ degrees about the point p in the plane of rotation, the rotation formulas above give us

$$\vec{\mathbf{v}'} = p + \vec{\mathbf{v}}_1 \cos\theta + \vec{\mathbf{v}}_2 \sin\theta.$$

To convert this to a rotation matrix, we apply this formula to the basis vectors $\vec{\mathbf{v}} = (1,0,0)$, $\vec{\mathbf{v}} = (0,1,0)$, and $\vec{\mathbf{v}} = (0,0,1)$ to get the values of the rows of the matrix. The result of all this is encapsulated in the function below.

⟨*Transform Methods*⟩+≡
```
Transform Rotate(Float angle, const Vector &axis) {
    Vector a = axis.Hat();
    Float s = sinf(Radians(angle));
    Float c = cosf(Radians(angle));
    Float m[4][4];

    m[0][0] = a.x * a.x + (1.f - a.x * a.x) * c;
    m[0][1] = a.x * a.y * (1.f - c) - a.z * s;
    m[0][2] = a.x * a.z * (1.f - c) + a.y * s;
    m[0][3] = 0;

    m[1][0] = a.x * a.y * (1.f - c) + a.z * s;
    m[1][1] = a.y * a.y + (1.f - a.y * a.y) * c;
    m[1][2] = a.y * a.z * (1.f - c) - a.x * s;
    m[1][3] = 0;

    m[2][0] = a.x * a.z * (1.f - c) - a.y * s;
    m[2][1] = a.y * a.z * (1.f - c) + a.x * s;
    m[2][2] = a.z * a.z + (1.f - a.z * a.z) * c;
    m[2][3] = 0;

    m[3][0] = 0;
    m[3][1] = 0;
    m[3][2] = 0;
    m[3][3] = 1;

    Matrix4x4 *mat = new Matrix4x4(m);
    return Transform(mat, mat->Transpose());
}
```

19	Hat
510	Matrix4x4
514	Radians
32	Transform
511	Transpose
16	Vector

The look-at transformation

There is a transformation that is particularly useful for placing a camera in the scene; it is known as the *look-at transformation*. The user species the desired position of the camera, the point the camera is looking at, and an "up" vector that orients the camera along the viewing direction specified by the first two parameters. All of these values are given in world-space coordinates. The look-at transformation uses these values to initialize a transformation matrix that describes the transformation between camera space and world space. See Figure 2.10.

The derivation of the look-at transformation just requires application of the principles described earlier in this section where we described how the columns of a transformation matrix show what effect the transformation has on the basis of the coordinate system that it is acting upon.

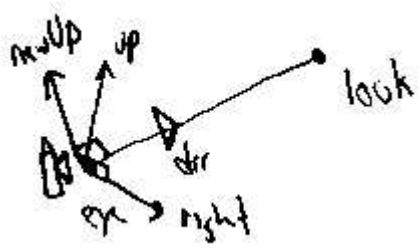


Figure 2.10: lookat setting

⟨*Transform Methods*⟩+≡
```
Transform LookAt(const Point &pos, const Point &look,
        const Vector &up) {
    Float m[4][4];
    ⟨Initialize fourth column of viewing matrix⟩
    ⟨Initialize first three columns of viewing matrix⟩
    Matrix4x4 *camToWorld = new Matrix4x4(m);
    return Transform(camToWorld->Inverse(), camToWorld);
}
```

Inverse 512
Matrix4x4 510
Point 21
Transform 32
Vector 16

The easiest column is the fourth one, which gives the point that the camera-space origin, $[0001]^T$, maps to in world space. This is clearly just the coordinates of the camera position, supplied by the user.

⟨*Initialize fourth column of viewing matrix*⟩≡
```
m[0][3] = pos.x;
m[1][3] = pos.y;
m[2][3] = pos.z;
m[3][3] = 1;
```

XXX note just need to generate an orthonormal coordinate system for these guys XXX

And the other three columns aren't much worse. First, we normalize the direction vector from the camera point to the look-at point; this gives us the vector coordinates that the z axis should map to and thus, the third column of the matrix. (Recall that camera space is defined with the viewing direction down the $+z$ axis.) The first column, giving the world space direction that the $+x$ axis in camera space maps to, is found by taking the cross product of the user-supplied "up"' vector with the viewing direction vector. Finally, the "up" vector is recomputed by taking the cross product of the viewing direction vector with the x axis vector, thus ensuring that we have an orthonormal viewing coordinate system. (Otherwise, if y and z axes weren't perpendicular, we wouldn't have an orthonormal coordinate system.)

⟨*Initialize first three columns of viewing matrix*⟩≡
```
Vector dir = (look - pos).Hat();
Vector right = Cross(dir, up.Hat());
Vector newUp = Cross(right, dir);
m[0][0] = right.x;
m[1][0] = right.y;
m[2][0] = right.z;
m[3][0] = 0.;
m[0][1] = newUp.x;
m[1][1] = newUp.y;
m[2][1] = newUp.z;
m[3][1] = 0.;
m[0][2] = dir.x;
m[1][2] = dir.y;
m[2][2] = dir.z;
m[3][2] = 0.;
```

Applying Transforms

We can now define routines that perform the appropriate matrix multiplications to transform points and vectors. We will overload the function application operator to describe these transformations; this lets us write code like:

```
Point Pold = ...;
Transform T = ...;
Point Pnew = T(Pold);
```

20 Cross
19 Hat
21 Point
32 Transform
16 Vector

Points

We compute the inner products of rows of the matrix with the column vector defined by the homogeneous point that we're transforming in order to compute the transformed result. For efficiency, we skip the divide by the resulting homogeneous weight `w` when its value is one; this is a common case for most of the transformations that we'll be using–only the projective transformations defined in Chapter 6 will require this divide.

⟨*Transform Inline Functions*⟩≡
```
inline Point Transform::operator()(const Point &pt) const {
    Float x = pt.x, y = pt.y, z = pt.z;

    Float xp = m->m[0][0]*x + m->m[0][1]*y + m->m[0][2]*z + m->m[0][3];
    Float yp = m->m[1][0]*x + m->m[1][1]*y + m->m[1][2]*z + m->m[1][3];
    Float zp = m->m[2][0]*x + m->m[2][1]*y + m->m[2][2]*z + m->m[2][3];
    Float wp = m->m[3][0]*x + m->m[3][1]*y + m->m[3][2]*z + m->m[3][3];

    if (wp == 1.) return Point(xp, yp, zp);
    else          return Point(xp / wp, yp / wp, zp / wp);
}
```

For efficiency, we also provide transformation methods that let the caller pass in a pointer to an object for the result; this saves passing structures by value on the

stack. Note that we copy the original (x,y,z) coordinates to local variables in case the result pointer points at the same point as `pt`.

⟨*Transform Inline Functions*⟩+≡
```
inline void Transform::operator()(const Point &pt,
        Point *ptrans) const {
    Float x = pt.x, y = pt.y, z = pt.z;

    ptrans->x = m->m[0][0]*x + m->m[0][1]*y + m->m[0][2]*z + m->m[0][3];
    ptrans->y = m->m[1][0]*x + m->m[1][1]*y + m->m[1][2]*z + m->m[1][3];
    ptrans->z = m->m[2][0]*x + m->m[2][1]*y + m->m[2][2]*z + m->m[2][3];
    Float w   = m->m[3][0]*x + m->m[3][1]*y + m->m[3][2]*z + m->m[3][3];

    if (w != 1.) {
        ptrans->x /= w;
        ptrans->y /= w;
        ptrans->z /= w;
    }
}
```

Point 21
Transform 32
Vector 16

Vectors

We compute the transformations of vectors in a similar fashion. However, the multiplication of the matrix and the row vector is simplified since the homogeneous w coordinate is zero.

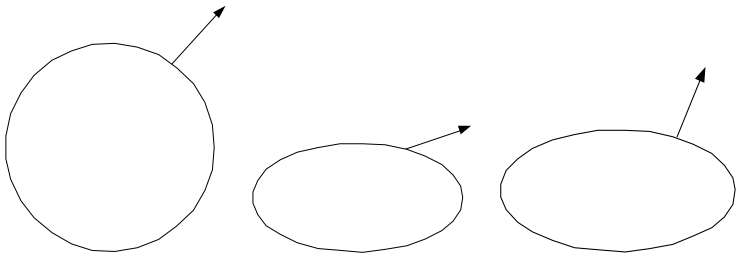
⟨*Transform Inline Functions*⟩+≡
```
inline Vector Transform::operator()(const Vector &v) const {
    Float x = v.x, y = v.y, z = v.z;
    return Vector(m->m[0][0]*x + m->m[0][1]*y + m->m[0][2]*z,
                  m->m[1][0]*x + m->m[1][1]*y + m->m[1][2]*z,
                  m->m[2][0]*x + m->m[2][1]*y + m->m[2][2]*z);
}
```

⟨*Transform Inline Functions*⟩+≡
```
inline void Transform::operator()(const Vector &v,
        Vector *vt) const {
    Float x = v.x, y = v.y, z = v.z;
    vt->x = m->m[0][0] * x + m->m[0][1] * y + m->m[0][2] * z;
    vt->y = m->m[1][0] * x + m->m[1][1] * y + m->m[1][2] * z;
    vt->z = m->m[2][0] * x + m->m[2][1] * y + m->m[2][2] * z;
}
```

Normals

Normals do not transform in the same way that vectors do, as shown in Figure 2.11. Although the tangent vectors to the surface that define the normal transform as expected, normals require special treatment. Because the normal vector $\vec{\mathbf{N}}$ and any tangent vector $\vec{\mathbf{T}}$ are orthogonal by construction, we know that

$$\vec{\mathbf{N}} \cdot \vec{\mathbf{T}} = \vec{\mathbf{N}}^T \vec{\mathbf{T}} = 0.$$



(a) Original object

(b) Scaled object with incorrect normal

(c) Scaled object with correct normal

Figure 2.11: Transforming surface normals. The circle in (a) is scaled by 50% in the *y* direction. Note that simply treating the normal as a direction and scaling it in the same manner, as shown in (b), will lead to incorrect results.

When we transform a point on the surface by some matrix M, the new tangent vector $\vec{\mathbf{T}}'$ at the transformed point is simply $M\vec{\mathbf{T}}$. The transformed normal $\vec{\mathbf{N}}'$ should be equal to $S\vec{\mathbf{N}}$ for some 4×4 matrix S. To maintain the orthogonality requirement, we must have:

32 Transform

$$\begin{aligned} 0 &= \vec{\mathbf{N}}'^{T}\vec{\mathbf{T}}' \\ &= S\vec{\mathbf{N}}^{T}M\vec{\mathbf{T}} \\ &= \vec{\mathbf{N}}^{T}S^{T}M\vec{\mathbf{T}} \end{aligned}$$

This condition holds if $S^TM = I$, the identity matrix. Therefore, $S^T = M^{-1}$, so $S = M^{-1^T}$, and we see that normals must be transformed by the inverse transpose of the transformation matrix. This is the main reason why `Transform`s maintain their inverses.

⟨*Transform Method Declarations*⟩+≡

```
Transform GetInverse() const {
    return Transform(m_inv, m);
}
```

Note that we do not explicitly compute the transpose of the inverse when transforming normals ; we simply iterate through the inverse matrix in a different order (compare to the code for transforming `Vector`s).

⟨*Transform Inline Functions*⟩+≡

```
inline Normal Transform::operator()(const Normal &n) const {
    Float x = n.x, y = n.y, z = n.z;
    return Normal(m_inv->m[0][0] * x + m_inv->m[1][0] * y +
                      m_inv->m[2][0] * z,
                  m_inv->m[0][1] * x + m_inv->m[1][1] * y +
                      m_inv->m[2][1] * z,
                  m_inv->m[0][2] * x + m_inv->m[1][2] * y +
                      m_inv->m[2][2] * z);
}
```

⟨*Transform Inline Functions*⟩+≡
```
inline void Transform::operator()(const Normal &n,
        Normal *nt) const {
    Float x = n.x, y = n.y, z = n.z;
    nt->x = m_inv->m[0][0] * x + m_inv->m[1][0] * y +
        m_inv->m[2][0] * z;
    nt->y = m_inv->m[0][1] * x + m_inv->m[1][1] * y +
        m_inv->m[2][1] * z;
    nt->z = m_inv->m[0][2] * x + m_inv->m[1][2] * y +
        m_inv->m[2][2] * z;
}
```

Rays

Transforming rays is straightforward: we just transform the constituent origin and direction.

⟨*Transform Inline Functions*⟩+≡
```
inline Ray Transform::operator()(const Ray &r) const {
    Ray ret;
    (*this)(r.O, &ret.O);
    (*this)(r.D, &ret.D);
    ret.mint = r.mint;
    ret.maxt = r.maxt;
    return ret;
}
```

Normal 23
Ray 26
Transform 32

⟨*Transform Inline Functions*⟩+≡
```
inline void Transform::operator()(const Ray &r, Ray *rt) const {
    rt->mint = r.mint;
    rt->maxt = r.maxt;
    (*this)(r.O, &rt->O);
    (*this)(r.D, &rt->D);
}
```

Bounding Boxes

The easiest way to transform an axis-aligned bounding box is to transform all eight of the vertices at its corners and then compute a new bounding box that encompasses those points. We will present code for this method below; one of the exercises for this chapter is to find a way to do this more efficiently.

⟨*Transform Methods*⟩+≡
```
BBox Transform::operator()(const BBox &b) const {
    const Transform &M = *this;
    BBox ret(      M(Point(b.pMin.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMin.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMin.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret, M(Point(b.pMin.x, b.pMax.y, b.pMax.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMax.y, b.pMax.z)));
    return ret;
}
```

Composition of Transformations

Having defined how the matrices representing individual types of transformations are constructed, we can now consider the transformation resulting from a series of individual transformations. It is in this setting that we can see the real value of representing transformations with 4*x*4 matrices.

Consider a series of transformations **ABC**. We'd like to compute a new transformation **T** such applying **T** gives the same result as applying each of **A**, **B**, and **C** in order; i.e. $\mathbf{A}(\mathbf{B}(\mathbf{C}(\mathbf{p}))) = \mathbf{T}(\mathbf{p})$. Such a transformation **T** can be computed by multiplying the matrices of the transformations **A**, **B**, and **C** together. In code, we can write:

```
Transform T = A * B * C;
```

510 Matrix4x4
512 Mul
28 pMax
28 pMin
21 Point
509 Reference
32 Transform
29 Union

Then we can apply `T` to `Point`s `p` as usual `Point pp = T(p)` instead of applying each transformation in turn: `Point pp = A(B(C(p)));`.

We use the C++ `*` operator to compute the new transformation that results from post-multiplying the current transformation with a new transformation `t2`. From the definition of matrix multiplication, the (i, j)th element of the resulting matrix `ret` is the product of the *i*th row of the first matrix with the *j*th column of the second.

The inverse of the resulting transformation, is equal to the product of `t2.m_inv * m_inv`; this is a result of the matrix identity

$$(AB)^{-1} = B^{-1}A^{-1}.$$

⟨*Transform Methods*⟩+≡
```
Transform Transform::operator*(const Transform &t2) const {
    Reference<Matrix4x4> m1 = Matrix4x4::Mul(m, t2.m);
    Reference<Matrix4x4> m2 = Matrix4x4::Mul(t2.m_inv, m_inv);
    return Transform(m1, m2);
}
```

Example:Rotation Around an Arbitrary Point

The rotations described so far all center the rotation around the origin of the active coordinate space. We can use the composition of three transformations in

order to rotate around an arbitrary axis that does not pass through the origin. Given an arbitrary axis of rotation defined by a point (x,y,z) and an axis α, and an angle θ to rotate by, the transformation can be constructed in three steps:

1. The coordinate frame is translated by $(-x,-y,-z)$ so that the axis passes through the origin.
2. The rotation is performed.
3. The coordinate frame is translated back by (x,y,z) so that the origin returns to its original location.

.

Thus we have

$$\mathbf{R}(x,y,z,\theta) = \mathbf{T}(x,y,z) * \mathbf{R}(\theta,\alpha) * \mathbf{T}(-x,-y,-z)$$

We will not include code for this operation as it won't be necessary for implementing `lrt`. However, this kind of coordinate system change is an important and powerful way of solving problems in computer graphics.

2.7 Differential Geometry

We will wrap up this chapter by developing a self-contained representation that holds the geometric information about a particular point on a surface (e.g. the point of a ray intersection). In particular, this abstraction needs to hide the particular type of geometric shape the point lies on, allowing the shading and geometric operations in the rest of the renderer to be implemented generically, not considering different shape types (e.g. spheres vs triangles, etc.).

The information that we will store to do this includes:

- The 3D hit point `P`
- A local coordinate system at the hit point, given by the surface normal `N` and two tangent vectors `S` and `T`
- The parametric partial derivatives $\partial P/\partial u$ and $\partial P/\partial v$.
- The partial derivatives of the change in surface normal $\partial N/\partial u$ and $\partial N/\partial v$.
- `u`, `v` coordinates from the parameterization of the surface.
- A pointer to the `Shape` that the differential geometry lies on; the shape class will be introduced in the next chapter. See Figure 2.12 for an depiction of these values.

This representation assumes that shapes have a parametric description–i.e. that for some range of (u,v) values, points on the surface are given by some function f such that $P = f(u,v)$. Though this isn't true for all of the shape representations that are used in graphics, all of the shapes that `lrt` supports do have at least a local a parametric description, so we will stick with the parametric representation since

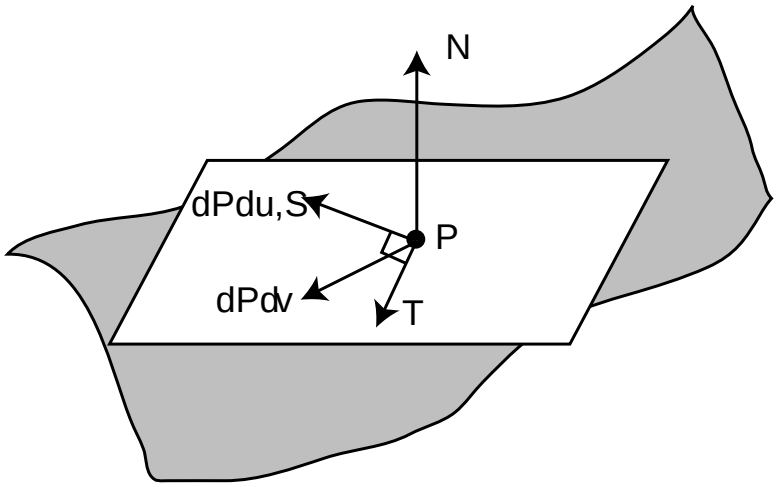


Figure 2.12: The local differential geometry around a point P. The tangent vectors S and T are orthogonal vectors in the plane that is tangent to the surface at P. The parametric partial derivatives of the surface, $\partial P/\partial u$ and $\partial P/\partial v$, also lie in the tangent plane but are not necessarily orthogonal. The surface normal N, is given by the cross product of $\partial P/\partial u$ and $\partial P/\partial v$. The vectors $\partial N/\partial u$ and $\partial N/\partial v$ (not shown here) record the differential change in surface normal as we move in u and v along the surface.

23 Normal
21 Point
16 Vector

this assumption will be helpful to us elsewhere (e.g. for anti-aliasing of textures in Chapter 11.)

⟨*DifferentialGeometry Declarations*⟩≡
```
struct DifferentialGeometry {
    DifferentialGeometry() { u = v = 0.; shape = NULL; }
    ⟨DifferentialGeometry Method Declarations⟩
    ⟨DifferentialGeometry Data⟩
};
```

⟨*DifferentialGeometry Data*⟩≡
```
Point P;
Normal Nb, Nn;
Vector S, T;
Float u, v;
const Shape *shape;
```

⟨*DifferentialGeometry Data*⟩+≡
```
Vector dPdu, dPdv;
Vector dNdu, dNdv;
```

The `DifferentialGeometry` constructor only needs a few parameters–the point of interest, the partial derivatives, and the (u,v) coordinates. It computes the normal as the cross product of the partial derivatives and initializes `S` to be the normalized $\partial P/\partial u$ vector. It then computes `T` by crossing `S` with `N`, which gives us a vector that is orthonormal to both of them and thus lies in the tangent plane.

⟨*DifferentialGeometry Method Declarations*⟩≡
```
DifferentialGeometry(const Point &p, const Vector &dpdu,
        const Vector &dpdv, const Vector &dndu,
        const Vector &dndv, Float uu, Float vv,
        const Shape *sh)
    : P(p), dPdu(dpdu), dPdv(dpdv), dNdu(dndu), dNdv(dndv) {
    ⟨Initialize DifferentialGeometry from parameters⟩
}
```

⟨*Initialize* `DifferentialGeometry` *from parameters*⟩≡
```
Nb = Normal(Cross(dPdu, dPdv));
Nn = Nb.Hat();
S = dPdu.Hat();
T = Cross(S, Nn);
u = uu;
v = vv;
shape = sh;
```

It is useful to be able to transform direction vectors from world space to the coordinate frame defined by the three basis directions $\vec{\mathbf{S}}$, $\vec{\mathbf{T}}$, and $\vec{\mathbf{N}}$. This maps the object's surface normal to the direction $(0,0,1)$, for example, and can help to simplify computations by letting us think of them in a standard coordinate system. It is easy to show that given three such orthogonal vectors $\vec{\mathbf{S}}$, $\vec{\mathbf{T}}$, and $\vec{\mathbf{N}}$ in world-space, the matrix M that transforms vectors in world space to the local differential geometry space is:

Cross 20
DifferentialGeometry 47
Hat 19
Normal 23
Point 21
Vector 16

$$M = \begin{pmatrix} \mathbf{S}_x & \mathbf{S}_y & \mathbf{S}_z \\ \mathbf{T}_x & \mathbf{T}_y & \mathbf{T}_z \\ \mathbf{N}_x & \mathbf{N}_y & \mathbf{N}_z \end{pmatrix} = \begin{pmatrix} \vec{\mathbf{S}} \\ \vec{\mathbf{T}} \\ \vec{\mathbf{N}} \end{pmatrix}$$

To confirm this yourself, consider the value of $M\vec{\mathbf{N}} = (\vec{\mathbf{S}}\cdot\vec{\mathbf{N}}, \vec{\mathbf{T}}\cdot\vec{\mathbf{N}}, \vec{\mathbf{N}}\cdot\vec{\mathbf{N}})$. Since $\vec{\mathbf{S}}$, $\vec{\mathbf{T}}$, and $\vec{\mathbf{N}}$ are all orthonormal, the x and y two components of $M\vec{\mathbf{N}}$ are zero. Since $\vec{\mathbf{N}}$ is normalized, $\vec{\mathbf{N}}\cdot\vec{\mathbf{N}} = 1$. Thus, $M\vec{\mathbf{N}} = (0,0,1)$. (In this case, we don't need to compute the inverse transpose of M to transform normals (recall the discussion of transforming normals in Section 2.6 on page 42.) Because M is an orthonormal matrix (its rows and columns are mutually orthogonal and are normalized), its inverse is equal to its transpose, so it is its own inverse transpose already.)

⟨*DifferentialGeometry Method Declarations*⟩+≡
```
Vector WorldToLocal(const Vector &v) const {
    return Vector(Dot(v, S), Dot(v, T), Dot(v, Nn)).Hat();
}
```

The function that takes vectors back from local space to world space just implements the transpose to invert M and does the appropriate dot products:

⟨*DifferentialGeometry Method Declarations*⟩+≡
```
Vector LocalToWorld(const Vector &v) const {
    return Vector(S.x * v.x + T.x * v.y + Nn.x * v.z,
                  S.y * v.x + T.y * v.y + Nn.y * v.z,
                  S.z * v.x + T.z * v.y + Nn.z * v.z);
}
```

Further Reading

DeRose and Goldman and others have pushed the coordinate-free geometry approach.

Geometric tools for Computer Graphics full of useful geometry for graphics, including excellent development of the coordiante-free approach (SE03).

Lots of stuff is useful. For example, *Mathematical Elements for Computer Graphics* by Rogers and Adams(RA90) is a winner. Note that they use a row-vector representation of points and vectors, though, which means that everything is backwards.

Linear algebra books: Lang(Lan86).

Homogeneous stuff: Stolfi(Sto91).

Advanced calculus (vector stuff), Buck(Buc78).

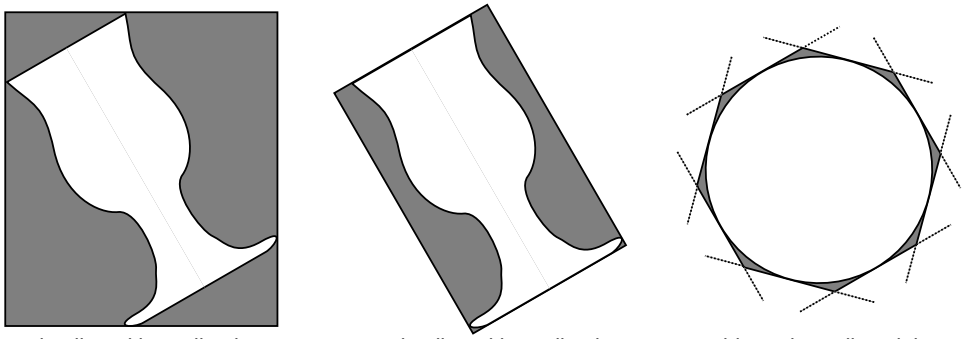
Möller and Haines for graphics-based introduction to linear algebra(MH02), lots of ray bounds stuff and ray–obb stuff.

obb stuff

Exercises

2.1 (Jim Arvo) Find a more efficient way to transform axis-aligned bounding boxes by taking advantage of the symmetries of the problem: because the eight corner points are linear combinations of three axis-aligned basis vectors and a single corner point, their transformed bounding box can be found much more efficiently than by the method we presented.

2.2 Instead of boxes, we could compute tighter bounds by using the intersections of many non-orthogonal slabs. Extend our bounding box class to allow the user to specify a bound comprised of arbitrary slabs.



Axis-aligned bounding box | Non-axis-aligned bounding box | Arbitrary bounding slabs

3. Shapes

⟨*shapes.h**⟩≡
  ⟨*Source Code Copyright*⟩
  `#ifndef SHAPES_H`
  `#define SHAPES_H`
  `#include "lrt.h"`
  `#include "geometry.h"`
  `#include "transform.h"`
  `#include "paramset.h"`
  ⟨*DifferentialGeometry Declarations*⟩
  ⟨*Shape Declarations*⟩
  `#endif // SHAPES_H`

⟨*shapes.cc**⟩≡
  ⟨*Source Code Copyright*⟩
  `#include "shapes.h"`
  `#include "../shapes/trianglemesh.h"`
  ⟨*Shape Method Definitions*⟩
  ⟨*DifferentialGeometry Method Definitions*⟩

Shapes in `lrt` are the basic representations of geometry in a scene. Each specific shape in `lrt` is a subclass of the `Shape` base class. Thus, we can describe a general interface to shapes that hides information about the actual type of shape that we have (triangle, sphere, etc). This abstraction strategy makes extending the geometric capabilities of the system quite straightforward; the rest of `lrt` doesn't need to make any distinctions based on what specific shape it may be using. The `Shape` class is purely geometric; it contains no information about the appearance of an object. The `Primitive` class, introduced in Chapter 1, holds additional information

about a shape such as its material properties.

3.1 Basic Shape Interface

⟨*Shape Declarations*⟩≡
```
class Shape : public ReferenceCounted<Shape> {
public:
    ⟨Shape Interface⟩
    virtual ~Shape() { }
protected:
    ⟨Shape Protected Data⟩
};
```

All shapes are defined in object coordinate space; for example, all spheres are defined in an object space where the center of the sphere is at the origin. In order to place a sphere at another position in the scene, a transformation that describes the mapping from object space to world space can be provided. The shape stores both this transformation and its inverse.

⟨*Shape Method Definitions*⟩≡
```
Shape::Shape(const Transform &o2w)
    : ObjectToWorld(o2w) {
    WorldToObject = ObjectToWorld.GetInverse();
    ⟨Update shape creation statistics⟩
}
```

GetInverse 43
ReferenceCounted 508
StatsCounter 501
Transform 32

⟨*Shape Protected Data*⟩≡
```
Transform ObjectToWorld, WorldToObject;
```

⟨*Update shape creation statistics*⟩≡
```
static StatsCounter nShapesMade("Geometry",
    "Total shapes created");
++nShapesMade;
```

Bounding

Each `Shape` subclass must be capable of bounding itself with a bounding box. There are two different bounding methods. The first, `ObjectBound()`, returns a bounding box in the shape's object space, and the second, `WorldBound()`, returns a bounding box in world space. The implementation of the first method is left up to each individual shape, thought there is a default implementation of the second method that transforms the object bound to world space and computes the bound of the result. Shapes that can easily compute a world-space bound that is tighter than the one computed by transforming the object-space bound to world space should override this method, however–an example of such a shape is a triangle; see Figure 3.1.

⟨*Shape Interface*⟩+≡
```
virtual BBox ObjectBound() const = 0;
```

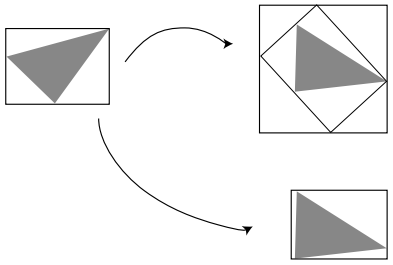


Figure 3.1: If we compute a world-space bounding box of a triangle by transforming its object-space bounding box to world space and then finding the bounding box that encloses the resulting bounding box, a sloppy bound may result (top). However, if we first transform its vertices from object space to world space and then bound those vertices (bottom), we can do much better.

⟨*Shape Interface*⟩+≡
```
virtual BBox WorldBound() const {
    return ObjectToWorld(ObjectBound());
}
```

Refinement

54 RefinedShape
498 Severe

Not every shape needs to be capable of determining whether a ray intersects it. For example, a complex curved surface might need to tessellate into triangles, which would then be intersected. Or, we might have a special shape that is a placeholder for a large amount of geometry that is stored on disk. We could store just the filename of the geometry file and the bounding box of the geometry inside of it in memory, only reading the geometry in from disk if a ray pierced the bounding box. We can't intersect a ray with such a shape directly, so its `CanIntersect` routine would return a false boolean value.

The default implementation of this function indicates that a shape can provide an intersection, so only shapes that are non-intersectable need to override this method.

⟨*Shape Interface*⟩+≡
```
virtual bool CanIntersect() const { return true; }
```

If the shape can not be intersected directly, a `Shape::Refine` method must be provided; this splits the shape into a group of new shapes, some of which may be intersectable and some of which may need further refinement. Repeated application of this method should eventually lead to intersectable shapes. We provide a default implementation of the `Shape::Refine` method that issues an error message. This is so that shapes that are in fact intersectable (which is the common case) do not have to provide an empty instance of this method. `lrt` will never call `Shape::Refine` if `Shape::CanIntersect` returns true.

⟨*Shape Interface*⟩+≡
```
virtual void Refine(vector<RefinedShape> &refined) const {
    Severe("Unimplemented Shape::Refine() method called");
}
```

The `Refine` method returns its results in the `RefinedShape` structure. It allows the `Shape` to return textures that describe the shading normal and tangent along

with the new Shapes. Textures are described in Chapter 11 and shading normals are described in Section 10.1.

⟨*Shape Declarations*⟩+≡
```
struct RefinedShape {
    RefinedShape(const Reference<Shape> &s, Texture<Normal> *ns = NULL,
            Texture<Vector> *ss = NULL) {
        shape = s;
        Ns = ns;
        Ss = ss;
    }
    Reference<Shape> shape;
    Texture<Normal> *Ns;
    Texture<Vector> *Ss;
};
```

Intersection

We will provide two separate intersection routines. The first, Intersect, returns information about a a single ray-shape intersection corresponding to the intersection in the $[\text{mint}, \text{maxt}]$ parametric range along the ray. The other, IntersectP is a predicate function that determines whether or not an intersection occurs, without returning any details about the nature of the intersection itself. Some shapes may be able to provide a more efficient implementation for IntersectP.

Normal 23
Reference 509
Texture 323
Vector 16

There are a few important things to keep in mind when reading and writing intersection routines:

- Recall that the Ray structure contains mint and maxt variables which define a ray *segment* from the point ray(mint)) to the point ray(maxt). Intersection routines should ignore any intersections that do not occur along this segment.
- If an intersection is found, the parametric distance along the ray where it happened should be stored in the pointer thitp that is passed into the intersection routine. If multiple intersections are present, the closest one should be returned.
- Information about intersection positions is stored in the DifferentialGeometry structure, which completely captures the local geometric properties of a surface. This type will be used heavily throughout lrt, and it serves to cleanly isolate the geometric portion of our ray-tracer from the shading and illumination portion. The differential geometry class was defined in Section 2.7 on page 46.
- The rays passed into these routines will be in world space, so shapes are responsible for transforming them to object space if needed for intersection tests. Furthermore, the differential geometry returned should be in world space.

Instead of making the intersection routines pure virtual functions, we provide default implementations of the intersect routine that report a severe error message

if they are called. All shapes that return `true` from `Shape::CanIntersect` must provide implementations of these functions; those that return `false` can depend on the rest of the system to not call these routines on non-intersectable shapes. If these were pure virtual functions, then even non-intersectable shapes would have to implement them, which would be awkward.

⟨*Shape Interface*⟩+≡
```
virtual bool Intersect(const Ray &ray, Float *thitp,
        DifferentialGeometry *dg) const {
    Severe("Unimplemented Shape::Intersect() method called");
    return false;
}

virtual bool IntersectP(const Ray &ray) const {
    Severe("Unimplemented Shape::IntersectP() method called");
    return false;
}
```

Surface Area

In order to properly use `Shape`s as area lights, wee need to be able to compute the surface area of a shape in object space. As with the intersection methods, this method will only be called for intersectable shapes.

47 DifferentialGeometry
26 Ray
498 Severe

⟨*Shape Interface*⟩+≡
```
virtual Float Area() const {
    Severe("Unimplemented Shape::Area() method called");
    return 0.;
}
```

3.2 Spheres

⟨*sphere.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "shapes.h"
#include "mc.h"
⟨Sphere Declarations⟩
⟨Sphere Methods⟩
```

⟨*Sphere Declarations*⟩≡
```
class Sphere: public Shape {
public:
    ⟨Sphere Interface⟩
private:
    ⟨Sphere Data⟩
};
```

Spheres are a special case of a general type of surface called *quadrics*. Quadrics are surfaces described by quadratic polynomials in x, y, and z; they are the simplest

type of curved surface that is useful to a ray tracer, and are an interesting introduction to more general ray intersection routines. The sphere is the simplest quadric. `lrt` supports six types of quadrics: spheres, cones, disks (a special case of a cone), cylinders, hyperboloids, and paraboloids.

Surfaces like quadrics are described mathematically in two main ways: in *implicit form* and in *parametric form*. An implicit function describes a surface (in the three-dimensional case) as:

$$f(x,y,z) = 0$$

The set of x, y, and z that fulfill this condition define the surface. For a unit sphere at the origin, the familiar implicit equation is $x^2 + y^2 + z^2 - 1 = 0$. Only the set of (x,y,z) one unit from the origin satisfies this constraint, giving us the unit sphere's surface.

Many surfaces can also be described parametrically: a functions maps a 2D set of points to 3D points on the surface. For example, a sphere can be described as a function of 2D spherical coordinates (θ,ϕ) where θ ranges from 0 to π and ϕ ranges from 0 to 2π for a complete sphere.

$$\begin{aligned} x &= r\cos\phi\cos\theta \\ y &= r\sin\phi\cos\theta \\ z &= r\sin\theta \end{aligned}$$

We can transform this function $f(\theta,\phi) = (x,y,z)$ into a function $f(u,v)$ over $[0,1]^2$ with the substitution

$$\begin{aligned} \phi &= u\cdot\phi_{\max} \\ \theta &= \theta_{\min} + v\cdot(\theta_{\max} - \theta_{\min}) \end{aligned}$$

This form is particularly useful for texture mapping, where we can directly use the (u,v) values to map a texture map over $[0,1]^2$ over the sphere.

As we describe the implementation of the sphere shape, we will make use of both the implicit and parametric descriptions of the shape, depending on which is a more natural way to approach the particular problem we're facing.

Construction

Our `Sphere` class specifies a shape that is centered at the origin in object space; to place them elsewhere in the scene, the user must apply appropriate transformations when specifying spheres in the input file.

The radius of the sphere can have an arbitrary value, though the sphere's extent can be truncated in two different ways. First, minimum and maximum z values may be set; the parts of the sphere below and above these, respectively, are cut off. Second, if we consider the parameterization of the sphere in spherical coordinates (as in its parametric form), we can set a maximum θ value. The sphere sweeps out θ values from 0 to the given $\theta_{\max}$ such that the section of the sphere with spherical θ values above this θ is also removed.

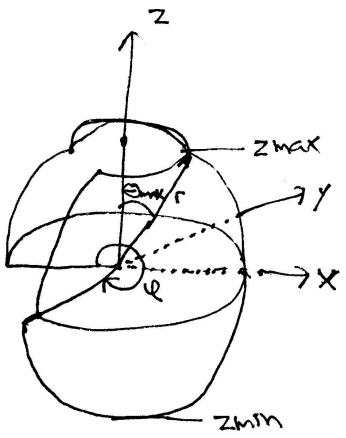


Figure 3.2: Basic setting for the sphere shape. It has a radius of r and XXX. A partial sphere may be swept by specifying a maximum ϕ value.

⟨*Sphere Methods*⟩≡
```
Sphere::Sphere(const Transform &o2w, Float rad, Float z0,
        Float z1, Float pm)
    : Shape(o2w) {
    radius = rad;
    zmin = Clamp(min(z0, z1), -radius, radius);
    zmax = Clamp(max(z0, z1), -radius, radius);
    thetaMin = asinf(zmin/radius);
    thetaMax = asinf(zmax/radius);
    phiMax = Radians(Clamp(pm, 0.0f, 360.0f));
}
```

513 Clamp
513 max
513 min
21 Point
514 Radians
55 Sphere
32 Transform

⟨*Sphere Data*⟩≡
```
Float radius;
Float phiMax;
Float zmin, zmax;
Float thetaMin, thetaMax;
```

Bounding

Computing a bounding box for a sphere is straightforward. We will use the values of $z_{\min}$ and $z_{\max}$ provided by the user to tighten up the bound when less than an entire sphere is being rendered. However, we won't do the extra work to look at $\theta_{\max}$ and see if we can compute a tighter bounding box when that is less than 2π.

⟨*Sphere Methods*⟩+≡
```
BBox Sphere::ObjectBound() const {
    return BBox(Point(-radius, -radius, zmin),
                Point( radius,  radius, zmax));
}
```

Intersection

Because we know that the sphere is centered at the origin, our task for deriving an intersection test is easier than it would be in a more general setting. However, if the sphere has been transformed so that it is at another position in world space, then we need to transform rays before intersecting them with the sphere. Given a ray in world space, it's necessary to apply the inverse of the transformation that places the sphere in world space–i.e. the world to object transformation. Given a ray in object space, we can go ahead and perform the intersection computation in object space.[1]

The entire intersection method is shown below.

⟨*Sphere Methods*⟩+≡
```
bool Sphere::Intersect(const Ray &r, Float *thitp,
        DifferentialGeometry *dg) const {
    Float phi;
    Point Phit;
    ⟨Transform Ray to object space⟩
    ⟨Compute quadratic sphere coefficients⟩
    ⟨Solve quadratic equation for t values⟩
    ⟨Compute sphere hit position and φ⟩
    ⟨Test sphere intersection against clipping parameters⟩
    ⟨Fill in DifferentialGeometry from sphere hit⟩
    ⟨Update thitp for quadric intersection⟩
    return true;
}
```

DifferentialGeometry 47, Point 21, Ray 26, Sphere 55

We need to start by transforming the given world-space ray to the sphere's object space. The remainder of the intersection test will happen in that coordinate system.

⟨*Transform* Ray *to object space*⟩≡
```
Ray ray;
WorldToObject(r, &ray);
```

If we have a sphere centered at the origin with radius r, its implicit representation is

$$x^2 + y^2 + z^2 - r^2 = 0.$$

By substituting the ray equation, 2.4.2 into the implicit sphere equation, we have:

$$\left(o(\mathbf{r})_x + t\tilde{\mathbf{d}}(\mathbf{r})_x\right)^2 + \left(o(\mathbf{r})_y + t\tilde{\mathbf{d}}(\mathbf{r})_y\right)^2 + \left(o(\mathbf{r})_z + t\tilde{\mathbf{d}}(\mathbf{r})_z\right)^2 = r^2.$$

Note that all elements of this equation besides t are known values. The t values where the equation holds give the parametric positions along the ray where the implicit sphere equation holds and thus the points along the ray where it intersects the sphere.

We can expand this equation out and gather the coefficients for a general quadratic in t:

$$At^2 + Bt + C = 0.$$

[1] This is something of a classic theme in computer graphics: by transforming the problem to a particular restricted case, we can more easily and effi ciently do an intersection test (i.e. lots of stuff cancels out since the sphere is always at $(0,0,0)$. No overall generality is lost, since we can just apply an appropriate translation to the ray to account for spheres at other positions, etc.

where

$$\begin{aligned} A &= \tilde{\mathbf{d}}(\mathbf{r})_x^2 + \tilde{\mathbf{d}}(\mathbf{r})_y^2 + \tilde{\mathbf{d}}(\mathbf{r})_z^2 \\ B &= 2(\tilde{\mathbf{d}}(\mathbf{r})_x o(\mathbf{r})_x + \tilde{\mathbf{d}}(\mathbf{r})_y o(\mathbf{r})_y + \tilde{\mathbf{d}}(\mathbf{r})_z o(\mathbf{r})_z) \\ C &= o(\mathbf{r})_x^2 + o(\mathbf{r})_y^2 + o(\mathbf{r})_z^2 - r^2 \end{aligned}$$

This directly translates to this fragment of source code.

⟨*Compute quadratic sphere coefficients*⟩≡
```
Float A = ray.D.x*ray.D.x + ray.D.y*ray.D.y + ray.D.z*ray.D.z;
Float B = 2 * (ray.D.x*ray.O.x + ray.D.y*ray.O.y +
               ray.D.z*ray.O.z);
Float C = ray.O.x*ray.O.x + ray.O.y*ray.O.y +
          ray.O.z*ray.O.z - radius*radius;
```

By the quadratic equation, we know there are two possible solutions to this equation, giving zero, one, or two `t` values where the ray intersects the sphere:

$$\begin{aligned} t_0 &= \frac{-B - \sqrt{B^2 - 4AC}}{2A} \\ t_1 &= \frac{-B + \sqrt{B^2 - 4AC}}{2A} \end{aligned}$$

57 `Sphere::radius`

We will provide a utility `Quadratic` function that solves a quadratic equation, returning `false` if there are no real solutions and returning `true` and setting `t0` and `t1` appropriately if there are solutions.

⟨*Solve quadratic equation for* `t` *values*⟩≡
```
Float t0, t1;
if (!Quadratic(A, B, C, &t0, &t1))
    return false;
⟨Compute intersection distance along ray⟩
```

⟨*Global Inline Functions*⟩≡
```
inline bool Quadratic(Float A, Float B, Float C, Float *t0, Float *t1) {
    ⟨Find quadratic discriminant⟩
    ⟨Compute quadratic t values⟩
}
```

If the discriminant ($B^2 - 4AC$) is negative, then there are no real roots and the ray must miss the sphere.

⟨*Find quadratic discriminant*⟩≡
```
Float discrim = B * B - 4.f * A * C;
if (discrim < 0.) return false;
Float rootDiscrim = sqrtf(discrim);
```

The usual version of quadratic equation can give poor numeric precision when $B \approx \pm\sqrt{B^2 - 4AC}$ due to cancellation error. It can be rewritten algebraically to be in a more stable form.

$$t_0 = \frac{q}{A}$$
$$t_1 = \frac{C}{q}$$

where

$$q = \begin{cases} -.5(B - \sqrt{B^2 - 4AC}) & : \quad B < 0 \\ -.5(B + \sqrt{B^2 - 4AC}) & : \quad \text{otherwise} \end{cases}$$

⟨*Compute quadratic* t *values*⟩≡
```
Float q;
if (B < 0) q = -.5f * (B - rootDiscrim);
else       q = -.5f * (B + rootDiscrim);
*t0 = q / A;
*t1 = C / q;
if (*t0 > *t1) swap(*t0, *t1);
return true;
```

Given the two intersection t values, we need to check them against the ray segment from `mint` to `maxt`. Since t_0 is guaranteed to be less than t_1 (and `mint` less than `maxt`), if t_0 is greater than `maxt` or t_1 is less than `mint`, then it is certain that both hits are out of the range of interest. Otherwise, t_0 is the tentative hit distance. If may be behind `mint`, however, in which case we ignore it and try t_1. If that is also out of range, we have no valid intersection. Otherwise `thit` holds the distance to the hit.

swap 513

⟨*Compute intersection distance along ray*⟩≡
```
if (t0 > ray.maxt || t1 < ray.mint)
    return false;
Float thit = t0;
if (t0 < ray.mint) {
    thit = t1;
    if (thit > ray.maxt) return false;
}
```

Partial Spheres

Now that we have the distance along the ray to the intersection with a full sphere, we need to handle partial spheres, specified with clipped z or ϕ ranges. Intersections that are in clipped areas need to be ignored.

We start by computing the object space position of the intersection, `Phit` and the ϕ value for the hit point. Taking the parametric equations for the sphere,

$$\frac{y}{x} = \frac{r \sin\phi \cos\theta}{r \cos\phi \cos\theta} = \tan\phi$$

so $\phi = \arctan y/x$.

⟨*Compute sphere hit position and* ϕ⟩≡
```
Phit = ray(thit);
phi = atan2f(Phit.y, Phit.x);
if (phi < 0.) phi += 2.f*M_PI;
```

We remap the result from the standard library's `atan2` function to be between 0 and 2π, to match the sphere's original definition.

We can now test the hit point against the specified minima and maxima for z and ϕ. If the intersection wasn't actually valid intersection and we were using the t_0 intersection, we try again with t_1.

⟨*Test sphere intersection against clipping parameters*⟩≡
```
if (Phit.z < zmin || Phit.z > zmax || phi > phiMax) {
    if (thit == t1) return false;
    if (t1 > ray.maxt) return false;
    thit = t1;
    ⟨Compute sphere hit position and φ⟩
    if (Phit.z < zmin || Phit.z > zmax || phiMax)
        return false;
}
```

At this point, we are sure that the ray hits the sphere, and we can fill in the `DifferentialGeometry` structure. We compute parametric u and v values by scaling the previously-computed ϕ value for the hit to lie between 0 and 1 and by computing a θ value for the hit point which is also mapped to $[0,1]$, based on the range of θ values for the given sphere. Next, we compute the parametric partial derivatives $\partial P/\partial u$ and $\partial P/\partial v$, fill in the `DifferentialGeometry` object for the intersection, and transform it out to world space.

47 DifferentialGeometry
57 Sphere::phiMax
57 Sphere::radius
57 Sphere::thetaMax
57 Sphere::thetaMin
57 Sphere::zmax
57 Sphere::zmin

⟨*Fill in* `DifferentialGeometry` *from sphere hit*⟩≡
```
Float u = phi / phiMax;
Float theta = asinf(Phit.z / radius);
Float v = (theta - thetaMin) / (thetaMax - thetaMin);
⟨Compute sphere ∂P/∂u and ∂P/∂v ⟩
⟨Compute sphere ∂N/∂u and ∂N/∂v ⟩
*dg = DifferentialGeometry(ObjectToWorld(Phit), ObjectToWorld(dPdu),
    ObjectToWorld(dPdv), ObjectToWorld(dNdu), ObjectToWorld(dNdv),
    u, v, this);
```

Computing the partial derivatives of a point on the sphere is a short exercise in algebra. Using the parametric definition of the sphere, we have for instance

$$\begin{aligned} x &= r\cos\phi\cos\theta \\ &= r\cos(\phi_{\max}u)\cos(\theta_{\min}+v(\theta_{\max}-\theta_{\min})) \end{aligned}$$

Consider the first component of $\partial P/\partial u$, $\partial x/\partial u$:

$$\begin{aligned} \frac{\partial x}{\partial u} &= \frac{\partial}{\partial u}(r\cos\phi\cos\theta) \\ &= r\cos\theta\frac{\partial}{\partial u}(\cos\phi) \\ &= r\cos\theta(-\phi_{\max}\sin\phi) \end{aligned}$$

Using a substitution based on the parametric definition of the sphere's y coordinate, this simplifies to

$$\partial x/\partial u = -\phi_{\max}y.$$

Similarly

$$\partial y/\partial u = \phi_{\max} x,$$

and

$$\partial z/\partial u = 0.$$

A similar process gives us $\partial P/\partial v$.

$$\begin{aligned}\frac{\partial P}{\partial u} &= (-\phi_{\max} y, \phi_{\max} x, 0)\\ \frac{\partial P}{\partial v} &= (\theta_{\max} - \theta_{\min})(-z\cos\phi, -z\sin\phi, r\cos\theta)\end{aligned}$$

⟨*Compute sphere $\partial P/\partial u$ and $\partial P/\partial v$*⟩≡

```
Float cosphi = cosf(phi), sinphi = sinf(phi);
Vector dPdu(-phiMax * Phit.y, phiMax * Phit.x, 0);
Vector dPdv = (thetaMax-thetaMin) *
    Vector(-Phit.z * cosphi, -Phit.z * sinphi,
        radius * cosf(thetaMin + v * (thetaMax - thetaMin)));
```

It can also be useful to determine how the normal changes as we move along the surface in the u and v directions. For example, some of the anti-aliasing techniques in Chapter 10 will use this information. The differential change in normal $\partial N/\partial u$ and $\partial N/\partial v$ is given by the *Weingarten equations* from differential geometry. They are:

Sphere::phiMax 57
Sphere::radius 57
Sphere::thetaMax 57
Sphere::thetaMin 57
Vector 16

$$\begin{aligned}\frac{\partial N}{\partial u} &= \frac{fF - eG}{EG - F^2}\frac{\partial P}{\partial u} + \frac{eF - fE}{EG - F^2}\frac{\partial P}{\partial v}\\ \frac{\partial N}{\partial v} &= \frac{gF - fG}{EG - F^2}\frac{\partial P}{\partial u} + \frac{fF - gE}{EG - F^2}\frac{\partial P}{\partial v}\end{aligned}$$

where E, F, and G are coefficients of the *first fundamental form* and are given by

$$\begin{aligned}E &= \left|\frac{\partial P}{\partial u}\right|^2\\ F &= \left(\frac{\partial P}{\partial u}\cdot\frac{\partial P}{\partial v}\right)\\ G &= \left|\frac{\partial P}{\partial v}\right|^2.\end{aligned}$$

These are easily computed with the $\partial P/\partial u$ and $\partial P/\partial v$ values that are already availabe. e, f, and g are coefficients of the *second fundamental form*,

$$\begin{aligned}e &= (N\cdot\frac{\partial^2 P}{\partial u^2})\\ f &= (N\cdot\frac{\partial^2 P}{\partial u\partial v})\\ g &= (N\cdot\frac{\partial^2 P}{\partial v^2}).\end{aligned}$$

For these, we need to compute the second order partial derivatives $\partial^2 P/\partial u^2$ and friends. (The two fundamental forms have basic connections with the local curvature of a surface; see any differential geometry textbook (for example, Gray (Gra93)) for details.)

For spheres, a little algebra gives the various second derivatives:

$$\begin{aligned}\frac{\partial^2 P}{\partial u^2} &= -\phi_{\max}^2(x, y, 0)\\ \frac{\partial^2 P}{\partial u \partial v} &= (z_{\max} - z_{\min}) z \phi_{\max}(\sin\phi, -\cos\phi, 0)\\ \frac{\partial^2 P}{\partial v^2} &= -(\theta_{\max} - \theta_{\min})^2(x, y, z)\end{aligned}$$

⟨*Compute sphere $\partial N/\partial u$ and $\partial N/\partial v$*⟩≡
```
Vector d2Pduu = -phiMax * phiMax * Vector(Phit.x, Phit.y, 0);
Vector d2Pduv = (zmax - zmin) * Phit.z * phiMax *
    Vector(sinphi, -cosphi, 0.);
Vector d2Pdvv = -(thetaMax - thetaMin) * (thetaMax - thetaMin) *
    Vector(Phit.x, Phit.y, Phit.z);
```
⟨*Compute coefficients for fundamental forms*⟩
⟨*Compute $\partial N/\partial u$ and $\partial N/\partial v$ from fundamental form coefficients*⟩

20 Cross
16 Vector

⟨*Compute coefficients for fundamental forms*⟩≡
```
Float E = Dot(dPdu, dPdu);
Float F = Dot(dPdu, dPdv);
Float G = Dot(dPdv, dPdv);
Vector N = Cross(dPdu, dPdv);
Float e = Dot(N, d2Pduu);
Float f = Dot(N, d2Pduv);
Float g = Dot(N, d2Pdvv);
```

⟨*Compute $\partial N/\partial u$ and $\partial N/\partial v$ from fundamental form coefficients*⟩≡
```
Float invEGF2 = 1.f / (E*G - F*F);
Vector dNdu = (f*F - e*G) * invEGF2 * dPdu + (e*F - f*E) * invEGF2 * dPdv;
Vector dNdv = (g*F - f*G) * invEGF2 * dPdu + (f*F - g*E) * invEGF2 * dPdv;
```

Since there is an intersection at parametric distance `thit` along the ray, we update the `thitp` value in the ray passed in to the intersect routine to hold the hit distance. This will allow subsequent intersection tests to stop testing for intersection if they determine that the ray would hit beyond an already-found closer intersection.

⟨*Update* `thitp` *for quadric intersection*⟩≡
```
*thitp = thit;
```

The sphere's `IntersectP` routine is almost identical to `Intersect`, but it does not fill in the `DifferentialGeometry` structure. Because `Intersect` and `IntersectP` are always so closely related, we will not show `IntersectP` for the remaining shapes.

⟨*Sphere Methods*⟩+≡
```
bool Sphere::IntersectP(const Ray &r) const {
    Float phi;
    Point Phit;
    ⟨Transform Ray to object space⟩
    ⟨Compute quadratic sphere coefficients⟩
    ⟨Solve quadratic equation for t values⟩
    ⟨Compute sphere hit position and φ⟩
    ⟨Test sphere intersection against clipping parameters⟩
    return true;
}
```

Surface Area

To compute surface area, a useful formula to use reflects the fact that if we revolve a curve $y = f(x)$ from $y = a$ to $y = b$ completely around the x axis, the surface area of the resulting swept surface is

$$2\pi \int_a^b f(x)\sqrt{1 + (f'(x))^2}\,\mathrm{d}x,$$

Point 21
Ray 26
Sphere 55
Sphere::phiMax 57
Sphere::radius 57
Sphere::zmax 57
Sphere::zmin 57

where $f'(x)$ denotes the derivative $d/dx f(x)$. Since most of our surfaces of revolution are only partially swept around the axis, we will actually use the formula:

$$\phi_{\max} \int_a^b f(x)\sqrt{1 + (f'(x))^2}\,\mathrm{d}x.$$

Our sphere is a surface of revolution of a circular arc. Recall that the sphere is clipped at $z_{\min}$ and $z_{\max}$. So the function that defines the profile curve of the sphere is

$$f(x) = \sqrt{r^2 - x^2},$$

and its derivative is

$$f'(x) = -\frac{x}{\sqrt{r^2 - x^2}}.$$

The surface area is therefore

$$\begin{aligned} \mathrm{A} &= \phi_{\max} \int_{z_0}^{z_1} \sqrt{r^2 - x^2}\sqrt{1 + \frac{x^2}{r^2 - x^2}}dx \\ &= \phi_{\max} \int_{z_0}^{z_1} \sqrt{r^2 - x^2 + x^2}dx \\ &= \phi_{\max} \int_{z_0}^{z_1} r dx \\ &= \phi_{\max} r(z_1 - z_0) \end{aligned}$$

This makes sense, because if $\phi_{\max} = 2\pi$, $z_{\min} = -r$ and $z_{\max} = r$, we have the standard formula $4\pi r^2$.

⟨*Sphere Methods*⟩+≡
```
Float Sphere::Area() const {
    return phiMax * radius * (zmax-zmin);
}
```

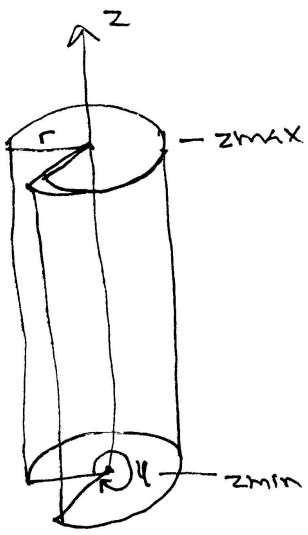


Figure 3.3: Basic setting for the cylinder shape. It has a radius of r and is covers a range of heights along the z-axis. A partial cylinder may be swept by specifying a maximum ϕ value.

3.3 Cylinders

⟨*cylinder.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "shapes.h"
⟨Cylinder Declarations⟩
⟨Cylinder Methods⟩
```

⟨*Cylinder Declarations*⟩≡
```
class Cylinder: public Shape {
public:
    ⟨Cylinder Interface⟩
protected:
    ⟨Cylinder Data⟩
};
```

Construction

Another useful quadric is the cylinder; `lrt` provides cylinder `Shapes` that are centered around the z axis. The user supplies a minimum and maximum z value for the cylinder as well as a radius and maximum ϕ sweep value. In parametric form, a cylinder is described by the equations:

$$\begin{aligned} \phi &= u\phi_{\max} \\ x &= r\cos\phi \\ y &= r\sin\phi \\ z &= z_{\min} + v(z_{\max} - z_{\min}) \end{aligned}$$

⟨*Cylinder Methods*⟩≡
```
Cylinder::Cylinder(const Transform &o2w, Float rad, Float z0,
        Float z1, Float pm)
    : Shape(o2w) {
    radius = rad;
    zmin = min(z0, z1);
    zmax = max(z0, z1);
    phiMax = Radians(Clamp(pm, 0.0f, 360.0f));
}
```

⟨*Cylinder Data*⟩≡
```
Float radius;
Float zmin, zmax;
Float phiMax;
```

Bounding

Like the sphere, we compute a conservative bounding box for the cylinder using the z range but without taking into account the maximum ϕ.

⟨*Cylinder Methods*⟩+≡
```
BBox Cylinder::ObjectBound() const {
    Point p1 = Point(-radius, -radius, zmin);
    Point p2 = Point( radius,  radius, zmax);
    return BBox(p1, p2);
}
```


Clamp 513
Cylinder 65
max 513
min 513
Point 21
Radians 514
Transform 32

Intersection

In a similar manner to the sphere, we can derive the algorithm for finding intersections with cylinders by substituting the ray equation into the cylinder's implicit equation. The implicit equation for an infinitely long cylinder centered on the z axis with radius r is

$$x^2 + y^2 - r^2 = 0.$$

Substituting the ray equation, 2.4.2, we have:

$$\left(o(\mathbf{r})_x + t\tilde{\mathbf{d}}(\mathbf{r})_x\right)^2 + \left(o(\mathbf{r})_y + t\tilde{\mathbf{d}}(\mathbf{r})_y\right)^2 = r^2$$

When we expand this and find the coefficients of the quadratic equation $At^2 + Bt + C$, we get:

$$\begin{aligned} A &= \tilde{\mathbf{d}}(\mathbf{r})_x^2 + \tilde{\mathbf{d}}(\mathbf{r})_y^2 \\ B &= 2(\tilde{\mathbf{d}}(\mathbf{r})_x o(\mathbf{r})_x + \tilde{\mathbf{d}}(\mathbf{r})_y o(\mathbf{r})_y) \\ C &= o(\mathbf{r})_x^2 + o(\mathbf{r})_y^2 - r^2 \end{aligned}$$

⟨*Compute quadratic cylinder coefficients*⟩≡
```
Float A = ray.D.x*ray.D.x + ray.D.y*ray.D.y;
Float B = 2 * (ray.D.x*ray.O.x + ray.D.y*ray.O.y);
Float C = ray.O.x*ray.O.x + ray.O.y*ray.O.y - radius*radius;
```

The solution process for the quadratic equation is the same for all quadric shapes, so some fragments from the `Sphere` intersection method will be re-used below.

⟨*Cylinder Methods*⟩+≡
```
bool Cylinder::Intersect(const Ray &r, Float *thitp,
        DifferentialGeometry *dg) const {
    Float phi;
    Point Phit;
    ⟨Transform Ray to object space⟩
    ⟨Compute quadratic cylinder coefficients⟩
    ⟨Solve quadratic equation for t values⟩
    ⟨Compute cylinder hit point and φ⟩
    ⟨Test cylinder intersection against clipping parameters⟩
    ⟨Fill in DifferentialGeometry from cylinder hit⟩
    ⟨Update thitp for quadric intersection⟩
    return true;
}
```

Partial Cylinders

As with the sphere, we invert the parametric description of the cylinder to compute a ϕ value by inverting the x and y parametric equations to solve for ϕ. In fact, the result is the same as for the sphere.

65 Cylinder
66 Cylinder::phiMax
66 Cylinder::zmax
66 Cylinder::zmin
47 DifferentialGeometry
21 Point
26 Ray

⟨*Compute cylinder hit point and φ*⟩≡
```
Phit = ray(thit);
phi = atan2f(Phit.y, Phit.x);
if (phi < 0.) phi += 2.f*M_PI;
```

We now make sure that the hit is between the specified z range, and that the angle is acceptable. If not, we reject the hit and possibly try again with t_1, if we weren't using it the first time through.

⟨*Test cylinder intersection against clipping parameters*⟩≡
```
if (Phit.z < zmin || Phit.z > zmax || phi > phiMax) {
    if (thit == t1) return false;
    thit = t1;
    if (t1 > ray.maxt) return false;
    ⟨Compute cylinder hit point and φ⟩
    if (Phit.z < zmin || Phit.z > zmax || phi > phiMax)
        return false;
}
```

Again like the sphere the u value is computed by scaling ϕ to lie between 0 and 1. Straightforward inversion of the parametric equation for the cylinder's z value gives us the v parametric coordinate.

⟨*Fill in* `DifferentialGeometry` *from cylinder hit*⟩≡
```
Float u = phi / phiMax;
Float v = (Phit.z - zmin) / (zmax - zmin);
⟨Compute cylinder ∂P/∂u and ∂P/∂v ⟩
⟨Compute cylinder ∂N/∂u and ∂N/∂v ⟩
*dg = DifferentialGeometry(ObjectToWorld(Phit), ObjectToWorld(dPdu),
    ObjectToWorld(dPdv), ObjectToWorld(dNdu), ObjectToWorld(dNdv),
    u, v, this);
```

The partial derivatives for a cylinder are quite easy to derive: they are

$$\begin{aligned}\frac{\partial P}{\partial u} &= (-\phi_{\max} y, \phi_{\max} x, 0)\\ \frac{\partial P}{\partial v} &= (0, 0, z_{\max} - z_{\min})\end{aligned}$$

⟨*Compute cylinder ∂P/∂u and ∂P/∂v* ⟩≡
```
Vector dPdu(-phiMax * Phit.y, phiMax * Phit.x, 0);
Vector dPdv(0, 0, zmax - zmin);
```

We again use the Weingarten equations to compute the parametric change in cylinder normal. The relevant partial derivatives are

$$\begin{aligned}\frac{\partial^2 P}{\partial u^2} &= -\phi_{\max}^2 (x, y, 0)\\ \frac{\partial^2 P}{\partial u \partial v} &= (0, 0, 0)\\ \frac{\partial^2 P}{\partial v^2} &= (0, 0, 0)\end{aligned}$$

⟨*Compute cylinder ∂N/∂u and ∂N/∂v* ⟩≡
```
Vector d2Pduu = -phiMax * phiMax * Vector(Phit.x, Phit.y, 0);
Vector d2Pduv(0, 0, 0), d2Pdvv(0, 0, 0);
⟨Compute coefficients for fundamental forms⟩
⟨Compute ∂N/∂u and ∂N/∂v from fundamental form coefficients⟩
```

Surface Area

A cylinder is just a rolled rectangle. The height of the rectangle is $z_{\max} - z_{\min}$, and the width is $r\phi_{\max}$:

⟨*Cylinder Methods*⟩+≡
```
Float Cylinder::Area() const {
    return (zmax-zmin)*phiMax*radius;
}
```


Cylinder 65
Cylinder::phiMax 66
Cylinder::radius 66
Cylinder::zmax 66
Cylinder::zmin 66
DifferentialGeometry 47
Vector 16

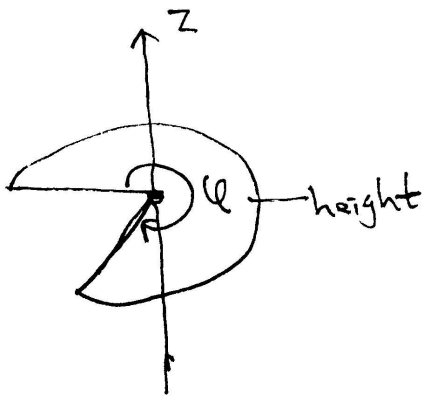


Figure 3.4: Basic setting for the disk shape. The disk has radius of r and is located at some height along the z-axis. A partial disk may be swept by specifying a maximum ϕ value.

3.4 Disks

⟨*disk.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "shapes.h"
⟨Disk Declarations⟩
⟨Disk Methods⟩
```

70 Disk

⟨*Disk Declarations*⟩≡
```
class Disk : public Shape {
public:
    ⟨Disk Interface⟩
private:
    ⟨Disk Private Data⟩
};
```

The disk is an interesting quadric since it has a particularly straightforward intersection routine that avoids solving the quadratic equation. In `lrt`, a `Disk` is a circular disk of user-supplied radius at some height along the z axis. In order to make partial disks, the user may specify a maximum ϕ value beyond which the disk is cut off (see Figure 3.4). In parametric form, it is described by:

$$\begin{aligned} \phi &= u\phi_{\max} \\ x &= r(1-v)\cos\phi \\ x &= r(1-v)\sin\phi \\ z &= \text{height} \end{aligned}$$

Construction

⟨*Disk Methods*⟩≡
```
Disk::Disk(const Transform &o2w, Float ht, Float r, Float tmax)
    : Shape(o2w) {
    height = ht;
    radius = r;
    phiMax = Radians(Clamp(tmax, 0.0f, 360.0f));
}
```

⟨*Disk Private Data*⟩≡
```
Float height, radius, phiMax;
```

Bounding

The bounding method is quite straightforward; we create a bounding box centered at the height of the disk along z, with extent of `radius` in both the x and y directions.

⟨*Disk Methods*⟩+≡
```
BBox Disk::ObjectBound() const {
    return BBox(Point(-radius,-radius,height),
                Point(radius, radius,height));
}
```


Clamp 513
DifferentialGeometry 47
Point 21
Radians 514
Ray 26
Transform 32

Intersection

Intersecting a ray with a disk is also quite easy. We intersect the ray with the

$$z = \text{height}$$

plane that the disk lies in and then see if the intersection point lies inside the disk.

⟨*Disk Methods*⟩+≡
```
bool Disk::Intersect(const Ray &r, Float *thitp,
        DifferentialGeometry *dg) const {
    ⟨Transform Ray to object space⟩
    ⟨Compute plane intersection for disk⟩
    ⟨See if hit point is inside disk radius and φmax⟩
    ⟨Fill in DifferentialGeometry from disk hit⟩
    ⟨Update thitp for quadric intersection⟩
    return true;
}
```

The first thing we do is compute the parametric t value where the ray intersects the plane that the disk lies in. Using the same approach as for intersecting rays with boxes, we want to find t such that the z component of the ray's position is equal to the height where the user placed the disk. Thus,

$$h = o(\mathbf{r})_z + t * \tilde{\mathbf{d}}(\mathbf{r})_z$$

So t is

$$t = \frac{h - o(\mathbf{r})_z}{\tilde{\mathbf{d}}(\mathbf{r})_z}$$

After checking to be sure that the ray isn't parallel to the disk's plane, in which case we report no intersection, we compare this t value and see if it is inside the legal range of t values, $[\texttt{mint}, \texttt{maxt}]$. If not, we can return false.

⟨*Compute plane intersection for disk*⟩≡
```
if (fabsf(ray.D.z) < 1e-7) return false;
Float thit = (height - ray.O.z) / ray.D.z;
if (thit < ray.mint || thit > ray.maxt)
    return false;
```

We now compute the point where the ray intersects the plane, `Phit`. Once the plane intersection is known, we check if the distance from the hit to the center of the disk is less than `radius`. If it's farther away, we return false. We optimize this process by actually computing the squared distance to the center, taking advantage of the fact that the x and y coordinates of the center point $(0, 0, \text{height})$ are zero, and that the z coordinate of `Phit` is equal to `height`.

⟨*See if hit point is inside disk radius and* ϕ_{max} ⟩≡
```
Point Phit = ray(thit);
Float Dist2 = Phit.x * Phit.x + Phit.y * Phit.y;
if (Dist2 > radius * radius)
    return false;
⟨Test disk φ value against φmax ⟩
```

If the distance check passes, we perform the final test, making sure that the ϕ value of the hit point is between zero and $\phi_{\max}$ specified by the user. Inverting the disk's parameterization gives us the same expression for ϕ as the other quadric shapes have.

⟨*Test disk* ϕ *value against* ϕ_{max} ⟩≡
```
Float phi = atan2f(Phit.y, Phit.x);
if (phi < 0) phi += 2. * M_PI;
if (phi > phiMax)
    return false;
```

If we've gotten this far, we know that there is an intersection with the disk. The parameter `u` is scaled to reflect the partial disk specified by $\phi_{\max}$ and `v` is computed by inverting the parametric equation. The equations for the partial derivatives at the hit point can be derived with a similar process as was used for the previous quadrics. Because the normal of a disk is the same everywhere, the partial derivatives $\partial N/\partial u$ and $\partial N/\partial v$ are both trivially $(0, 0, 0)$.

⟨*Fill in* `DifferentialGeometry` *from disk hit*⟩≡
```
Float u = phi / phiMax;
Float v = 1.f - (sqrtf(Dist2) / radius);
Vector dPdu(-phiMax * Phit.y, phiMax * Phit.x, 0.);
Vector dPdv(-Phit.x / (1-v), -Phit.y / (1-v), 0.);
*dg = DifferentialGeometry(ObjectToWorld(Phit), ObjectToWorld(dPdu),
    ObjectToWorld(dPdv), Vector(0,0,0), Vector(0,0,0), u, v, this);
```

47 DifferentialGeometry
70 Disk::phiMax
21 Point
16 Vector

Surface Area

Disks have trivial surface area, since they're just portions of a circle:

⟨*Disk Methods*⟩+≡
```
Float Disk::Area() const {
    return phiMax * 0.5f * radius * radius;
}
```

3.5 Other Quadrics

Filenames: `cone.cc`, `paraboloid.cc` and `hyperboloid.cc`.

`lrt` supports three more various quadrics: cones, paraboloids, and hyperboloids. We won't include their full implementations here, since there is little to be gained by walking through them; the same techniques are used to derive their quadratic intersection coefficients, parametric coordinates and partial derivatives as have been used for the previous quadrics. However, we will briefly describe the implicit and parametric forms of these shapes.

The implicit equation of a cone centered on the z axis with radius r and height h is

$$\left(\frac{hx}{r}\right)^2 + \left(\frac{hy}{r}\right)^2 - (z-h)^2 = 0.$$

They also have the parametric description

Disk 70
Disk::phiMax 70

$$\begin{aligned} \phi &= u\phi_{\max} \\ x &= r(1-v)\cos\phi \\ y &= r(1-v)\sin\phi \\ z &= v\,\text{height} \end{aligned}$$

The partial derivatives are:

$$\begin{aligned} \frac{\partial P}{\partial u} &= (-\phi_{\max} y, \phi_{\max} x, 0) \\ \frac{\partial P}{\partial v} &= (-x/(1-v), y/(1-v), \text{height}) \end{aligned}$$

and

$$\begin{aligned} \frac{\partial^2 P}{\partial u^2} &= -\phi_{\max}^2 (x, y, 0) \\ \frac{\partial^2 P}{\partial u \partial v} &= \frac{\phi_{\max}}{1-v}(y, -x, 0) \\ \frac{\partial^2 P}{\partial v^2} &= (0,0,0) \end{aligned}$$

The implicit equation of a paraboloid centered on the z axis with radius r at $z = h$ is:

$$\frac{hx^2}{r^2} + \frac{hy^2}{r^2} - z = 0$$

and the parametric form is

$$\begin{aligned} \phi &= u\phi_{\max} \\ z &= v(z_{\max} - z_{\min}) \\ r &= r_{\max}\sqrt{z/z_{\max}} \\ x &= r\cos\phi \\ y &= r\sin\phi \end{aligned}$$

The partial derivatives are:

$$\begin{aligned}\frac{\partial P}{\partial u} &= (-\phi_{\max} y, \phi_{\max} x, 0)\\ \frac{\partial P}{\partial v} &= (z_{\max} - z_{\min})(x/z, y/z, 1)\end{aligned}$$

and

$$\begin{aligned}\frac{\partial^2 P}{\partial u^2} &= -\phi_{\max}^2 (x, y, 0)\\ \frac{\partial^2 P}{\partial u \partial v} &= \phi_{\max}(z_{\max} - z_{\min})(-y/z, x/z, 0)\\ \frac{\partial^2 P}{\partial v^2} &= -2(z_{\max} - z_{\min})^2 (x/z^2, y/z^2, 0)\end{aligned}$$

Finally, the implicit form of the hyperboloid is

$$x^2 + y^2 - z^2 = -1$$

and the parametric form is

$$\begin{aligned}\phi &= u\phi_{\max}\\ x_r &= (1-v)x_1 + vx_2\\ y_r &= (1-v)y_1 + vy_2\\ x &= x_r \cos\phi - y_r \sin\phi\\ y &= x_r \sin\phi + y_r \cos\phi\\ z &= (1-v)z_1 + vz_2\end{aligned}$$

The partial derivatives are:

$$\begin{aligned}\frac{\partial P}{\partial u} &= (-\phi_{\max} y, \phi_{\max} x, 0)\\ \frac{\partial P}{\partial v} &= ((x_2 - x_1)\cos\phi - (y_2 - y_1)\sin\phi, (x_2 - x_1)\sin\phi + (y_2 - y_1)\cos\phi, z_2 - z_1)\end{aligned}$$

and

$$\begin{aligned}\frac{\partial^2 P}{\partial u^2} &= -\phi_{\max}^2 (x, y, 0)\\ \frac{\partial^2 P}{\partial u \partial v} &= \phi_{\max}(-\partial y/\partial v, \partial x/\partial v, 0)\\ \frac{\partial^2 P}{\partial v^2} &= (0, 0, 0)\end{aligned}$$

3.6 Triangles and Meshes

⟨*trianglemesh.h**⟩≡
⟨*Source Code Copyright*⟩
```
#ifndef TRIANGLEMESH_H
#define TRIANGLEMESH_H
#include "lrt.h"
#include "shapes.h"
#include "paramset.h"
```
⟨*TriangleMesh Declarations*⟩
```
#endif // TRIANGLEMESH_H
```

⟨*trianglemesh.cc**⟩≡
⟨*Source Code Copyright*⟩
```
#include "lrt.h"
#include "shapes.h"
#include "paramset.h"
#include "trianglemesh.h"
```
⟨*TriangleMesh Methods*⟩

⟨*TriangleMesh Declarations*⟩≡
```
class TriangleMesh: public Shape {
public:
```
    ⟨*TriangleMesh Interface*⟩
```
protected:
```
    ⟨*TriangleMesh Data*⟩
```
};
```

The triangle is one of the most commonly used shapes in computer graphics. `lrt` supports triangle meshes, where a number of triangles are stored together so that their per-vertex data can be shared among multiple triangles that reference it. Single triangles are simply treated as degenerate meshes.

The arguments to the `TriangleMesh` constructor are as follows:

- `nt` Number of triangles in this mesh
- `nv` Number of vertices in this mesh
- `vi` Pointer to an array of vertex indices. For the `i`th triangle, its three vertex positions are `P[vi[3*i]]`, `P[vi[3*i+1]]`, and `P[vi[3*i+2]]`.
- `P` Array of `nv` vertex positions.
- `uv` An optional array of a parametric (u,v) value for each vertex.

We just copy the relevant information and store it in the `TriangleMesh` object. In particular, must make our own copies of `vi` and `P`, since the caller retains ownership of the data being passed in.

⟨*TriangleMesh Methods*⟩≡
```
TriangleMesh::TriangleMesh(const Transform &o2w, int nt, int nv,
        const int *vi, const Point *P, const Float *uv)
    : Shape(o2w) {
    ntris = nt;
    nverts = nv;
    vertexIndex = new int[3 * ntris];
    memcpy(vertexIndex, vi, 3 * ntris * sizeof(int));
    if (uv) {
        uvs = new Float[2*nverts];
        memcpy(uvs, uv, 2*nverts*sizeof(Float));
    }
    else uvs = NULL;
    p = new Point[nverts];
    ⟨Transform mesh vertices to world space⟩
}
```

⟨*TriangleMesh Data*⟩≡
```
int ntris;
int nverts;
int *vertexIndex;
Point *p;
Float *uvs;
```

21 Point
32 Transform
29 Union

Unlike the other primitives, where we leave the primitive description in object space and then transform incoming rays from world space to object space, here we do the opposite, and transform the primitive into world space. As a result, we won't need to transform the incoming rays or the intersection differential geometry.

⟨*Transform mesh vertices to world space*⟩≡
```
for (int i  = 0; i < nverts; ++i)
    p[i] = ObjectToWorld(P[i]);
```

⟨*TriangleMesh Methods*⟩+≡
```
TriangleMesh::~TriangleMesh() {
    delete[] vertexIndex;
    delete[] p;
}
```

The object-space bound of a triangle mesh is easily found by computing a bounding box that encompasses all of the vertices of the mesh. We transform the world-space `p` positions back to object space before computing their bound.

⟨*TriangleMesh Methods*⟩+≡
```
BBox TriangleMesh::ObjectBound() const {
    BBox bobj;
    for (int i = 0; i < nverts; i++)
        bobj = Union(bobj, WorldToObject(p[i]));
    return bobj;
}
```

The TriangleMesh shape is one of the shapes that can usually compute a better world space bound than can be found by transforming its object-space bounding box to world space. We just directly compute a bounding box of the world-space vertices.

⟨*TriangleMesh Methods*⟩+≡
```
BBox TriangleMesh::WorldBound() const {
    BBox worldBounds;
    for (int i = 0; i < nverts; i++)
        worldBounds = Union(worldBounds, p[i]);
    return worldBounds;
}
```

The TriangleMesh shape does not directly compute intersections. Instead, it splits itself into many separate Triangles, each representing a single triangle. This allows all of the individual triangles to reference the shared set of vertices in p, saving us from needing to replicate the shared data for each triangle. We override the CanIntersect method of Shape to indicate that TriangleMeshs can not be intersected directly.

⟨*TriangleMesh Interface*⟩+≡
```
bool CanIntersect() const { return false; }
```

When lrt encounters a shape that cannot be intersected directly, it calls its Refine method. Refine is expected to produce a list of simpler shapes in the refined vector. The implementation here is simple; we just make a new Triangle for each of the triangles in the mesh.

⟨*TriangleMesh Methods*⟩+≡
```
void TriangleMesh::Refine(vector<RefinedShape> &refined) const {
    for (int i = 0; i < ntris; ++i)
        refined.push_back(RefinedShape(
            new Triangle(ObjectToWorld, (TriangleMesh *)this, i)));
}
```

Triangle

⟨*TriangleMesh Declarations*⟩+≡
```
class Triangle : public Shape {
public:
    ⟨Triangle Interface⟩
//private:
    ⟨Triangle Data⟩
};
```

The Triangle doesn't store much data; just a pointer to the parent TriangleMesh that it came from and a pointer to its three vertex indices in the mesh.

ntris 75
nverts 75
push_back 494
RefinedShape 54
Union 29

⟨*Triangle Interface*⟩≡
```
Triangle(const Transform &o2w, TriangleMesh *m, int n)
        : Shape(o2w) {
    mesh = m;
    v = &mesh->vertexIndex[3*n];
#ifndef OLD_SCHOOL
    p1 = m->p[v[0]];
    p2 = m->p[v[1]];
    p3 = m->p[v[2]];
#endif
    ⟨Update created triangles stats⟩
}
```

⟨*Triangle Data*⟩≡
```
Reference<TriangleMesh> mesh;
int *v;
#ifndef OLD_SCHOOL
Point p1, p2, p3;
#endif
```

⟨*Update created triangles stats*⟩≡
```
static StatsCounter trisMade("Geometry", "Triangles created");
++trisMade;
```

As with TriangleMeshes, we can compute better world space bounding boxes for individual triangles by bounding the world space vertices directly.

⟨*TriangleMesh Methods*⟩+≡
```
BBox Triangle::ObjectBound() const {
    ⟨Get triangle vertices in p1, p2, and p3⟩
    return Union(BBox(WorldToObject(p1), WorldToObject(p2)),
        WorldToObject(p3));
}

BBox Triangle::WorldBound() const {
    ⟨Get triangle vertices in p1, p2, and p3⟩
    return Union(BBox(p1, p2), p3);
}
```

⟨*Get triangle vertices in* p1, p2, *and* p3⟩≡
```
#ifdef OLD_SCHOOL
const Point &p1 = mesh->p[v[0]];
const Point &p2 = mesh->p[v[1]];
const Point &p3 = mesh->p[v[2]];
#endif
```

Triangles have a dual role among the primitives in lrt: not only are they used as a user-specified primitive, but other primitives may tessellate themselves into triangle meshes; for example, subdivision surfaces end up creating a mesh of triangles to approximate the smooth subdivision limit surface–ray intersections are performed against these triangles, rather than directly against the subdivision surface.

21 Point
509 Reference
501 StatsCounter
32 Transform
29 Union
75 vertexIndex

Because of this second role, it's important that code that creates triangle meshes be able to specify the parameterization of the triangles. If a triangle was created by evaluating the position of a parametric surface at three particular (u,v) coordinate values, those (u,v) values should be interpolated to compute the (u,v) value at ray intersection points inside the triangle.

The `GetUVs` method of the `Triangle` class returns the parametric coordinates for the three vertices of a triangle. If the `TriangleMesh` has a non-NULL `uvs` value, the appropriate values are retrieved and returned. Otherwise, we use default coordinates of $(0,0)$, $(1,0)$, and $(1,1)$.

⟨*Shape Method Definitions*⟩+≡
```
void Triangle::GetUVs(Float uv[3][2]) const {
    if (mesh->uvs) {
        uv[0][0] = mesh->uvs[2*v[0]];
        uv[0][1] = mesh->uvs[2*v[0]+1];
        uv[1][0] = mesh->uvs[2*v[1]];
        uv[1][1] = mesh->uvs[2*v[1]+1];
        uv[2][0] = mesh->uvs[2*v[2]];
        uv[2][1] = mesh->uvs[2*v[2]+1];
    }
    else {
         uv[0][0] = uv[0][1] = uv[1][1] = 0.;
         uv[1][0] = uv[2][0] = uv[2][1] = 1.;
    }
}
```

Triangle Intersection

An algorithm for ray-triangle intersection can be computed using *barycentric coordinates*. Barycentric coordinates provide a way to parameterize a triangle in terms of two variables, b_1 and b_2:

$$\mathbf{p}(b_1,b_2) = (1-b_1-b_2)\mathbf{p}_0 + b_1\mathbf{p}_1 + b_2\mathbf{p}_2$$

The conditions on b_1 and b_2 are that $b_1 \geq 0$, $b_2 \geq 0$, and $b_1 + b_2 \leq 1$. This is the parametric form of a triangle. The barycentric coordinates are also a natural way to interpolate across the surface of the triangle; given values defined at the vertices a_0, a_1, and a_2 and given the barycentric coordinates for a point on the triangle, we can compute an interpolated value of a at that point as $(1-b_1-b_2)a_0 + b_1a_1 + b_2a_2$. (See Section 11.5 on page 331 for a texture that interpolates shading values over a triangle mesh in this manner.)

To derive an algorithm for intersecting a ray with a triangle, we insert the parametric ray equation into the triangle equation.

$$o(\mathbf{r}) + t\vec{\mathbf{d}}(\mathbf{r}) = (1-b_1-b_2)\mathbf{p}_0 + b_1\mathbf{p}_1 + b_2\mathbf{p}_2 \tag{3.6.1}$$

Following the technique described by Möller and Trumbore(MT97), we use the shorthand notation $\vec{\mathbf{e}}_1 = \mathbf{p}_1 - \mathbf{p}_0$, $\vec{\mathbf{e}}_2 = \mathbf{p}_2 - \mathbf{p}_0$, and $\vec{\mathbf{t}} = o(\mathbf{r}) - \mathbf{p}_0$. We can now

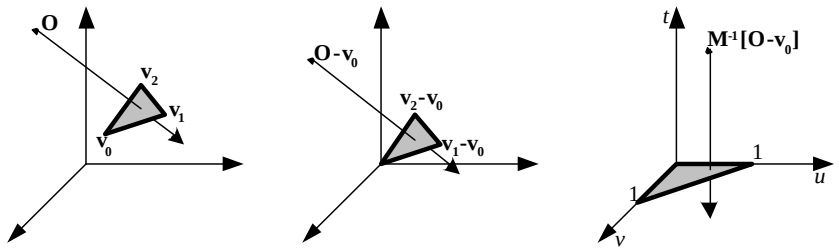


Figure 3.5: Transforming the ray into a more convenient coordinate system for intersection. First, a translation is applied to make a corner of the triangle coincide with the origin. Then, the triangle is rotated and scaled to a unit right-triangle.

rearrange terms of Equation 3.6.1 to obtain the matrix equation:

$$\begin{bmatrix} -\vec{\mathbf{d}}(\mathbf{r}) & \vec{\mathbf{e}}_1 & \vec{\mathbf{e}}_2 \end{bmatrix} \begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \vec{\mathbf{t}} \tag{3.6.2}$$

Solving this linear system will give us both the barycentric coordinates of the intersection point (which can easily be used to compute the 3D intersection point) as well as the distance along the ray.

Geometrically, we can interpret this system as a translation of the triangle to the origin, and a transformation of the triangle to a unit triangle in *y* and *z*, keeping the ray direction aligned with *x*, as shown in Figure 3.5.

Cramer's rule gives a solution to equation 3.6.2:

XXX Need to explain the $|\vec{\mathbf{a}}\vec{\mathbf{b}}\vec{\mathbf{c}}|$ notation–determinant of a 3x3 matrix. XXX

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\left|-\vec{\mathbf{d}}(\mathbf{r}) \quad \vec{\mathbf{e}}_1 \quad \vec{\mathbf{e}}_2\right|} \begin{bmatrix} \left| \vec{\mathbf{t}} \quad \vec{\mathbf{e}}_1 \quad \vec{\mathbf{e}}_2 \right| \\ \left|-\vec{\mathbf{d}}(\mathbf{r}) \quad \vec{\mathbf{t}} \quad \vec{\mathbf{e}}_2\right| \\ \left|-\vec{\mathbf{d}}(\mathbf{r}) \quad \vec{\mathbf{e}}_1 \quad \vec{\mathbf{t}}\right| \end{bmatrix} \tag{3.6.3}$$

This can be rewritten as $\left|\vec{\mathbf{A}} \quad \vec{\mathbf{B}} \quad \vec{\mathbf{C}}\right| = -\left(\vec{\mathbf{A}}\times\vec{\mathbf{C}}\right)\cdot\vec{\mathbf{B}} = -\left(\vec{\mathbf{C}}\times\vec{\mathbf{B}}\right)\cdot\vec{\mathbf{A}}$. We can thus rewrite Equation 3.6.3 as:

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\left(\vec{\mathbf{d}}(\mathbf{r})\times\vec{\mathbf{e}}_2\right)\cdot\vec{\mathbf{e}}_1} \begin{bmatrix} \left(\vec{\mathbf{t}}\times\vec{\mathbf{e}}_1\right)\cdot\vec{\mathbf{e}}_2 \\ \left(\vec{\mathbf{d}}(\mathbf{r})\times\vec{\mathbf{e}}_2\right)\cdot\vec{\mathbf{t}} \\ \left(\vec{\mathbf{t}}\times\vec{\mathbf{e}}_1\right)\cdot\vec{\mathbf{d}}(\mathbf{r}) \end{bmatrix} \tag{3.6.4}$$

If we use the substitution $\vec{\mathbf{s}}_1 = \vec{\mathbf{d}}(\mathbf{r})\times\vec{\mathbf{e}}_2$ and $\vec{\mathbf{s}}_2 = \vec{\mathbf{t}}\times\vec{\mathbf{e}}_1$ we can make the common subexpressions more explicit:

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\vec{\mathbf{s}}_1\cdot\vec{\mathbf{e}}_1} \begin{bmatrix} \vec{\mathbf{s}}_2\cdot\vec{\mathbf{e}}_2 \\ \vec{\mathbf{s}}_1\cdot\vec{\mathbf{t}} \\ \vec{\mathbf{s}}_2\cdot\vec{\mathbf{d}}(\mathbf{r}) \end{bmatrix} \tag{3.6.5}$$

In order to compute $\vec{\mathbf{e}}_1$, $\vec{\mathbf{e}}_2$, and $\vec{\mathbf{t}}$ we need 9 subtractions. To compute $\vec{\mathbf{s}}_1$ and $\vec{\mathbf{s}}_2$, we need two cross products, which is a total of 12 multiplies and 6 subtractions. Finally, to compute t, b_1, and b_2, we need 4 dot products (12 multiplies and 8 additions), 1 reciprocal, and 3 multiplies. Thus, the total cost of ray-triangle intersection is 1 divide, 27 multiplies, and 17 additions (counting additions and subtractions together). Note that some of these operations can be avoided if it is determined mid-calculation that the ray does not intersect the triangle.

⟨*TriangleMesh Methods*⟩+≡
```
bool Triangle::Intersect(const Ray &ray, Float *thitp,
        DifferentialGeometry *dg) const {
    ⟨Initialize triangle intersection statistics⟩
    ⟨Update triangle tests count⟩
    ⟨Compute s1⟩
    ⟨Compute first barycentric coordinate⟩
    ⟨Compute second barycentric coordinate⟩
    ⟨Compute t to intersection point⟩
    ⟨Fill in DifferentialGeometry from triangle hit⟩
    *thitp = t;
    return true;
}
```

Cross 20
DifferentialGeometry 47
Dot(v,v) 19
Ray 26
StatsRatio 501
Vector 16

⟨*Initialize triangle intersection statistics*⟩≡
```
static StatsRatio triangleHits("Geometry", "Triangle Ray Intersections");
```

⟨*Update triangle tests count*⟩≡
```
triangleHits.add(0, 1);
```

First, we compute the divisor from Equation 3.6.5. We figure out which three mesh vertices are the ones for this particular `Triangle`, and then compute the edge vectors and divisor. Note that if the divisor is zero, this triangle is degenerate and therefore cannot intersect a ray.

⟨*Compute* $\vec{\mathbf{s}}_1$⟩≡
```
⟨Get triangle vertices in p1, p2, and p3⟩
Vector E1 = p2 - p1;
Vector E2 = p3 - p1;
Vector S_1 = Cross(ray.D, E2);
Float divisor = Dot(S_1, E1);
if (divisor == 0.)
    return false;
Float invDivisor = 1.f / divisor;
```

We can now compute the desired barycentric coordinate b_1. Recall that barycentric coordinates that are less than zero or greater than one represent points outside the triangle, so those are non-intersections.

⟨*Compute first barycentric coordinate*⟩≡
```
Vector T = ray.O - p1;
Float b1 = Dot(T, S_1) * invDivisor;
if (b1 < 0. || b1 > 1.)
    return false;
```

The second barycentric coordinate, b_2, is computed in a similar way:

⟨*Compute second barycentric coordinate*⟩≡
```
Vector S_2 = Cross(T, E1);
Float b2 = Dot(ray.D, S_2) * invDivisor;
if (b2 < 0. || b1 + b2 > 1.)
    return false;
```

Now that we know the ray intersects the triangle, we compute the distance along the ray at which the intersection occurs. This gives us one last opportunity to exit the procedure early, in case the t value falls outside our `mint` and `maxt` bounds.

⟨*Compute* `t` *to intersection point*⟩≡
```
Float t = Dot(E2, S_2) * invDivisor;
if (t < ray.mint || t > ray.maxt)
    return false;
triangleHits.add(1, 0);
```

We now have all the information we need to compute the `DifferentialGeometry` structure for this intersection. In contrast to previous shapes, we don't need to transform the partial derivatives to world-space, since the triangle's vertices were already transformed to world-space themselves. Like the disk, the triangles normal partial derivatves are also both $(0,0,0)$.

20 `Cross`
47 `DifferentialGeometry`
19 `Dot(v,v)`
16 `Vector`

⟨*Fill in* `DifferentialGeometry` *from triangle hit*⟩≡
```
⟨Compute triangle partial derivatives⟩
⟨Interpolate (u,v) triangle parametric coordinates⟩
*dg = DifferentialGeometry(ray(t), dPdu, dPdv, Vector(0,0,0),
    Vector(0,0,0), tu, tv, this);
```

In order to have consistent tangents and bitangents over triangle meshes we'll compute the partial derivatives $\partial P/\partial u$ and $\partial P/\partial v$ of the triangle using the parametric (u,v) values provided at the triangle vertices, if any. Although the partial derivatives are the same at all points on the triangle, we will just recompute them each time an intersection is found.

The triangle is the set of points

$$P_o + u\partial P/\partial u + v\partial P/\partial v,$$

for some P_o, where u and v range over the parametric coordinates of the triangle. We also know the three vertex positions V_i, $i = 0,1,2$ and the texture coordinates (u_i, v_i) at each vertex. From this it follows that

$$V_i = P_o + u_i\partial P/\partial u + v_i\partial P/\partial v.$$

We can write this in matrix form:

$$\begin{pmatrix} V_0 \\ V_1 \\ V_2 \end{pmatrix} = \begin{pmatrix} u_0 & v_0 & 1 \\ u_1 & v_1 & 1 \\ u_2 & v_2 & 1 \end{pmatrix} \begin{pmatrix} \partial P/\partial u \\ \partial P/\partial v \\ P_o \end{pmatrix}$$

In other words, there is a unique affine mapping from the two-dimensional (u,v) space to points on the triangle (such a mapping exists since although the triangle

is specified in 3D space, it is a within a 2D plane through 3D space.) To compute expressions for $\partial P/\partial u$ and $\partial P/\partial v$, we just need to solve the matrix equation. We subtract the bottom row of each matrix from the top two rows, giving:

$$\begin{pmatrix} V_0 - V_2 \\ V_1 - V_2 \end{pmatrix} = \begin{pmatrix} u_0 - u_2 & v_0 - v_2 \\ u_1 - u_2 & v_1 - v_2 \end{pmatrix} \begin{pmatrix} \partial P/\partial u \\ \partial P/\partial v \end{pmatrix}$$

So

$$\begin{pmatrix} \partial P/\partial u \\ \partial P/\partial v \end{pmatrix} = \begin{pmatrix} u_0 - u_2 & v_0 - v_2 \\ u_1 - u_2 & v_1 - v_2 \end{pmatrix}^{-1} \begin{pmatrix} V_0 - V(2) \\ V(1) - V(2) \end{pmatrix}$$

⟨*Compute triangle partial derivatives*⟩≡
```
Vector dPdu, dPdv;
Float uvs[3][2];
GetUVs(uvs);
⟨Compute deltas for triangle partial derivatives⟩
Float determinant = du1 * dv2 - dv1 * du2;
if (determinant == 0) {
    ⟨Handle zero determinant for triangle partial derivative matrix⟩
}
else {
    Float invdet = 1.f / determinant;
    dPdu = Vector((dx1 * dv2 - dv1 * dx2) * invdet,
        (dy1 * dv2 - dv1 * dy2) * invdet,
        (dz1 * dv2 - dv1 * dz2) * invdet);
    dPdv = Vector((du1 * dx2 - dx1 * du2) * invdet,
        (du1 * dy2 - dy1 * du2) * invdet,
        (du1 * dz2 - dz1 * du2) * invdet);
}
```

CoordinateSystem 21
Cross 20
GetUVs 78
Hat 19
Vector 16

⟨*Compute deltas for triangle partial derivatives*⟩≡
```
Float du1 = uvs[1][0] - uvs[0][0];
Float du2 = uvs[2][0] - uvs[0][0];
Float dv1 = uvs[1][1] - uvs[0][1];
Float dv2 = uvs[2][1] - uvs[0][1];
Float dx1 = p2.x - p1.x;
Float dx2 = p3.x - p1.x;
Float dy1 = p2.y - p1.y;
Float dy2 = p3.y - p1.y;
Float dz1 = p2.z - p1.z;
Float dz2 = p3.z - p1.z;
```

Just do something arbitrary... Make sure they are all orthonormal, though..

⟨*Handle zero determinant for triangle partial derivative matrix*⟩≡
```
CoordinateSystem(Cross(E2, E1).Hat(), &dPdu, &dPdv);
```

To compute the (u,v) parametric coordinates at the hit point, we just apply the barycentric interpolation formula to the (u,v) parametric coordinates at the vertices.

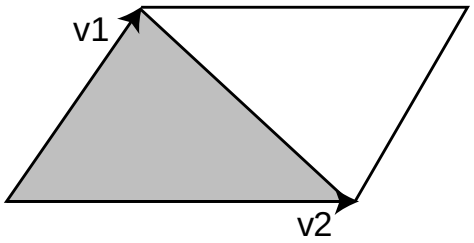


Figure 3.6: The area of a triangle with two edges given by vectors v_1 and v_2 is one half of the area of the parallelogram, which is given by the length of the cross product of v_1 and v_2.

⟨*Interpolate* (u, v) *triangle parametric coordinates*⟩≡
```
Float b0 = 1 - b1 - b2;
Float tu = b0*uvs[0][0] + b1*uvs[1][0] + b2*uvs[2][0];
Float tv = b0*uvs[0][1] + b1*uvs[1][1] + b2*uvs[2][1];
```

Surface Area

Recall from Section 2.1 that the area of a parallelogram is given by the length of the cross product of the two vectors along its sides. From this, it's easy to see that given the vectors for two edges of a triangle, its area is $1/2$ of the area of the parallelogram given by those two vectors–see Figure 3.6.

20 Cross
19 Length

⟨*TriangleMesh Methods*⟩+≡
```
Float Triangle::Area() const {
    ⟨Get triangle vertices in p1, p2, and p3⟩
    return 0.5f * Cross(p2-p1, p3-p1).Length();
}
```

3.7 Subdivision Surfaces

⟨*subdiv.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "shapes.h"
#include "paramset.h"
#include "dynload.h"
#include "texture.h"
#include <set>
#include <map>
using std::set;
using std::map;
⟨SubdivisionMesh Macros⟩
⟨SubdivisionMesh Local Structures⟩
⟨SubdivisionMesh Declarations⟩
⟨SubdivisionMesh Inline Functions⟩
⟨SubdivisionMesh Methods⟩
```

Figure 3.7: tetra control mesh and 4 levels of subdivision.

We will wrap up this chapter by defining a shape that implements *subdivision surfaces*, a type of surface that is particularly well-suited to describing complex smooth shapes. A subdivision surface is defined by a mesh of *control vertices*; the surface that results from repeatedly subdividing the faces of the mesh into more faces and applying rules that compute new positions for the mesh vertices based on weighted combinations of vertex positions at the previous level gives the subdivision surface for that mesh.

For appropriately chosen subdivision rules, this process converges to give a smooth *limit surface* as the number of subdivision steps goes to infinity. (In practice, just a few levels of subdivision typically suffice to give a good approximation to the limit surface.) Figure 3.7 shows the effect of applying one set of subdivision rules to a tetrahedron; on the left is the original control mesh–one, two, three, and four levels of subdivision are shown moving from left to right.

Though originally developed in the 1970s, subdivision surfaces have receitnly received a fair amount of attention in computer graphics thanks to their advantages over polygonal and spline-based representations of surfaces. The advantages of subdivision include:

- Subdivision surfaces are smooth (as opposed to polygon meshes, which appear faceted when viewed sufficiently closely, regardless of how finely they are modeled); subdivision surface are a generally compact way to represent smooth surfaces.
- The classic toolbox of techniques for modeling polygon meshes can be applied to modeling subdivision control meshes–a lot of existing infrastructure in modeling systems can be retargeted to subdivision.
- Subdivision methods are often generalizations of spline-based surface representations, so spline surfaces can often just be run through general subdivision surface renderers.
- Subdivision surfaces naturally describe objects with complex topology, since control meshes with complex topology can be modeled. Parametric surface models generally don't handle complex topology well.
- It is easy to add detail to a localized region of a subdivision surface, simply

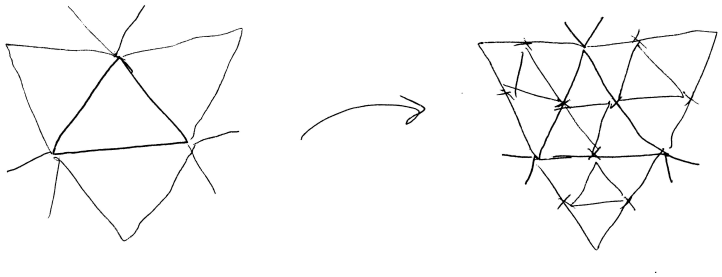


Figure 3.8: Basic refinement process for Loop subdivision: the control mesh on the left has been subdivided once to create the new mesh on the right. Each triangular face of the mesh has been subdivided into four new faces by splitting each of the edges and connecting the new vertices with new edges.

by adding faces to appropriate parts of the control mesh. This is much less easily done with spline representations.

Here, we will describe an implementation of *Loop subdivision surfaces*. The Loop rules are based on triangular faces in the control mesh; faces with more than three vertices are just triangulated at the start. At each subdivision step, each face splits into four child faces– see Figure 3.8. New vertices are added along all of the edges of the original mesh; their positions are computed with weighted averages of nearby vertices. Furthermore, the position of each vertex in the previous step is also updated with a weighted average of its previous position and its neighbors' positions.

Mesh Representation

⟨*SubdivisionMesh Declarations*⟩≡
```
class LoopSubdiv : public Shape {
public:
    ⟨LoopSubdiv Method Declarations⟩
private:
    ⟨LoopSubdiv Private Methods⟩
    ⟨LoopSubdiv Private Data⟩
};
```

We will start by describing the data structures used for representing the subdivision mesh; they need to be carefully designed in order to support all of the operations necessary to cleanly implement the subdivision algorithm. The parameters to the `LoopSubdiv` constructor specify a triangle mesh in exactly the same format as is passed to the `TriangleMesh` constructor (see Section 3.6 on page 74.): each face is described by three integer vertex indices, giving offsets into the vertex array `P` for the face's three vertices. We will need to process this data to compute a representation of which faces are adjacent to which other faces, which faces are adjacent to each vertex, etc., in order to implement subdivision efficiently.

⟨*SubdivisionMesh Methods*⟩≡
```
LoopSubdiv::LoopSubdiv(const Transform &o2w, int nfaces,
        int nvertices, const int *vertexIndices,
        const Point *P, int nl)
    : Shape(o2w) {
    nLevels = nl;
    ⟨Allocate LoopSubdiv vertices and faces⟩
    ⟨Set face to vertex pointers⟩
    ⟨Set neighbor pointers in faces⟩
    ⟨Finish vertex initialization⟩
}
```

We will shortly define `SDVertex` and `SDFace` structures, which hold data for vertices and faces in the subdivision mesh, respectively. We start by allocating one instance of the `SDVertex` class for each vertex in the mesh and an `SDFace` for each face. For now, these are mostly uninitialized, except for the position stored in each vertex.

⟨*Allocate* `LoopSubdiv` *vertices and faces*⟩≡
```
int i;
SDVertex *verts = new SDVertex[nvertices];
for (i = 0; i < nvertices; ++i) {
    verts[i] = SDVertex(P[i]);
    vertices.push_back(&verts[i]);
}
SDFace *fs = new SDFace[nfaces];
for (i = 0; i < nfaces; ++i)
    faces.push_back(&fs[i]);
```


LoopSubdiv	85
Point	21
push_back	494
SDFace	87
SDVertex	87
Transform	32

The `LoopSubdiv` destructor, which we won't include here, just deletes all of the faces and vertices allocated above.

⟨*LoopSubdiv Private Data*⟩≡
```
int nLevels;
vector<SDVertex *> vertices;
vector<SDFace *> faces;
```

The Loop subdivision scheme, like most other subdivision schemes, assumes that the control mesh is *manifold*; no more than two faces share any given edge. Such a mesh may be closed or open: a *closed mesh* has no boundary–all faces have other faces adjacent to them across all of their edges. An open mesh has some faces that do not have all three neighbors. The `LoopSubdiv` implementation supports both closed and open meshes.

It can be shown that in the interior of a triangle mesh, most vertices are adjacent to six faces and have six neighbor vertices directly connected to them with edges. On the boundaries of a non-closed mesh, most vertices are adjacent to three faces and four vertices. The number of vertices directly adjacent to a vertex is called the vertex's *valence*. Interior vertices with valence other than six, or boundary vertices with valence other than four are called *extraordinary vertices*; otherwise they are called *regular*. Loop subdivision surfaces are smooth everywhere except at their extraordinary vertices.

Each vertex stores its position `P`, a boolean that records if it's a regular or extraordinary vertex, and a boolean that records if it lies on the boundary of the mesh. It also holds a pointer to one of the faces adjacent to it; later, we will be able to use this pointer to start an iteration over all of the faces adjacent to the vertex by following pointers that faces store to record which faces are adjacent to them. Finally, we have a pointer to store the new `SDVertex` for this vertex at the next level of subdivision, if any.

⟨*SubdivisionMesh Local Structures*⟩+≡
```
struct SDVertex {
    ⟨SDVertex Constructor⟩
    ⟨SDVertex Methods⟩
    Point P;
    SDFace *startFace;
    SDVertex *child;
    bool regular, boundary;
};
```

The constructor for `SDVertex` does the obvious initialization; we won't include it here.

The `SDFace` structure is were we maintain most of the topological information about the mesh. Because all faces are triangular, we always store three pointers to the vertices for this face and three pointers to the faces adjacent to this one. (The face neighbor pointers may be `NULL`.)

21 `Point`

The face neighbor pointers are indexed such that if we label the edge from `v[i]` to `v[(i+1)%3]` as the *i*th edge, then the neighbor face across that edge is stored in `f[i]`–see Figure 3.9. This labeling convention is important to keep in mind; later when we are updating the topology of a newly subdivided mesh, we will make extensive use of it to navigate around the mesh. Similarly to the `SDVertex` class, we also store pointers to child faces at the next level of subdivision.

⟨*SubdivisionMesh Local Structures*⟩+≡
```
struct SDFace {
    ⟨SDFace Constructor⟩
    ⟨SDFace Methods⟩
    SDVertex *v[3];
    SDFace *f[3];
    SDFace *children[4];
};
```

The `SDFace` constructor is similarly straightforward–setting pointers to `NULL`, etc.–so we will also elide it here.

In order to simplify navigation of the `SDFace` data structure, we'll provide macros that make it easy to determine the vertex and face indices after or before a particular index. These macros add appropriate offsets and compute the result modulus three to handle cycling around properly. Rather than subtracting 1 and taking the modulus for `PREV`, we add 2, which avoids taking the modulus of a negative number, the result of which isn't well-defined in C++.

⟨*SubdivisionMesh Macros*⟩≡
```
#define NEXT(i) (((i)+1)%3)
#define PREV(i) (((i)+2)%3)
```

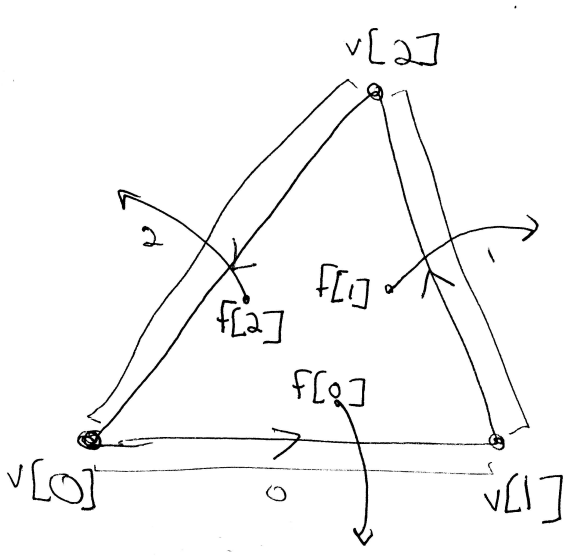


Figure 3.9: Each triangular faces stores three pointers to `SDVertex` objects `v[i]` and three pointers to neighboring faces `f[i]`. Neighbor faces are indexed using the convention that the *i*th edge is the edge from `v[i]` to `v[(i+1)%3]` such that the neighbor across the *i*th edge is in `f[i]`.

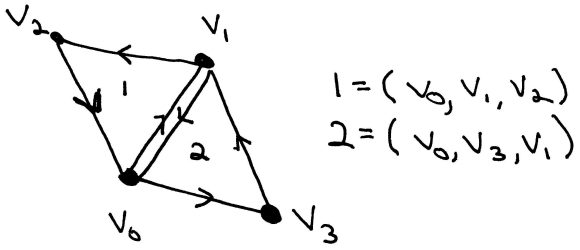


Figure 3.10: All of the faces in the input mesh must be specified so that each shared edge is given once in one direction and the other time in the other direction. Here, the edge from v_0 to v_1 is traversed from v_0 to v_1 by face number one, and from v_1 to v_0 by face number two. Another way to think of this is in terms of face orientation: all faces' vertices should be given consisdently in either clockwise or counter-clockwise order, as seen from outside the mesh.

In addition to requiring a manifold mesh, the `LoopSubdiv` class expects that the control mesh specified by the user will be well-behaved in one additional important way: the mesh must be *consistently ordered*–each *directed edge* in the mesh can be present only once. Consider two vertices, v_0 and v_1, with an edge between then. For the edge from v_0 to v_1, we expect that one of the triangular faces that has that edge will specify its three vertices so that v_0 is before v_1, and that the other face will specify its vertices so that v_1 is before v_0–Figure 3.10. Thus, no more than two faces may have this edge. A Möbius strip is an example of a surface that cannot be consistently ordered. In practice, however, this requirement is rarely troublesome.

Given this assumption about the input data, we will initialize this mesh's topological data structures. We'll first loop over all of the faces and set their `v` pointers to point to their adjacent vertices. This is just simple indexing from the vertex indices passed in to describe each face. We also set each vertex's `startFace` pointer to point to one of the faces adjacent to it. It doesn't matter which of its incident faces we point to, so we just keep re-setting it each time we come across another face that it is incident to, ensuring that all vertices have some non-NULL face pointer by the time we're done.

⟨*Set face to vertex pointers*⟩≡
```
const int *vp = vertexIndices;
for (i = 0; i < nfaces; ++i) {
    SDFace *f = faces[i];
    for (int j = 0; j < 3; ++j) {
        SDVertex *v = vertices[vp[j]];
        f->v[j] = v;
        v->startFace = f;
    }
    vp += 3;
}
```

87 SDFace
87 SDVertex
87 startFace

Now we need to set the face neighbor pointers for each face. This is a bit trickier, since face adjacency information isn't directly given in the mesh specification from the user. We'll loop over the faces and store a `SDEdge` object for each of their three edges; when we come to another face that shares the same edge, we can update both faces' neighbor pointers for that edge.

⟨*SubdivisionMesh Local Structures*⟩+≡
```
struct SDEdge {
    ⟨SDEdge Constructor⟩
    ⟨SDEdge Comparison Function⟩
    SDVertex *v[2];
    SDFace *f[2];
    SDFace **fptr;
};
```

The constructor takes pointers to the two vertices at each end of the edge. It orders them so that `v[0]` holds the one that is first in memory; this way we properly recognize that the edge (v_a, v_b) is the same as the edge (v_b, v_a), regardless of the order the vertices are given in.

⟨*SDEdge Constructor*⟩≡
```
SDEdge(SDVertex *v0 = NULL, SDVertex *v1 = NULL) {
    v[0] = min(v0, v1);
    v[1] = max(v0, v1);
    f[0] = f[1] = NULL;
    fptr = NULL;
}
```

We also define an ordering operation for edges so that we can store `SDEdge`s in data structures that depend on being able to compute an ordering for them.

⟨*SDEdge Comparison Function*⟩≡
```
bool operator<(const SDEdge &e2) const {
    if (v[0] == e2.v[0]) return v[1] < e2.v[1];
    return v[0] < e2.v[0];
}
```

Now we can get to work, looping over the edges in all of the faces and updating the neighbor pointers as we go. We use an STL `set<>` to store the edges where we're still looking for the face on the other side of it; the `set<>` uses the comparison function above to provide $O(\log n)$ searches in the `SDEdge`s.

max 513
min 513
SDEdge 89
SDFace 87
SDVertex 87

⟨*Set neighbor pointers in* `faces`⟩≡
```
set<SDEdge> edges;
for (i = 0; i < nfaces; ++i) {
    SDFace *f = faces[i];
    for (int edge = 0; edge < 3; ++edge) {
        ⟨Update neighbor pointer for edge⟩
    }
}
```

For each edge in each face, we create an edge object and see if the same edge was seen previously. If so, we initialize both faces' neighbor pointers across the edge. If not, we add the edge to the set of edges.

⟨*Update neighbor pointer for* `edge`⟩≡
```
int v0 = edge, v1 = NEXT(edge);
SDEdge e(f->v[v0], f->v[v1]);
if (edges.find(e) == edges.end()) {
    ⟨Handle new edge⟩
}
else {
    ⟨Handle previously-seen edge⟩
}
```

Given an edge that we haven't seen before, we store the current face's pointer in the edge object's `f[0]` member. When we come across the other face that shares this edge (if any), we can thus know what the neighboring face is. Also, so that the current face's neighbor pointer can be set to point to the other, not-yet-found face, we store a pointer to the relevant neighbor pointer in the edge as well.

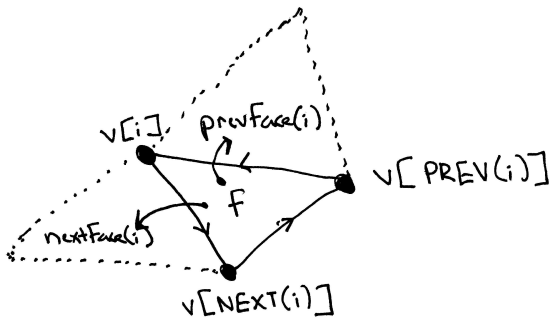


Figure 3.11: Given a vertex `v[i]` and a face that it is incident to, `f`, we define the *next face* as the face adjacent to `f` across the edge from `v[i]` to `v[NEXT(i)]`. The previous face is defined analogously.

⟨*Handle new edge*⟩≡
```
e.f[0] = f;
e.fptr = &(f->f[edge]);
edges.insert(e);
```

After the other edge is found, we can set the neighbor pointers for each of the two faces. We then remove the edge from the edge set, since we are assuming assume that no edge is shared by more than two faces.

⟨*Handle previously-seen edge*⟩≡
```
e = *edges.find(e);
*e.fptr = f;
f->f[edge] = e.f[0];
edges.erase(e);
```

Now that all faces have proper neighbor pointers, we can set the `boundary` and `regular` flags in each of the vertices. In order to deterime if a vertex is a boundary vertex, we'll introduce the idea of an ordering of faces around a vertex; see Figure 3.11. For a vertex `v[i]` on a face `f`, we define the vertex's *next face* as the face across the edge from `v[i]` to `v[NEXT(i)]` and the *previous face* as the face across the edge from `v[PREV(i)]` to `v[i]`.

By successively going to the next face around `v`, `f=f->nextFace(v)`, we can iterate over the faces adjacent to it. If we eventually return to the face we started at, then we are at an interior vertex; if we come to an edge with a `NULL` neighbor pointer, then we're at a boundary vertex–see Figure 3.12. Once we've determined if we have a boundary vertex, we compute to valence of the vertex and set the `regular` flag if the valence is 6 for an interior vertex or 4 for a boundary vertex.

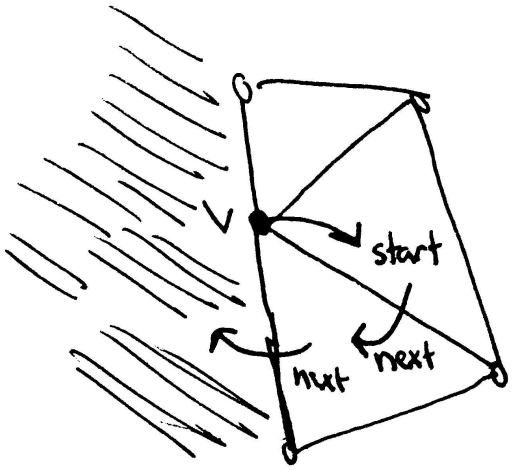


Figure 3.12: We can determine if a vertex is a boundry vertex by starting from the adjacent face `startFace` and following next face pointers around the vertex. If we come to a face that has no next neighbor face, then the vertex is on a boundary. If we return to `startFace`, it's an interior vertex.

boundary 87
regular 87
SDFace 87
SDVertex 87
startFace 87
valence 93
vnum 93

⟨*Finish vertex initialization*⟩≡
```
for (i = 0; i < nvertices; ++i) {
    SDVertex *v = vertices[i];
    SDFace *f = v->startFace;
    do {
        f = f->nextFace(v);
    } while (f && f != v->startFace);
    v->boundary = (f == NULL);
    v->regular = (!v->boundary && (v->valence() == 6) ||
                  (v->boundary && (v->valence() == 4)));
}
```

Since the next face for a vertex `v` on a face `f` is over the *i*th edge, where *i* is the vertex index such that `f->v[i]==v` (recall Figure 3.11 and the mapping of edge neighbor pointers, Figure 3.9), we can find the appropriate face neighbor pointer easily given he index for the vertex, which the `vnum()` utility function provides. Since the previous face is across the edge from `PREV(i)` to `i`, we return `f[PREV(i)]` for the previous face.

⟨*SDFace Methods*⟩≡
```
SDFace *nextFace(SDVertex *vert) {
    return f[vnum(vert)];
}
```

⟨*SDFace Methods*⟩+≡
```
SDFace *prevFace(SDVertex *vert) {
    return f[PREV(vnum(vert))];
}
```

Here is the utility function that finds which vertex number a given vertex is on

one of the faces adjacent to it. It's a fatal error to pass a pointer to a vertex that isn't one of the vertices of the given face–this case would represent a bug elsewhere in the subdivision code.

⟨*SDFace Methods*⟩+≡
```
int vnum(SDVertex *vert) const {
    for (int i = 0; i < 3; ++i)
        if (v[i] == vert) return i;
    Assert(1 == 0);
    return -1;
}
```

⟨*SubdivisionMesh Inline Functions*⟩≡
```
inline int SDVertex::valence() {
    SDFace *f = startFace;
    if (!boundary) {
        ⟨Compute valence of interior vertex⟩
    }
    else {
        ⟨Compute valence of boundary vertex⟩
    }
}
```

498	Assert
87	boundary
92	nextFace
92	prevFace
87	SDFace
87	SDVertex
87	startFace

To compute the valence of a non-boundary vertex, we count the number of of the adjacent faces to the vertex by following neighbor pointers for the faces around it until we reach the original face we started at. The valence is equal to as the number of faces visited.

⟨*Compute valence of interior vertex*⟩≡
```
int nf = 1;
while ((f = f->nextFace(this)) != startFace)
    ++nf;
return nf;
```

For boundary vertices we use the same approach, though in this case, the valence is one more than the number of adjacent faces. The loop over adjacent faces is slightly more complicated here: we follow pointers to the next face around the vertex until we reach the boundary, counting the number of faces seen. We then start again at `startFace` and follow previous face pointers until we hit the boundary going the other way.

⟨*Compute valence of boundary vertex*⟩≡
```
int nf = 1;
while ((f = f->nextFace(this)) != NULL)
    ++nf;
f = startFace;
while ((f = f->prevFace(this)) != NULL)
    ++nf;
return nf+1;
```

Bounds

Loop subdivision surfaces have the *convex hull property*: the limit surface is guaranteed to be inside the convex hull of the original control mesh. Thus, for the bounding methods, we can just bound the original control vertices. The bounding methods are essentially equivalent to those in `TriangleMesh`, so we won't include them here.

⟨*LoopSubdiv Method Declarations*⟩+≡
```
BBox ObjectBound() const;
BBox WorldBound() const;
```

Subdivison

Now we can show how subdivision proceeds with the Loop rules. The `LoopSubdiv` shape doesn't support intersection directly, but will apply subdivision a fixed number of times to generate a `TriangleMesh` for rendering. An exercise at the end of the chapter discussed how adaptive subdivision might be implemented, such that each original face is subdivided just enough so that the result looks smooth from the particular viewpoint.

⟨*SubdivisionMesh Methods*⟩+≡
```
bool LoopSubdiv::CanIntersect() const {
    return false;
}
```

LoopSubdiv 85
MemoryArena 505
RefinedShape 54
SDFace 87
SDVertex 87

The `Refine` method handles all of the subdivision. We repeatedly apply the subdivision rules to the mesh, each time generating a new mesh to be used as the input to the next step. After each subdivision step, the `f` and `v` arrays in the `Refine` function below will be updated to point to the faces and vertices from the level of subdivision just computed. After we are done subdividing, a `TriangleMesh` representation of the surface will be created and returned to the caller.

⟨*SubdivisionMesh Methods*⟩+≡
```
void LoopSubdiv::Refine(vector<RefinedShape> &refined) const {
    vector<SDFace *> f = faces;
    vector<SDVertex *> v = vertices;
    MemoryArena<SDVertex> vertexArena;
    MemoryArena<SDFace> faceArena;
    u_int i;
    for (i = 0; i < nLevels; ++i) {
        ⟨Update f and v for next level of subdivision⟩
    }
    ⟨Push vertices to limit surface⟩
    ⟨Compute vertex tangents on limit surface⟩
    ⟨Create TriangleMesh from subdivision mesh⟩
}
```

Here are the contents the main loop of a subdivision step. We create `vector`s for all of the vertices and faces at this level of subdivision and then proceed to compute new vertex positions and update the topological representation for the refined mesh. Figure 3.13 shows the basic refinement rules for faces in the mesh. Each face is split into four children faces, such that the *i*th child face is next to the *i*th vertex of the input face. Three new vertices need to be computed along the split edges of the face.

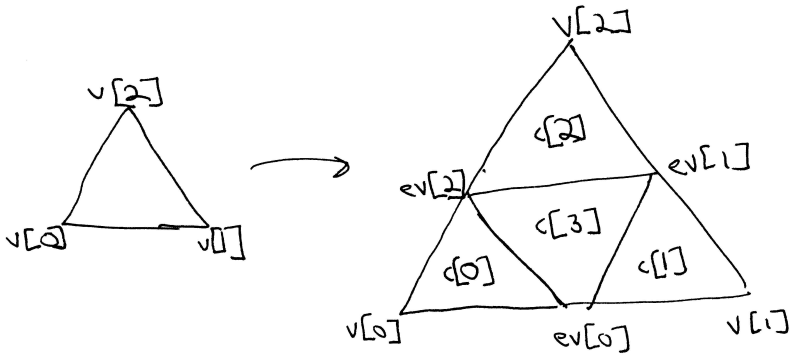


Figure 3.13: Basic Loop subdivision of a single face: four child faces are created, ordered such that the *i*th child face is adjacent to the *i*th vertex of the original face and the third child face is in the center of the subdivided face. Three edge vertices need to be computed; they are numbered so that the *i*th edge vertex is along the *i*th edge of the original face.

⟨*Update* f *and* v *for next level of subdivision*⟩≡
```
vector<SDFace *> newFaces;
vector<SDVertex *> newVertices;
⟨Allocate next level of children in mesh tree⟩
⟨Update vertex positions and create new edge vertices⟩
⟨Update new mesh topology⟩
⟨Prepare for next level of subdivision⟩
```

```
 87 boundary
494 push_back
 87 regular
 87 SDFace
 87 SDVertex
494 size
```

First, we allocate storage for the updated vertices for the vertices in the input mesh and for the subdivided faces at the next level. We don't yet do any initialization of the new vertices and faces, though we do go ahead and set the `regular` and `boundary` flags for the vertices; subdivision leaves boundary vertices on the boundary and interior vertices in the interior. Furthermore, it doesn't change the valence of vertices in the mesh.

⟨*Allocate next level of children in mesh tree*⟩≡
```
u_int j;
for (j = 0; j < v.size(); ++j) {
    v[j]->child = new (vertexArena) SDVertex;
    v[j]->child->regular = v[j]->regular;
    v[j]->child->boundary = v[j]->boundary;
    newVertices.push_back(v[j]->child);
}
for (j = 0; j < f.size(); ++j)
    for (int k = 0; k < 4; ++k) {
        f[j]->children[k] = new (faceArena) SDFace;
        newFaces.push_back(f[j]->children[k]);
    }
```

Computing new vertex positions

Before we worry about the topology of the subdivided mesh, we compute positions for all of the vertices in the mesh. First, we will consider the problem of computing updated positions for all of the vertices that were present in the mesh after the previous subdivision step; these vertices are called *even vertices*. We will then compute the new vertices for the split edges–these are called *odd vertices*.

⟨*Update vertex positions and create new edge vertices*⟩≡
⟨*Update vertex positions for even vertices*⟩
⟨*Compute new odd edge vertices*⟩

Different techniques are used to compute the updated positions for each of the different types of even vertices–regular and extraordinary, boundary and interior. The cross product of these two possibilities gives us four cases to handle.

⟨*Update vertex positions for even vertices*⟩≡
```
for (j = 0; j < v.size(); ++j) {
    if (!v[j]->boundary) {
        ⟨Apply one-ring rule for even vertex⟩
    }
    else {
        ⟨Apply boundary rule for even vertex⟩
    }
}
```

boundary 87
size 494

For both types interior vertices, we take the set of vertices adjacent to each vertex (called the *one-ring* around it, reflecting the fact that it's a ring of neighbors) and weight each of them by a weight β. (See Figure 3.14.) The vertex we are updating, in the center, is weighted by $1-n\beta$, where n is the valence of the vertex. Thus, the new position v' for a vertex v is:

$$v' = (1-n\beta)v + \sum_{i=1}^{N} \beta v_i.$$

This formulation ensures that the sum of weights is one, which is what guarantees the convex hull property we used above for bounding the surface. The fact that only vertices nearby the a vertex being updated affect its new position is called *local support*; Loop subdivision is particularly efficient to implement since its subdivision rules all have this property.

The particular weight β used for this step is a key component of the subdivision method: it must be chosen carefully in order to ensure smoothness of the limit surface among other desirable properties. In the Loop scheme, for regular interior vertices, a β value of $1/16$ should be used; for extraordinary interior vertices, the `beta` function below computes a value based on the vertex's valence that ensures smoothness.

⟨*Apply one-ring rule for even vertex*⟩≡
```
if (v[j]->regular)
    v[j]->child->P = weightOneRing(v[j], 1.f/16.f);
else
    v[j]->child->P = weightOneRing(v[j], beta(v[j]->valence()));
```

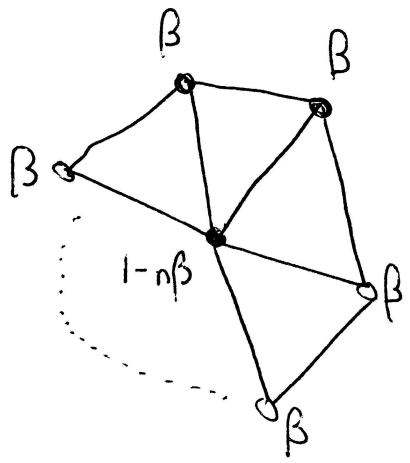


Figure 3.14: The new position v' for a vertex v is computed by weighting the adjacent vertices v_i by a weight β and weighting v by $(1-n\beta)$, such that $v' = (1-n\beta)v + \sum_i \beta v_i$, where n is the valence of v. The adjacent vertices v_i are collectively referred to as the *one ring* around v.

The `beta()` function computes the weight β to use to weight the neighbors of an extraordinary vertex with given valence. Note that the implementation below returns $\beta = 1/16$ for regular vertices, though we only call it for extraordinary vertices.

495 alloca
85 LoopSubdiv
98 oneRing
21 Point
87 regular
87 SDVertex
93 valence

⟨*LoopSubdiv Private Methods*⟩≡
```
static Float beta(int val) {
    if (val == 3) return 3.f/16.f;
    else return 3.f / (8.f * val);
}
```

The `weightOneRing` function loops over the one-ring of vertices adjacent to a given vertex and applies the given weight to compute a new vertex position. It uses the `oneRing` function, defined below, which returns the positions of the vertices around the vertex `vert`.

⟨*SubdivisionMesh Methods*⟩+≡
```
Point LoopSubdiv::weightOneRing(SDVertex *vert, Float beta) {
    ⟨Put vert one-ring in Pring⟩
    Point P = (1 - val * beta) * vert->P;
    for (int i = 0; i < val; ++i)
        P += beta * Pring[i];
    return P;
}
```

⟨*Put* `vert` *one-ring in* `Pring`⟩≡
```
int val = vert->valence();
Point *Pring = (Point *)alloca(val * sizeof(Point));
vert->oneRing(Pring);
```

⟨*SubdivisionMesh Methods*⟩+≡
```
void SDVertex::oneRing(Point *P) {
    if (!boundary) {
        ⟨Get one ring vertices for interior vertex⟩
    }
    else {
        ⟨Get one ring vertices for boundary vertex⟩
    }
}
```

It's relatively easy to get the one-ring around an interior vertex: we loop over the faces adjacent to the vertex, and for each one, grab the next vertex around the face from the center vertex.

⟨*Get one ring vertices for interior vertex*⟩≡
```
SDFace *face = startFace;
do {
    *P++ = face->nextVert(this)->P;
    face = face->nextFace(this);
} while (face != startFace);
```

boundary 87
nextFace 92
Point 21
prevFace 92
SDFace 87
SDVertex 87
startFace 87
vnum 93

The one-ring around a boundary vertex is a bit more tricky. We will carefully store the one ring in the given `Point` array so that the 0th and `valence-1`st entries are the vertices adjacent to the vertex along the boundary. This requires that we first loop around neighbor faces until we reach a face on the boundary and then loop around the other way, grabbing vertices one by one.

⟨*Get one ring vertices for boundary vertex*⟩≡
```
SDFace *face = startFace, *f2;
while ((f2 = face->nextFace(this)) != NULL)
    face = f2;
*P++ = face->nextVert(this)->P;
do {
    *P++ = face->prevVert(this)->P;
    face = face->prevFace(this);
} while (face != NULL);
```

The `oneRing()` function uses these face, the `nextVert()` and `prevVert()` methods, which return the next and previous vertices around the face, respectively. (See Figure 3.15.)

⟨*SDFace Methods*⟩+≡
```
SDVertex *nextVert(SDVertex *vert) {
    return v[NEXT(vnum(vert))];
}
```

⟨*SDFace Methods*⟩+≡
```
SDVertex *prevVert(SDVertex *vert) {
    return v[PREV(vnum(vert))];
}
```

For vertices on the boundary, the new vertex's position is only based on the two neighboring vertices on the boundary (see Figure 3.16); by not depending on

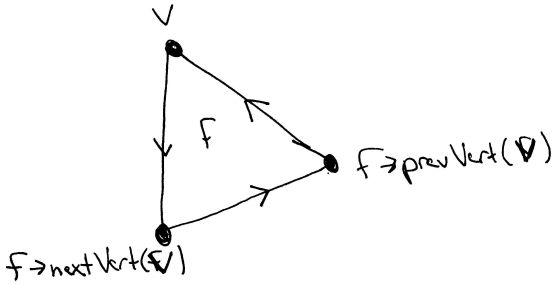


Figure 3.15: Given a vertex `v` on a aface `f`, the method `f->prevVert(v)` returns the previous vertex around the face from `v` and `f->nextVert(v)` returns the next vertex. The ordering of vertices about the faces, as originally specified in the input mesh, determines this ordering.

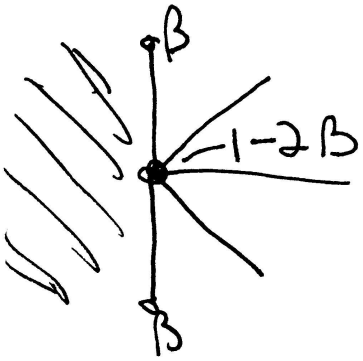


Figure 3.16: Subdivision on a boundary edge: the new position for the vertex in the center is computed by weighting it and its two neighbor vertices by the weights shown.

interior vertices, we ensure that two abutting surfaces that share the same vertices on the boundary will have abutting limit surfaces. The `weightBoundary` utility function applies the given weighting on the two neighbor vertices v_1 and v_2 to compute the new position v' for v as

$$v' = (1-2\beta)v + \beta v_1 + \beta v_2.$$

The same weight, $1/8$, is used for both regular and extraordinary vertices.

⟨*Apply boundary rule for even vertex*⟩≡
```
v[j]->child->P = weightBoundary(v[j], 1.f/8.f);
```

The `weightBoundary()` function applies the given weights at a boundary vertex. Because the `oneRing()` function ordered the boundary vertex's one ring such that the first and last entries are the boundary neighbors, the implementation here is particularly straightforward.

100 `weightBoundary`

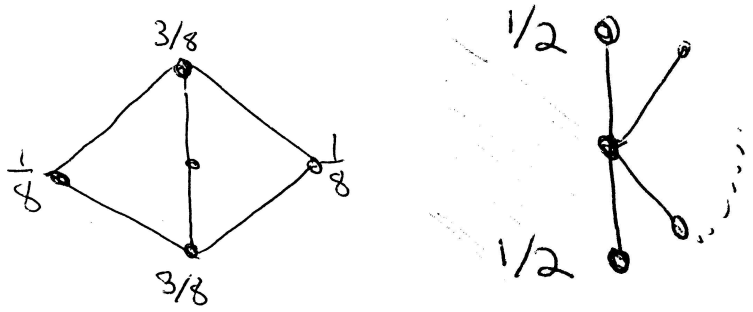


Figure 3.17: Subdivision rule for edge split: the position of the new odd vertex, marked with an "x", is found by weighting the two vertices at the end of the edge and the two vertices opposite it on the adjacent triangles. On the left are the weights for an interior vertex; on the right are the weights for a boundary vertex.

⟨*SubdivisionMesh Methods*⟩+≡
```
Point LoopSubdiv::weightBoundary(SDVertex *vert, Float beta) {
    ⟨Put vert one-ring in Pring⟩
    Point P = (1-2*beta) * vert->P;
    P += beta * Pring[0];
    P += beta * Pring[val-1];
    return P;
}
```

```
      beta  97
LoopSubdiv  85
     Point  21
    SDEdge  89
    SDFace  87
  SDVertex  87
      size  494
```

Now we'll compute the positions of the new odd vertices, the vertices along the split edges of the mesh. We loop over each edge of each face in the mesh, computing the new vertex that splits the edge. Figure 3.17 shows the general setting. For interior edges, the new vertex, marked by an "x", is found by weighting the two vertices as the ends of the edge, v_0 and v_1 and the two vertices across from the edge on the adjacent faces, v_2 and v_3. For each edge on each face, the first time we come to the edge, we compute and store the new odd vertex in the `splitEdges` associative array.

⟨*Compute new odd edge vertices*⟩≡
```
map<SDEdge, SDVertex *> splitEdges;
for (j = 0; j < f.size(); ++j) {
    SDFace *face = f[j];
    for (int k = 0; k < 3; ++k) {
        ⟨Compute odd vertex on kth edge⟩
    }
}
```

As when we were originally setting the neighbor pointers in the faces of the original mesh, we'll create an `SDEdge` object for the edge and see if we've already visited that edge. If we haven't, we compute the new vertex and add it to the `map`. The `map` is an associative array structure that performs efficient lookups.

⟨*Compute odd vertex on* k*th edge*⟩≡
```
SDEdge edge(face->v[k], face->v[NEXT(k)]);
SDVertex *vert = splitEdges[edge];
if (!vert) {
    ⟨Create and initialize new odd vertex⟩
    ⟨Apply edge rules to compute new vertex position⟩
    splitEdges[edge] = vert;
}
```

In Loop subdivision, the new vertices added by subdivision are always regular. (Thus, the number of extraordinary vertices as a fraction of all vertices decreases with each level of subdivision.) This fact lets us immediately initialize the `regular` member of the new vertex. The `boundary` member can be similarly easily initialized by checking to see if there is a neighbor face across the edge that we're splitting. Finally, we'll go ahead and set the vertex's `startFace` pointer here; for all odd vertices on the edges of a face, the inner child face of that face, number three, is guaranteed to be adjacent to the new vertex.

⟨*Create and initialize new odd vertex*⟩≡
```
vert = new (vertexArena) SDVertex;
newVertices.push_back(vert);
vert->regular = true;
vert->boundary = (face->f[k] == NULL);
vert->startFace = face->children[3];
```

87 boundary
102 otherVert
494 push_back
87 regular
89 SDEdge
87 SDVertex
87 startFace

For odd boundary vertices, the new vertex is just the average of the two adjacent vertices. For interior odd vertices, the two vertices at the end of the edge are given weight $3/8$, and the two vertices opposite the edge are given weight $1/8$ (Figure 3.17). We have all the information handy that we need to apply these weights; the `otherVert()` utility function helps out by returning the vertex on a face that is opposite a given edge.

⟨*Apply edge rules to compute new vertex position*⟩≡
```
if (vert->boundary) {
    vert->P =  0.5f * edge.v[0]->P;
    vert->P += 0.5f * edge.v[1]->P;
}
else {
    vert->P =  3.f/8.f * edge.v[0]->P;
    vert->P += 3.f/8.f * edge.v[1]->P;
    vert->P += 1.f/8.f *
        face->otherVert(edge.v[0], edge.v[1])->P;
    vert->P += 1.f/8.f *
        face->f[k]->otherVert(edge.v[0], edge.v[1])->P;
}
```

The `otherVert` function loops through the face's three vertices until it finds the one that isn't equal to either of the two given vertices.

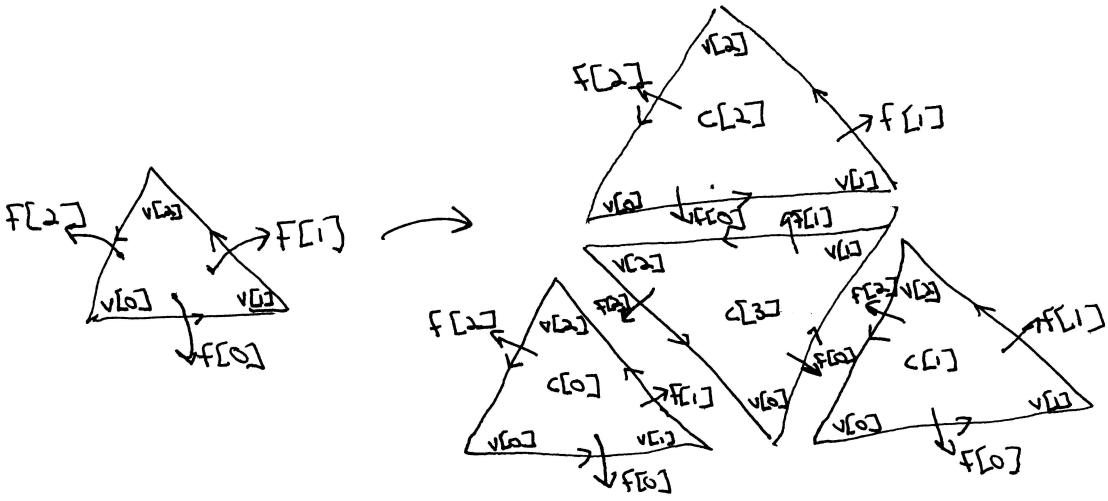


Figure 3.18: Carefully set up the children of the subdivided face...

Assert 498
SDVertex 87

⟨*SDFace Methods*⟩+≡

```
SDVertex *otherVert(SDVertex *v0, SDVertex *v1) {
    for (int i = 0; i < 3; ++i)
        if (v[i] != v0 && v[i] != v1)
            return v[i];
    Assert(1 == 0);
    return NULL;
}
```

Updating mesh topology

In order to keep the details of the topology update as straightforward as possible, the numbering scheme for the subdivided faces and their vertices has been chosen carefully–see Figure 3.18 for a summary. Each face is split int four child faces, such that the *i*th child is adjacent to the *i*th vertex of the original face, and such that the *i*th child face's *i*th vertex is the child of the *i*th vertex of the original face. The vertices of the center child are oriented such that the *i*th vertex is the odd vertex along the *i*th edge of the parent face. Review the figure and re-read this paragraph; these conventions are key to the next few pages.

There are four main tasks to take care of in order to update the topological pointers of the refined mesh:

1. The new even vertices need to store a pointer to one of their adjacent faces in `startFace`.
2. Similarly, the odd vertices `startFace` pointers need to be set.
3. The new faces' neighbor `f[i]` pointers need to be initialized.
4. The new faces' `v[i]` pointers need to point to their incident vertices.

We went ahead and set the `startFace` pointers of the odd vertices when we first created them; we'll handle the other three tasks in order here.

⟨*Update new mesh topology*⟩≡
⟨*Update even vertex face pointers*⟩
⟨*Update face neighbor pointers*⟩
⟨*Update face vertex pointers*⟩

We will first set the `startFace` pointer for the children of the even vertices. Because the vertex and face numbers of the child vertices and faces were carefully chosen, if a vertex is the *i*th vertex of its `startFace`, then it is guaranteed that it will be adjacent to the *i*th child face of `startFace`. Therefore, we just need to loop through the parent vertices of the new even vertices and find their vertex index in their `startFace`.

⟨*Update even vertex face pointers*⟩≡
```
for (j = 0; j < v.size(); ++j) {
    SDVertex *vert = v[j];
    int vertNum = vert->startFace->vnum(vert);
    vert->child->startFace = vert->startFace->children[vertNum];
}
```

Next we update the face neighbor pointers for the newly-created faces. We break this into two steps: one to update neighbors among children of the same parent, and one to do neighbors across children of different parents. This involves some tricky pointer setting.

```
87  SDFace
87  SDVertex
494 size
87  startFace
93  vnum
```

⟨*Update face neighbor pointers*⟩≡
```
for (j = 0; j < f.size(); ++j) {
    SDFace *face = f[j];
    for (int k = 0; k < 3; ++k) {
        ⟨Update children f pointers for siblings⟩
        ⟨Update children f pointers for neighbor children⟩
    }
}
```

First we'll do the easy bits. Recall that the interior child face is always stored in `children[3]`. Furthermore, the $k+1$st child face (for $k = 0, 1, 2$) is across the kth edge of the interior face, and the interior face is across the $k+1$st edge of the kth face.

⟨*Update children* `f` *pointers for siblings*⟩≡
```
face->children[3]->f[k] = face->children[NEXT(k)];
face->children[k]->f[NEXT(k)] = face->children[3];
```

We'll now update the childrens' face neighbor pointers that point to children of the faces adjacent to the parent face. Only the first three children point to children of their parent's neighbors; the interior child's neighbor pointers have already been fully initialized. Inspection of Figure 3.18 reveals that the *i*th and `PREV`(i)th edges of the *i*th child need to be set. We find the vertex of the *i*th vertex's parent on the neighbor (if this isn't a boundary edge); that index is also the face child number on the parent's neighbor that is adjacent to the vertex, which gives us the child over the edge.

⟨*Update children* f *pointers for neighbor children*⟩≡
```
SDFace *f2 = face->f[k];
face->children[k]->f[k] =
    f2 ? f2->children[f2->vnum(face->v[k])] : NULL;
f2 = face->f[PREV(k)];
face->children[k]->f[PREV(k)] =
    f2 ? f2->children[f2->vnum(face->v[k])] : NULL;
```

Finally, we handle the fourth step in the topological updates: setting the face `v[i]` vertex pointers.

⟨*Update face vertex pointers*⟩≡
```
for (j = 0; j < f.size(); ++j) {
    SDFace *face = f[j];
    for (int k = 0; k < 3; ++k) {
        ⟨Update child vertex pointer to new even vertex⟩
        ⟨Update child vertex pointer to new odd vertex⟩
    }
}
```

For the *i*th child face, the *i*th vertex corresponds to the even vertex that is adjacent to it. (For the non-interior children faces, there is one even vertex and two odd vertices; for the interior child face, there are three odd vertices). We can get a pointer to this vertex by following the child pointer of the parent vertex, available from the parent face.

SDEdge 89
SDFace 87
SDVertex 87
size 494
vnum 93

⟨*Update child vertex pointer to new even vertex*⟩≡
```
face->children[k]->v[k] = face->v[k]->child;
```

To update the face vertex pointers to the new odd vertices, we re-use the `splitEdges` associative array to find the odd vertex for each split edge of the parent face. Three child faces have that vertex as an incident vertex. Fortunately, the vertex indices for the three faces are easily found, again based on the numbering scheme established in Figure 3.18.

⟨*Update child vertex pointer to new odd vertex*⟩≡
```
SDVertex *vert = splitEdges[SDEdge(face->v[k], face->v[NEXT(k)])];
face->children[k]->v[NEXT(k)] = vert;
face->children[NEXT(k)]->v[k] = vert;
face->children[3]->v[k] = vert;
```

After the geometric and topological work has been done for a subdivision step, we copy the newly-created vertices and faces into the `v` and `f` arrays, first deleting the old ones, since we no longer need them. We only do these deletions after the first time through the loop, however; the original faces and vertices of the control mesh are left intact.

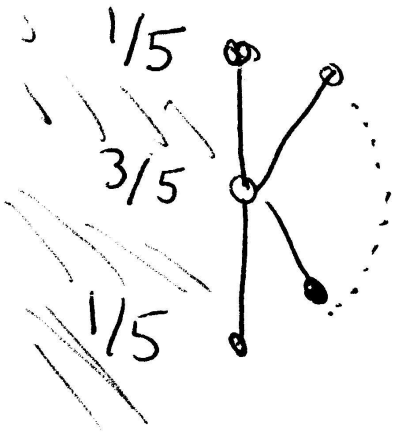


Figure 3.19: To push a boundary vertex onto the limit surface, we apply the weights shown to the vertex and its neighbors along the edge.

⟨*Prepare for next level of subdivision*⟩≡

```
#if 0
if (i != 0) {
    for (u_int j = 0; j < f.size(); ++j)
        delete f[j];
    for (u_int j = 0; j < v.size(); ++j)
        delete v[j];
}
#endif
f = newFaces;
v = newVertices;
```

494 size

To the limit surface and output

One of the remarkable properties of subdivision surfaces is that there are special subdivision rules that let us take the vertices of the mesh and compute the positions they would have if we continued subdividing infinitely. We apply these rules here to initialize an array of limit surface positions, `Plimit`. Note that it's important to store the limit surface positions away somewhere other than in the vertices until all of them have been computed–otherwise we would be incorrectly limit surface positions from previously-processed vertices when applying the limit surface rules for other vertices.

The limit rule for a boundary vertex weights the two neighbor vertices by $1/5$ and the center vertex by $3/5$ (Figure 3.19); the rule for interior vertices is based on a function `gamma()`, which comuptes appropriate vertex weights based on the valence of the vertex.

⟨*Push vertices to limit surface*⟩≡
```
Point *Plimit = new Point[v.size()];
for (i = 0; i < v.size(); ++i) {
    if (v[i]->boundary)
        Plimit[i] = weightBoundary(v[i], 1.f/5.f);
    else
        Plimit[i] = weightOneRing(v[i], gamma(v[i]->valence()));
}
for (i = 0; i < v.size(); ++i)
    v[i]->P = Plimit[i];
```

⟨*LoopSubdiv Private Methods*⟩+≡
```
static Float gamma(int val) {
    return 1.f / (val + 3.f / (8.f * beta(val)));
}
```

In order to generate a smooth-looking triangle mesh with per-vertex surface normals, we'll also compute a pair of non-parallel tangent vectors at each vertex. As with the limit rule for positions, this is also an analytic comuptation that gives the precise tangents on the actual limit surface.

beta 97
boundary 87
Cross 20
Normal 23
Point 21
push_back 494
reserve 494
SDVertex 87
size 494
valence 93
Vector 16
weightBoundary 100
weightOneRing 97

⟨*Compute vertex tangents on limit surface*⟩≡
```
vector<Normal> Ns;
Ns.reserve(v.size());
for (i = 0; i < v.size(); ++i) {
    SDVertex *vert = v[i];
    Vector S(0,0,0), T(0,0,0);
    ⟨Put vert one-ring in Pring⟩
    if (!vert->boundary) {
        ⟨Compute tangents of interior face⟩
    }
    else {
        ⟨Compute tangents of boundary face⟩
    }
    Ns.push_back(Normal(Cross(S, T)));
}
```

Figure 3.20 shows the setting for computing tangents in the mesh interior. The center vertex is given a weight of zero and the neighbors are given weights w_i. To compute the first tangent vector, S, the weights are

$$w_i = \cos\left(\frac{2\pi i}{n}\right),$$

where n is the valence of the vertex. The second tangent, T, is computed with weights

$$w_i = \sin\left(\frac{2\pi i}{n}\right).$$

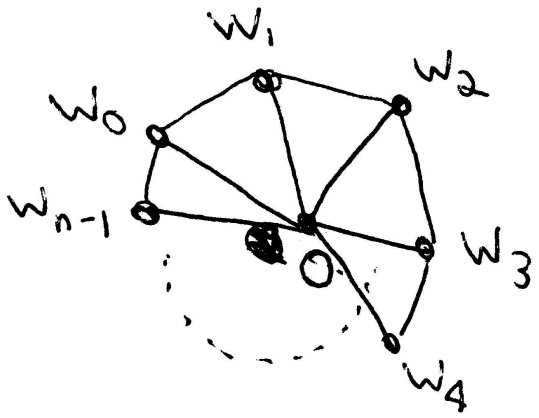


Figure 3.20: To compute tangents for interior vertices, the one-ring vertices are weighted with weights w_i. The center vertex, where the tangent is being computed, always has a weight of 0.

⟨*Compute tangents of interior face*⟩≡

```
for (int k = 0; k < val; ++k) {
    S += cosf(2.f*M_PI*k/val) * Vector(Pring[k]);
    T += sinf(2.f*M_PI*k/val) * Vector(Pring[k]);
}
```

16 Vector

Tangents on boundary vertices are a bit trickier; Figure 3.21 shows the expected ordering of vertices in the one ring that we'll assume in the discussion below.

The first tangent, S, known as the *across tangent* is given by the vector between the two neighboring boundary vertices:

$$S = v_{n-1} - v_0.$$

The second tangent, known as the *transverse tangent* is computed differently based on the vertex's valence. The center vertex is given a (possibly zero) weight w_c and the one-ring vertices are given weights specified by a vector $(w_0, w_1, \ldots, w_{n-1})$. The transverse tangent rules we will use are:

valence	w_c	w_i
2	-2	(1, 1)
3	-1	(0,1,0)
4 (regular)	-2	(-1, 2, 2, -1)

For valences of 5 and higher, $w_c = 0$ and

$$\begin{aligned} w_0 &= w_{n-1} = \sin\theta \\ w_i &= (2\cos\theta - 2)\sin(\theta i) \end{aligned}$$

where

$$\theta = \frac{\pi}{n-1}.$$

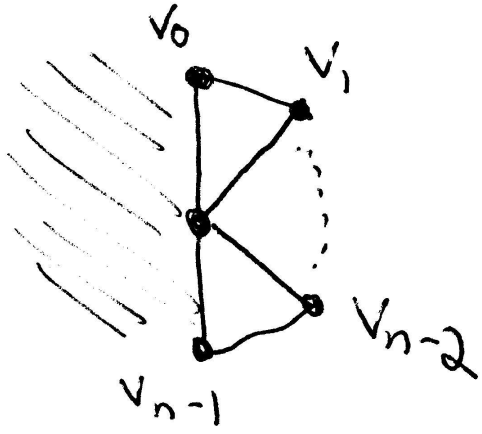


Figure 3.21: Tangents at boundary vertices are also computed as weighted averages of the adjacent vertices. Some of the boundary tangent rules also incorporate the value of the center vertex as well, however.

regular 87
Vector 16

⟨*Compute tangents of boundary face*⟩≡

```
S = Pring[val-1] - Pring[0];
if (val == 2)
    T = Vector(Pring[0] + Pring[1] - 2 * vert->P);
else if (val == 3)
    T = Pring[1] - vert->P;
else if (val == 4) // regular
    T = Vector(-1*Pring[0] + 2*Pring[1] + 2*Pring[2] +
        -1*Pring[3] + -2*vert->P);
else {
    Float theta = M_PI / float(val-1);
    T = Vector(sinf(theta) * (Pring[0] + Pring[val-1]));
    for (int k = 1; k < val-1; ++k) {
        Float wt = (2 * cosf(theta) - 2) * sinf((k) * theta);
        T += Vector(wt * Pring[k]);
    }
    T = -T;
}
```

Finally, the fragment ⟨*Create* `TriangleMesh` *from subdivision mesh*⟩ creates the triangle mesh object and adds it to the `refined` vector passed to the refinement method. We won't include it here, since it's just straightforward transformation of the subdivided mesh into an indexed triangle mesh.

Further Reading

Introduction to Ray Tracing has an extensive survey of algorithms for ray-shape intersection (Gla89a). Heckbert has written a technical report that discusses the mathematics of quadrics for graphics applications in detail, with many citations to literature in mathematics and other fields (Hec84). The ray-triangle intersection test in Section 3.6 was developed by Möller and Trumbore(MT97).

The notion of shapes that repeatedly could refine themselves into collections of other shapes until ready for rendering was first introduced in the REYES renderer (CCC87).

An excellent introduction to differential geometry is Gray's book (Gra93); Section 14.3 of it presents the Weingarten equations. Turkowski's technical report has expressions for first and second derivatives of a handful of parametric primitives (Tur90).

The Loop subdivision method was originally developed by Charles Loop (Loo87). Our implementation here uses improved rules for subdivision and tangents along boundary edges developed by Hoppe et al (HDD+94). There has been extensive work in subdivision recently; the SIGGRAPH course notes give a good summary of the state-of-the-art and also have extensive references (ZSD+00).

Exercises

3.1 One nice property of mesh-based shapes like triangle meshes and subdivision surfaces is that we can transform the shape's vertices into world space, so that it isn't necessary to transform rays into object space before performing ray intersection tests. Interestingly enough, it is possible to do the same thing for ray-quadric intersections.

The implicit forms of the quadrics in this chapter were all of the form

$$Ax^2 + Bxy + Cxz + Dy^2 + Eyz + Fz^2 + G = 0,$$

where some of the constants $A \ldots G$ were zero. More generally, we can define quadric surfaces by the equation

$$Ax^2 + By^2 + Cz^2 + 2Dxy + 2Eyz + 2Fxz + 2Gz + 2Hy + 2Iz + J = 0,$$

(where most of the parameters $A \ldots J$ don't directly correspond to the $A \ldots G$ above.) In this form, the quadric can be represented by a four by four matrix Q:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} A & D & F & G \\ D & B & E & H \\ F & E & C & I \\ G & H & I & J \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = P^T \cdot Q \cdot P = 0$$

Given this representation, first show that the matrix Q' representing a quadric transformed by the matrix M is:

$$Q' = (M^T)^{-1} Q M^{-1}.$$

To do so, show that for any point p where $p^T Q p = 0$, if we apply a transformation M to p to compute $p' = Mp$, we'd like to find Q' so that $(p')^T Q' p' = 0'$.

Next, substitute the ray equation into the more general quadric equation above to compute a, b, and c values for the quadratic equation in terms of entries of the matrix Q to pass to the `Quadratic` function.

Now implement this approach in `lrt` and use it instead of the original quadric intersection routines. Note that you will still need to transform the resulting

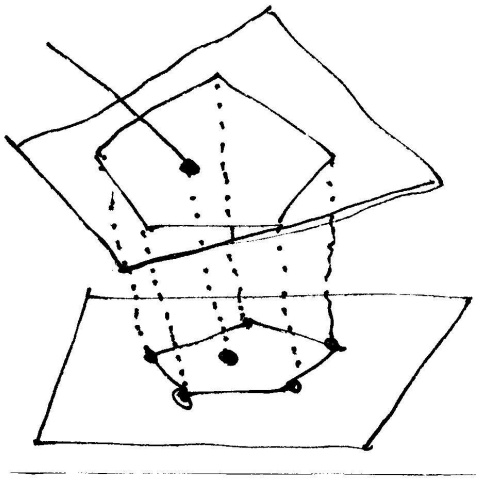


Figure 3.22: Polygon projection onto plane for intersection.

world-space hit points into object space to test against θ_{max}, if it is not 2π, etc. How does performance compare to the original scheme?

3.2 Implement a general polygon primitive. `lrt` currently transforms polygons with more than three vertices into a collection of triangles by XXX. This is actually only correct for convex polygons without holes. Support all kinds of polygons as as first-class primitive. How to compute plane equation from a normal and a point on the plane.... Then intersect ray with the plane the polygon sits in. Project that point and the polygon vertices to 2D. Then apply a 2D point in polygon test; easy one is to essentially ray-trace in 2d–intersect the ray with each of the edge segments, count how many it goes through. If odd number, are inside the polygon and have an intersection. Figure 3.22.

Haines (Hai94).

3.3 subdiv extensions: ”crease”, n integer vertices to specify chain of edges, one float, infinity, giving sharpness. for crease, use boundary subdivision rules along the edges, giving a sharp feature there.

”hole” face property, inherit to children, just don’t output at end

3.4 Implement adaptive subdivision for the subdivision surface `Shape`. A weakness of the basic implementation is that each face is always refined a fixed number of times: this may mean that some faces are under-refined, leading to visible faceting in the triangle mesh, and some faces are over-refined, leading to excessive memory use and rendering time. Instead, stop subdividing faces once a particular error threshold has been reached.

An easy error threshold to implement computes the face normals of each face and its directly adjacent faces. If they are sufficiently close to each other (e.g. as tested via dot products), then the limit surface for that face will be reasonably flat.

The trickiest part of this exercise is that some faces that don't need subdivision due to the flatness test will still need to be subdivided in order to provide vertices so that neighboring faces that do need to subdivide can get their vertex one-rings. In particular, adjacent faces can differ by no more than one level of subdivision.

3.5 Use the triangular face refinement infrastructure from the `LoopSubdiv` shape to implement displacement mapping. Displacement mapping is a technique related to bump mapping, where an offset function is defined over the entire surface. Rather than just adjusting the surface normal as in bump mapping, the actual surface shape is modified by displacement mapping. The usual approach to displacement mapping is to finely tessellate the geometric shape and to then evaluate the displacement function at its vertices, moving each vertex the given distance along its normal.

Because displacement mapping may make the extent of the shape larger, the bounding box of the un-displaced shape will need to be expanded by the maximum displacement distance that a particular displacement function will ever generate.

Refine each face of the mesh until it is roughly the size of a pixel. To do this, you will need to be able to estimate the image pixel-based length of an edge in the scene when it is projected onto the screen. After you have done this, use the texturing infrastructure in Chapter 11 to evaluate displacement functions.

3.6 Ray-tracing point-sampled geometry: Schaufler and Jensen (SJ00)...

3.7 Implicit functions. More general functions, sums of them to define complex surface. Good for molecules, water drops, etc. Introduced by Blinn (Bli82a). Wyvill and Wyvill give new falloff function with a number of advantages (WW89). Kalra and Barr (KB89) and Hart (Har96) give methods for ray-tracing them.

4. Intersection Acceleration

579 primitives

⟨*primitives.cc**⟩≡
⟨*Source Code Copyright*⟩
```
#include "lrt.h"
#include "primitives.h"
#include "light.h"
```
⟨*Primitive Methods*⟩
⟨*GeometricPrimitive Methods*⟩
⟨*PrimitiveList Methods*⟩
⟨*Surf Method Definitions*⟩

⟨*primitives.h**⟩≡
⟨*Source Code Copyright*⟩
```
#ifndef PRIMITIVES_H
#define PRIMITIVES_H
#include "lrt.h"
#include "geometry.h"
#include "shapes.h"
#include "transform.h"
#include "color.h"
#include "materials.h"
```
⟨*Primitive Declarations*⟩
⟨*PrimitiveList Declarations*⟩
```
#endif // PRIMITIVES_H
```

⟨*scene.h**⟩≡
⟨*Source Code Copyright*⟩
```
#ifndef SCENE_H
#define SCENE_H
#include "lrt.h"
#include "primitives.h"
#include "transport.h"
```
⟨*Scene Declarations*⟩
```
#endif // SCENE_H
```

⟨*scene.cc**⟩≡
⟨*Source Code Copyright*⟩
```
#include "scene.h"
#include "camera.h"
#include "film.h"
#include "sampling.h"
#include "dynload.h"
#include "transport.h"
#include "volume.h"
```
⟨*Scene Methods*⟩

film 173
primitives 579

One of the keys to making ray-tracing efficient is having algorithms that reduce the cost of finding intersetions of rays with shapes in the scene. Since a ray throught a scene will generally only intersect a handful of the primitives in it, there is substantial room for improvement compared to naively performing an intersection test with each primitive. A variety of approaches to this problem have been developed; in this chapter, we will describe a number of them and then show the implementation of two: grids and kd-trees.

4.1 Approaches To Reducing Intersections

Given a scene with a million primitives in it, it's clearly quite wasteful to perform one million ray–primitive intersections for each ray traced–the ray will generally be nowhere near most of the primitives, so we should be able to avoid doing most of those intersection tests while still finding any intersections. In the absence of a mechanism to cull the primitives down to a small set of candidates for each ray, ray tracing would be an inordinately expensive algorithm. This section will survey general techniques used to approach this problem.

The `BBox` class that we introduced previously in Section 2.5 is one building block for reducing intersection tests. We can test a ray for intersection with the bounding box of a primitive or collection of geometry first, and only try to find an intersection with the geometry if the ray intersects the box. As long as the bounding box is a good fit for the geometry and computing the actual intersection with the geometry is significantly more expensive than testing the ray against the box, we can save a lot of time in this manner.

Ray-Box Intersections

One way to think of bounding boxes is as the intersection of three slabs. A *slab* is simply the region of space between two parallel planes. To intersect a ray against

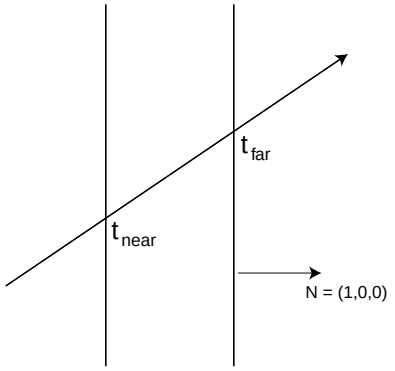


Figure 4.1: Intersecting a ray with a pair of axis-aligned slabs. Each of the slabs shown here is a plane given by $x = c$, for some constant value c. The normal of each slab is $(1,0,0)$.

a box, we intersect the ray against each of the box's three slabs in turn. Because we know that the slabs are aligned with the three coordinate axes, we can make a number of optimizations in a ray-bounding box intersection routine.

XXX is this discussion duplicated in the disk primitive?? XXX

We will first describe the basic geometry of planes and how to compute the intersection point of a ray with a plane. A plane in 3-space can be specified in a number of ways; here, we will define a plane by a point on the plane $\mathbf{p}$ and the plane normal $\hat{\mathbf{n}}$. Given a ray $\mathbf{r}$, we'd like to find the parametric point t along $\mathbf{r}$ that gives the point along $\mathbf{r}$ that lies on the plane. We write an equation that describes the set of points $\mathbf{p}'$ that lie on the plane: this is just is the set of all points such that the vector from $\mathbf{p}$ to $\mathbf{p}'$ is perpendicular to $\hat{\mathbf{n}}$. Because perpendicular vectors have a dot product of zero, we have:

$$((\mathbf{p}' - \mathbf{p}) \cdot \hat{\mathbf{n}}) = 0$$

Thus, given a ray $\mathbf{r}$ defined by $\mathbf{r} = o(\mathbf{r}) + t\vec{\mathbf{d}}(\mathbf{r})$, we substitute the ray equation for $\mathbf{p}'$ to find the point where the ray intersects the plane:

$$((o(\mathbf{r}) + t\vec{\mathbf{d}}(\mathbf{r}) - \mathbf{p}) \cdot \hat{\mathbf{n}}) = 0.$$

Using basic definitions of the dot product, we have

$$\begin{aligned}
((o(\mathbf{r}) - \mathbf{p}) \cdot \hat{\mathbf{n}}) + (t\vec{\mathbf{d}}(\mathbf{r}) \cdot \hat{\mathbf{n}}) &= 0 \\
((o(\mathbf{r}) - \mathbf{p}) \cdot \hat{\mathbf{n}}) + t(\vec{\mathbf{d}}(\mathbf{r}) \cdot \hat{\mathbf{n}}) &= 0 \\
t(\vec{\mathbf{d}}(\mathbf{r}) \cdot \hat{\mathbf{n}}) &= -((o(\mathbf{r}) - \mathbf{p}) \cdot \hat{\mathbf{n}}) \\
t &= -\frac{((o(\mathbf{r}) - \mathbf{p}) \cdot \hat{\mathbf{n}})}{(\vec{\mathbf{d}}(\mathbf{r}) \cdot \hat{\mathbf{n}})}
\end{aligned}$$

As long as $(\vec{\mathbf{d}}(\mathbf{r}) \cdot \hat{\mathbf{n}})$ is not zero (which would indicate that the ray is parallel to the plane), t is defined. If t is less than zero, the ray faces away from the plane and never intersects it. See Figure 4.1 for the basic geometry of the situation.

The basic ray–bounding box intersection algorithm works as follows: we start with a parametric interval that covers that range of positions t along the ray where

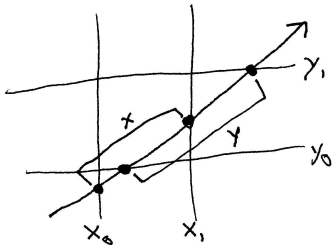


Figure 4.2: Intersecting a ray with an axis-aligned bounding box. We compute intersection points with each pair of slabs, progressively narrowing the pair of intersection points. Here, we see that the intersection of the *x* and *y* extents along the ray gives the extent where the ray is inside the box.

we're interested in finding intersections; typically, this is $[0,\infty)$. We will then successively compute the two parametric positions where the ray intersects each pair of axis-aligned slabs. We successively compute the set-intersection of this interval with our original interval, returning failure if we find that the resulting interval is degenerate, which indicates that there are no points t along the ray where it is between all of the slabs, and thus the ray does not intersect the box. If after checking all three slabs, the interval is non-degenerate, we have the parametric range of the ray that is inside the box. Figure 4.2 illustrates this process.

Ray 26

The routine to compute the intersection is called `IntersectP` . `IntersectP` is a *predicate* function, meaning that its main purpose is to return a boolean value. If the function returns true, the intersection parametric range can be returned in the optional arguments `hitt0`and `hitt1`. Intersections outside of the `mint`/`maxt` range of the ray that is passed in are not considered.

⟨*BBox Method Definitions*⟩+≡
```
bool BBox::IntersectP(const Ray &ray, Float *hitt0,
        Float *hitt1) const {
    ⟨Initialize parametric interval⟩
    ⟨Check X slab⟩
    ⟨Check Y slab⟩
    ⟨Check Z slab⟩
    if (hitt0) *hitt0 = t0;
    if (hitt1) *hitt1 = t1;
    return true;
}
```

⟨*Initialize parametric interval*⟩≡
```
Float t0 = ray.mint, t1 = ray.maxt;
```

For each pair of slabs, we need to compute two ray–plane intersections, giving the parametric t values where the intersections occur. Consider the pair of slabs along the x axis: they are can be described by the two planes through the points $(x_1,0,0)$ and $(x_2,0,0)$, each with normal $(1,0,0)$. We need to compute two t values, one for each plane. Consider the first one, t_1. From the ray–plane equation above, we have:

$$t_1 = -\frac{((o(\mathbf{r}) - (x_1,0,0)) \cdot (1,0,0))}{(\vec{\mathbf{d}}(\mathbf{r}) \cdot (1,0,0))}$$

Because the *y* and *z* components of the normal are zero, we can use the definition of the dot product to simplify this substantially:

$$t_1 = -\frac{o(\mathbf{r_x}) - x_1}{\vec{\mathbf{d}}(\mathbf{r_x})} = \frac{x_1 - o(\mathbf{r_x})}{\vec{\mathbf{d}}(\mathbf{r_x})}$$

The code implementing this equation for the *x* slab is shown here; the code for the *y* and *z* slabs is nearly identical and is omitted. We start by computing the reciprocal of the *x* component of the ray direction. We will then multiply by this factor when we would otherwise divide by the *x* direction component; this saves a potentially-expensive divide. We do not need to verify that the *x* direction component is not zero; if it is, then `invRayDir` will hold an infinite value, either $-\infty$ or ∞[1], and the rest of the algorithm works correctly.

⟨*Check X slab*⟩≡
```
Float invRayDir = 1.f / ray.D.x;
Float tNear = (pMin.x - ray.O.x) * invRayDir;
Float tFar  = (pMax.x - ray.O.x) * invRayDir;
⟨Update parametric interval⟩
```

We then swap the two distances, so that t_{near} holds the closer intersection and t_{far} the farther one. This gives us a parametric range $[t_{\text{near}}, t_{\text{far}}]$. We compute the intersection of this with the current range $[t_0, t_1]$ to compute a new range. If this new range is empty (i.e. $t_0 > t_1$), then we return failure.

513 max
513 min
28 pMax
28 pMin
513 swap

⟨*Update parametric interval*⟩≡
```
if (tNear > tFar) swap(tNear, tFar);
t0 = max(tNear, t0);
t1 = min(tFar,  t1);
if (t0 > t1) return false;
```

Any shape that can bound a more complex shape and can easily be intersected with a ray can be used in place of axis-aligned bounding boxes. Bounding spheres and oriented bounding boxes, which aren't necessarily aligned with the coordinate axes, are notable examples. Placing a single bounding volume around all of the objects in the scene only helps for simple scenes. For more complex scenes, we need more complex spatial data structures to partition the scene geometry into smaller subsets so that we can only consider the subsets that the ray actually approaches. If we can roughly order these subsets from near to far, all the better: we can stop performing intersection tests once we have found an intersection and know that it's not possible to have any closer intersections.

Regular Grid

The *regular grid* divides a rectangular region of space into equal-sized *voxels* that store references to the primitives that overlap them. (see Figure 4.3). Each Given a ray to trace, we step through each of the voxels that it passes through in turn, checking for intersections only with the primitives in the voxel that the ray is currently in.

[1] This assumes that the architecture being used supports IEEE floating-point arithmetic; this is universal on modern systems. The relevant properties of IEEE floating-point arithmetic are that for all $v_+ > 0$, $v - +/0 = \infty$ and for all $v_- < 0$, $v_-/0 = -\infty$, where ∞ is a special value such that any positive number multiplied by ∞ gives ∞, any negative number multiplied by ∞ gives $-\infty$, etc.

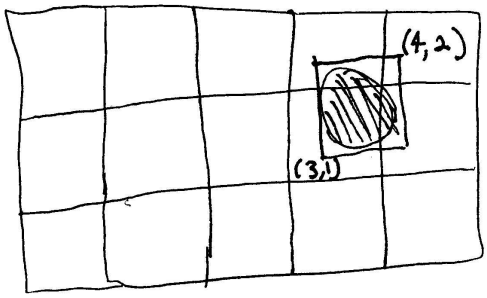


Figure 4.3: References to primitives in the scene (such as the sphere shown here) are stored in all of the voxels that they overlap in the grid. Typically, the primitive's bounding box is used to determine which voxels it overlaps. In this case, the sphere is inaccurately stored in the upper-right voxel since its bounding box overlaps the voxel even though the primitive does not.

The regular grid usually performs reasonably well. It can be initialized from a collection of geometry relatively quickly, and it takes relatively little computation to compute the sequence of voxels that a ray passes through. However, due to this simplicity, it can suffer from performance problems when the data in the scene isn't distributed regularly; if there's a small region of space with a lot of geometry in it such that all of that geometry is in a single voxel, performance suffers greatly when a ray reaches that voxel as many intersection tests are performed. The basic problem is that the data structure doesn't adapt well to the distribution of the data.

Hierarchical bounding volumes

An approach that better adapts to the distribution of geometry in the scene is the *hierarchical bounding volume* (HBV). Given some method of bounding primitives (e.g. axis aligned bounding boxes), a hierarchy of these bounding primitives is built. The top node of the hierarchy encompasses all of the primitives in the scene (see Figure 4.4). It has two or more children nodes, each of which bounds a subset of the scene. This continues recursively until the bottom of the tree, at which point a single primitive is bound. The hierarchical bounding volume is traversed by first intersecting the ray with the top-level bounding volume. If it misses the volume, it cannot possibly intersect any geometry in the scene, so we're done. Otherwise we "open up" that volume and test the ray against the children bounding volumes. For any of those that are hit, the recursion continues throughout the tree. In order to ensure that the primitives are intersected in roughly front-to-back order, a priority queue is often used to sort the sub-volumes that the ray intersects by the parametric distance to the intersection.

HBVs can work well for a wide variety of scenes because they are naturally adaptive to the distribution of primitives. They can be difficult to construct, however, since when they're being built, the algorithm needs to repeatedly partition the primitives into sets and try to simultaneously minimize the amount of overlap between the sets as well as the size of the bounding volumes that encompass groups of geometry.

BSP trees and friends

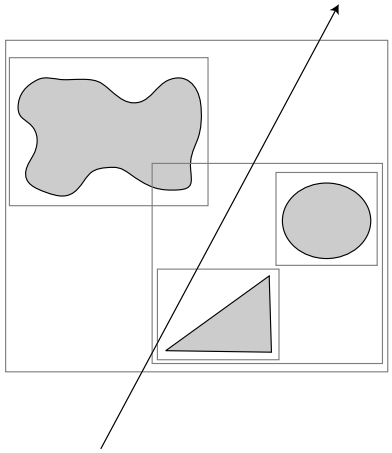


Figure 4.4: A set of primitives are stored in a bounding volume hierarchy. When a ray is being traced, we first see if it intersects the top-level bounding volume. If so, we recursively process the children bounding volumes, continuing on with those that are intersected, until we reach the geometric primitives.

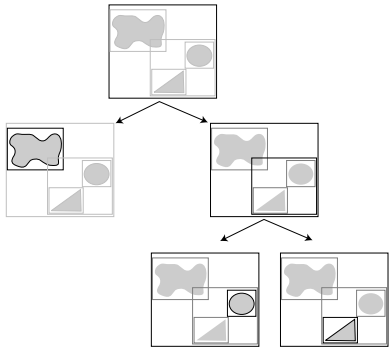


Figure 4.5: Structure of a bounding volume hierarchy. The top node of the tree holds the bounding box of the entire scene and then pointers to children nodes that hold subsets of the scene. This continues recursively until the leaf nodes, which hold pointers to geometric primitives in the scene.

Sitting somewhere between HBVs and grids are *BSP trees*, which adaptively subdivide space, so that they work well for irregular collections of geometry, but in a more constrained fashion, so that they are easier to traverse. A BSP tree starts with a bounding box that encompasses the entire scene. If the number of primitives in the box is greater than some threshold, it is split in half by a plane that separates the bounding box into two pieces. Primitives are then redistributed to either one or both of the halves, depending if they overlap one or both sides of the splitting plane. This process continues recursively until either a small enough number of primitives is in each box or a maximum depth is reached.

Because the BSP tree adaptively divides up space in an irregular manner, it takes longer to traverse the tree than more regular structures, like uniform grids.

Two variations of BSP trees are *k-d trees* and *octrees*. A k-d tree adds the restriction that the splitting plane must be aligned perpendicular to one of the coordinate axes; this makes traversal and construction of the tree more efficient. The octree also splits along coordinate axes, but splits the bound into eight equal-sized regions at each step.

Meta-Hierarchies

The idea of using spatial data structures can be generalized to include spatial data structures that themselves hold other spatial data structures, rather than just primitives. Not only could we have a grid that has sub-grids inside the grid cells that have many primitives in them (thus partially solving the adaptive refinement problem), but we could also have the scene organized into a HBV where the leaf nodes are grids that hold smaller collections of spatially-nearby primitives. Such hybrid techniques can bring the best of a variety of spatial data structure-based ray intersection acceleration methods. In `lrt`, because both geometric primitives and intersection accelerators inherit from the `Primitive` base class and thus provide the same interface, it's easy to mix and match in this way.

Refinements to basic approaches

There are a number of other important optimizations that can reduce the number of intersection tests made; some of them are implemented in `lrt` and some are left as exercises.

- Shadow rays can be processed more efficiently than camera rays, since we only need to find any intersection along the ray–it's not necessary to find the *closest* intersection. Once we have found anything that blocks the ray, we can immediately stop testing ray intersections and return. Furthermore, we don't need to compute the differential geometry at the hit point. Therefore, we can use the more efficient `IntersectP` routines of the `Shape`s and `Primitive`s to do these tests.

- Another technique that takes advantage of this property of shadow rays is the *shadow cache*; for each light source in the scene, we keep a pointer to the last primitive that occluded light from the emitter. Subsequent shadow rays are fist checked against this blocker–since the blocking object will often block a number of shadow rays in a row, this can make it much faster to find the blocker.

- For non-shadow rays, after we have found an intersection we keep track of the parametric distance to that hit. We have effectively turned our semi-infinite ray into a line segment, and we can cull from testing any primitives that are further along the ray than the hit point. We use this optimization in `lrt`; in Section 4.3 we will describe how this is used to reduce work in the grid accelerator.
- Shaft culling....
- A last technique has been dubbed *mailboxes*. Because a primitive may overlap multiple cells in grid or octree-type accelerators, we can keep track of which primitives have already been tested against the ray and void testing them multiple times as the ray goes through multiple cells that they overlap. Our grid implementation below will use this optimization.

4.2 Primitive Lists

In addition to `GeometricPrimitive`s, the type of `Primitive` that we have in `lrt` is a `PrimitiveList`. This is simply a collection of `Primitive`s that can be iterated through. Although not generally used by itself, it will form the basis for our acceleration structures. This encapsulation leads naturally to being able to have nested acceleration structures, such as grids within grids.

```
 9 Primitive
 6 prims
509 Reference
494 size
29&Union
```

⟨*PrimitiveList Methods*⟩≡
```
PrimitiveList::PrimitiveList(const vector<Reference<Primitive> > &p)
    : prims(p) {
    for (u_int i = 0; i < prims.size(); ++i)
        bounds = Union(bounds, prims[i]->WorldBound());
}
```

A `PrimitiveList` maintains an internal `vector` of `Primitive`s. It also maintains a bounding box that is the union of the bounds of all the primitives it contains.

⟨*PrimitiveList Protected Data*⟩≡
```
vector<Reference<Primitive> > prims;
BBox bounds;
```

We provide a simple method to query the number of `Primitive`s currently being stored:

⟨*PrimitiveList Public Interface*⟩+≡
```
int NumPrims() const { return (int) prims.size(); }
```

Finally, a simple method to get the bounding box of the collection of primitives.

⟨*PrimitiveList Public Interface*⟩+≡
```
BBox WorldBound() const { return bounds; }
```

⟨*PrimitiveList Methods*⟩+≡
```
bool PrimitiveList::CanIntersect() const {
    for (u_int i = 0; i < prims.size(); ++i)
        if (!prims[i]->CanIntersect())
            return false;
    return true;
}
```

⟨*PrimitiveList Methods*⟩+≡
```
bool PrimitiveList::Intersect(const Ray &r, Surf *s) const {
    bool hit = false;
    for (u_int i = 0; i < prims.size(); ++i)
        if (prims[i]->Intersect(r, s))
            hit = true;
    return hit;
}
```

⟨*PrimitiveList Methods*⟩+≡
```
bool PrimitiveList::IntersectP(const Ray &r) const {
    for (u_int i = 0; i < prims.size(); ++i)
        if (prims[i]->IntersectP(r))
            return true;
    return false;
}
```

⟨*PrimitiveList Methods*⟩+≡
```
void PrimitiveList::Refine(vector<Reference<Primitive> > &refined) const {
    for (u_int i = 0; i < prims.size(); ++i)
        if (prims[i]->CanIntersect())
            refined.push_back(prims[i]);
        else
            prims[i]->Refine(refined);
}
```

Primitive	9
primitives	579
prims	6
push_back	494
Ray	26
Reference	509
size	494
Surf	10

4.3 Regular Grid Accelerator

⟨*grid.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "primitives.h"
#include "geometry.h"
⟨GridAccelerator Declarations⟩
⟨GridAccelerator Method Definitions⟩
```

Here we will describe the implementation of `lrt`'s regular grid accelerator. It chooses a resolution for the grid based on the number of primitives it has to bound. Though the regular grid is not robust to very irregularly-distributed geometry, it usually works well in practice and is relatively easy to implement.

⟨*GridAccelerator Declarations*⟩≡
```
class GridAccelerator : public Primitive {
    ⟨GridAccelerator Forward Declarations⟩
public:
    ⟨GridAccelerator Method Declarations⟩
private:
    ⟨GridAccelerator Private Data⟩
};
```

Creation

⟨*GridAccelerator Method Definitions*⟩≡
```
GridAccelerator::GridAccelerator(
        const vector<Reference<Primitive> > &prims) {
    ⟨Compute grid bounds⟩
    ⟨Choose grid resolution⟩
    ⟨Compute voxel widths and allocate voxels⟩
    ⟨Add primitives to grid voxels⟩
    ⟨Initialize mailbox⟩
}
```

⟨*Compute grid bounds*⟩≡
```
for (u_int i = 0; i < prims.size(); ++i)
    bounds = Union(bounds, prims[i]->WorldBound());
```

⟨*GridAccelerator Private Data*⟩≡
```
BBox bounds;
```

⟨*GridAccelerator Method Declarations*⟩+≡
```
BBox WorldBound() const { return bounds; }
bool CanIntersect() const { return true; }
```

Given the set of primitives to bound, we need to choose a resolution for the grid. We take the cube root of the number of primitives and use that to set the grid resolution in whichever of the x, y or z dimensions that has the largest extent. The sizes in the other directions are set such that they are proportional to the sizes in the maximum dimension, according to the ratio of the grid extents in the two directions, in an effort to create voxels that are as square as possible.

⟨*Choose grid resolution*⟩≡
```
⟨Expand grid bounds by small factor⟩
Vector diag = bounds.pMax - bounds.pMin;
Float invmaxWidth = 1.0f/max(diag.x, max(diag.y, diag.z));
Assert(invmaxWidth > 0.f);
if (prims.size() < 5)
    XVoxels = YVoxels = ZVoxels = 1;
else {
    Float cubeRoot = powf(Float(prims.size()), 1.f/3.f);
    Float voxScale = 8.f * cubeRoot * invmaxWidth;
    XVoxels = Clamp(Round2Int(diag.x * voxScale), 1, 100);
    YVoxels = Clamp(Round2Int(diag.y * voxScale), 1, 100);
    ZVoxels = Clamp(Round2Int(diag.z * voxScale), 1, 100);
}
```

⟨*GridAccelerator Private Data*⟩+≡
```
int XVoxels, YVoxels, ZVoxels;
```

We'll expand the bounding box of all the primitives by a small factor, proportional to the grid's maximum extent. This helps avoid numerical error when primitives abut the sides of the voxel grids.

498 Assert
513 Clamp
513 max
28 pMax
28 pMin
9 Primitive
6 prims
509 Reference
514 Round2Int
494 size
29 Union
16 Vector

⟨*Expand grid bounds by small factor*⟩≡
```
Float delta = max(fabsf(bounds.pMin.x),
    max(fabsf(bounds.pMin.y), fabsf(bounds.pMin.z)));
delta = max(delta, max(fabsf(bounds.pMax.x),
        max(fabsf(bounds.pMax.y), fabsf(bounds.pMax.z))));
bounds.pMin -= 1e-4f * Vector(delta, delta, delta);
bounds.pMax += 1e-4f * Vector(delta, delta, delta);
```

We now use the chosen voxel resolutions to set `XWidth` and friends, which are the world-space widths of a voxel in each direction. We also precompute `InvXWidth` et al, so that routines that would otherwise divide by `XWidth` can be that much faster by multiplying rather than dividing.

⟨*Compute voxel widths and allocate voxels*⟩≡
```
XWidth = diag.x / XVoxels;
YWidth = diag.y / YVoxels;
ZWidth = diag.z / ZVoxels;
InvXWidth = (XWidth == 0.f) ? 0.f : 1.f / XWidth;
InvYWidth = (YWidth == 0.f) ? 0.f : 1.f / YWidth;
InvZWidth = (ZWidth == 0.f) ? 0.f : 1.f / ZWidth;
int nVoxels = XVoxels * YVoxels * ZVoxels;
voxels = (Voxel **)AllocL1CacheAligned(nVoxels * sizeof(Voxel *));
memset(voxels, 0, nVoxels * sizeof(Voxel *));
```

Small structure to hold informaiton needed for each voxel...

⟨*GridAccelerator Private Data*⟩+≡
```
struct Voxel {
    Voxel() { allCanIntersect = false; }
    vector<MailboxPrim *> primitives;
    bool allCanIntersect;
};
```

⟨*GridAccelerator Private Data*⟩+≡
```
Float XWidth, YWidth, ZWidth;
Float InvXWidth, InvYWidth, InvZWidth;
Voxel **voxels;
```

We make a small `MailboxPrim` structure for each `Primitive` in the grid. It stores both a pointer to the primitive as well as the integer mailbox id tag of the last ray that was tested against the primitive.

Each ray that comes into the `GridAccelerator::Intersect` routine is given a new, unique mailbox id number. After we test the ray against a primitive, we set the primitive's `lastMailboxId` value to the ray's id. Then, if the ray advances to another voxel that the primitive also overlaps, we can skip re-testing the primitive with the ray by just seeing if the ids match.

AllocL1CacheAligned 506
MailboxPrim 125
max 513
pMax 28
pMin 28
primitives 579
Vector 16
XVoxels 123
YVoxels 123
ZVoxels 123

⟨*GridAccelerator Private Data*⟩+≡
```
struct MailboxPrim {
    MailboxPrim() {
        primitive = NULL;
        lastMailboxId = -1;
    }
    Reference<Primitive> primitive;
    int lastMailboxId;
};
```

To add primitives to the grid, we loop through the primitives in turn, adding each one to the vectors of pointers to `MailboxPrim`s in the cells that its bounding box overlaps.

⟨*Add primitives to grid voxels*⟩≡
```
nMailboxes = prims.size();
mailboxes = (MailboxPrim *)AllocL1CacheAligned(prims.size() * sizeof(MailboxPrim));
for (u_int i = 0; i < prims.size(); ++i) {
    new (&mailboxes[i]) MailboxPrim;
    ⟨Find cell extent of primitive⟩
    ⟨Add primitive to overlapping cells⟩
}
⟨Update fraction of empty voxels⟩
```

We store a pointer to the array of `MailboxPrim`s allocated above so that the grid's destructor can free this memory.

⟨*GridAccelerator Private Data*⟩+≡
```
u_int nMailboxes;
MailboxPrim *mailboxes;
```

We find the world space bounds of the primitive and compute the integer set of voxels that it overlaps. We use the utility functions `x2v` et al, which turn a world space x, y, or z coordinate into voxel offsets. For safety in case of small numerical errors, these values are then clamped to the range of valid voxel addresses.

⟨*Find cell extent of primitive*⟩≡
```
BBox primBounds = prims[i]->WorldBound();
int x0 = max(x2v(primBounds.pMin.x), 0);
int x1 = min(x2v(primBounds.pMax.x), XVoxels-1);
int y0 = max(y2v(primBounds.pMin.y), 0);
int y1 = min(y2v(primBounds.pMax.y), YVoxels-1);
int z0 = max(z2v(primBounds.pMin.z), 0);
int z1 = min(z2v(primBounds.pMax.z), ZVoxels-1);
```

These utility functions turn coordinates in world space into integer voxel coordinates and integer voxel coordinates into the coordinates of their lower-left corners.

506 AllocL1CacheAligned
513 max
513 min
28 pMax
28 pMin
9 Primitive
6 prims
509 Reference
494 size
126 x2v
123 XVoxels
126 y2v
123 YVoxels
126 z2v
123 ZVoxels

⟨*GridAccelerator Method Declarations*⟩+≡
```
int x2v(Float x) const { return Float2Int((x - bounds.pMin.x) *
                    InvXWidth); }
int y2v(Float y) const { return Float2Int((y - bounds.pMin.y) *
                    InvYWidth); }
int z2v(Float z) const { return Float2Int((z - bounds.pMin.z) *
                    InvZWidth); }
Float v2x(int x) const { return x * XWidth + bounds.pMin.x; }
Float v2y(int y) const { return y * YWidth + bounds.pMin.y; }
Float v2z(int z) const { return z * ZWidth + bounds.pMin.z; }
```

After initializing the `MailboxPrim` for the `Primitive`, we just loop over the voxel addresses that the primitive covers, compute the offset into the array of voxel `vectors`s, and add the pointer to the end of it.

⟨*Add primitive to overlapping cells*⟩≡
```
MailboxPrim *mp = mailboxes + i;
mp->primitive = prims[i];
for (int z = z0; z <= z1; ++z)
    for (int y = y0; y <= y1; ++y)
        for (int x = x0; x <= x1; ++x) {
            int offset = z*XVoxels*YVoxels + y*XVoxels + x;
            if (!voxels[offset])
                voxels[offset] = new (voxelArena) Voxel;
            voxels[offset]->primitives.push_back(mp);
            ⟨Update grid voxel statistics⟩
        }
⟨Update grid/primitive statistics⟩
```

⟨*GridAccelerator Private Data*⟩+≡
```
MemoryArena<Voxel> voxelArena;
```

It's useful to keep a number of statistics about the grid and how well it's working. First, we'll track the maximum number of primitives in any one voxel. If there are voxels with many tens of primitives in them, then our acceleration structure probably isn't working too well.

⟨*Update grid voxel statistics*⟩≡
```
static StatsCounter maxPrimsInVoxel("Acceleration",
    "Max # of primitives in a grid voxel");
maxPrimsInVoxel.stat_max(voxels[offset]->primitives.size());
```

We'll also keep track of ratio of the total number of voxels overlapped by a primitive to the total number of primitives (i.e. the average number of voxels covered by a primitive.) Very fine grid resolutions will give a high number here, which may be indicative of wasted memory.

⟨*Update grid/primitive statistics*⟩≡
```
static StatsRatio nPrimitiveVoxels("Acceleration",
    "Voxels covered vs # / primitives");
nPrimitiveVoxels.add((1 + x1-x0) * (1 + y1-y0) * (1 + z1-z0), 1);
```


Float2Int 514
InvXWidth 124
InvYWidth 124
InvZWidth 124
mailboxes 125
MailboxPrim 125
MemoryArena 505
mp 131
pMin 28
primitives 579
prims 6
push_back 494
size 494
StatsCounter 501
StatsRatio 501
Voxel 124
voxels 124
XVoxels 123
XWidth 124
YVoxels 123
YWidth 124
ZWidth 124

Finally, we keep track of how many voxels ended up not having any primitives at all overlap them. This statistic is only computed after all primitives have been added to voxels, whereas the above statistics are updated per-primitive.

⟨*Update fraction of empty voxels*⟩≡
```
static StatsRatio nEmptyVoxels("Acceleration", "Empty voxels");
nEmptyVoxels.add(0, XVoxels * YVoxels * ZVoxels);
for (int i = 0; i < XVoxels * YVoxels * ZVoxels; ++i)
    if (!voxels[i]) nEmptyVoxels.add(1, 0);
```

The grid needs to keep track of the next valid, not-previously-used mailbox id value; the last thing we do in the constructor is initialize it.

⟨*Initialize mailbox*⟩≡
```
curMailboxId = 0;
```

⟨*GridAccelerator Private Data*⟩+≡
```
mutable int curMailboxId;
```

The destructor just has to free up the array of voxels that we made and the array of `MailboxPrim`s allocated. The primitives themselves are deleted when no other objects are holding references to them, which is likely (but not certain) to be when the grid accelerator is destroyed.

⟨*GridAccelerator Method Definitions*⟩+≡
```
GridAccelerator::~GridAccelerator() {
    for (u_int i = 0; i < nMailboxes; ++i)
        mailboxes[i].~MailboxPrim();
    FreeCacheAligned(voxels);
    FreeCacheAligned(mailboxes);
}
```

```
507 FreeCacheAligned
125 mailboxes
125 MailboxPrim
 26 Ray
501 StatsRatio
 10 Surf
124 voxels
123 XVoxels
123 YVoxels
123 ZVoxels
```

Traversal

We now come to the most interesting part of the grid, where we have a ray to compute primitive intersections with. We need to step through all of the cells that the ray passes through in order, first to last, and bail out as soon as we have found an intersection and can guarantee that there is no closer intersection (or, for shadow rays, any intersection will do.)

⟨*GridAccelerator Method Definitions*⟩+≡
```
bool GridAccelerator::Intersect(const Ray &ray,
        Surf *surf) const {
    ⟨Check ray against overall grid bounds⟩
    ⟨Set up 3D DDA for this ray⟩
    ⟨Walk grid⟩
}
```

We first check to see at what point the ray enters the grid. We first check the ray's origin with the grid's bounding box: if it's inside, then that's our starting point. Otherwise we try to intersect the ray with the grid's bounding box; if it hits, the parametric hit distance along the ray is our starting point. Otherwise, there can be no intersection with any of the geometry in the grid, so we return immediately.

⟨*Check ray against overall grid bounds*⟩≡
```
Float rayT = ray.maxt;
if (bounds.Inside(ray(ray.mint)))
    rayT = ray.mint;
else if (!bounds.IntersectP(ray, &rayT))
    return false;
Point gridIntersect = ray(rayT);
```

Next, we set up the initial (x,y,z) integer voxel coordinates for this ray, and set up difference values for stepping along. Our basic strategy will be to keep track of four important things (see Figure 4.6):

1. Which voxel we're currently in.
2. The parametric position along the ray where we make our next crossing in each of the *x*, *y*, and *z* directions.
3. How much farther we'll have to go parametrically along the ray after stepping to a new voxel in some direction before we step in the same direction again.
4. The (x,y,z) coordinates of the last voxel we pass through before we exit the grid.

Inside 30
Intersection 29
Point 21
StatsRatio 501

The first two items will be updated as we step through the grid, while the last two remain constant. We'll describe these computations for the *x* direction and won't include the *y* and *z* implementations here, as they are essentially identical.

⟨*Set up 3D DDA for this ray*⟩≡
```
⟨Update statistics for ray inside grid⟩
⟨Set up X stepping⟩
⟨Set up Y stepping⟩
⟨Set up Z stepping⟩
```

Two additional useful statistics are the average number of ray-primitive tests performed per-ray and the average number of intersections found. Here, we just increment the count for the number of rays that entered the grid.

⟨*Update statistics for ray inside grid*⟩≡
```
static StatsRatio rayTests("Acceleration", "Intersection tests per ray", false)
static StatsRatio rayHits("Acceleration", "Intersections found per ray", false)
rayTests.add(0, 1);
rayHits.add(0, 1);
```

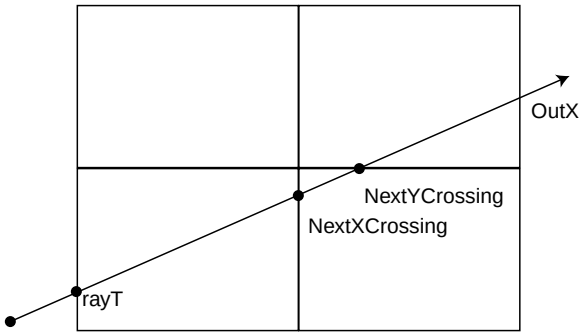


Figure 4.6: Stepping a ray through a voxel grid. We first compute `rayT`, the distance along the ray to the first intersection with the grid. We then compute distances along the ray to the next time we cross into the next voxel in the *x* direction, `NextXCrosing`, and in the *y* and *z* (not shown) directions. When we cross into the next *x* voxel, for example, we can immediately update the value of `NextXCrossing` by adding a fixed value, the voxel width in *x* divided by the ray's *x* direction, to it.

⟨*Set up X stepping*⟩≡
```
⟨Compute current x voxel⟩
Float NextXCrossing, DeltaX;
int StepX, OutX;
if (fabsf(ray.D.x) < 1e-6) {
    ⟨Handle ray perpendicular to x⟩
}
else if (ray.D.x > 0) {
    ⟨Handle ray with positive x direction⟩
}
else {
    ⟨Handle ray with negative x direction⟩
}
```

498 Assert
128 gridIntersect
126 x2v
123 XVoxels

Computing the voxel address that we start out in is pretty easy–we take the position where we enter the grid and compute its voxel number, being careful to handle the case where we've computed it to be outside the set of valid voxels (this may happen due to floating-point error, if `gridIntersect` is actually slightly outside of the grid).

⟨*Compute current x voxel*⟩≡
```
int x = x2v(gridIntersect.x);
if (x == XVoxels) x--;
else if (x < 0) ++x;
Assert(x >= 0 && x < XVoxels);
```

Now for each of *x*, *y*, and *z*, we compute crossing distances, changes in crossing distances when we step in that direction, and the exiting voxel numbers. If the ray's *x* component is nearly zero, then we'll *never* step in the *x* direction. We set the *x* crossing distance to infinity, so that we always decide that one of the other directions has the shortest parametric distance to the next voxel. As such, the values of `DeltaX` and `OutX` won't be used, but we'll set them to silence over-aggressive compiler warnings about uninitialized variables.

⟨*Handle ray perpendicular to x*⟩≡
```
NextXCrossing = INFINITY;
DeltaX = 0;
OutX = -1;
```

Things are more interesting in the common case. For a ray with a positive *x* direction component, the parametric value along the ray where we cross into the next voxel in *x*, `NextXCrossing` is our parametric starting point, `rayT` plus the *x* distance to the next voxel, divided by the *x* direction component. Similarly, dividing the width of a voxel in *x* by the ray's direction component gives us the parametric distance along the ray that we have to travel to get from one side of a voxel to the other, in the *x* direction.

`StepX` just tells us that when we leave a voxel in the *x* direction, we move 1 voxels. `OutX` says that when we reach a voxel with component `XVoxels`, we've left the grid and are done.

⟨*Handle ray with positive x direction*⟩≡
```
NextXCrossing = rayT + (v2x(x+1) - gridIntersect.x)/ray.D.x;
DeltaX = XWidth / ray.D.x;
StepX = 1;
OutX = XVoxels;
```

Similar computations compute these values for rays with negative *x* components.

⟨*Handle ray with negative x direction*⟩≡
```
NextXCrossing = rayT + (v2x(x) - gridIntersect.x)/ray.D.x;
DeltaX = - XWidth / ray.D.x;
StepX = -1;
OutX = -1;
```

This leads us to the code that walks through the grid. Starting with the first voxel, we check for intersection with the primitives inside that voxel. If we find a hit, `hitSomething` is set to true. Since we may have found a hit that is outside of the current voxel, however, we don't immediately return when through processing a voxel with an intersection. Instead, since the primitive's intersection routine will update the `maxt` variable, setting it to the parametric hit distance, we'll leave the grid stepping code to detect when we've walked into a voxel that's past an already-found hit.

If no hit is found in the current voxel, we step forward to the next voxel that the ray enters.

DeltaX 129
gridIntersect 128
INFINITY 514
NextXCrossing 129
OutX 129
rayT 128
StepX 129
v2x 126
XVoxels 123
XWidth 124

⟨*Walk grid*⟩≡
```
bool hitSomething = false;
⟨Get mailbox id for the ray⟩
for (;;) {
    int offset = z*XVoxels*YVoxels + y*XVoxels + x;
    Voxel *voxel = voxels[offset];
    if (voxel != NULL) {
        ⟨Check single voxel⟩
    }
    ⟨Advance to next voxel⟩
}
return hitSomething;
```

We grab a unique mailbox id number for the ray here. As we step through the grid, if e find a primitive that has a mailbox id number equal to the current ray's id, then we know that the ray has already tested for intersection with the primitive while in a previous voxel.

⟨*Get mailbox id for the ray*⟩≡
```
int rayId = curMailboxId++;
```

To check the primitives in a voxel, we first call `Refine()` methods if needed until we have primitives that are all able to test for ray intersections. We then loop through the primitives and call their intersection routines.

127 curMailboxId
125 MailboxPrim
579 primitives
494 size
124 Voxel
124 voxels
123 XVoxels
123 YVoxels

⟨*Check single voxel*⟩≡
```
vector<MailboxPrim *> &primitiveList = voxel->primitives;
⟨Refine primitives in voxel if needed⟩
for (u_int i = 0; i < primitiveList.size(); ++i) {
    MailboxPrim *mp = primitiveList[i];
    ⟨Do mailbox check between ray and primitive⟩
    ⟨Check for ray-primitive intersection⟩
}
```

A boolean value for each voxel is stored in the `allCanIntersect` array; it records whether all of the primitives in the voxel are known to be intersectable. If this value is false, we need to check them, calling their refinement routines until we have intersectable geometry.

⟨*Refine primitives in voxel if needed*⟩≡
```
if (!voxel->allCanIntersect) {
    for (u_int i = 0; i < primitiveList.size(); ++i) {
        MailboxPrim *mp = primitiveList[i];
        ⟨Refine primitive if needed⟩
    }
    voxel->allCanIntersect = true;
}
```

Handling primitives that need refinement is quite easy; we just get the vector of refined primitives from them and create a new `GridAccelerator` to hold the returned primitives all if more than one was returned. We then update the pointer in `mp->primitive` appropriately and continue. The `Intersect` call in ⟨*Check for ray-primitive intersection*⟩ will then call the intersect routine of the refined

primitive or of the newly-created grid. Note that we may need to repeatedly refine the resulting primitives, since shapes are allowed to refine themselves into shapes that aren't themselves intersectable, so long as continued refinement eventually gives intersectable shapes.

⟨*Refine primitive if needed*⟩≡
```
while (!mp->primitive->CanIntersect()) {
    vector<Reference<Primitive> > p;
    mp->primitive->Refine(p);
    Assert(p.size() > 0);
    if (p.size() == 1)
        mp->primitive = p[0];
    else
        mp->primitive = new GridAccelerator(p);
}
```

We do the mailbox check before calling the `Primitive`'s actual intersection routine; if we've already intersected this ray against this primitive in a previous voxel that this primitive was also stored in that the ray has already passed through, we can trivially skip doing a redundant intersection test.

Assert	498
hitSomething	131
lastMailboxId	125
mp	131
Primitive	9
Reference	509
size	494

⟨*Do mailbox check between ray and primitive*⟩≡
```
if (mp->lastMailboxId == rayId)
    continue;
```

If we are going to do the ray intersection test, we first update the mailbox for the primitive. We can then call the `Primitive::Intersect` method, recording whether any intersection has been found along the ray.

⟨*Check for ray-primitive intersection*⟩≡
```
mp->lastMailboxId = rayId;
rayTests.add(1, 0);
if (mp->primitive->Intersect(ray, surf)) {
    rayHits.add(1, 0);
    hitSomething = true;
}
```

We now have the code to step to the next voxel. We see which direction is the first where we step into a new voxel; whichever of these has the lowest `Next?Crossing` value is the one. We then do the appropriate computations to step as needed. If we determine that we've stepped out of the voxel grid, or if we've stepped beyond the t distance of an intersection we've already found, then we'll break out of the traversal loop.

⟨*Advance to next voxel*⟩≡
```
if (NextXCrossing < NextYCrossing &&
    NextXCrossing < NextZCrossing) {
    ⟨Step in X⟩
}
else if (NextZCrossing < NextYCrossing) {
    ⟨Step in Z⟩
}
else {
    ⟨Step in Y⟩
}
```

We first see if an intersection has been found that is inside the current voxel. If so, we're done and can exit. This is the case if `maxt` is less than the parametric distance at which we enter the next *x* voxel, `NextXCrossing` Otherwise we update the variable that holds the current voxel address by adding `StepX` (which is either -1 or 1) to it. If we have left the grid (`x == OutX`), then we also break. Otherwise we increment the value of `NextXCrossing` to the `DeltaX` value, so that we know how far we need to go parametrically before stepping in *x* again.

⟨*Step in X*⟩≡
```
if (ray.maxt < NextXCrossing)
    break;
x += StepX;
if (x == OutX)
    break;
NextXCrossing += DeltaX;
```

129 DeltaX
129 NextXCrossing
129 OutX
26 Ray
129 StepX

The cases for stepping in *y* and *z* are equivalent and are omitted.

We also provide a specialized version of `GridAccelerator::IntersectP()` that is optimized for checking for intersection along shadow rays, where we only are interested if there is an intersection, rather than knowing the full details of the closest intersection. It is almost completely identical to the normal `GridAccelerator::Intersect()` routine, except that it calls the `Primitive::IntersectP()` method of the primitives, rather than `Primitive::Intersect()`, and it immediately stops traversal when any intersection is found. Because of the small number of differences, we won't include the implementation here.

⟨*GridAccelerator Method Declarations*⟩+≡
```
bool IntersectP(const Ray &ray) const;
```

4.4 Kd Tree

⟨*kdtree.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "primitives.h"
#include "geometry.h"
⟨KdTreeAccelerator Declarations⟩
⟨KdTreeAccelerator Method Definitions⟩
```

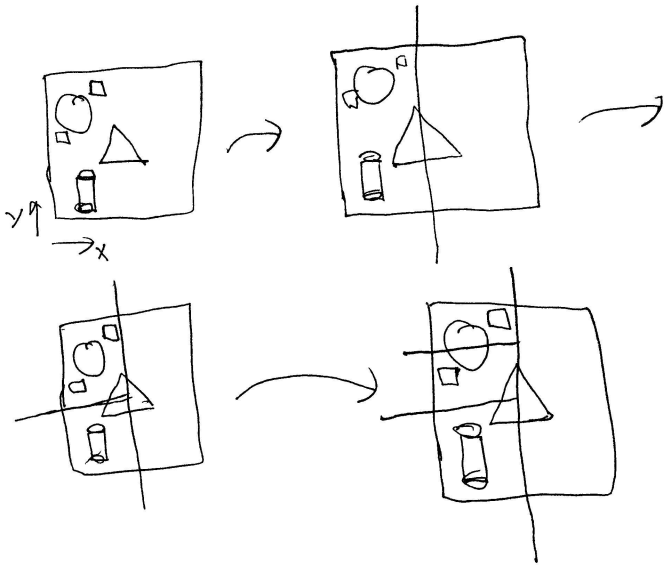


Figure 4.7: The kd-tree is built by recursively splitting the scene's bound along one of the coordinate axes. Here, we start by splitting along the *x* axis; all of the primitives still overlap the resulting left region, though only the triangle overlaps the right region. Therefore, we stop refining the right region any further. Continuing along, we split the left region along the *y* axis, giving a region with only two primitives in it in the bottom left. We split the upper left one more time, again along the *y* axis, before terminating. The details of the refinement criteria: which axis we split along, at which position we split, and at what point we stop, can all substantially affect the performance of the tree in practice.

Primitive 9
primitives 579

To complement the `GridAccelerator`, `lrt` also has an accelerator based on kd-trees. Recall that the kd-tree recursively splits up space with axis-aligned planes; splitting stops when the region of space that a node represents has a small number of primitives in it or when we reach a maximum depth. Each leaf of the tree holds a list of the primitives that overlap the region of space that it represents; see Figure 4.7 for an overview of how the tree is built. Because the kd-tree adaptively splits up 3D space based on the spatial distribution of primitives in the scene, It can have better performance than uniform grids for scenes with irregular distributions of primitives, where a grid might have an enormous number of empty cells in order to ensure that the cells in the dense regions don't have too many primitives in them.

⟨*KdTreeAccelerator Declarations*⟩+≡
```
class KdTreeAccelerator : public Primitive {
public:
    ⟨KdTreeAccelerator Method Declarations⟩
private:
    ⟨KdTreeAccelerator Private Data⟩
    ⟨KdTreeAccelerator Private Methods⟩
};
```

For simplicity of implementation, the `KdTreeAccelerator` requires that all of its primitives be intersectable. We leave as an exercise the task of improving the

implementation to do lazy refinement like the GridAccelerator does. Therefore, the constructor starts out by refining all primitives until we only have intersectable primitives. We then do some general preparation and build the tree.

⟨*KdTreeAccelerator Method Definitions*⟩≡
```
KdTreeAccelerator::KdTreeAccelerator(const vector<Reference<Primitive> > &prims) {
    ⟨Refine all prims until they are intersectable⟩
    ⟨Initialize mailboxes for KdTreeAccelerator⟩
    ⟨Set up memory pools for kd tree nodes⟩
    ⟨Build kd tree for accelerator⟩
}
```

Because the new Primitives returned from the Primitive::Refine() method may themselves need to be refined before we have intersectable primitives, we maintain both a vector of known intersectable primitives, prefined, as well as a vector of primitives yet to be processed–some may be intersectable, and some may yet need more refinement. For those that do need refinement, we take advantage of the fact that Primitive::Refine() adds new primitives to the end of the vector that is passed in.

⟨*Refine all* prims *until they are intersectable*⟩≡
```
vector<Reference<Primitive> > prefined, todo = prims;
while (todo.size()) {
    Reference<Primitive> prim = todo.back();
    todo.pop_back();
    if (prim->CanIntersect())
        prefined.push_back(prim);
    else
        prim->Refine(todo);
}
```

```
127 curMailboxId
140 KdAccelNode
134 KdTreeAccelerator
125 MailboxPrim
  9 Primitive
  6 prims
494 push_back
509 Reference
494 size
```

As with the GridAccelerator, we'll use mailboxing to avoid repeated intersections with primitives that straddle splitting planes and overlap multiple regions of the tree. In fact, we'll use the exact same MailboxPrim structure.

⟨*Initialize mailboxes for* KdTreeAccelerator⟩≡
```
curMailboxId = 0;
mailboxPrims = new MailboxPrim[prefined.size()];
for (u_int i = 0; i < prefined.size(); ++i)
    mailboxPrims[i].primitive = prefined[i];
```

⟨*KdTreeAccelerator Private Data*⟩≡
```
KdAccelNode *root;
MailboxPrim *mailboxPrims;
mutable int curMailboxId;
BBox bounds;
```

Tree construction

At each step of building the tree, we choose a splitting plane and classify the remaining primitives with respect to the plane. We then recursively build trees for each of the children of the current node, processing the primitives that overlapped

each one. Because we will be repeatedly referring to the bounding boxes of the primitives along the way, we precompute them and store them in a vector so that we don't repeatedly call primitives' potentially-slow `WorldBound()` methods. So that we don't need to repeatedly build vectors for the bounds of the sets of primitives on each side of the splitting plane, we pass a vector of integers, recording which primitive numbers overlapped each side of the split. Just recording integers in this manner improves tree building efficiency by reducing the amount of data to be copied along the way.

XXX need to make clear that we've got this bound that is the volume of interest– it will either be split in half or it will store a list of the primitives that overlap it... XXX

⟨*Build kd tree for accelerator*⟩≡
```
vector<BBox> primBounds;
vector<int> primNums;
primBounds.reserve(prefined.size());
primNums.reserve(prefined.size());
for (u_int i = 0; i < prefined.size(); ++i) {
    BBox b = prefined[i]->WorldBound();
    bounds = Union(bounds, b);
    primBounds.push_back(b);
    primNums.push_back(i);
}
buildTree(&root, bounds, mailboxPrims, primBounds, primNums, 0);
```

KdAccelNode	140
KdTreeAccelerator	134
MailboxPrim	125
primBounds	125
push_back	494
reserve	494
size	494
Union	29

`buildTree()` is called for each node of the tree as we're building it. Given the integer primitive numbers to be considered and the bounding box of the current region, it decides if the recursive splitting should continue or if we've reached a leaf node, updating the tree appropriately.

⟨*KdTreeAccelerator Method Definitions*⟩+≡
```
void KdTreeAccelerator::buildTree(KdAccelNode **node, const BBox &nodeBounds,
        MailboxPrim *mailboxPrims, const vector<BBox> &allPrimBounds,
        const vector<int> &primNums, int depth) {
    if (!primNums.size()) {
        *node = NULL;
        return;
    }
    ⟨Initialize leaf node if termination criteria met⟩
    ⟨Initialize interior node and continue recursion⟩
}
```

We stop building the tree if we've either got a sufficently small number of primitives in the region or if we've reached a maximum depth. We relax the "small number of primitives" test as the tree gets deeper because deep trees take longer to traverse for intersection tests–it's not necessarily worth increasing the tree depth substantially versus doing a few more intersection tests while traversing it.

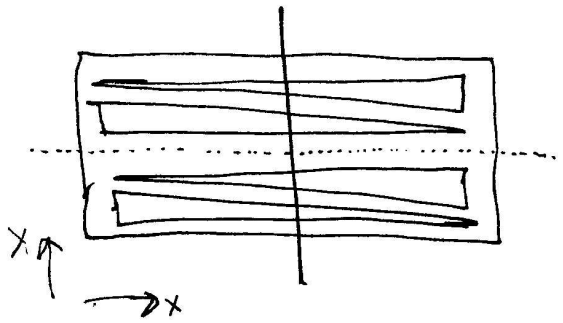


Figure 4.8: A case where splitting along the axis with the largest extent isn't necessarily the optimal choice: an *x* split, shown with the solid line, leaves all of the primitives overlapping both sides of the splitting plane, while a split along the *y* axis, shown with a dotted line, would cleanly split the primitives into independent sets.

⟨*Initialize leaf node if termination criteria met*⟩≡
```
if (primNums.size() == 1 ||
    (primNums.size() < 5 && depth > 10) ||
    depth > 20) {
    *node = allocNode(depth, primNums, nodeBounds);
    return;
}
```

```
28  pMax
28  pMin
494 size
16  Vector
```

Otherwise, we choose a splitting plane, classify the primitives, and work on down the tree.

⟨*Initialize interior node and continue recursion*⟩≡
⟨*Choose split axis for interior node*⟩
⟨*Compute node split position and allocate interior* `KdAccelNode`⟩
⟨*Classify primitives with respect to split*⟩
⟨*Recursively initialize children nodes*⟩

Which axis to split along is determined by the coordinate axis along which the node's bounds have the largest extent. Other reasonable approaches include cycling through *x*, *y*, and *z* at successive levels of the tree, or trying each axis and choosing the one that gives the smallest number of primitives that straddle both sides of the splitting plane. Figure 4.8 shows a situation where this case may lead to a suboptimal tree.

⟨*Choose split axis for interior node*⟩≡
```
int nextAxis;
Vector diag = nodeBounds.pMax - nodeBounds.pMin;
if (diag.x > diag.y && diag.x > diag.z) nextAxis = SPLIT_X;
else if (diag.y > diag.z) nextAxis = SPLIT_Y;
else nextAxis = SPLIT_Z;
```

⟨*KdTreeAccelerator Declarations*⟩+≡
```
#define SPLIT_X 0
#define SPLIT_Y 1
#define SPLIT_Z 2
```

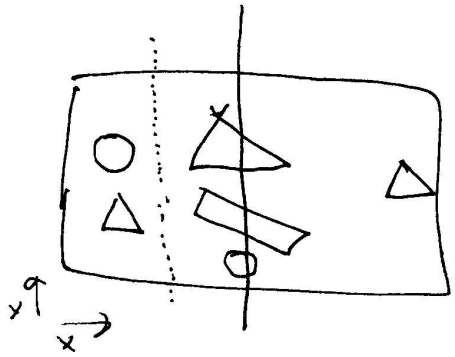


Figure 4.9: Splitting along the center of the bounds (solid line) may lead to an excessive number of primitives overlapping both children of the tree. Here, splitting along the dotten line would create a tree without any overlapping primitives at this split.

We always split down the middle of the node's bounds. Here also we could try to choose more carefully based on primitive bounds to reduce number of primitives that straddle the splitting plane and overlap both children. Figure 4.9 shows a case where a different spliting strategy could be more effective.

Having chosen the split position, it's straightforward to compute the bounding boxes of the child nodes. We then allocate the interior node of the tree to hold pointers to the children before continuing onward.

⟨*Compute node split position and allocate interior* KdAccelNode⟩≡

```
BBox bounds0 = nodeBounds, bounds1 = nodeBounds;
Float tsplit;
if (nextAxis == SPLIT_X) {
    tsplit = Lerp(.5f, nodeBounds.pMin.x, nodeBounds.pMax.x);
    bounds0.pMax.x = bounds1.pMin.x = tsplit;
}
else if (nextAxis == SPLIT_Y) {
    tsplit = Lerp(.5f, nodeBounds.pMin.y, nodeBounds.pMax.y);
    bounds0.pMax.y = bounds1.pMin.y = tsplit;
}
else {
    tsplit = Lerp(.5f, nodeBounds.pMin.z, nodeBounds.pMax.z);
    bounds0.pMax.z = bounds1.pMin.z = tsplit;
}
*node = allocNode(depth, nextAxis, tsplit);
```

Lerp 512
pMax 28
pMin 28

And we can now build the vectors that record which primitives overlap each side of the split. Because we classify the primitives using their bounding boxes, we may sometimes think that a primitive overlaps a region of space that it actually doesn't, leading to a tree that will require unnecessary ray–primitive intersection tests when it is traversed. Figure 4.10 shows an example of this problem. The excess work usually isn't too much in practice; an exercise at the end of the chapter outlines one approach to improving the classificaiton of primitives in the tree.

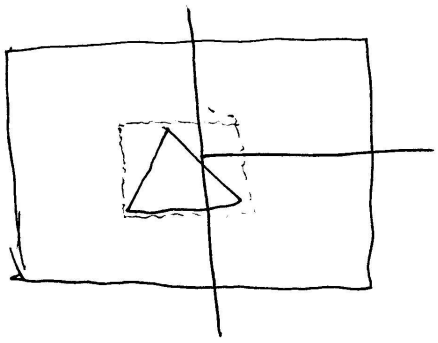


Figure 4.10: Using the bounding box to classify primitives with respect to the kd tree means that sometimes we will incorrectly believe that they overlap regions that they actually don't. Here, the triangle shown doesn't actually overlap the upper right region of the tree, even though the bounding box test says that it does.

⟨*Classify primitives with respect to split*⟩≡

```
vector<int> prims0, prims1;
for (u_int i = 0; i < primNums.size(); ++i) {
    int primNum = primNums[i];
    if (bounds0.Overlaps(allPrimBounds[primNum]))
        prims0.push_back(primNum);
    if (bounds1.Overlaps(allPrimBounds[primNum]))
        prims1.push_back(primNum);
}
```

30	Overlaps
494	push_back
494	size

⟨*Recursively initialize children nodes*⟩≡

```
buildTree(&((*node)->u.children[0]), bounds0, mailboxPrims, allPrimBounds,
    prims0, depth+1);
buildTree(&((*node)->u.children[1]), bounds1, mailboxPrims, allPrimBounds,
    prims1, depth+1);
```

Each node of the kd tree–leaf or interior–is represented by a `KdAccelNode` structure. Assuming `Float`s and pointers are four bytes large, each node uses 16 bytes of storage, thanks to careful use of bitfields and a `union` that lets us overlap memory used by leaf and interior nodes, since we won't need to store both leaf-related and interior node-related data in the same node. Keeping the structure this size lets two nodes fit exactly in a 32 byte cache line, which improves performance at traversal time by limiting cache misses to no more than one each time a node is accessed.

⟨*KdTreeAccelerator Declarations*⟩+≡
```
struct KdAccelNode {
    ⟨KdAccelNode Constructors⟩
    ⟨KdAccelNode Destructor⟩
    u_int axis:2;
    u_int isLeaf:1;
    u_int nPrimitives:24;
    Float split;
    union {
        MailboxPrim **primitives;
        KdAccelNode *children[2];
    } u;
};
```

There are two `KdAccelNode` constructors; the first one is for interior nodes, where the split axis and position are passed in.

⟨*KdAccelNode Constructors*⟩≡
```
KdAccelNode(int a, Float s) {
    axis = a;
    isLeaf = 0;
    split = s;
    u.children[0] = u.children[1] = NULL;
}
```

MailboxPrim 125
primitives 579
size 494

The second constructor is for leaf nodes; it takes the overlapping primitives and the bounding box for the node.

⟨*KdAccelNode Constructors*⟩+≡
```
KdAccelNode(MailboxPrim *allPrimitives, const vector<int> &primNums,
        const BBox &nodeBound) {
    nPrimitives = u_int(primNums.size());
    u.primitives = new MailboxPrim *[nPrimitives];
    for (u_int i = 0; i < nPrimitives; ++i)
        u.primitives[i] = &allPrimitives[primNums[i]];
    isLeaf = 1;
}
```

⟨*KdAccelNode Destructor*⟩≡
```
~KdAccelNode() {
    if (isLeaf)
        delete[] u.primitives;
}
```

Cache-friendly memory allocation

Because acceleration data structure traversal is at the heart of `lrt`'s inner loop, it's worth going through some effort to ensure cache-friendly layout of `KdAccelNode`s in memory. Applying the techniques in this section sped up `lrt` by 3–5% for a handful of test scenes, thanks to reduced cache misses. While this isn't an enormous speedup, it's a relatively easy one to take advantage of. For a review of principles of cache-friendly programming issues, see Section XXX.

We provide two `allocNode()` functions, one for allocating and initializing interior `KdAccelNode`s and one for leaf nodes. Both use the same key fragment, ⟨*Get pointer to new* `KdAccelNode`⟩ to get a pointer to uninitialized memory in `node`; they then use C++'s *placement new operator* to construct a `KdAccelNode` at the given memory location.

⟨*KdTreeAccelerator Method Definitions*⟩+≡
```
KdAccelNode *KdTreeAccelerator::allocNode(int depth, int axis,
        Float split) {
    ⟨Update kd interior node allocation statistics⟩
    ⟨Get pointer to new KdAccelNode⟩
    new (node) KdAccelNode(axis, split);
    return node;
}
```

⟨*KdTreeAccelerator Method Definitions*⟩+≡
```
KdAccelNode *KdTreeAccelerator::allocNode(int depth,
        const vector<int> &primNums, const BBox &nodeBound) {
    ⟨Update kd leaf node allocation statistics⟩
    ⟨Get pointer to new KdAccelNode⟩
    new (node) KdAccelNode(mailboxPrims, primNums, nodeBound);
    return node;
}
```

140 KdAccelNode
134 KdTreeAccelerator

Our strategy for improving the cache layout of `KdAccelNode`s has two main components. First, rather than allocating nodes one at a time as needed, we allocate large contiguous blocks of them and parcel them out as needed. Doing this ensures that our careful work to keep `KdAccelNode`s at 16 bytes doesn't go to waste–dynamic memory allocation typically adds an overhead of four to eight bytes per allocation request, which would mean that two nodes would no longer exactly fit into a 32 byte cache line. Further, allocating these chunks with the aligned `AllocL2CacheAligned()` function ensures that none of the individual nodes straddles more than one cache entry (as long as the size of cache lines is an even multiple of the size of `KdAccelNode`s.)

Second, we make sure that the nodes at the top few levels of the tree won't map to the same cache entries, ensuring that there won't be any cache misses due to conflicts among high nodes in the tree. Our assumption here is that the nodes in the top part of the tree will be the most frequently accessed ones, so minimizing conflicts among them is worthwhile. We can easily do this by allocating a large contiguous chunk of memory for all of the top levels of the tree–so long as the size of this chunk is less than or equal to the cache size, no two locations inside the chunk will map to the same cache entry.

⟨*Get pointer to new* `KdAccelNode`⟩≡
```
KdAccelNode *node;
if (depth < MAX_TOP_DEPTH) {
    ⟨Allocate kd tree node for top part of tree⟩
}
else {
    ⟨Allocate kd tree node for bottom part of tree⟩
}
```

Nodes up to depth MAX_TOP_DEPTH re allocated from the contiguous chunk of memory for nodes at the top of the tree. numTop is initialized to hold the total number of such nodes. Memory for these nodes is handed out in breadth-first order–the root node of the tree is given the first node in the chunk, the two nodes at the next level are given the next two, and so forth. The topNodeOffset[] array is used to record how many nodes have been allocated so far at each depth.

⟨*Allocate kd tree node for top part of tree*⟩≡
```
int offset = (1 << depth) - 1 + topNodeOffset[depth];
Assert(offset < (1 << MAX_TOP_DEPTH) - 1);
++topNodeOffset[depth];
node = topNodes + offset;
```

For the bottom levels of the tree, we just hand out nodes as needed from a chunk of memory for LOWER_NODE_ALLOC_SIZE nodes. Whenever we need to allocate a new chunk, we add its starting address to the allocedNodeBlocks vector, so that we can free all of the allocated memory when we're done.

⟨*KdTreeAccelerator Private Data*⟩+≡
```
#define LOWER_NODE_ALLOC_SIZE 2048
```

⟨*Allocate kd tree node for bottom part of tree*⟩≡
```
if (lowerNodePos == LOWER_NODE_ALLOC_SIZE) {
    lowerNodePos = 0;
    lowerNodes = (KdAccelNode *)AllocL2CacheAligned(
        LOWER_NODE_ALLOC_SIZE*sizeof(KdAccelNode));
    allocedNodeBlocks.push_back(lowerNodes);
}
node = lowerNodes + lowerNodePos;
++lowerNodePos;
```

AllocL2CacheAligned 507
Assert 498
KdAccelNode 140
push_back 494

⟨*Set up memory pools for kd tree nodes*⟩≡
```
int numTop = (1 << MAX_TOP_DEPTH) - 1;
topNodes = (KdAccelNode *)AllocL2CacheAligned(numTop * sizeof(KdAccelNode));
allocedNodeBlocks.push_back(topNodes);
for (int i = 0; i < MAX_TOP_DEPTH; ++i)
    topNodeOffset[i] = 0;
lowerNodes = NULL;
lowerNodePos = LOWER_NODE_ALLOC_SIZE;
```

⟨*KdTreeAccelerator Private Data*⟩+≡
```
KdAccelNode *topNodes;
#define MAX_TOP_DEPTH 10
int topNodeOffset[MAX_TOP_DEPTH];
KdAccelNode *lowerNodes;
int lowerNodePos;
vector<KdAccelNode *> allocedNodeBlocks;
```

XXX need to run destructors for the ones that we constructed...

⟨*Free memory pools for* KdTreeNodes⟩≡
```
for (u_int i = 0; i < allocedNodeBlocks.size(); ++i)
    FreeCacheAligned(allocedNodeBlocks[i]);
```

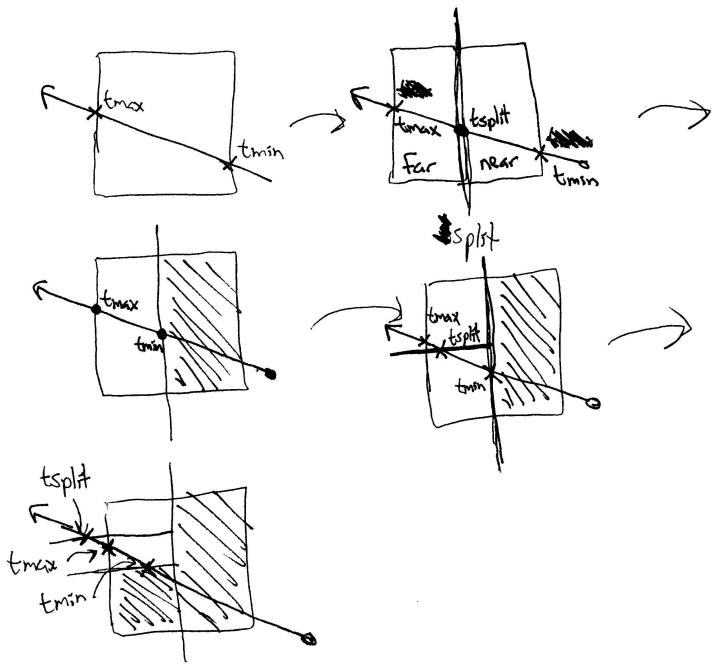


Figure 4.11: Traversal of a ray through the kd tree: the ray is intersected with the bounds of the tree, giving an initial parametric $[t_{\min}, t_{\max}]$ range to consider. Because this range is non-empty, we need to consider the two children of the root node, here. The ray first enters the child on the right, labeled "near", where it has a parametric range $[t_{\min}, t_{\text{split}}]$. If the near node is a leaf with primitives in it,we intersect the ray with the primitives; otherwise we process its children nodes. If no hit is found, or if a hit is found beyond $[t_{\min}, t_{\text{split}}]$, then the far node, on the left, is processed. This sequence continues–processing tree nodes in a depth-first, front-to-back traversal–until the closest intersection is found or the ray exits the tree.

507 FreeCacheAligned
494 size

Traversal

Figure 4.11 shows the basic process of ray traversal through the tree; if the ray intersects the tree's bounds, we "open up" the root node, first processing the child of the root that the ray enters first and then processing the other child only after processing of the near node and its children is done. We stop traversal either when the ray exits the tree or when we find the closest intersection.

⟨*KdTreeAccelerator Method Declarations*⟩+≡
```
BBox WorldBound() const { return bounds; }
bool CanIntersect() const { return true; }
```

Traversal walks the tree in the order that the ray passes through tis nodes; see Figure 4.11. We start by intersecting the ray with the tree's overall obunds, giving us initial `tmin` and `tmax` values, marked with "x"s in the figure. Because the ray does intersect the top-level bounds, ...

⟨*KdTreeAccelerator Method Definitions*⟩+≡
```
bool KdTreeAccelerator::Intersect(const Ray &ray, Surf *surf) const {
    ⟨Compute initial parametric range of ray inside kd tree extent⟩
    ⟨Prepare to traverse kd-tree for ray⟩
    bool hit = false;
    while (node != NULL) {
        ⟨Process kd tree node node for ray traversal⟩
    }
    return hit;
}
```

We start by finding the overall parametric range $[t_{\min}, t_{\max}]$ of the ray overlapping the tree, exiting immediately if there is no overlap.

⟨*Compute initial parametric range of ray inside kd tree extent*⟩≡
```
Float tmin, tmax;
if (!bounds.IntersectP(ray, &tmin, &tmax))
    return false;
tmax *= 1.001f;
```

Before traversal starts, we get a new mailbox id for the ray and precompute the inverse of the components of the direction vector, in order to replace divides with multiplies in the main traversal loop. We also set up an array of `KdToDo` structures, which are used to record the nodes yet to be processed for the ray. These are ordered so that the last active entry in the array is the next node to be considered. It can be shown that the maximum number of entries needed in this array is the maximum depth of the kd tree; the array size used below should be more than enough in practice.

curMailboxId 127
KdAccelNode 140
KdTreeAccelerator 134
Ray 26
Surf 10
Vector 16

⟨*Prepare to traverse kd-tree for ray*⟩≡
```
int rayId = curMailboxId++;
const KdAccelNode *node = root;
Vector invDir(1.f/ray.D.x, 1.f/ray.D.y, 1.f/ray.D.z);
#define MAX_TODO 64
KdToDo todo[MAX_TODO];
int todoPos = 0;
```

⟨*KdTreeAccelerator Declarations*⟩+≡
```
struct KdToDo {
    const KdAccelNode *node;
    Float tmin, tmax;
};
```

For each node of the tree that we process, we first see if we can stop traversing due to having found an intersection that is closer along the ray than the ray's overlap with the node. If this is not so, we either do ray–primitive intersections, for a leaf node, or determine which of an interior node's children the ray overlaps.

⟨*Process kd tree node* node *for ray traversal*⟩≡
```
⟨Update kd-tree traversal statistics⟩
⟨Bail out if we found a hit closer than the current node⟩
if (node->isLeaf) {
    ⟨Check for intersections inside leaf node⟩
    ⟨Grab next node to process from todo list⟩
}
else {
    ⟨Process kd tree interior node⟩
}
```

⟨*Update kd-tree traversal statistics*⟩≡
```
static StatsCounter nodesTraversed("Acceleration",
    "Number of kd-tree nodes traversed by normal rays");
++nodesTraversed;
```

We may have previously found an intersection in a primitive that overlaps multiple nodes; if the intersection was outside the current node when first detected, we need to keep traversing the tree until we come to a node where the node entry-point $t_{\min}$ is beyond the intersection; only then do we know that there is no closer intersection.

⟨*Bail out if we found a hit closer than the current node*⟩≡
```
if (ray.maxt < tmin) break;
```

If the current node is a leaf, we loop over the primitives in the leaf, using the mailbox test to avoid re-testing primitives that have already been processed for this ray.

⟨*Check for intersections inside leaf node*⟩≡
```
u_int nPrimitives = node->nPrimitives;
MailboxPrim **prims = node->u.primitives;
for (u_int i = 0; i < nPrimitives; ++i) {
    MailboxPrim *mp = prims[i];
    if (mp->lastMailboxId != rayId) {
        mp->lastMailboxId = rayId;
        if (mp->primitive->Intersect(ray, surf))
            hit = true;
    }
}
```

After doing the intersection tests, we find the next node to process from the `todo` array. If there are no more nodes, we break out of the traversal loop.

⟨*Grab next node to process from todo list*⟩≡
```
if (todoPos > 0) {
    --todoPos;
    node = todo[todoPos].node;
    tmin = todo[todoPos].tmin;
    tmax = todo[todoPos].tmax;
}
else
    break;
```

125 lastMailboxId
125 MailboxPrim
131 mp
579 primitives
6 prims
501 StatsCounter

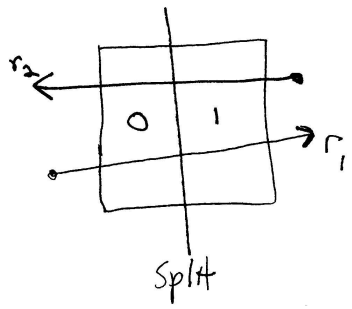


Figure 4.12: The position of the origin of the ray with respect to the splitting plane can be used to determine which of the node's children should be processed first. Because the child on the "below" side of the splitting plane is always stored in `children[0]` and the "above" side in `children[1]`, if the ray is on the below side of the split plane, we should process `children[0]` before `children[1]` and vice versa.

For interior tree nodes, we intersect the ray with the node's splitting plane and determine if one or both of the children nodes needs to be processed and in what order to do so.

⟨*Process kd tree interior node*⟩≡
  ⟨*Compute distance along ray to split plane*⟩
  ⟨*Get near and far child pointers for ray*⟩
  ⟨*Advance to next child node, possibly enqueue far child*⟩

The parametric distance to the split plane is computed in the same manner as was done in the ray–bounding box test, for example.

⟨*Compute distance along ray to split plane*⟩≡
```
Float tplane;
if (node->axis == SPLIT_X)
    tplane = (node->split - ray.O.x) * invDir.x;
else if (node->axis == SPLIT_Y)
    tplane = (node->split - ray.O.y) * invDir.y;
else
    tplane = (node->split - ray.O.z) * invDir.z;
```

We also need to determine which of the node's children should be processed first, so that we traverse the tree in front-to-back order along the ray. Figure 4.12 shows the geometry of this computation.

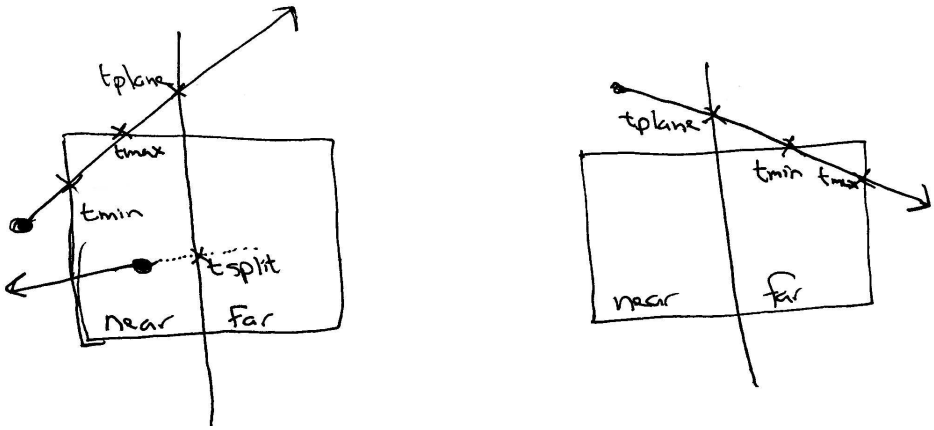


Figure 4.13: Two cases where both children of a node don't need to be processed because the ray doesn't overlap them. On the left, the top ray intersects the splitting plane beyond the ray's $t_{\max}$ position and thus doesn't enter the far child. The bottom ray is facing away from the splitting plane, indicated by a negative t_{split} value. On the right, the ray intersects the ray before the ray's $t_{\min}$ value, indicating that the near plane doesn't need processing.

140 KdAccelNode

⟨*Get near and far child pointers for ray*⟩≡
```
KdAccelNode *nearChild, *farChild;
bool zeroIsNear;
if (node->axis == SPLIT_X)
    zeroIsNear = (ray.O.x <= node->split);
else if (node->axis == SPLIT_Y)
    zeroIsNear = (ray.O.y <= node->split);
else
    zeroIsNear = (ray.O.z <= node->split);
if (zeroIsNear) {
    nearChild = node->u.children[0];
    farChild  = node->u.children[1];
}
else {
    nearChild = node->u.children[1];
    farChild  = node->u.children[0];
}
```

However, we don't necessarily need to process both children of this node; the details of this are slightly tricky. Figure 4.13 shows some configurations that we handle here. The first `if` test below corresponds to the left side of the figure: only the near node needs to be processed if it can be shown that the ray doesn't overlap the far node because it faces away from the far node or doesn't overlap it. The right side of the figure shows the case tested in the second `if` test: the near node may not need processing if the ray doesn't overlap it. Otherwise, the `else` clause handles the case of both children needing processing; we process the near node and enqueue the far node. However, because one (but not both!) of an inner node's children pointers may be `NULL`, we go directly to the far node if the near node is

NULL and we don't enqueue the far node if it is NULL. After all this, we make sure that node isn't NULL (as would happen with a NULL near or far child for the first two cases, respectively). If it is, we get the next node from the todo array.

⟨*Advance to next child node, possibly enqueue far child*⟩≡
```
if (tplane > tmax || tplane < 0)
    node = nearChild;
else if (tplane < tmin)
    node = farChild;
else {
    if (!nearChild) {
        node = farChild;
        tmin = tplane;
    }
    else {
        if (farChild) {
            ⟨Enqueue farChild in todo list⟩
        }
        node = nearChild;
        tmax = tplane;
    }
}
if (!node) {
    ⟨Grab next node to process from todo list⟩
}
```

Assert 498
Ray 26

⟨*Enqueue* farChild *in todo list*⟩≡
```
todo[todoPos].node = farChild;
todo[todoPos].tmin = tplane;
todo[todoPos].tmax = tmax;
++todoPos;
Assert(todoPos < MAX_TODO);
```

Basically just like the usual intersect method, just calls the Primitive's IntersectP() method and returns true as soon as it finds any intersection–doesn't need to worry about waiting for the closest one... (Worth a 3-5% speedup on typical scenes...)

⟨*KdTreeAccelerator Method Declarations*⟩+≡
```
bool IntersectP(const Ray &ray) const;
```

Further Reading

After the introduction of the ray-tracing algorithm, an enormous amount of research was done to try to find effective ways to speed it up, primarily by developing improved ray-tracing acceleration structures. Arvo and Kirk's chapter in *An Introduction to Ray Tracing* summarizes the state of the art as of 1989.

Rubin and Whitted developed the first hierarchical data structures for scene representation for fast ray tracing (RW80). Fujimoto et al were the first to intorduce uniform voxel grids, similar to what we describe in this chapter (FTI86).

Glassner introduced octrees for ray intersection acceleration (Gla84); this approach was more robust to scenes with non-uniform distributions of geometry. Another adaptive approach was the hierarchial bounding volumes of Goldsmith and

Salmon (GS87).

Arvo and Kirk introduced the unifying principle of *meta-hierarchies* (AK87); they showed that by implementing acceleration data structures to conform to the same interface as is used for primitives in the scene, it's easy to mix and match multiple intersection schemes in a scene without needing to have particular knowledge of it.

Sung and Shirley describe the implementation of a BSP tree accelerator in in *Graphics Gems III* (SS92); our `KdTreeAccelerator` is loosely based on their implementation.

Revelles octree traversal, including pointers to previous approaches (RUL).

Kay Kajiya (KK86).

Snyder and Barr nested grids, various improvements like ray bounding box (SB87).

Papers by Woo, Pearce, etc. with additional clever tricks

Ray Tracing News full of discussion, tricks of the trade.

Who came up with mailboxing?

Exercises

4.1 try using bounding box tests to improve the grid's performace. what about testing the ray against an object's world-space bound before testing it against the object? or transform to object space and then test against object-space bound (likely to be a better test). can avoid the transformation in the first case, but will generally reject more in the second.

general trade-off in these sorts of culling schemes of how successful is the extra test and how much time does it take, versus how much time does it take to just test against the object anyway.

4.2 implement ray bound in each voxel; then check for overlap of ray bound with world bound of the objects first–very cheap test...

4.3 when sub-grids, we finish isecting in subgrid before continuing current cell: may spend time on unneeded way far away isections!

4.4 discuss statistics above; number of intersection tests per ray and number of intersections found per ray are the big key ones. general tradeoff, though, of improving those values at the expense of more complex traversal schemes, etc.

4.5 explain teapot in a stadium problem.. then,

Extend the grid accelerator so that it is *hierarchical*: for any grid cell that has more than a fixed number of primitives, generate a new grid inside that cell and re-grid the overlapping primitives. Investigate the performance of this scheme.

note that if child grids have power of 2 size with respect to parent, can re-use DDA values just by scaling them appropriately.

4.6 Implement additional ray intersection acceleration schemes. How does performance compare to the regular grid?

4.7 smarter overlap tests for bounding structures–bounding box slop causes inefficiency for both grids and kd-trees. For to handle both of those, could add a `bool Shape::Overlaps(const BBox &) const` method that takes a world-space bounding box. Default could get world bound from primitive, do overlap, smarter ones could be smarter. Would work well for both of the ones here...

4.8 fix the kd tree so that it doesn't refine all primitives immediately. easy is to build sub kd-trees as needed, though this isn't optimal, since same problem of finding far away intersections before checking closer stuff. Better is to re-build sub-trees as needed when refinement is done.

4.9 smarter splitting axis/split position selection for kd trees?

5. Color and Radiometry

In order to set the stage for describing how light ing the scene is represented and sampled to compute images, we will first establish some background in *radiometry*. Radiometry is the area of study of the propagation of electromagnetic radiation in environments. The wavelengths of electromagnetic radiation between roughly 370nm and 730nm account for light visible to the human visual system and are of particular interest in rendering. The lower wavelengths, $\lambda \approx 400nm$ are the blue-ish colors, the middle wavelengths $\lambda \approx 550nm$ are the greens, and the upper wavelengths $\lambda \approx 650nm$ are the reds.

We will introduce four key radiometric quantities–flux, intensity, irradiance, and radiance–that describe electromagnetic radiation. By evaluating the amount of radiation arriving on the camera's image plane, we can model the process of image formation. These radiometric quantities generally vary according to wavelength. Such quantities are generally described by a *spectral power distribution (SPD)*, which is a function of wavelength, λ. This chapter starts by describing the `Spectrum` class, including its operations, that `lrt` uses to represent SPDs throughout the system. We will then introduce basic concepts of radiometry and some theory behind light scattering from surfaces.

For now, we will ignore the effects of smoke, fog, and atmospheric scattering and assume that the scene is a collection of surfaces in a vacuum. Radiometric principles for the more general case will be introduced in Chapter 13.

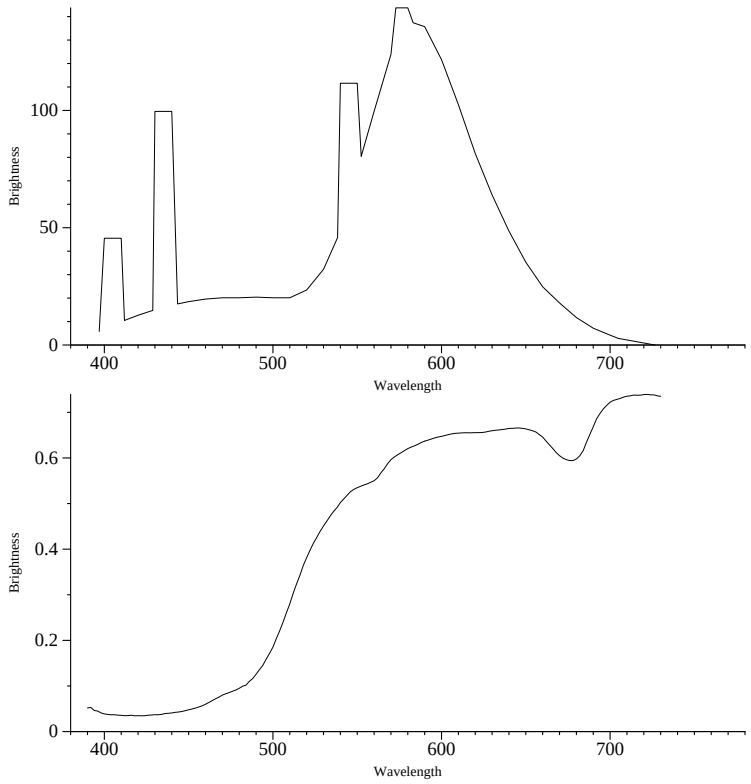


Figure 5.1: Spectral power distributions of a fluorescent light (top) and the reflectance of lemon skin (bottom). Wavelengths around 400nm are blue-ish colors, greens and yellows are in the middle range of wavelengths, and reds have wavelengths around 700nm. The fluorescent light's SPD is even spikier than shown here, where the SPDs have been binned into 10nm ranges; it emits much of its illumination at single frequencies.

5.1 Spectral Representation

⟨*color.h**⟩≡
  ⟨*Source Code Copyright*⟩
  `#ifndef COLOR_H`
  `#define COLOR_H`
  `#include "lrt.h"`
  ⟨*Color Declarations*⟩
  `#endif // COLOR_H`

⟨*color.cc**⟩≡
  ⟨*Source Code Copyright*⟩
  `#include "color.h"`
  ⟨*Spectrum Method Definitions*⟩

The SPDs of real-world objects can be quite complex; Figure 5.1 shows a graph of the spectral distribution of emission from a fluorescent light and the spectral distribution of the reflectance of lemon skin. Given such complex functions, we would like a compact, efficient, and accurate way to represent them. A number

of approaches have been developed that are based on finding good *basis functions* to represent complex SPDs. The idea behind basis functions is to map the infinite-dimensional space of possible SPD functions to a low-dimensional space of coefficients $c_i \in \mathbb{R}$. For example, a trivial basis function is the constant function $B(\lambda) = 1$. An arbitrary SPD would be represented by a single coefficient c equal to its average value, so that its basis function approximation would be $cB(\lambda) = c$. This is obviously a poor approximation, since it has no chance to account for the SPD's possible complexity.

It is often convenient to limit ourselves to *linear basis functions*; this means that the basis functions are pre-determined functions of wavelength and aren't themselves parameterized. For example, if we were using Gaussians as basis functions and wanted to have a linear basis, we need to set their respective widths and central wavelengths ahead of time. If we allowed the widths and center positions to vary based on the SPD we were trying to fit, the basis would be non-linear. Though non-linear basis functions can naturally adapt to the complexity of SPDs, they are in general less computationally efficient. Because it is not a primary goal of `lrt` to provide the most comprehensive spectral representations, we will only implement infrastructure for linear basis functions.

Given a set of linear basis functions B_i, coefficients c_i for a SPD $S(\lambda)$ can be computed by

$$c_i = \int_\lambda B_i(\lambda) S(\lambda)\, \mathrm{d}\lambda, \tag{5.1.1}$$

so that

$$S(\lambda) \approx \sum_i c_i B_i(\lambda).$$

Measured SPDs of real-world objects are often given in 10nm increments; this corresponds to basis functions that are step functions:

$$B(\lambda)_{a,b} = \begin{cases} 1 & : \quad a \leq \lambda < b \\ 0 & : \quad \text{otherwise} \end{cases}$$

Another common basis function is the delta function that evaluates the SPD at single wavelengths. Others that have been investigated include polynomials and Gaussians.

Given an SPD and its associated set of linear basis function coefficients, a number of operations on the spectral distributions can be easily expressed directly in terms of the coefficients. For example, to compute the coefficients c_i' for the SPD given by multiplying a scalar k with a SPD $S(\lambda)$, where the coefficients for $S(\lambda)$ are c_i, we have:

$$\begin{aligned} c_i' &= \int_\lambda B_i(\lambda)\,(kS(\lambda))\, \mathrm{d}\lambda \\ c_i' &= k \int_\lambda B_i(\lambda) S(\lambda)\, \mathrm{d}\lambda \\ c_i' &= k c_i \end{aligned}$$

Such a multiplication might be used to adjust the brightness of a light source. Similarly, for two SPDs $S_1(\lambda)$ and $S_2(\lambda)$ represented by coefficients c_i^1 and c_i^2, the sum $S_1(\lambda) + S_2(\lambda)$ can be shown to be

$$c_i' = \sum c_i^1 + c_i^2.$$

Thus, by converting to a basis function representation, a number of otherwise potentially-tricky operations with SPDs are made straightforward.

We will often need to multiply two SPDs together; for example, the product of the SPD of light arriving from a light with the SPD for a surface's reflectance gives the SPD of light reflected from the surface. In general, the coefficients for the SPD representing product of two SPDs doesn't work out quite so cleanly, even with linear basis functions:

$$\begin{aligned} c_i &= \int_\lambda B_i(\lambda)\,(S_1(\lambda)S_2(\lambda))\,\mathrm{d}\lambda \\ &\approx \int_\lambda B_i(\lambda)\left(\sum_j c_j^1 B_j(\lambda)\right)\left(\sum_k c_k^2 B_k(\lambda)\right)\mathrm{d}\lambda \\ &= \sum_j \sum_k c_j^1 c_k^2 \int_\lambda B_i(\lambda)B_j(\lambda)B_k(\lambda)\mathrm{d}\lambda \end{aligned}$$

The integrals of the product of the three basis functions can be precomputed and stored in n matrices of size n^2 each, where n is the number of basis functions. Thus, n^3 multiplications are necessary to compute the new coefficients. Alternatively, If one of the colors is known ahead of time (e.g. a surface's reflectance), we can precompute an matrix S defined so that the $S_{i,j}$ element is

$$S_{i,j} = \int_\lambda S_1(\lambda)B_i(\lambda)B_j(\lambda).$$

Then, multiplication with another SPD is just a matrix-vector multiply with S and the vector c_i^2, requiring n^2 multiplications.

In `lrt`, we will choose computational efficiency over generality and further limit the supported basis functions to be orthonormal. This means that for $i \neq j$,

$$\int_\lambda B_i(\lambda)B_j(\lambda)\mathrm{d}\lambda = 0$$

and

$$\int_\lambda B_i(\lambda)B_i(\lambda)\mathrm{d}\lambda = 1.$$

Under these assumptions, the coefficients for the product of two SPDs is just the produce of their coefficients

$$c_i = c_i^1 c_i^2,$$

requiring n multiplications.

XXX need to note, though, that the coefficients for the product of two SPDs will not in general have the same values as the products of their coefficients:

$$\int_\lambda B_i(\lambda)S_1(\lambda)S_2(\lambda)\mathrm{d}\lambda \neq \int_\lambda B_i(\lambda)\left(\sum_j c_j^1 B_j(\lambda)\right)\left(\sum_k c_k^2 B_k(\lambda)\right)\mathrm{d}\lambda.$$

This is a natural consequence of both the error introduced i nthe original transformation to a basis functino representatino as well as the need to reproject the result of the multiplication onto the basis functions..

Other than requiring that the basis functions used be linear and orthonormal, `lrt` places no further restriction on them. In fact, `lrt` operates purely on basis function

coefficients: colors are specified in input files and texture maps as coefficients and `lrt` can write out images of coefficients–almost no knowledge of the form of the particular basis functions being used is needed by the system.

Spectrum Class

The `Spectrum` class holds a compile-time fixed number of basis function coefficients, given by `COLOR_SAMPLES`.

⟨*Global Constants*⟩+≡
```
#define COLOR_SAMPLES 3
```

⟨*Color Declarations*⟩≡
```
class Spectrum {
public:
    ⟨Spectrum Constructor Declarations⟩
    ⟨Spectrum Method Declarations⟩
    ⟨Spectrum Public Data⟩
private:
    ⟨Spectrum Private Data⟩
    Float c[COLOR_SAMPLES];
};
```

Two `Spectrum` constructors are provided, one initializing a spectrum with the same value for all coefficients, and one initializing it with `COLOR_SAMPLES` given coefficients.

⟨*Spectrum Constructor Declarations*⟩≡
```
Spectrum(Float intens = 0.) {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        c[i] = intens;
}
```

⟨*Spectrum Constructor Declarations*⟩+≡
```
Spectrum(Float cs[COLOR_SAMPLES]) {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        c[i] = cs[i];
}
```

A variety of arithmetic operations on `Spectrum` objects are supported; the implementations are all quite straightforward. First are operations to add pairs of spectral distributions.

⟨*Spectrum Method Declarations*⟩+≡
```
Spectrum &operator+=(const Spectrum &s2) {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        c[i] += s2.c[i];
    return *this;
}
```

⟨*Spectrum Method Declarations*⟩+≡
```
Spectrum operator+(const Spectrum &s2) const {
    Spectrum ret = *this;
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        ret.c[i] += s2.c[i];
    return ret;
}
```

Similarly, subtraction, multiplication and division of spectra is defined componentwise. We won't include all of the code for those cases, or for multiplying or dividing them by scalar values, since there's little additional value to seeing it all.

We also provide the obvious equality test.

⟨*Spectrum Method Declarations*⟩+≡
```
bool operator==(const Spectrum &sp) const {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        if (c[i] != sp.c[i]) return false;
    return true;
}
```

⟨*Spectrum Method Declarations*⟩+≡
```
bool Black() const {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        if (c[i] != 0.) return false;
    return true;
}
```

Spectrum 155

Also useful are functions that take the square-root of a spectrum and raise the components of a `Spectrum` to a given power, also given as a `Spectrum`. Because the product of two spectra is computed with products of their coefficients, taking the square root of the coefficients gives the square root of the SPD.

Needed for Fresnel formulas..

⟨*Spectrum Method Declarations*⟩+≡
```
Spectrum Sqrt() const {
    Spectrum ret;
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        ret.c[i] = sqrtf(c[i]);
    return ret;
}
```

Needed for some BRDF models...

⟨*Spectrum Method Declarations*⟩+≡
```
Spectrum Pow(const Spectrum &e) const {
    Spectrum ret;
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        ret.c[i] = c[i] > 0 ? powf(c[i], e.c[i]) : 0.f;
    return ret;
}
```

5.2 Basic Radiometry

Radiometry gives us a set of ideas and mathematical tools to describe light propagation and reflection in environments; it forms the basis of the derivation of the rendering algorithms that will be used throughout the rest of this book. Interestingly enough, radiometry wasn't originally directly derived from first principles using the basic physics of light, but was based on an abstraction of light based on particle flows. As such, effects like polarization of light aren't naturally a part of it, though connections have since been made between it and Maxwell's equations, giving it a solid basis in physics.

Radiative transfer is the phenomenological study of the transfer of radiant energy. It is based on radiometric principles and operates at the *geometrical optics* level, where macroscopic properties of light suffice to describe how light interacts with objects much larger than the wavelength of light, it is not at all uncommon to incorporate results from wave optics models. These results just need to be expressed in the language of radiative transfer's basic abstractions.[1] In this manner, it is possible describe interactions of light with objects close to the wavelength and this describe effects like dispersion and interference. At an even finer level of detail, quantum mechanics is needed to describe light's interaction with atoms; as direct simulation of quantum mechanical principles is unnecessary for solving rendering problems in computer graphics, the problem of the intractability of such an approach is avoided anyway.

In `lrt`, we will assume that geometrical optics are an adequate basis for the description of light and light scattering. As such, we will make following assumptions about how the behavior of light.

- *Linearity*: the combined effect of two inputs to an optical system is always equal to the sum of the effects of each of the inputs individually.
- *Energy conservation*: more energy is never produced by a scattering event than there was to start with.
- *No polarization*: we will ignore polarization of the electromagnetic field; as such, the only relevant property of light particles is their wavelength (or frequency). While the radiative transfer framework has been extended to include the effects of polarization, we will ignore this effect for simplicity.
- *No fluorescence or phosphorescence*: we make the assumption that the behavior of light at one wavelength is completely independent of light's behavior at other wavelengths. As with polarization, it is relatively straightforward to include these effects in this work, but largely serves to make the presentation more complex, with little practical advantage.
- *Steady state*: light in the environment is assumed to have reached equlibrium, such that its radiance distribution ins't changing with time. This happens nearly instantaneously with light in realistic scenes.

[1] Preisendorfer has connected radiative transfer theory to Maxwell's classical equations describing electromagnetic fields (Pre65, Chapter 14); his framework both demonstrates their equivalence and makes it easier to apply results from one world-view to the other. More recent work was done in this area by Fante (**?**).

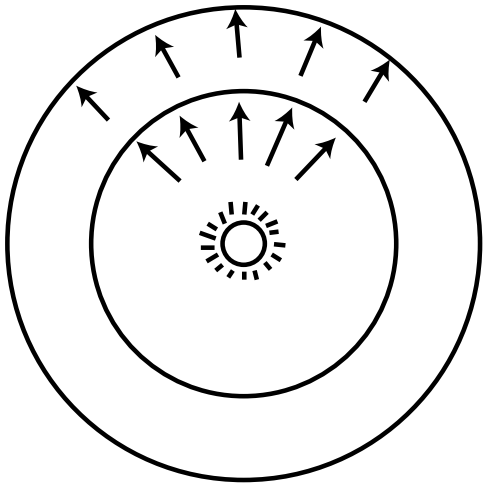


Figure 5.2: Radiant flux, Φ, measures energy passing through a surface or region of space. Here, flux from a point light source is being measured at a sphere that surrounds the light.

The most significant loss from assuming geometrical optics is that diffraction and interference effects cannot easily be accounted for. As noted by Preisendorfer, this is hard to fix given these assumptions because, for example, the total flux over two areas isn't necessarily equal to sum of flux over each individually (Pre65, p. 24).

Basic quantities

There are four radiometric quantities that are central to rendering:

- flux
- irradiance
- intensity
- radiance

All of these quantities are generally functions that vary by wavelength, λ. For the remainder of this chapter, we will not make this dependence explicit, but it's important to keep in mind.

Radiant flux, also known as *power*, is the total amount of energy passing through a surface or region of space per unit time. Its units are Joules/second and it is normally signified by the symbol Φ. Total emission from light sources is generally described in terms of flux; Figure 5.2 shows flux from a point light measured by the total amount of energy passing through the imaginary sphere around the light. Note that the amount of flux measured on either of the two spheres in Figure 5.2 is the same–although less energy is passing through any local part of the large sphere than the small sphere, the greater area of the large sphere accounts for this.

Irradiance (E) is the area density of flux, flux/square meter. For the point light example in Figure 5.2, irradiance on the outer sphere is less than the irradiance on

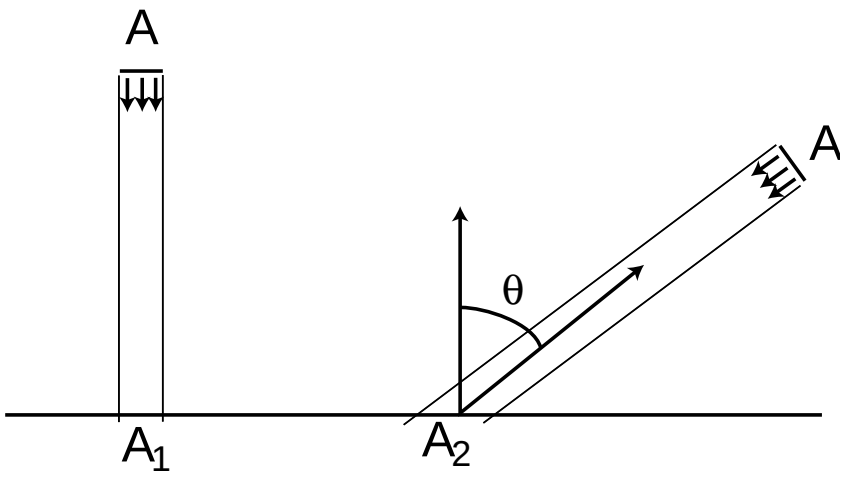


Figure 5.3: Irradiance (E) arriving at a surface varies according to the cosine of the angle of incidence of illumination, since illumination is over a larger area at lower incident directions. This effect was first described by Lambert; it is known as Lambert's Law.

the inner sphere, since the area on the outer sphere is larger. In particular, for a sphere in this configuration that has radius r,

$$E = \frac{\Phi}{4\pi r^2}.$$

This falloff with distance explains why received energy from a light falls off with the squared distance from the light.

The irradiance equation can also help us understand the origin of *Lambert's Law*, which says that the amount of light arriving at a surface is related to the cosine of the angle between the light direction and the surface normal–see Figure 5.3. Consider a light source with area A and flux Φ that is shining on a surface. If the light is shining directly down on the surface (left), then the area on the surface receiving light A_1 is equal to A and irradiance at any point inside A_1 is

$$E_1 = \frac{\Phi}{A}.$$

However, if the light is at an angle to the surface (right), the total area on the surface receiving light is larger. If the area of the light source is small, then the area receiving flux, A_2, is roughly $A/\cos\theta$. For points inside A_2, the irradiance is therefore

$$E_2 = \frac{\Phi \cos\theta}{A}.$$

This is the origin of the cosine law for radiance.

More formally, to cover the cases like when the emitted flux distribution isn't constant, irradiance at a point is actually defined as

$$E = \frac{d\Phi}{dA}, \tag{5.2.2}$$

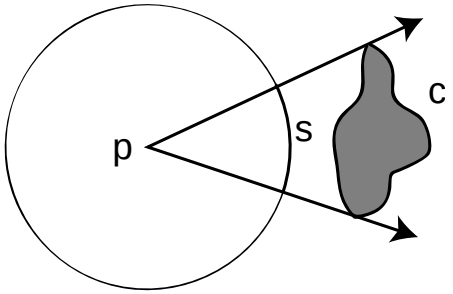


Figure 5.4: The plane angle of an object *c* as seen from a point *p* is equal to the angle it subtends as seen from *p*, or equivalently as the length of the arc *s* on the unit sphere.

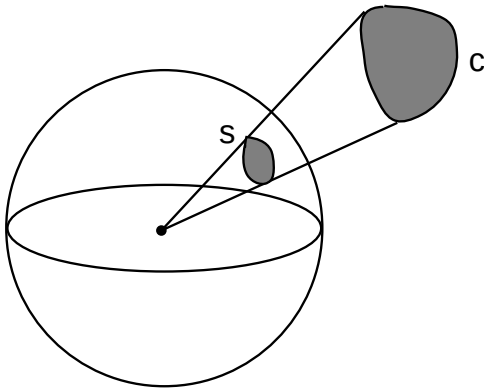


Figure 5.5: The solid angle *s* subtended by an object *c* in three dimensions is similarly computed by projecting *c* onto the unit sphere and measuring its area there.

where the differential flux from the light is computed at the differential point receiving flux.

In order to define the radiometric quantity intensity, we first need to define the notion of the *solid angle*. Solid angles are just the extension of two-dimensional angles in a plane to angle on a sphere. The *plane angle* is the total angle subtended by some object with respect to some position; see Figure 5.4. Consider the unit circle around the point *p*; if we project the shaded object on to that circle, some length of the circle *s* will be covered by its projection. The arc-length of *s* (which is the same as the angle θ) is the angle *subtended* by the object. Plane angle is given the unit *radians*.

The solid angle extends the unit circle in two-dimensions to a unit sphere in three-dimensions (Figure 5.5). The total area *s* is the solid angle subtended by the object. Solid angle is given the unit *steradians*. The entire sphere subtends a solid angle of 4π and a hemisphere subtends 2π.

We will use the symbol $\vec{\omega}$ to describe directions on the unit sphere centered around some point. (These directions can thus also be thought of as point on the unit sphere around *p*. We will therefore follow use the convention that $\vec{\omega}$ is always a normalized vector). We can now define intensity, which is flux density per solid

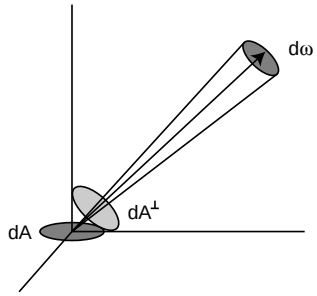


Figure 5.6: Radiance L is defined at a point by the ratio of the differential flux incident along a direction $\vec{\omega}$ to the differential solid angle $\mathrm{d}\vec{\omega}$ times the differential projected area of the receiving point.

angle,

$$I = \frac{\mathrm{d}\Phi}{\mathrm{d}\vec{\omega}}. \tag{5.2.3}$$

Intensity is generally only used when describing the distribution of light by direction from point light sources.

Finally, radiance (L) is the flux density per unit area, per unit solid angle. In terms of flux, it is

$$L = \frac{\mathrm{d}^2\Phi}{\mathrm{d}\vec{\omega}\, dA^{\perp}} \tag{5.2.4}$$

where $dA^{\perp}$ is the projected area of dA on a hypothetical surface perpendicular to $\vec{\omega}$–see Figure 5.6. All those differential terms don't need to be as confusing as they initially appear–just think of radiance as the limit of the measurement of incident light at the surface as a small cone of incident directions of interest $\mathrm{d}\vec{\omega}$ becomes very small, and as the local area of interest on the surface dA also becomes very small.

Now that we have defined these various units, it's easy to derive relations between them. For instance, irradiance at a point x due to radiance over a set of directions Ω is

$$E(x) = \int_{\Omega} L(x, \vec{\omega}) \cos\theta \, \mathrm{d}\vec{\omega}, \tag{5.2.5}$$

where $L(x, \vec{\omega})$ denotes the arriving radiance at position x as seen along direction $\vec{\omega}$ (see Figure 5.7). (The $\cos\theta$ term in this integral is due to the $dA^{\perp}$ term in the definition of radiance.) We are often interested in irradiance over the hemisphere of directions about a given surface normal $\mathcal{H}^2$ or the entire sphere of directions $\mathcal{S}^2$.

5.3 Working with Radiometric Integrals

One of the main tasks in rendering is integrating information about the values of particular radiometric quantities to compute information about other radiometric quantities. There are a few important tricks that can be used to make this task easier.

Integrals over projected solid angle

The various cosine terms in integrals for radiometric quantities can clutter things up and distract from what is being expressed in the integral. There is an different

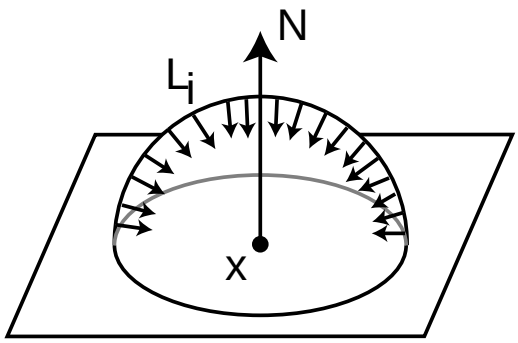


Figure 5.7: Irradiance at a point x is given by the integral of radiance times the cosine of the incident direction over the entire upper hemisphere above the point.

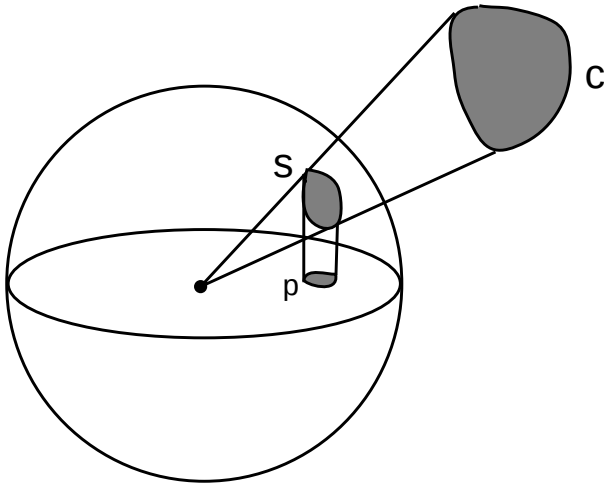


Figure 5.8: The projected solid angle subtended by an object c is the cosine-weighted solid angle that it subtends. It can be computed by finding the object's solid angle s, projecting it down to the plane, and measuring its area there. Thus, the projected solid angle depends on the surface normal where it is being measured, since the normal orients the plane of projection.

way that the integrals can be written that removes this distraction. The *projected solid angle* subtended by an object is determined by projecting the object on to the unit sphere, as is done for solid angle, but then projecting the resulting shape down on to the unit disk–see Figure 5.8. Integrals over hemispheres of directions with respect to solid angle can equivalently be written as integrals over projected solid angles.

The projected solid angle measure is related to the solid angle measure by

$$d\vec{\omega}^{\perp} = \cos\theta \, d\vec{\omega},$$

so the irradiance-from-radiance integral can be written more simply as

$$E = \int_{\mathcal{H}^2} L(\vec{\omega}) \, d\vec{\omega}^{\perp}.$$

For the rest of this book, we'll write integrals over directions in terms of solid angle, rather than projected solid angle. When reading rendering integrals in other

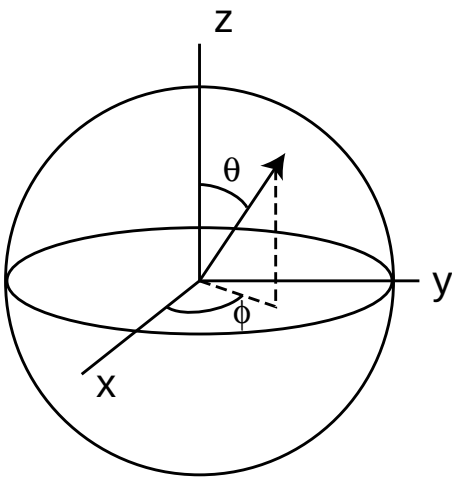


Figure 5.9: A given direction vector can be written in terms of spherical coordinates (θ, ϕ) if the *x*, *y*, and *z* basis vectors are given as well. The spherical angle formulae make it easy to convert between the two representations.

contexts, however, be sure to be aware of the measure of the space that is being integrated over to disambiguate these cases.

16 Vector

Integrals over spherical coordinates

It is often convenient to transform integrals over solid angle into integrals over spherical coordinates (θ, ϕ). Recall that an (x,y,z) direction vector can be alternatively written in terms of spherical angles (see Figure 5.9):

$$\begin{aligned} x &= \sin\theta \cos\phi \\ y &= \sin\theta \sin\phi \\ z &= \cos\theta \end{aligned}$$

For convenience, we'll define two functions that turn θ and ϕ values into (x,y,z) direction vectors. The first applies the equations above directly.

⟨*Geometry Inline Functions*⟩+≡
```
inline Vector SphericalDirection(Float sintheta, Float costheta,
        Float phi) {
    return Vector(sintheta * cosf(phi),
        sintheta * sinf(phi), costheta);
}
```

The second function takes three basis vectors to replace the *x*, *y* and *z* axes and returns the appropriate direction vector with respect to the coordinate frame that they define.

⟨*Geometry Inline Functions*⟩+≡
```
inline Vector SphericalDirection(Float sintheta, Float costheta,
        Float phi, const Vector &x, const Vector &y,
        const Vector &z) {
    return sintheta * cosf(phi) * x +
        sintheta * sinf(phi) * y + costheta * z;
}
```

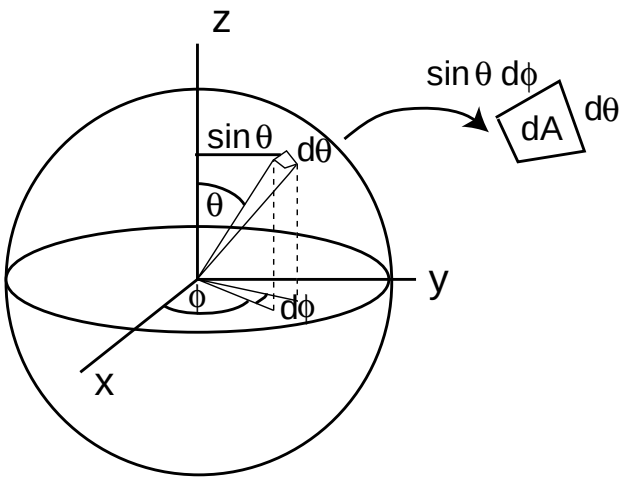


Figure 5.10: The differential area dA subtended by a differential solid angle is the product of the differential lengths of the two edges $\sin\theta d\phi$ and $d\theta$. The resulting relationship, $d\vec{\omega} = \sin\theta d\theta d\phi$, is the key to converting between integrals over solid angles and integrals over spherical angles.

Vector 16

The spherical angles for a direction can be found by:

$$\begin{aligned} \theta &= \arccos z \\ \phi &= \arctan \frac{y}{z} \end{aligned}$$

Corresponding functions are below. Note that `SphericalTheta()` assumes that the vector `v` has been normalized before being passed in.

⟨*Geometry Inline Functions*⟩+≡
```
inline Float SphericalTheta(const Vector &v) {
    return acosf(v.z);
}
```

⟨*Geometry Inline Functions*⟩+≡
```
inline Float SphericalPhi(const Vector &v) {
    return atan2f(v.y, v.x) + M_PI;
}
```

In order to write an integral over solid angle in terms of an integral over (θ, ϕ), we need to be able to express the relationship between the differential area of a set of directions $d\vec{\omega}$ and the differential area of a (θ, ϕ) pair–see Figure 5.10. The differential area $d\vec{\omega}$ is the product of the differential lengths of the sides of $d\vec{\omega}$, $\sin\theta d\phi$ and $d\theta$. Therefore,

$$d\vec{\omega} = \sin\theta \, d\theta \, d\phi.$$

We can thus see that the irradiance integral over the hemisphere (Equation 5.2.5 with $\Omega = \mathcal{H}^2$) can equivalently be written

$$E = \int_0^{2\pi} \int_0^{\pi/2} L(x, \theta, \phi) \cos\theta \sin\theta \, d\theta \, d\phi$$

So if the radiance is the same from all directions, this simplifies to $E = \pi L$.

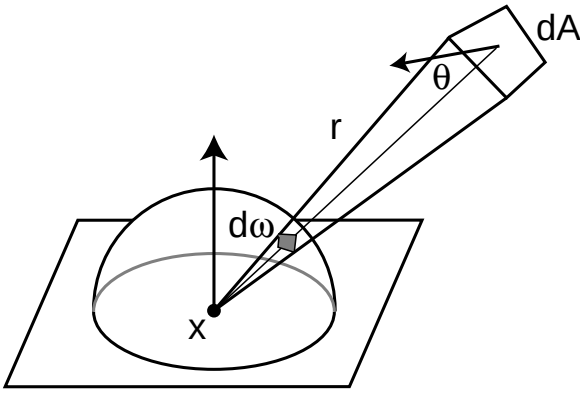


Figure 5.11: The differential solid angle subtended by a differential area dA is equal to $\mathrm{d}A\cos\theta/r^2$, where θ is the angle between dA's surface normal and the vector to the point x and r is the distance from x to dA.

Just as we found irradiance in terms of incident radiance, we can also compute the total flux emitted from some object over the hemisphere about the normal by integrating over the object's surface area A:

$$\Phi = \int_A \int_{\mathcal{H}^2} L(x, \vec{\omega}) \cos\theta \, \mathrm{d}\vec{\omega} \, \mathrm{d}A$$

Integrals over area

One last transformation of integrals that can be simplify computation is to turn integrals over directions into integrals over area. Consider the irradiance integral, 5.2.5 again, where there is a quadrilateral with constant outgoing radiance and where we'd like to compute the resulting irradiance at a point x. The easiest way to write this integral is as an integral over the area of the quadrilateral; writing it as an integral over directions is less straightforward, since given a particular direction, the computation to determine if the quadrilateral is visible in that direction is non-trivial.

Differential area is related to differential solid angle by

$$\mathrm{d}\vec{\omega} = \frac{\mathrm{d}A \cos\theta}{r^2} \tag{5.3.6}$$

where θ is the angle between the surface normal of dA and r^2 is the squared distance from x to dA. See Figure 5.11.

We will not derive this result here, but it can be understood intuitively: if dA is distance 1 from x and is aligned exactly so that it is facing down d$\vec{\omega}$, then $\mathrm{d}\vec{\omega} = \mathrm{d}A$, $\theta = 0$, and Equation 5.3.6 holds. As dA moves farther away from x, or as it rotates so that it's not aligned with the direction of d$\vec{\omega}$, the r^2 and $\cos\theta$ terms compensate accordingly to reduce d$\vec{\omega}$.

Therefore, we can write the irradiance integral for the quadrilateral source as

$$E(x) = \int_A L \cos\theta_i \frac{\cos\theta_o \, \mathrm{d}A}{r^2}$$

where θ_i is the angle between the surface normal at x and the direction from x to the point x' on the light, and θ_o is the angle between the surface normal at x' on the light and the direction from x' to x (see Figure 5.12.)

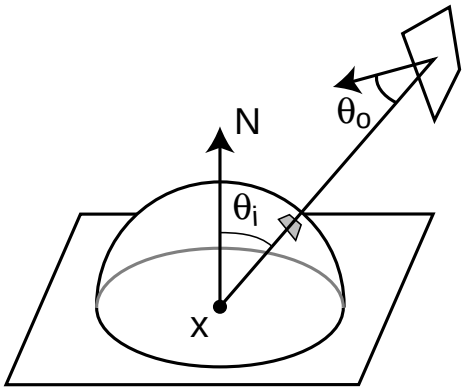


Figure 5.12: To compute irradiance at a point x from a quadrilateral source, it's easier to integrate over the surface area of the source than to integrate over the irregular set of directions that it subtends. The relationship between solid angles and areas given by Equation 5.3.6 lets us go back and forth between the two approaches.

5.4 Surface Reflection

When light in an environment is incident on a surface, the surface scatters the light, re-reflecting some of it back into the environment. For example, the skin of a lemon mostly absorbs light in the blue wavelengths, but reflects most of light in the red and green wavelengths (recall the lemon skin reflectance SPD in Figure 5.1.) Therefore, when it is illuminated with white light, its color is yellow. The skin has pretty much the same color no matter what direction it's being observed from, although for some directions a highlight is visible, where it is more white than yellow.

In contrast, the color seen in a mirror depends almost entirely on the viewing direction. At a fixed point on the mirror, as the viewing angle changes, the object that is reflected in the mirror changes accordingly. Furthermore, mirrors generally don't change the color of the object they are reflecting very much.

The BRDF

There are a few concepts in radiometry that give formalisms for describing these types of reflection. One of the most important is the *bidirectional reflectance distribution function*, (BRDF). Consider the setting in Figure 5.13: we'd like to know how much radiance is leaving the surface in the direction $\vec{\omega}_o$ toward the viewer, $L_o(\vec{\omega}_o)$ as a result of incident radiance along the direction $\vec{\omega}_i$, $L_i(\vec{\omega}_i)$.

If the direction $\vec{\omega}_i$ is considered a differential cone of directions, we can compute the resulting differential irradiance at x by

$$dE(\vec{\omega}_i) = L(\vec{\omega}_i) \cos\theta_i \, d\vec{\omega}_i.$$

A differential amount of radiance will be reflected in the direction $\vec{\omega}_o$. An important assumption made in radiometry is that the system is linear: doubling the amount of energy going into it will lead to a doubling of the amount going out of it. This is a reasonable assumption as long energy levels are not extreme.

Therefore, the reflected differential radiance is

$$dL_o(\vec{\omega}_o) \propto dE(\vec{\omega}_i).$$

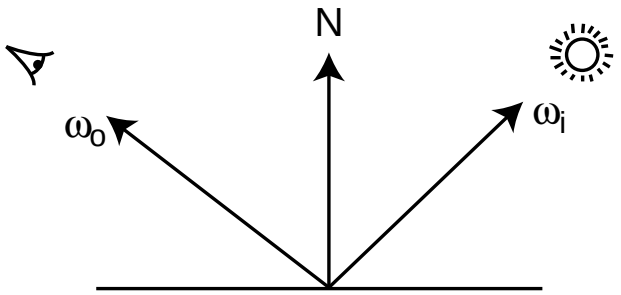


Figure 5.13: The bidirectional reflectance distribution function (BRDF) is a four-dimensional function over pairs of directions $\vec{\omega}_i$ and $\vec{\omega}_o$ that describes how much incident light along $\vec{\omega}_i$ is scattered from the surface in the direction $\vec{\omega}_o$.

The constant proportionality for the particular pair of directions $\vec{\omega}_i$ and $\vec{\omega}_o$ is defined to be the surface's BRDF:

$$f_r(\vec{\omega}_i, \vec{\omega}_o) = \frac{dL(\vec{\omega}_o)}{dE(\vec{\omega}_i)} = \frac{dL(\vec{\omega}_o)}{L(\vec{\omega}_i)\cos\theta_i d\vec{\omega}_i} \tag{5.4.7}$$

Real-world BRDFs have two important qualities:

1. *Reciprocity*: for all pairs of directions $\vec{\omega}_i$ and $\vec{\omega}_o$, $f_r(\vec{\omega}_i, \vec{\omega}_o) = f_r(\vec{\omega}_o, \vec{\omega}_i)$.
2. *Energy conservation*: the total energy of light reflected is less than or equal to the energy of incident light. For all directions $\vec{\omega}$, $\int_{S^2} f(\vec{\omega}_i, \vec{\omega}) \cos\theta_i d\vec{\omega}_i \leq 1$.

The surface's *bidirectional transmittance distribution function* (BTDF) can be defined in a similar manner to the BRDF. The BTDF is generally denoted by $f_t(\vec{\omega}_i, \vec{\omega}_o)$, where $\vec{\omega}_i$ and $\vec{\omega}_o$ are in opposite hemispheres around x. Interestingly enough, the BTDF does not obey reciprocity; we will discuss this in detail in Section 9.2.

For convenience in equations, we will denote the BRDF and BTDF considered together as $f(\vec{\omega}_i, \vec{\omega}_o)$; we will call this the *bidirectional scattering distribution function* (BSDF). Chapter 9 is entirely devoted to describing BSDFs that are used in graphics.

Using the definition of the BSDF, we have

$$dL_o(\vec{\omega}_o) = L_i(\vec{\omega}_i)\, f(\vec{\omega}_i, \vec{\omega}_o) \cos\theta_i d\vec{\omega}_i.$$

We can integrate this over the sphere of incident directions around x to compute the outgoing radiance in direction $\vec{\omega}_o$ due to the incident illumination at x:

$$L_o(\vec{\omega}_o) = \int_{S^2} L_i(\vec{\omega}_i)\, f(\vec{\omega}_i, \vec{\omega}_o)\, |\cos\theta_i| d\vec{\omega}_i \tag{5.4.8}$$

This is a fundamental equation in rendering; it describes how an incident distribution of light at a point is transformed into an outgoing distribution, based on the scattering properties of the surface. It is often called the *reflectance equation*, when just the upper hemisphere $\mathcal{H}^2$ is being integrated over, or the *scattering equation* when the sphere $\mathcal{S}^2$ is the domain, as it is here.

Further Reading

Hall's book summarizes the state-of-the-art in spectral representations through 1989 (Hal89) and Glassner's *Principles of Digital Image Synthesis* covers the topic through the mid-90s (Gla95). Meyer was the one of the first researchers to closely investigate spectral representations in graphics; XXX. Later, Raso and Fournier proposed a polynomial representation for spectra (RF91).

Our discussion of SPD representation with basis functions is based on Peercy's 1993 SIGGRAPH paper (Pee93). In that paper, Peercy chose particular basis functions in a scene-dependent manner: by looking at the SPDs of the lights and reflecting objects in the scene, a small number of basis functions that could accurately represent the scene's SPDs were found using characteristic vector analysis.

Another approach to spectral representation was investigated by Sun et al; they partitioned SPDs into a smooth base SPD and a set of spikes (SFDC01). Each part was represented differently, using basis functions that worked well for each particular type of function.

He and Stam have use wave optics stuff in graphics (HTSG91; Sta99). Also cite appropriate part of Preisendorfer and Chandrasekhar.

Arvo has investigated the connection between rendering algorithms in graphics and previous work in *transport theory*, which applies classical physics to particles and their interactions to predict their overall behavior and global illumination algorithms (**?**; **?**).

XXX where to get real-world SPD data

McCluney's book on radiometry (McC94) is an excellent introduction to the topic. Preisendorfer also covers radiometry in an accessible manner and delves into the relationship between radiometry and the physics of light (Pre65). Moon and Spencer's books (MS36; **?**) and Gershun's article (Ger39) are classic early introductions to radiometry. Lambert's seminal early writings about photometry from the mid-186h Century were recently translated by DiLaura (Lam01).

Exercises

5.1 Experiment with different basis functions for spectral representation. How many coefficients are needed for accurate rendering of tricky situations like fluorescent lighting? How much does the particular choice of basis affect the number of coefficients needed?

5.2 Generalize the `Spectrum` class so that it's not limited to orthonormal basis functions. Implement Peercy's approach of choosing basis functions based on the main SPDs in the scene. Does the improvement in accuracy make up for the additional computational expense of computing the products of spectra.

5.3 Generalize the `Spectrum` class further to support non-linear basis functions. Compare the results to more straightforward spectral representations.

5.4 Compute the irradiance at a point due to a square quadrilateral with outgoing radiance of 10 J/m^2 sr that has sides of length 1 that is 5 units directly above it along its surface normal.

5.5 Similarly, compute irradiance at a point due to a unit-radius disk 3 units directly above its normal with constant outgoing radiance of 10 $J/m^2\,sr$. Do the computation twice, once as an integral over solid angle and once as an integral over area. (Hint: if the results don't match and you write the integral over the disks' area as an integral over radius r and an integral over angle θ, see Section XXX in the Monte Carlo chapter for a hint about XXXXXX.)

6. Camera Models and Film

In addition to describing the objects that make up the scene, we also need to describe how the scene is viewed and how its three-dimensional representation is imaged into a two-dimensional image. We will start by presenting the `Camera` class, which generates of rays from the camera that sample the scene to generate the image. By generating these rays in various ways, we can create many types of images of the same 3D scene. We will then show a few implementations of different particular types of cameras, each of which generates rays in a different way.

The point of the camera is to generate an image of the scene, so we next describe the `Film` class that handles storage of the resulting image. We wrap up by describing the imaging pipeline, which handles conversion of image pixel values stored by the film into final output values for display or storage in a file.

6.1 Camera Model

⟨*camera.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef CAMERA_H
#define CAMERA_H
#include "lrt.h"
#include "color.h"
#include "sampling.h"
#include "geometry.h"
#include "transform.h"
⟨Camera Declarations⟩
#endif // CAMERA_H
```

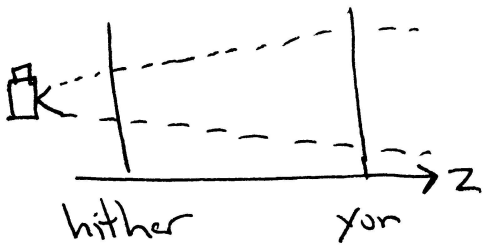


Figure 6.1: The camera's clipping planes give the range of space along the *z* axis that will be images; objects in front of the hither plane or beyond the yon plane will not be visible in the image. Setting the clipping planes to tightly encompass the objects in the scene is important for many scanline algorithms, but is less important for ray-tracing.

⟨*camera.cc**⟩≡
⟨*Source Code Copyright*⟩
```
#include "lrt.h"
#include "camera.h"
#include "film.h"
#include "mc.h"
```
⟨*Camera Method Implementations*⟩

film 173

We will define an abstract `Camera` base class that holds options that are used to specify generic camera parameters and that defines the interface that concrete camera implementations need to provide. The main method that camera subclasses need to implement is `GenerateRay()`, which was previously defined in Section 1.5.

The base `Camera` constructor takes a number of parameters that are appropriate for all camera types. They include the transformation that places the camera in the scene and near and far *clipping planes*, which give distances along the camera space *z* axis that delineate the scene being rendered. Any geometric primitives in front of the near plane or beyond the far plane will not be rendered; see Figure 6.1.

Real-world cameras have a shutter that opens for a short period of time to expose the film to light; one result of this non-zero exposure time is that objects that move during the film exposure time are blurred; this effect is called *motion blur*. To model this effect in `lrt`, each ray has a time value associated with it–by sampling the scene over a range of times, motion can be captured. Thus, all `Camera`s store a shutter open and shutter close time.

⟨*Camera Method Implementations*⟩+≡
```
Camera::Camera(const Transform &world2cam, Float hither, Float yon,
        Float sopen, Float sclose, Film *f) {
    WorldToCamera = world2cam;
    CameraToWorld = WorldToCamera.GetInverse();
    ClipHither = hither;
    ClipYon = yon;
    invClipHither = 1.f / ClipHither;
    ShutterOpen = sopen;
    ShutterClose = sclose;
    film = f;
}
```

⟨*Camera Options*⟩≡
```
Transform WorldToCamera, CameraToWorld;
Float ClipHither, ClipYon, invClipHither;
Float ShutterOpen, ShutterClose;
```

⟨*Camera Public Data*⟩≡
```
Film *film;
```

43 GetInverse
32 Transform

Camera Coordinate Spaces

We have already made use of two important modeling coordinate spaces, object space and world space. We will now introduce three more useful coordinate spaces that have to do with the camera and imaging. Including object and world space, we now have the following. (See Figure 6.2.)

- *Object space*: This is the coordinate system in which geometric primitives are defined. For example, spheres in `lrt` are defined to be centered at the origin of their object space.
- *World space*: While each primitive may have its own object space, there is a single world space that the objects in the scene are placed in relation to. Each primitive has an object to world transformation that determines how it is located in world space. World space is the standard frame that all spaces are defined in terms of.
- *Camera space*: A virtual camera is placed in the scene at some world-space point with a particular viewing direction and "up" vector. This defines new coordinate space around that point with the origin at the camera's location, the z axis is mapped to the viewing direction and the y axis mapped to the up direction. This is a handy space for reasoning about which objects are potentially visible to the camera. For example, if an object's camera-space bounding box is entirely behind the $z = 0$ plane (and the camera doesn't have a field of view wider than 180 degrees), the object will not be visible to the camera.
- *Screen space*: Screen space is defined on the image plane. The camera projects objects in camera space onto the image plane; the parts inside the *screen window* are visible in the image that is generated. Depth z values in

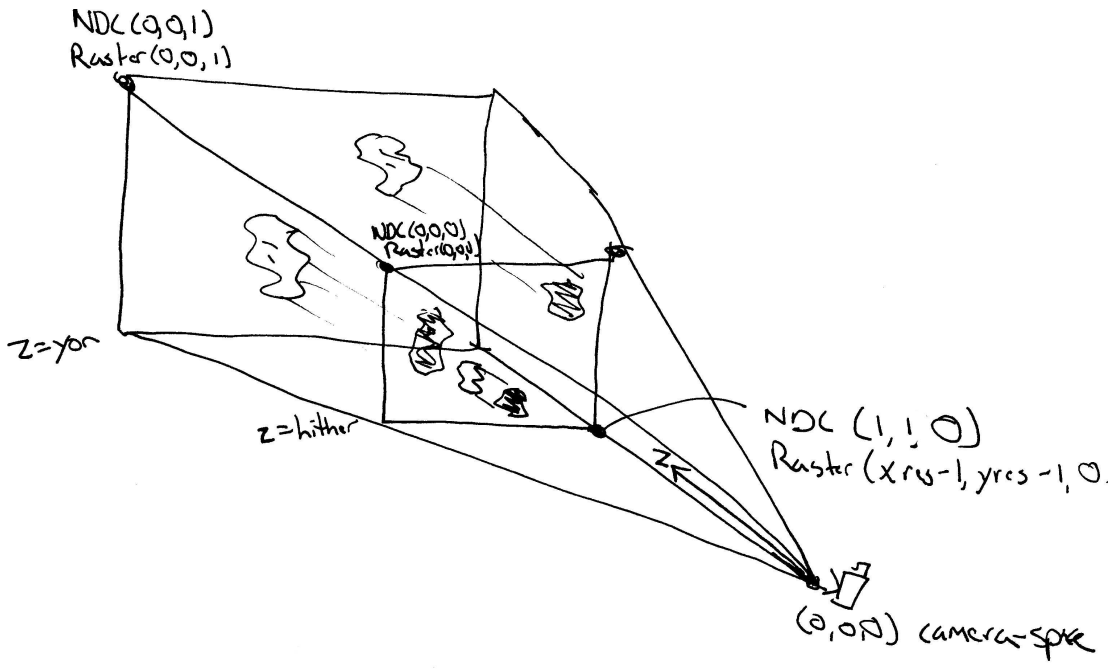


Figure 6.2: A handful of camera-related coordinate spaces help to simplify the implementation of `Camera`s. The camera class holds transformations between them. Scene objects in world space are viewed by the camera, which sits at the origin of camera space and looks down the $+z$ axis. Objects between the hither and yon planes are projected onto the image plane at $z =$ hither in camera space. The image plane is at $z = 0$ in raster space, where x and y range from $(0,0)$ to $(\text{xResolution} - 1, \text{yResolution} - 1)$. Normalized device coordinate (NDC) space normalizes raster space so that x and y range from $(0,0)$ to $(1,1)$.

screen space range from zero to one, corresponding to points at the near and far clipping planes, respectively.

- *Raster space*: Raster space is the coordinate system for the actual image being rendered–in x and y, it ranges from $(0,0)$ to $(\text{xResolution}-1, \text{yResolution}-1)$, the overall image resolution, where $(0,0)$ is the upper left corner of the image. Depth values are the same as in screen space and a linear transformation converts from screen to raster space.

- *NDC Normalized device coordinate* space: this is almost like raster space, except in x and y, the image is normalized to range from $(0,0)$ to $(1,1)$.

All cameras store a world space to camera space transformation; this can be used transform primitives in the scene into camera space. The origin of camera space is the camera's position, and the camera looks down the camera space z axis. The projection cameras in the next section will compute matrices to transform between all of these spaces as needed, but cameras with unusual imaging characteristics can't necessarily represent these transformations with 4x4 matrices.

So that other code can see if a point in the scene lies between the clipping planes, all cameras provide a `ScreenDepth()` function which computes the screen-space z depth of a given point in the scene. Points outside the depth range $[0,1]$ won't appear in the image.

21 Point

⟨*Camera Interface Declarations*⟩+≡
```
virtual Float ScreenDepth(const Point &Pworld) const = 0;
```

6.2 Projective Camera Models

One of the fundamental parts of 3D computer graphics is the *3D viewing problem*: how a three-dimensional scene is projected onto a two-dimensional image for display. Most of the classic approaches can be expressed by a 4x4 projective transformation matrix. Therefore, we will introduce a projection matrix camera class for such cameras and then define two simple camera models. The first of these implements an orthographic projection and the other implements a perspective projection–these are two classic and widely-used projections.

⟨*Camera Declarations*⟩+≡
```
class ProjectiveCamera : public Camera {
public:
    ⟨ProjectiveCamera Method Declarations⟩
protected:
    ⟨ProjectiveCamera Options⟩
};
```

In addition to the world to camera transformation and the projective transformation matrix, the `ProjectiveCamera` takes the screen-space extent of the image, clipping plane distances, a pointer to the `Film` class for the camera, and additional parameters for motion blur and depth of field. `sopen` and `sclose` give times when the camera's shutter opens and closes. If objects in the scene are moving during that time range or if the camera is moving, each ray traced can sample the scene

at a different point in time, such that objects in the image are blurred appropriately. Depth of field, the implementation of which will be shown at the end of this section, simulates blurriness of out-of-focus objects in real lens systems.

⟨*Camera Method Implementations*⟩+≡
```
ProjectiveCamera::ProjectiveCamera(const Transform &w2c,
        const Transform &proj, const Extent2D &Screen,
        Float hither, Float yon, Float sopen,
        Float sclose, Float lensr,
        Float focald, Film *f)
    : Camera(w2c, hither, yon, sopen, sclose, f) {
    ⟨Initialize depth of field parameters⟩
    ⟨Compute projective camera transformations⟩
}
```

The `ProjectiveCamera` implementations pass the projective transformation up to the base class constructor here. This transformation gives us the camera to screen projection; from that we can compute most of the others that we need.

⟨*Compute projective camera transformations*⟩≡
```
CameraToScreen = proj;
WorldToScreen = CameraToScreen * WorldToCamera;
⟨Compute projective camera screen transformations⟩
RasterToCamera = CameraToScreen.GetInverse() * RasterToScreen;
```

Extent2D 27
film 173
GetInverse 43
ProjectiveCamera 175
Scale 36
Transform 32
Translate 35
Vector 16

⟨*ProjectiveCamera Options*⟩≡
```
Transform CameraToScreen, WorldToScreen, RasterToCamera;
```

The only non-trivial one of the precomputed transformations is `ScreenToRaster` note the composition of transformations where (reading backwards), we start with a point in screen space, translate so that the upper left corner of the screen is at the origin, and then scale by one over the screen width and height, giving us a point with x and y coordinates between zero and one. Finally, we scale by the raster resolution, so that we end up covering the raster range from $(0,0)$ up to the overall raster resolution.

⟨*Compute projective camera screen transformations*⟩≡
```
ScreenToRaster = Scale(film->xResolution-1.f, film->yResolution-1.f, 1.f) *
    Scale(1.f / (Screen.x1 - Screen.x0),
          1.f / (Screen.y0 - Screen.y1), 1.f) *
    Translate(Vector(-Screen.x0, -Screen.y1, 0.f));
RasterToScreen = ScreenToRaster.GetInverse();
```

⟨*ProjectiveCamera Options*⟩+≡
```
Transform ScreenToRaster, RasterToScreen;
```

Once we have all of the transformations initialized appropriately, it's easy to compute the screen-space depth of a point in the scene by applying the appropriate transformation.

⟨*Camera Method Implementations*⟩+≡
```
Float ProjectiveCamera::ScreenDepth(const Point &Pworld) const {
    return WorldToScreen(Pworld).z;
}
```

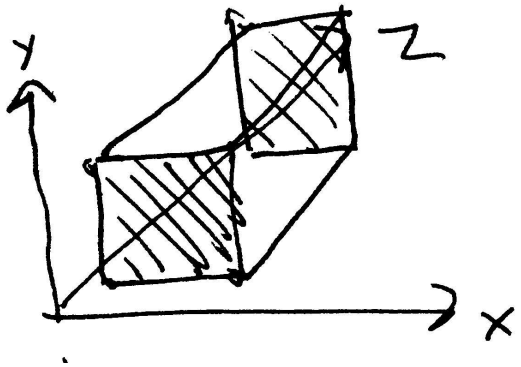


Figure 6.3: The orthographic view volume is an axis-aligned box in camera space, defined such that objects inside the region are projected onto the $z =$ hither face of the box.

Orthographic Camera

⟨*orthographic.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "camera.h"
#include "paramset.h"
⟨OrthographicCamera Declarations⟩
⟨OrthographicCamera Definitions⟩
```

⟨*OrthographicCamera Declarations*⟩≡
```
class OrthoCamera : public ProjectiveCamera {
public:
    ⟨OrthoCamera Method Declarations⟩
};
```

The orthographic transformation takes a rectangular region of the scene and projects it onto the front face of the box that defines the region. It doesn't give the effect of foreshortening–objects becoming smaller on the image plane as they get farther away–but it does leave parallel lines parallel and preserves relative distance between objects. Figure 6.3 shows how this rectangular volume gives the visible region of the scene.

The orthographic camera constructor takes a transform matrix to position the camera in the scene, various common camera parameters, the screen window, and lens parameters for depth of field. It generates the orthographic transformation matrix with the `Orthogrpahic()` transformation function which will be defined shortly.

21 Point
175 ProjectiveCamera
175 ScreenDepth
176 WorldToScreen

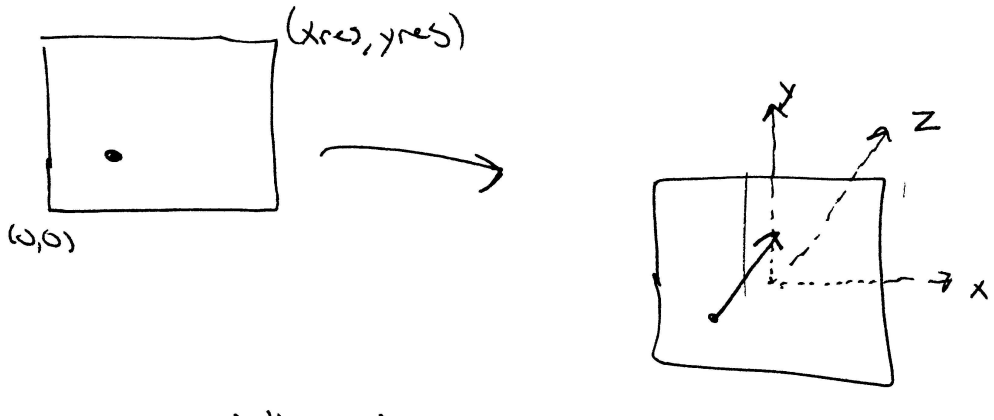


Figure 6.4: orthographic ray generation: raster space to ray...

⟨*OrthographicCamera Definitions*⟩≡
```
OrthoCamera::OrthoCamera(const Transform &world2cam,
        const Extent2D &Screen, Float hither, Float yon,
        Float sopen, Float sclose, Float lensr,
        Float focald, Film *f)
    : ProjectiveCamera(world2cam, Orthographic(hither, yon),
         Screen, hither, yon, sopen, sclose,
         lensr, focald, f) {
}
```

Extent2D 27
OrthoCamera 177
ProjectiveCamera 175
Scale 36
Transform 32
Translate 35
Vector 16

The orthographic viewing transformation leaves x and y coordinates unchanged, but maps z values at the hither plane to 0 and z values at the yon plane to 1. (See Figure 6.3.) It is easy to derive: first, the scene is translated along the z axis so that the near clipping plane is aligned with $z = 0$. Then, the scene is scaled in z so that the far clipping plane maps to $z = 1$. The composition of these two transformations gives the overall transformation.

⟨*Transform Methods*⟩+≡
```
Transform Orthographic(Float znear, Float zfar) {
    return Scale(1.f, 1.f, 1.f / (zfar-znear)) *
        Translate(Vector(0.f, 0.f, -znear));
}
```

We can now write the code to take a sample point in raster space and turn it into a camera ray. Recall that the `imagex` and `imagex` components of the camera `Sample` are raster-space x and y coordinates on the image plane. We use following process: first, we transform the raster-space sample position into a point in camera space; this gives us the origin of the camera ray–a point located on the near clipping plane. Because the camera-space viewing direction points down the z axis, the camera space ray direction is $(0, 0, 1)$. After the camera-space ray has been generated, we transform it to world space.

If depth of field has been enabled for this scene, the fragment ⟨*Modify ray for depth of field*⟩ takes care of modifying the ray so that depth of field is simulated.

Depth of field will be explained later in this section.

⟨*OrthographicCamera Definitions*⟩+≡
```
void OrthoCamera::GenerateRay(const Sample *sample, Ray &ray) const {
    ⟨Generate raster and camera samples⟩
    ray.O = Pcamera;
    ray.D = Vector(0,0,1);
    ray.mint = 0.;
    ray.maxt = INFINITY;
    ⟨Set ray time value⟩
    ⟨Modify ray for depth of field⟩
    CameraToWorld(ray, &ray);
}
```

⟨*Set ray time value*⟩≡
```
ray.time = Lerp(sample->time, ShutterOpen, ShutterClose);
```

Once all of the transformation matrices have been set up, we just set up the raster space sample point and transform it to camera space.

⟨*Generate raster and camera samples*⟩≡
```
Point Pras(sample->imagex, sample->imagey, 0);
Point Pcamera;
RasterToCamera(Pras, &Pcamera);
```

173 CameraToWorld
514 INFINITY
512 Lerp
177 OrthoCamera
21 Point
175 ProjectiveCamera
176 RasterToCamera
26 Ray
173 ShutterClose
173 ShutterOpen
16 Vector

Perspective Camera

⟨*perspective.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "camera.h"
#include "paramset.h"
⟨PerspectiveCamera Declarations⟩
⟨PerspectiveCamera Definitions⟩
```

The perspective projection is similar to the orthographic projection in that it projects a volume of space onto a 2D image plane. However, it includes the effect of *foreshortening*: objects that are far away are projected to be smaller than objects of the same size that are closer. Furthermore, unlike the orthographic projection, the perspective projection also doesn't preserve distances or angles in general, and parallel lines no longer remain parallel. The perspective projection is a reasonably close match for how the eye and camera lenses generate images of the three-dimensional world.

⟨*PerspectiveCamera Declarations*⟩≡
```
class PerspectiveCamera : public ProjectiveCamera {
public:
    ⟨PerspectiveCamera Method Declarations⟩
};
```

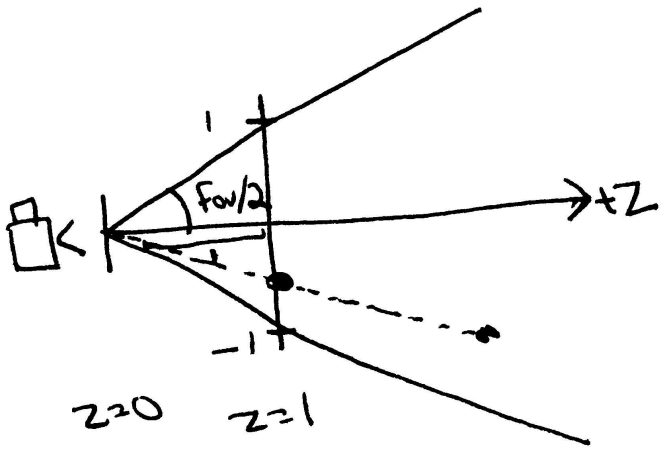


Figure 6.5: The perspective transformation matrix projects points in camera space onto the image plane. The x' and y' coordinates of the projected points are equal to the unprojected x and y coordinates divided by the z coordinate. The projected z' coordinate is computed so that z points on the hither plane map to $z' = 0$ and points on the yon plane map to $z' = 1$.

Extent2D 27
Perspective 181
PerspectiveCamera 179
ProjectiveCamera 175
Transform 32

⟨*PerspectiveCamera Definitions*⟩≡
```
PerspectiveCamera::PerspectiveCamera(const Transform &world2cam,
        const Extent2D &Screen, Float hither, Float yon,
        Float sopen, Float sclose, Float lensr, Float focald,
        Float fov, Film *f)
    : ProjectiveCamera(world2cam, Perspective(fov, hither, yon),
        Screen, hither, yon, sopen, sclose,
        lensr, focald, f) {
}
```

The perspective projection describes perspective viewing of the scene. Points in the scene are projected onto a viewing plane at $z = 1$; this is one unit away from the virtual camera at $z = 0$)–see Figure 6.5. The process is most easily understood in two steps:

- First, points p in camera space are projected onto the viewing plane. A little algebra shows that the projected x' and y' coordinates on the viewing plane can be computed by dividing x and y by the point's z coordinate value. The projected z depth is remapped so that z values at the hither plane go to 0 and z values at the yon plane go to 1. The computation we'd like to do is:

$$\begin{aligned} x' &= x/z \\ y' &= y/z \\ z' &= \frac{f(z-n)}{z(f-n)}. \end{aligned}$$

  Fortunately, all of this can easily be encoded in a four-by-four matrix using homogeneous coordinates (recall the discussion of homogeneous coordinates in Section 2.6 on page 30.) The `Transform` in the `Perspective()`

function below generates the appropriate matrix.

- Second, we account for the angular field of view specified by the user and scale the (x,y) values on the projection plane so that points inside the field of view project to coordinates between $[-1,1]$ on the view plane. (For square images, both x and y will lie between $[-1,1]$ in screen space. Otherwise, the direction in which the image is narrower will map to $[-1,1]$ and the wider direction will map to an appropriately larger range of screen-space values.) The scale that is applied after the projective transformation takes care of this. (Recall that the tangent is equal to the ratio of the opposite side of a right triangle to the adjacent side. Here the adjacent side is defined to have a length of 1, so the opposite side has the length $\tan(\text{fov}/2)$. Scaling by one over this maps the field of view to range from $[-1,1]$.

⟨*Transform Methods*⟩+≡
```
Transform Perspective(Float fov, Float n, Float f) {
    Float invTanAng = 1.f / tanf(Radians(fov) / 2.f);
    Matrix4x4 *persp =
        new Matrix4x4(1, 0,       0,          0,
                      0, 1,       0,          0,
                      0, 0, f/(f-n), -f*n/(f-n),
                      0, 0,       1,          0);
    return Scale(invTanAng, invTanAng, 1) *
           Transform(persp);
}
```

510 Matrix4x4
514 Radians
36 Scale
32 Transform

For a perspective projection, rays originate from the sample position on the hither plane and have the direction given by the vector from $(0,0,0)$ through the sample position. Therefore, we compute the ray's direction by subtracting $(0,0,0)$ from the sample's camera-space position. In other words, the ray's vector direction is component-wise equal to its point position. Rather than doing a useless subtraction to convert the point to a direction, we just component-wise initialize the vector `ray.D` from the point `Pcamera`.

In the perspective case, since the generated ray's direction may be quite short, we scale it up by the inverse of the near clip plane location; although this isn't strictly necessary (there's no particular need for the ray direction to be normalized), it can be more intuitive when debugging if the ray's direction has a magnitude somewhat close to one.

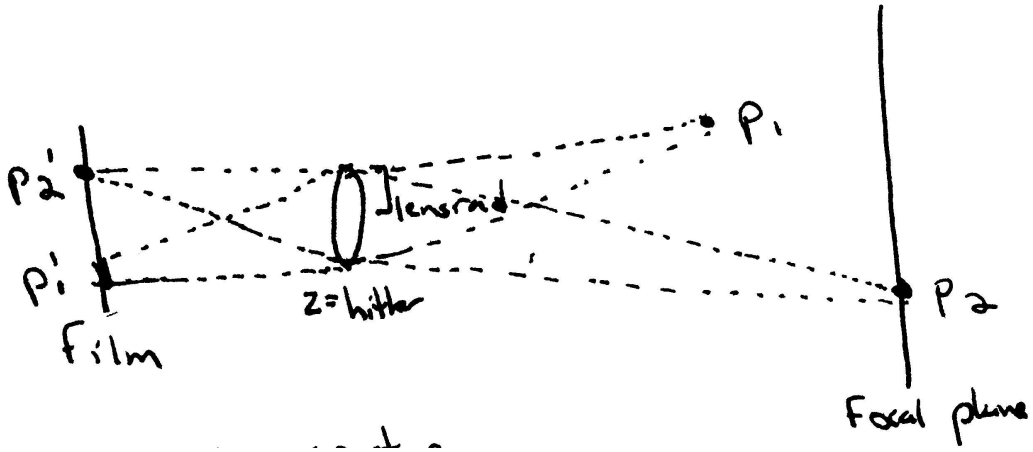


Figure 6.6: Real-world cameras have a lens with finite aperture and lens controls that adjust the lens position with respect to the film plane. Because the aperture is finite, objects in the scene aren't all imaged onto the film in perfect focus. Here, the point p_1 doesn't lie on the plane of points in perfect focus, so it images to an area p'_1 on the film and is blurred. The point p_2 does lie on the focal plane, so it images to a point p'_2 and is in focus. Both increasing aperture size and increasing an object's distance from the focal plane increase its blurriness.

CameraToWorld 173
INFINITY 514
invClipHither 173
PerspectiveCamera 179
Ray 26
Vector 16

⟨*PerspectiveCamera Definitions*⟩+≡
```
void PerspectiveCamera::GenerateRay(const Sample *sample,
        Ray &ray) const {
    ⟨Generate raster and camera samples⟩
    ray.O = Pcamera;
    ray.D = Vector(Pcamera.x, Pcamera.y, Pcamera.z);
    ray.mint = 0.;
    ray.maxt = INFINITY;
    ⟨Set ray time value⟩
    ⟨Modify ray for depth of field⟩
    CameraToWorld(ray, &ray);
    ray.D *= invClipHither;
}
```

Depth of Field

Real cameras have lens systems that focus light through a finite-sized aperture onto the film plane. Because the aperture has finite area, a single point in the scene may be projected onto an area on the film plane. (And correspondingly, a single point on the film plane may see different parts of the scene, depending on which part of the lens it's receiving light from.) Figure 6.6 shows this effect. The point p_1 doesn't lie on the plane of focus, so is projected through the lens onto an area p'_1 on the film plane. The point p_2 does lie on the plane of focus, so it projects to a single point p'_2 on the image plane. Therefore, p_1 will be blurring on the image plane while p_2 will be in sharp focus.

XXX need to differentiate between focal distance and lens focal length XXX

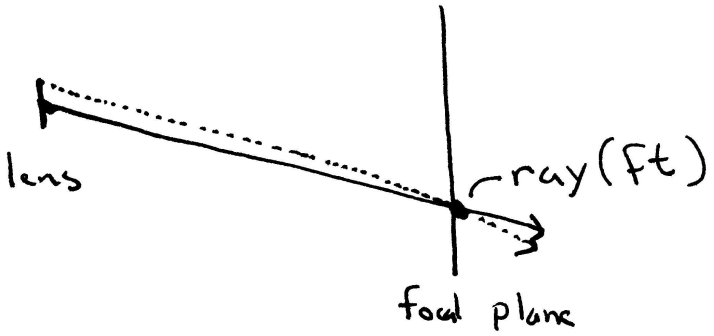


Figure 6.7: To adjust a camera ray for depth of field, we first compute the distance along the ray, `ft`, where it intersects the focal plane. We then shift the ray's origin from the center of the lens to the sampled lens position and construct a new ray (dashed line) from the new origin that sill does through the same point on the focal plane. This ensures that points on the focal plane remain in focus but that other points are blurred appropriately.

Lensmaker's equation makes this behavior explicit, relating the distance from the object being imaged to the lens, d_o, and the distance between the image and the lens, d_i,

$$\frac{1}{d_o} + \frac{1}{d_i} = \frac{1}{f},$$

where f is the *focal length* of the lens.

The area that the point projects to is called the *circle of confusion*. The size of the circle of confusion is dependent on the size of the aperture and how close the point is to the *focal plane*: the plane of points that are in perfect focus. The focusing controls of a camera adjust the lens system inside of it to shift the distance to the plane of focus. The larger the lens aperture, the more blurred out of focus points are. In the limit, a pinhole camera has an infinitessimal aperture, leaving all points in focus.

Therefore, the projective cameras take two extra parameters for depth of field: one sets the size of the lens aperture and the other sets the focal distance.

⟨*Initialize depth of field parameters*⟩≡
```
LensRadius = lensr;
FocalDistance = focald;
```

⟨*ProjectiveCamera Options*⟩+≡
```
Float LensRadius, FocalDistance;
```

It turns out that it just takes a few lines of code to simulate depth of field in a ray tracer. We associate each ray with a point on the lens and then adjust its direction to simulate the lens's effect: see Figure 6.7. Starting with the original ray, computed without accounting for depth of field, we have a ray through the center of the lens (corresponding to a pinhole camera.)

⟨*Modify ray for depth of field*⟩≡
```
if (LensRadius > 0.) {
    ⟨Sample point on lens⟩
    ⟨Compute point on plane of focus⟩
    ⟨Update ray for effect of lens⟩
}
```

We then choose a 2D point on the lens. The `ConcentricSampleDisk()` function, defined in Chapter 14, takes a (u,v) sample position in $[0,1]^2$ and maps it to the 2D disk with radius 1. To get a point on the lens, we scale these coordinates by the lens radius. The camera sample point passed into the `GenerateSample()` function uses the two lens sample positions from the `Sampler`.

⟨*Sample point on lens*⟩≡
```
Float lu, lv;
ConcentricSampleDisk(sample->lensx, sample->lensy, &lu, &lv);
lu *= LensRadius;
lv *= LensRadius;
```

We next comupte the t value along the ray where it intersects with the plane of focus. Because the plane of focus is orthogonal to the z axis and the ray starts on the hither plane, this is a particularly simple computation.

ConcentricSampleDisk 417
film 173
Point 21

⟨*Compute point on plane of focus*⟩≡
```
Float ft = (FocalDistance - ClipHither) / ray.D.z;
Point Pfocus = ray(ft);
```

Now we can adjust the ray: we want to compute the ray corresponding to the dashed line in Figure 6.7; the origin is shifted to the sampled point on the lens and the direciton is set so that the ray still passes through the point on the plane of focus, `Pfocus`.

⟨*Update ray for effect of lens*⟩≡
```
ray.O.x += lu;
ray.O.y += lv;
ray.D = Pfocus - ray.O;
```

6.3 Environment Camera

⟨*environment.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "camera.h"
#include "film.h"
#include "paramset.h"
⟨EnvironmentCamera Declarations⟩
⟨EnvironmentCamera Definitions⟩
```

Figure 6.8: An image rendered with the `EnvironmentCamera`, which traces rays in all directions from the camera position. The resulting image gives a representation of all light arriving at that point in the scene and can be used for interesting lighting techniques that will be described in Chapters 12 and 15.

⟨*EnvironmentCamera Declarations*⟩≡

```
class EnvironmentCamera : public Camera {
public:
    ⟨EnvironmentCamera Method Declarations⟩
    EnvironmentCamera(const Transform &world2cam, Float hither,
        Float yon, Float sopen, Float sclose, Film *film);
    virtual Float ScreenDepth(const Point &Pworld) const;
private:
    ⟨EnvironmentCamera Private Data⟩
};
```

173 film
21 Point
175 ScreenDepth
32 Transform

One advantage of ray-tracing renderers compared to scanline or rasterization rendering methods is that it's easy to have unusual image projections: we have great freedom in how the image sample positions are mapped into ray directions, since the rendering algorithm doesn't depend on properties such as straight lines in the scene always projecting to straight lines in the image, etc.

Here we will describe a camera model that traces rays in all directions around a point in the scene, giving a two-dimensional view of everything that is visible from that point. Consider a sphere around the camera position in the scene; choosing points on that sphere gives directions to trace rays in. If we parameterize the sphere with spherical coordinates, each point on the sphere is associated with a (θ, ϕ) pair, where $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi]$. (See Section 5.3 on page 163 for more details on spherical coordinates.) This type of image is particularly useful because it compactly captures a representation of all of the incident light at a point on the scene. It will be useful later when we discuss environment mapping and environment lighting: two rendering techniques that are based on image-based representations of light in a scene.

An image generated with this kind of projection is shown in Figure 6.8. Theta values range from 0, at the top of the image, to π, at the bottom of the image, and phi values range from 0 to 2π, moving across the image.

⟨*EnvironmentCamera Definitions*⟩≡
```
EnvironmentCamera::EnvironmentCamera(const Transform &world2cam,
        Float hither, Float yon, Float sopen, Float sclose,
        Film *film)
    : Camera(world2cam, hither, yon, sopen, sclose, film) {
    rayOrigin = CameraToWorld(Point(0,0,0));
}
```

All rays generated by this camera have the same origin; for efficiency we compute the world-space position of the camera once in the constructor.

⟨*EnvironmentCamera Private Data*⟩≡
```
Point rayOrigin;
```

⟨*EnvironmentCamera Definitions*⟩+≡
```
void EnvironmentCamera::GenerateRay(const Sample *sample,
        Ray &ray) const {
    ray.O = rayOrigin;
    ⟨Generate environment camera ray direction⟩
    ⟨Set ray time value⟩
    ray.mint = 0.;
    ray.maxt = INFINITY;
}
```

CameraToWorld 173
Distance 23
EnvironmentCamera 185
film 173
INFINITY 514
Point 21
Ray 26
ScreenDepth 175
Transform 32
Vector 16

To compute the (θ, ϕ) coordinates for this ray, we first compute NDC coodinates from the raster image sample position. These are then scaled up to cover the (θ, ϕ) range and then the spherical coordinate formula is used to comupte the ray direction.

⟨*Generate environment camera ray direction*⟩≡
```
Float theta = M_PI * sample->imagey / (film->yResolution - 1);
Float phi = 2 * M_PI * sample->imagex / (film->xResolution - 1);
Vector dir(sinf(theta) * cosf(phi), cosf(theta),
    sinf(theta) * sinf(phi));
CameraToWorld(dir, &ray.D);
```

To compute a screen depth value for a point in the scene, we treat the clip planes as clip spheres, with radii given by the hither and yon distances. From these, we can compute a point's depth by computing where it lies along the ray from the camera point passing through it with respect to these spheres.

⟨*EnvironmentCamera Definitions*⟩+≡
```
Float EnvironmentCamera::ScreenDepth(const Point &Pworld) const {
    return (Distance(Pworld, rayOrigin) - ClipHither) /
        (ClipYon - ClipHither);
}
```

6.4 Film

⟨*film.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef FILM_H
#define FILM_H
#include "lrt.h"
#include "color.h"
#include "geometry.h"
#include "transform.h"
⟨Film Declarations⟩
#endif // FILM_H
```

⟨*film.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "film.h"
#include "tonemap.h"
⟨Film Method Definitions⟩
```

The `Film` class takes care of string the values of the pixels computed from the image samples. Once all of the samples have arrived, it then applies a set of imaging operations which adjust the final image and prepare it for display before it is written out. The various information stored by the image is organized into a set of *channels*; numeric values at the regular grid of pixel sample locations. Each channel has a particular semantic meaning. Many image formats just store color channels, representing spectral color values. More generally, we can store some combination of a color representation, the depth of an object visible at the pixel, etc.

173 film

⟨*Film Declarations*⟩≡
```
class Film {
public:
    ⟨Film Interface⟩
    ⟨Film Public Data⟩
private:
    ⟨Film Private Data⟩
};
```

It is also useful to store information the coverage of objects at each pixel: how many of the rays contributing to it intersected an object in the scene and how many didn't hit anything. We store this fraction in the image's *alpha channel*. The lets us later disambiguate between pixels that are black because nothing was visible in them and pixels that are black because all of their rays hit a black object, for example. In general, the alpha channel is quite useful for image compositing: for instance, a rendered image can be put over a photograph, using the alpha channel to determine in which of the pixels the photograph is visible. For pixels with an alpha value between zero and one, the two images are blended together, giving smooth edges at the boundary of the rendered object.

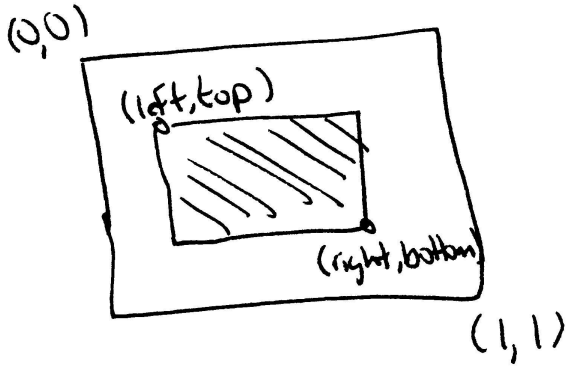


Figure 6.9: The image crow window specifies a subset of the image to be renderer. It is specified in NDC space, with coordinates ranging from $(0,0)$ to $(1,1)$. The `Film` class only allocates space for and stores pixel values in the region inside the crop window.

⟨*Film Method Definitions*⟩≡
```
Film::Film(int xres, int yres, const Extent2D &cropWindow) {
    xResolution = xres;
    yResolution = yres;
    ⟨Compute film image extent⟩
    ⟨Allocate film image storage⟩
}
```

Ceil2Int 514
Extent2D 27
xPixelWidth 189
yPixelWidth 189

The film constructor starts by intiializing a few parameters passed into the constructor. One of the most important parameters is the overall image resolution– `xResolution` and `yResolution` hold the total number of pixels in the x and y directions.

⟨*Film Public Data*⟩≡
```
int xResolution, yResolution;
```

The user may have also specified a *crop window* that defines a subsection of the image to render–this can be useful for debugging as well as for breaking a large image into chunks that can then be reassembled later. The crop window is specified in NDC space, with each coordinate ranging from zero to one–see Figure 6.9. In conjunction with the overall image resolution, the crop window gives us the extent of integer pixel locations that we'll actually store and write out. `xPixelStart` and `yPixelStart` store the pixel position of the upper left corner of the crop window, and `xPixelWidth` and `yPixelWidth` give the pixel widths in each direction. Given a pixel (x,y) inside the pixel crop window, the pixel arrays are indexed as

$$(y - \text{yPixelStart}) * \text{xPixelWidth} + (x - \text{xPixelStart}).$$

⟨*Compute film image extent*⟩≡
```
xPixelStart = Ceil2Int(xResolution * cropWindow.x0);
xPixelWidth = Ceil2Int(xResolution * cropWindow.x1) - xPixelStart;
yPixelStart = Ceil2Int(yResolution * cropWindow.y0);
yPixelWidth = Ceil2Int(yResolution * cropWindow.y1) - yPixelStart;
```

⟨*Film Private Data*⟩≡
```
int xPixelStart, yPixelStart, xPixelWidth, yPixelWidth;
```

Now that we know the pixel resolution of the image, we allocate an array of `Pixel` structures to store sample values as samples come in. Pixel radiance values are stored in `L`, their alpha values are stored in `alpha`, and their z depths are stored in `depth`. For now, just consider `weightSum` to be a tally of the total number of samples that contribute to the final pixel value; Chapter 7 explains the function of `WeightSums` in detail in the context of general principles of image sampling and reconstruction.

⟨*Allocate film image storage*⟩≡
```
pixels = new Pixel[xPixelWidth * yPixelWidth];
```

⟨*Film Private Data*⟩+≡
```
struct Pixel {
    Pixel() { alpha = 0.; depth = INFINITY; weightSum = 0.; }
    Spectrum L;
    Float alpha, depth, weightSum;
};
Pixel *pixels;
```

514 INFINITY
513 min
155 Spectrum

⟨*Film Method Definitions*⟩+≡
```
Film::~Film() {
    delete[] pixels;
}
```

As the renderer computes radiance along rays in the scene, the `Sampler` will call the film's `UpdatePixel()` method to report the radiance, alpha, and depth values along with the sample's weighted contribution at each film sample that it contributes to. We make sure that the asked-for pixel is inside the range of pixels the film stores and then update the appropriate parts of the `Pixel` structure if so.

⟨*Film Interface*⟩+≡
```
inline bool UpdatePixel(int x, int y, const Spectrum &L,
        Float alpha, Float depth, Float weight) {
    if (x < xPixelStart || x >= xPixelStart + xPixelWidth ||
        y < yPixelStart || y >= yPixelStart + yPixelWidth)
        return false;
    int offset = (y - yPixelStart) * xPixelWidth +
        (x - xPixelStart);
    Pixel *pixelp = pixels + offset;
    pixelp->L += L * weight;
    pixelp->alpha += alpha * weight;
    pixelp->depth = min(pixelp->depth, depth);
    pixelp->weightSum += weight;
    return true;
}
```

Further Reading

Möller and Haines have a particularly well-written derivation of the orthographic and perspective projection matrices in *Real Time Rendering* (MH02). Other good references for projections are Rogers and Adams' *Mathematical Elements for Computer Graphics* (RA90), Watt and Watt (WW92), Foley et al (FvDFH90) and Eberly's book on game engine design (Ebe01). (Originally Sutherland sketchpad stuff?)

Potmesil and Chakravarty did early work on depth of field and motion blur in computer graphics (PC81; PC82; PC83). Cook and collaborators developed a more accurate model for these effects based on *distribution ray tracing*; this is the approach we have implemented in this chapter (CPC84; Coo86).

Kolb et al investigated simulating complex camera lens systems with ray-tracing in order to model the imaging effects of real cameras (KHM95). Another unusual projection method was used by Greene and Heckbert for generating images for Omnimax theaters (GH86a).

Porter and Duff's paper on compositing digital images is the classic paper on the uses of images with alpha channels and explains why pre-multiplied alpha is a preferable preresentation for color (PD84). (The first use of an extra alpha channel in images in graphics dates to Smith and Catmull, however (Smi79). See also Wallace's paper for a refinement of Smith and Catmull's approach (Wal81).)

Gamma correction has a a long history in computer graphics; Poynton has written comprehensive FAQs on issues related to color and gamma-correction in computer graphics (Poy02b; Poy02a).

Display issues, mapping to reasonable RGB values, out of gamut colors, ... See Rougeron and Péroche's survey article for discussion and references (RP98).

Malacara's monograph gives a concise overview of color theory and basic properties of how the the human visual system processes color (Mal02).

Wandell's book?

wazczeki(?sp) and stiles

Glassner has written an article on the under-constrained problem of converting RGB values (e.g. as selected by the user from a display) to a SPD (Gla89b).

Tone reproduction for computer graphics became an active area of research around 1993 with the work of Tumblin and Rushmeier (TR93), Chiu et al (CHS+93), and Ward (War94a). The non-linear mapping we presented was developed by Reinhard et al (ERF02).

Exercises

6.1 Moving camera

6.2 Cek style lens systems?

6.3 Ward style histogram-based ton repro stuff: don't waste dynamic range in parts of the histogram where not many image samples lie

7. Sampling and Reconstruction

We'll now describe how the `Sampler` decides where the image should be sampled and how the pixels in the output image are computed from those samples. The mathematical background for this is given by *sampling theory*: the theory of taking discrete sample values from continuous signals and then reconstructing new signals from those samples. Most of the previous development of sampling theory has been for encoding and compressing audio (e.g. over the telephone), and for television signal encoding and transmission. In rendering, we face the two-dimensional instance of this problem, where we're sampling an image at particular positions, by tracing rays into the scene and then reconstructing a set of output pixels that form an image.

In the one dimensional case, consider a signal given by a function $f(x)$; we can evaluate f at any x value we choose. Each such x is a *sample position*, and the value of $f(x)$ is the *sample value*. The left half of Figure 7.1 shows a set of samples (black dots) of a smooth 1D function. From a set of such samples, $(x, f(x))$, we'd like to *reconstruct* a new signal $\tilde{f}$ that approximates f as closely as possible. On the right side of Figure 7.1 is a reconstructed function that approximated $f(x)$ by linearly interpolating neighboring sample values. In general, the only information we have about f comes from the sample values we have taken; as such, $\tilde{f}$ is likely to not match f perfectly, since we have no knowledge of f's behavior between the sample values that we have.

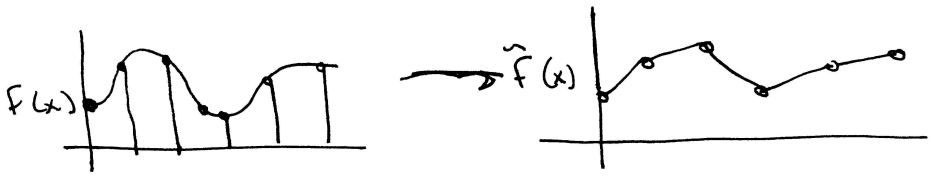


Figure 7.1: By taking a set of *point samples* of $f(x)$, we determine its value at those positions. From the sample values, we can *reconstruct* a function $\tilde{f}(x)$ which is an approximatino to $f(x)$. The sampling theorem, introduced in Section 7.1, makes a precise statement about the conditions on $f(x)$ and the number of samples taken under which $\tilde{f}(x)$ is exactly the same as $f(x)$. That the exact function can be found purely by point sampling $f(x)$ is a remarkable result.

7.1 Signal Processing and Sampling Theory

Intuitively, the smoother that a function is, the fewer samples will be necessary to reconstruct it accurately. In the limit, when the signal is constant, a single sample is enough to characterize the signal completely. As a signal gets progressively less smooth (i.e. as it has higher frequency undulations), progressively more samples are necessary to represent it accurately. In general, we can talk about the *sampling rate*, (or the inverse of the sampling rate, the *sampling frequency*): this is the separation in Δx between adjacent samples of the signal. If the sampling rate is uniform, the spacing between all of the samples is constant.

The frequency of the function $f(x)$ can be described precisely. For example $f(x) = \sin 2\pi x$ has a single frequency, $\omega = 1$, since a new cycle starts whenever a distance of $\Delta x = 1/\omega = 1$ passes along the x axis. $f(x) = \sin 8\pi x + \sin 2\pi x$ has two frequencies, $\omega = 1$ and $\omega = 4$. Interestingly enough, *any* continuous function $f(x)$ can be completely characterized by the distribution of all of its frequencies.

The *sampling theorem* makes an important statement about the sampling rate and how accurately a function can be reconstructed from a set of samples. Specifically, so long as the frequency of sample points ω_s is greater than twice the maximum frequency present in the signal ω_m, it is possible to reconstruct the original signal *perfectly* from the samples. This minimum sampling frequency is called the *Nyquist frequency*.

In order to perform this perfect reconstruction, a specific technique must be used to reconstruct the new function from the samples. Given the set of samples and their values $(x_i, f(x_i))$, the new function is defined by

$$\tilde{f}(x) = \sum_i f(x_i) r(x_i - x) \tag{7.1.1}$$

where $r(t)$ is the *ideal reconstruction filter* (also known as the *sinc function*):

$$r(x) = \text{sinc}\left(\frac{\omega}{2\pi} x\right)$$

where ω is the sampling frequency and $\text{sinc}(x) = (\sin x)/x$. A graph of the sinc function is shown in Figure 7.3.

Aliasing

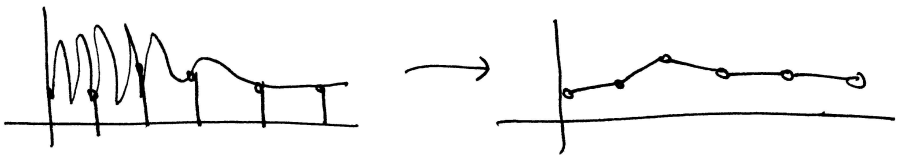


Figure 7.2: Undersampled 1D function: when the original function has undulations at a higher frequency than half the sampling frequency, it's not possible to reconstruct the original function. Aliasing, low-frequency errors in the reconstructed function that aren't present in the original function, is the result.

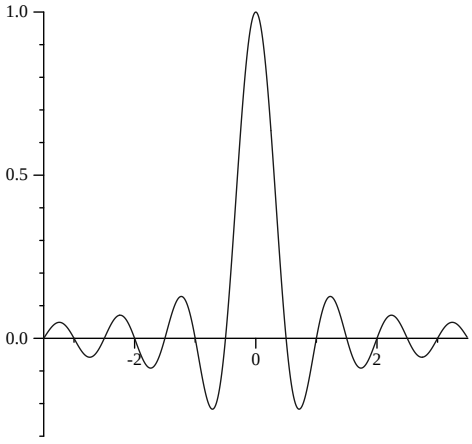


Figure 7.3: Graph of the sinc function, the filter that perfectly reconstructs the original function that was sampled, as long as the sampling frequency was sufficiently high. The entire sinc function actually has infinite support, spanning $(-\infty, \infty)$.

Figure 7.4: Aliasing from point sampling the function $\cos(x^2+y^2)$; at the left side of the image, the function has a low frequency–tens of pixels per cycle–so it is represented accurately. Moving to the right, however, aliasing artifacts appear in the top image since the sampling rate doesn't keep up with the function's highest frequency. If high frequency elements of the signal are removed with filtering before sampling, as was done in the bottom image, the right side of the image takes on a constant grey color. (Example due to Don Mitchell.)

If the original function isn't sampled with a sufficiently high sampling rate, *aliasing* can result. Aliasing happens when high-frequency components in the original signal appear in the reconstructed signal as lower-frequency artifacts. In 1D, Figure 7.2 shows aliasing in a reconstructed function due to under-sampling the original function. Figure 7.4 shows the effect of sampling the two-dimensional function $f(x,y) = \cos(x^2+y^2)$; the origin $(0,0)$ is at the center of the left edge of the image. At the left, we have accurately represented the signal, though as we move farther to the right and f has higher and higher frequency content, aliasing starts: the circular patterns that appear in the center and right of the image are sever aliasing artifacts.

It's often either impossible or very difficult to know the frequency content of the signal being sampled. Nevertheless, the sampling theorem is still useful. First, it tells us the effect of increasing the sampling frequency: the point at which aliasing starts is pushed out to a higher frequency. Second, given some particular sampling frequency, it tells us the frequency beyond which we should try to remove high frequency data from the signal; this will be useful in Section 11.6 when we introduce texture filtering, for instance. For a given sampling rate, the best way to avoid aliasing is to *pre-filter* the signal to remove any frequencies higher than the Nyquist limit.

The application of these ideas to the two-dimensional case of sampling and reconstructing images is straightforward; we have an image, which we can think of

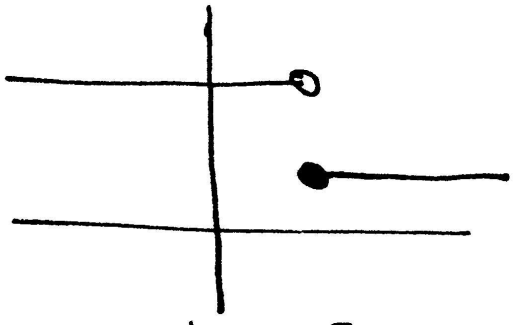


Figure 7.5: 1D step function: the function discontinuously jumps from one value to another. Such functions have infinitely-high frequency content. As such, point sampling can never adequately capture them for perfect reconstruction.

as a function of two-dimensional (x,y) image locations to radiance values L:

$$f(x,y) \rightarrow L$$

where $x \in [0, \text{xResolution})$ and $y \in [0, \text{yResolution})$. The good news is that, with our ray-tracer, we can evaluate this at any (x,y) point that we choose. The bad news is that we can only point sample the image function f: it's not generally possible to remove the high frequencies from the function before sampling it.

More generally, we can think of there being a multi-dimensional scene function that maps a set of sample parameters to radiance. In addition to sampling a particular (x,y) pixel, varying the time t at which it is sampled will give different radiance values if there are moving objects in the scene. Further, for cameras that simulate depth of field (Section 6.2), varying the (u,v) lens sample position gives different results. Sampling all of these dimensions well is an important part of generating high-quality imagery; the `Sampler` classes in the next few sections will address the issue of sampling all of them as well as possible.

Geometry is one of the biggest causes of aliasing in rendered images. When projected onto the image plane, an object's boundary introduces a *step function*, where the image function's value discontinuously jumps from one value to another. A one-dimensional example of a step function is shown in Figure 7.5. Unfortunately, step functions have infinite frequency content, which means that no amount of increasing the sampling density can correctly capture them. Furthermore, when the perfect reconstruction filter is applied to aliased samples, ringing artifacts appear in the reconstructed image–an effect known as *Gibb's phenomenon*. Another problem comes from very small objects in the scene: if geometry is small enough that it falls in between samples on the image plane, it can make no contribution to the final image at all. Both of these forms of *geometric aliasing* can cause some of the worst artifacts in rendered images.

Another source of aliasing can come from the colors and materials on an object. *Shading aliasing* can come from texture maps on objects that haven't been filtered correctly (see Section 11.6 on page 333), or from small highlights on shiny surfaces; if the sampling rate is not high enough to sample these features adequately,

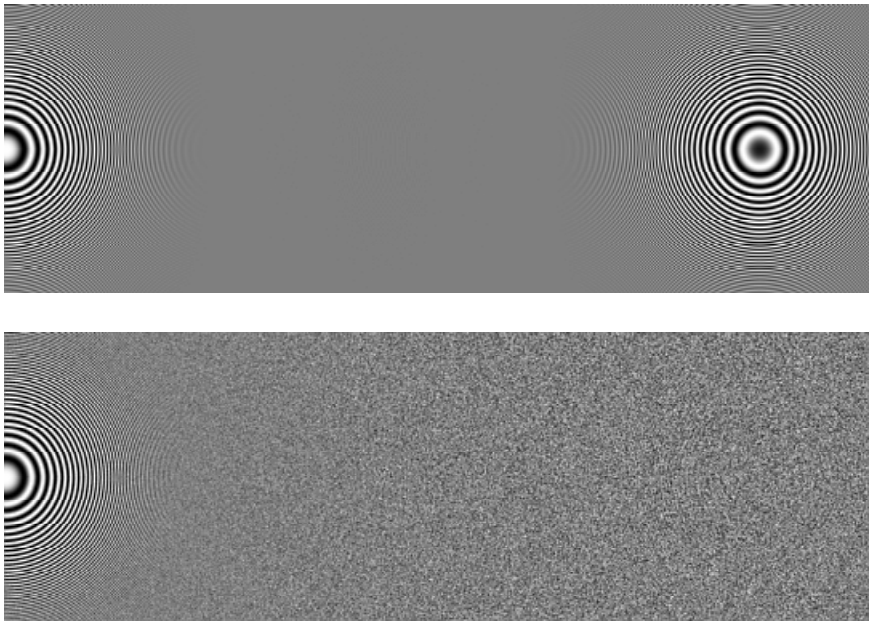


Figure 7.6: Jittered sampling (below) changes the regular, low-frequency aliasing artifacts from under-sampling the signal into high-frequency noise.

aliasing will result. Furthermore, a sharp shadow cast by an object introduces another step function in the final image; while it is possible to identify step functions from geometric edges, detecting step functions from shadow boundaries is much more difficult.

Non-uniform sampling

Although the image function that we're sampling is known to have infinite-frequency components and thus can't be perfectly reconstructed, not all is lost. It turns out that choosing the distribution of sample points carefully (and specifically, not using a uniform sampling pattern) can reduce the visual impact of aliasing. For a fixed sampling rate that isn't sufficient to capture the function, both uniform and non-uniform sampling produce incorrect reconstructed signals. However, non-uniform sampling tends to turn the regular aliasing artifacts into *noise*.

Figure 7.6 shows this effect with the same cosine function example as was used above. On top, we have the function sampled at a fixed rate using uniform samples. Below, we have *jittered* each sample location, adding a small random number to it in x and y. The aliasing patterns have been transformed into high-frequency noise artifacts, which are less visually objectionable.

This is an interesting result, since it shows that the best sampling patterns according to the signal processing view don't always give the best results perceptually. In particular, some image artifacts are more visually acceptable than others. This observation will guide our development of good image sampling patterns through the rest of this chapter.

Adaptive sampling

One approach that has been suggested to combat aliasing is *adaptive supersampling*: if we can identify the regions of the signal with frequencies higher than

the Nyquist limit, we can take additional samples in those regions without needing to incur the expense of increasing the sampling frequency everywhere. It is hard to get this to work well in practice, however, since in general it's hard to find all of the places where super-sampling is needed. Most schemes are based on examining adjacent sample values, finding ones where there is a significant change in sample value between the two; the hypothesis is that the signal may have high frequencies in that region. In general, however, adjacent sample values cannot tell us anything about what is really happening in between them: the function may have huge variation between the two of them, but just happen to return to the same value at each of them. Thus, some areas that need super-sampling will usually be missed, leaving the only recourse to be increasing the basic sampling rate anyway.

7.2 Image Sampling Interface

⟨*sampling.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef SAMPLING_H
#define SAMPLING_H
#include "lrt.h"
⟨Sampling Constants⟩
⟨Sampling Declarations⟩
#endif // SAMPLING_H
```

173 film

⟨*sampling.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "scene.h"
#include "sampling.h"
#include "film.h"
⟨Sampler Method Definitions⟩
⟨Sample Method Definitions⟩
```

We can now start to describe the operation of a few classes that generate good image sampling patterns. All of them inherit from an abstract `Sampler` class that defines their interface. `Samplers` have two main jobs:

1. They are responsible for generating a sequence of multi-dimensional sample positions. The first two dimensions give the raster-space image sample position. The third value gives the time at which the sample should be taken; this ranges from zero to one, and is scaled by the camera to cover the time period that the shutter is open appropriately. The next two samples give a (u, v) lens position to sample for depth of field; these also vary from zero to one. Finally, sample points in four more dimensions are generated for future use by some of the light transport routines in Chapter 15.

2. `Samplers` are responsible for taking the radiance values computed for particular image samples and computing final values for the output pixels. We will describe this part of their operation later, in Section 7.6.

⟨*Sampling Declarations*⟩≡
```
class Sampler {
public:
    ⟨Sampler Interface⟩
    ⟨Sampler Options⟩
};
```

All `Sampler`s take a few common parameters that must be passed on to the base class's constructor. They are the overall image resolution in the x and y dimensions, the number of samples per pixel to take in each direction, the NDC image crop window, and a pointer to the `Filter` to be used to filter the image samples to compute the final pixels. We store these values in member variables for later use.

⟨*Sampler Method Definitions*⟩+≡
```
Sampler::Sampler(int xres, int yres, int xsamp, int ysamp,
        const Extent2D &Crop, Filter *f) {
    xResolution = xres;
    yResolution = yres;
    xPixelSamples = xsamp;
    yPixelSamples = ysamp;
    filter = f;
    ⟨Initialize pixel extents from crop window⟩
}
```

Extent2D 27
Filter 229

⟨*Sampler Options*⟩≡
```
int xResolution, yResolution;
int xPixelSamples, yPixelSamples;
Filter *filter;
```

The constructor wraps up by initializing the variables below that give the range of pixels in x and y for which we need to generate samples. Samples for pixels ranging from `xPixelStart` to `xPixelEnd-1`, inclusive, in x (and analogously in y) should be generated by the `Sampler`. The fragment that implements ⟨*Initialize pixel extents from crop window*⟩ and details of how particular crop window values translate into sample pixel ranges will be explained later, in Section 7.6.

⟨*Sampler Options*⟩+≡
```
int xPixelStart, xPixelEnd, yPixelStart, yPixelEnd;
```

`Sampler`s need to implement the `GetNextSample()` method, which is here declared as a pure virtual function. The `Scene::Render()` method will call this function until it returns `false`; as long as it keeps returning `true`, it should fill in the `sample` that is passed in with sample values. All of the dimensions should be in the range $[0,1]$, except for the first two, which should be given in terms of the image size.

⟨*Sampler Interface*⟩+≡
```
virtual bool GetNextSample(Sample *sample) = 0;
```

So that it's easy for the main rendering loop to figure out what percentage of the scene has been rendered after some number of samples have been processed, the `TotalSamples()` method returns the total number of samples that the `Sampler` will be returning.

⟨*Sampler Interface*⟩+≡
```
int TotalSamples() const {
    return xPixelSamples * yPixelSamples * (xPixelEnd - xPixelStart) *
        (yPixelEnd - yPixelStart);
}
```

Sample representation

The `Sample` structure

Explain something about different numbers needed at different ray depths, etc...

⟨*Sampling Declarations*⟩+≡
```
struct Sample {
public:
    ⟨Sample Method Declarations⟩
    Float imagex, imagey;
    Float lensx, lensy;
    Float time;
    vector<int> nLightSamples, nBSDFSamples;
    Float **light, **bsdf;
};
```

⟨*Sample Method Definitions*⟩≡
```
Sample::Sample(const vector<int> &nLight, const vector<int> &nBSDF) {
    nLightSamples = nLight;
    nBSDFSamples = nBSDF;
    ⟨Compute total number of light and BSDF samples needed⟩
    ⟨Allocate storage for light and BSDF sample pointers⟩
    ⟨Allocate storage for light and BSDF sample memory⟩
}
```

⟨*Compute total number of light and BSDF samples needed*⟩≡
```
int totSamples = 0;
for (u_int i = 0; i < nLightSamples.size(); ++i)
    totSamples += nLightSamples[i];
for (u_int i = 0; i < nBSDFSamples.size(); ++i)
    totSamples += nBSDFSamples[i];
totSamples *= 2;
```

⟨*Allocate storage for light and BSDF sample pointers*⟩≡
```
int nPtrs = nLightSamples.size() + nBSDFSamples.size();
if (!nPtrs)
    light = bsdf = NULL;
else {
    Float **ptrs = (Float **)AllocL1CacheAligned(nPtrs * sizeof(Float *));
    light = ptrs;
    bsdf = ptrs + nLightSamples.size();
}
```

506 AllocL1CacheAligned
494 size
198 xPixelEnd
198 xPixelSamples
198 yPixelEnd
198 yPixelSamples

⟨*Allocate storage for light and BSDF sample memory*⟩≡
```
if (light) {
    Float *mem = (Float *)AllocL1CacheAligned(totSamples * sizeof(Float));
    for (u_int i = 0; i < nLightSamples.size(); ++i) {
        light[i] = mem;
        mem += 2 * nLightSamples[i];
    }
    for (u_int i = 0; i < nBSDFSamples.size(); ++i) {
        bsdf[i] = mem;
        mem += 2 * nBSDFSamples[i];
    }
}
```

7.3 Stratified Sampling

⟨*stratified.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "sampling.h"
#include "paramset.h"
⟨StratifiedSampler Declarations⟩
⟨StratifiedSampler Method Definitions⟩
```

AllocL1CacheAligned 506
size 494

The first sample generator that we will introduce divides the image plane into rectangular regions and generates a single sample inside each region. These regions are commonly called *strata*, and this sampler is thus called `StratifiedSampler`. Each sample is chosen by choosing a random point inside each of the stratum; this can be computed by *jittering* the center point of the stratum by a random amount, up to half its width and height. This sampler also offers a mode where this jittering is not done, giving uniform sampling in the strata; this unjittered mode is mostly useful for sampling pattern comparisons rather than rendering final images.

Figure 7.7 shows a comparison of a few basic sampling patterns. On the top is a completely random sampling pattern: we have chosen a number of image samples to take and have computed that many random image locations. The result is a terrible sampling pattern; some regions of the image have few samples and other areas have clumps of many samples. For reference, in the middle is an un-jittered stratified pattern. On the bottom, we have jittered the uniform pattern, adding a random offset to each sample's location but keeping it inside its cell. This gives a better overall distribution than the purely random pattern, although there are still some clumps of samples and some regions that are under-sampled. We will present a more sophisticated image sampling method in the next section that ameliorates some of these problems.

A visualization of strata over an image is shown in Figure 7.8; a grid has been superimposed over the image, where one sample point is chosen inside each grid cell. The total number of strata in each direction is the number of pixels times the number of samples per pixel in that direction. The default sampling rate, four samples (two in the x direction and two in y), gives reasonably good results on many images.

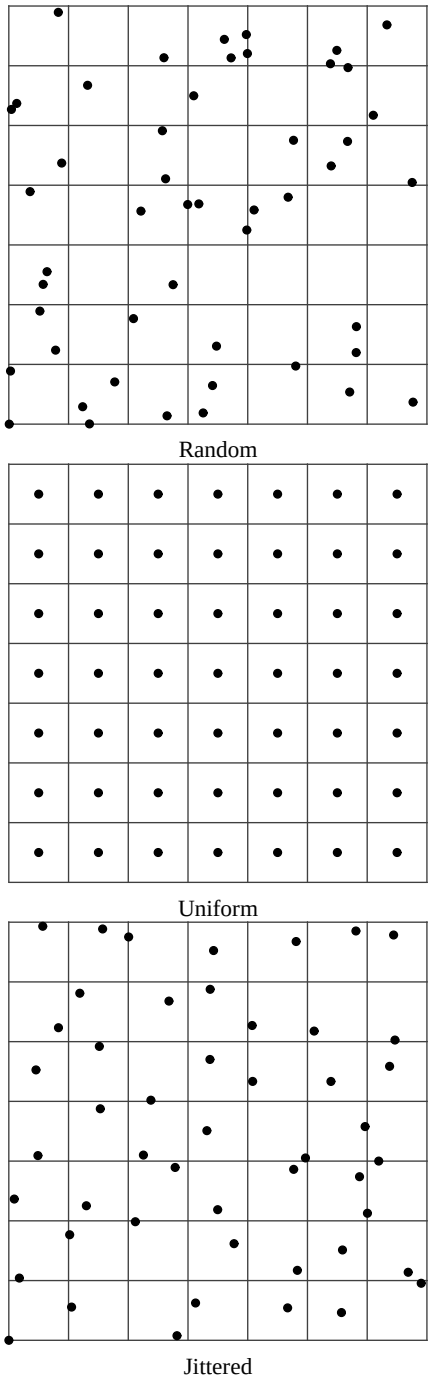


Figure 7.7: Three sampling patterns. The random pattern on the top is a poor pattern, with many clumps of samples that leave large sections of the image poorly sampled. In the middle is a uniform pattern which is better distributed but that can exacerbate aliasing artifacts. On the bottom is a jittered pattern, which turns aliasing from the uniform pattern into high-frequency noise.

There are a total of `xResolution*xPixelSamples` strata in the *x* direction and analogously in the *y* direction. Samples are generated by scanning over the pixel strata left-to-right and top-to-bottom. The sampler holds the offset of the current stratum in the `XPos` and `YPos` variables, which are initialized to point at first stratum in the upper left of the image's crop window to start out.

⟨*StratifiedSampler Declarations*⟩≡
```
class StratifiedSampler : public Sampler {
public:
    ⟨StratifiedSampler Method Declarations⟩
private:
    ⟨StratifiedSampler Private Data⟩
    ⟨StratifiedSampler Private Methods⟩
};
```

⟨*StratifiedSampler Method Definitions*⟩≡
```
StratifiedSampler::StratifiedSampler(int xres, int yres, int xpix,
        int ypix, bool jitter, const Extent2D &crop,
        Filter *f)
    : Sampler(xres, yres, xpix, ypix, crop, f) {
    JitterSamples = jitter;
    XPos = xPixelStart * xPixelSamples;
    YPos = yPixelStart * xPixelSamples;
    imageSamples = new Float[xPixelSamples * yPixelSamples * 2];
    lensSamples = new Float[xPixelSamples * yPixelSamples * 2];
    timeSamples = new Float[xPixelSamples * yPixelSamples];
    ⟨Generate samples for XPos,YPos⟩
}
```

Extent2D 27
Filter 229
lensSamples 216
Sampler 198
timeSamples 216
xPixelSamples 198
yPixelSamples 198

⟨*StratifiedSampler Private Data*⟩≡
```
bool JitterSamples;
int XPos, YPos;
int samplePos;
Float *imageSamples, *lensSamples, *timeSamples;
```

Generate image, lens, time for the whole pixel. Will do light and bsdf as needed, per sample (for now?).

⟨*Generate samples for* `XPos,YPos`⟩≡
```
sample2D(imageSamples);
sample2D(lensSamples);
sample1D(timeSamples);
⟨Scale and shift stratified image samples⟩
⟨Decorrelate sample dimensions⟩
samplePos = 0;
```

Rather than generating 5 dimensional stratified pattern, generate a collection of 2d and 1d patterns for all of the various dimensions. Then associate samples from the additional dimensions with each image sample.

Good since no exponential growth in number of samples, but good coverage of the sample space. In particular, good since *each pixel* has good coverage– intuition for why this matters, why pixel spacing is the rate at which we want

good distribution–not more and not less....

⟨*StratifiedSampler Method Definitions*⟩+≡
```
void StratifiedSampler::sample2D(Float *samp) const {
    Float dx = 1.f / xPixelSamples;
    Float dy = 1.f / yPixelSamples;
    for (int y = 0; y < yPixelSamples; ++y)
        for (int x = 0; x < xPixelSamples; ++x) {
            int o = (x + y * xPixelSamples) * 2;
            Float jx = 0., jy = 0.;
            if (JitterSamples) {
                jx = RandomFloat();
                jy = RandomFloat();
            }
            samp[o]   = (x + jx) * dx;
            samp[o+1] = (y + jy) * dy;
        }
}
```

⟨*StratifiedSampler Method Definitions*⟩+≡
```
void StratifiedSampler::sample1D(Float *samp) const {
    int totSamples = xPixelSamples * yPixelSamples;
    Float invTot = 1.f / totSamples;
    for (int i = 0;  i < totSamples; ++i) {
        Float delta = .5;
        if (JitterSamples)
            delta = RandomFloat();
        samp[i] = (i + delta) * invTot;
    }
}
```

202 JitterSamples
216 lensSamples
515 RandomFloat
202 StratifiedSampler
216 timeSamples
198 xPixelSamples
202 XPos
198 yPixelSamples
202 YPos

XXX this is wrong

Now we need to generate a sample in the current cell. If jittering is enabled, we add a random offset between $-.5$ and $.5$ to each position to place it randomly in its cell. We then convert the sample to raster-space by dividing by the total number of samples in that direction.

⟨*Scale and shift stratified image samples*⟩≡
```
for (int y = 0; y < yPixelSamples; ++y) {
    for (int x = 0; x < xPixelSamples; ++x) {
        int o = (x + y * xPixelSamples) * 2;
        imageSamples[o]   += XPos - .5f;
        imageSamples[o+1] += YPos - .5f;
    }
}
```

⟨*Decorrelate sample dimensions*⟩≡
```
shuffle2D(lensSamples);
shuffle1D(timeSamples);
```

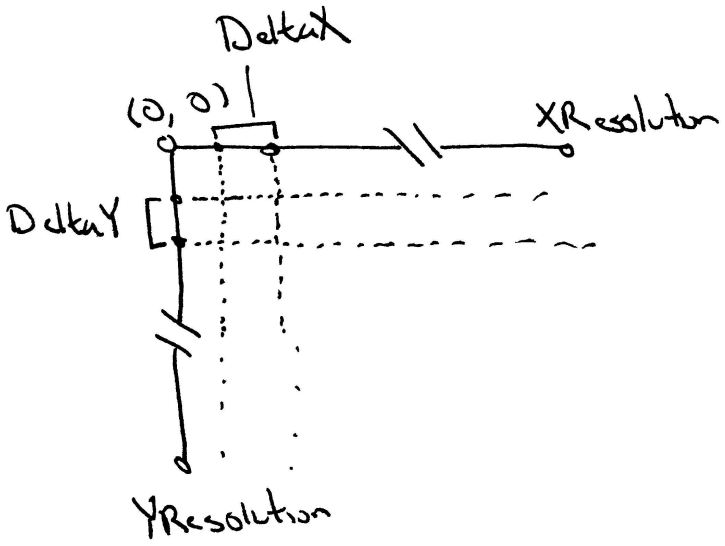


Figure 7.8: pixel strata for jittered sampling.

GetNextSample 198
RandomInt 515
StratifiedSampler 202
swap 513
xPixelSamples 198
yPixelEnd 198
yPixelSamples 198
YPos 202

⟨*StratifiedSampler Method Definitions*⟩+≡
```
void StratifiedSampler::shuffle2D(Float *samp) const {
    int totSamples = xPixelSamples * yPixelSamples;
    for (int i = 0; i < totSamples; ++i) {
        int other = RandomInt() % totSamples;
        swap(samp[2*i], samp[2*other]);
        swap(samp[2*i+1], samp[2*other+1]);
    }
}
```

We can now write the `GetNextSample()` function. It starts by checking to see if it has generated all of the necessary samples; if so, it returns false. It then generates a new sample and advances the variables that keep track of the next stratum that needs to be sampled.

⟨*StratifiedSampler Method Definitions*⟩+≡
```
bool StratifiedSampler::GetNextSample(Sample *sample) {
    ⟨Compute new set of samples if needed for next pixel⟩
    ⟨Return next StratifiedSampler sample point⟩
    return true;
}
```

⟨*Compute new set of samples if needed for next pixel*⟩≡
```
if (samplePos == xPixelSamples * yPixelSamples) {
    ⟨Advance to next stratum⟩
    if (YPos == yPixelEnd)
        return false;
    ⟨Generate samples for XPos,YPos⟩
}
```

The *y* strata counter `YPos` is only advanced when we reach the end of a row

of samples in the *x* direction. Therefore, once the *y* position counter has been advanced all the way down to the bottom of the image, we're done.

We finally advance to the next cell in the sampling grid, first trying to move to the next cell in *x*. If that takes us off of the end of the image, we reset the x position to the first stratum in the next *x* row to be sampled and try to advance the y position. If that ends up taking us off the bottom of the image, we're done–the next time this method is called, it will just return `false`.

⟨*Advance to next stratum*⟩≡
```
if (++XPos == xPixelEnd) {
    XPos = xPixelStart;
    ++YPos;
}
```

⟨*Return next* `StratifiedSampler` *sample point*⟩≡
```
sample->imagex = imageSamples[2*samplePos];
sample->imagey = imageSamples[2*samplePos+1];
sample->lensx = lensSamples[2*samplePos];
sample->lensy = lensSamples[2*samplePos+1];
sample->time = timeSamples[samplePos];
⟨Generate stratified lens and BSDF samples⟩
++samplePos;
```

Need to explain latin hypercube stuff somewhere....

⟨*Generate stratified lens and BSDF samples*⟩≡
```
for (u_int i = 0; i < sample->nLightSamples.size(); ++i)
    LatinHypercube(sample->light[i], sample->nLightSamples[i], 2);
for (u_int i = 0; i < sample->nBSDFSamples.size(); ++i)
    LatinHypercube(sample->bsdf[i], sample->nBSDFSamples[i], 2);
```

```
408 LatinHypercube
216 lensSamples
494 size
216 timeSamples
198 xPixelEnd
202 XPos
202 YPos
```

7.4 Low-Discrepancy Sequences

⟨*hammersley.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "sampling.h"
#include "paramset.h"
⟨HammersleySampler Declarations⟩
⟨HammersleySampler Method Definitions⟩
```

The underlying goal that the `StratifiedSampler` strives for is to generate a well-distributed set of sample points, where no two sample points are too close together, and where there aren't any excessively large regions of the image with no samples in them. As Figure 7.7 showed, the jittered pattern does this much better than a random pattern does, though its quality can suffer when samples in adjacent strata happen to be close to the shared boundary of the strata.

Definition of Discrepancy

Mathemeticians have developed a concept called *discrepancy* that can be used to evaluate the quality of a pattern of sample positions. Patterns that are well-distributed (in a manner to be formalized shortly) have low discrepancy values.

One can thus consider the sample pattern generation problem to be one of finding a suitable *low-discrepancy* pattern of points. A number of deterministic techniques have been developed that generate low-discrepancy point sets; this section will use one of them, the Hammersley point set, as the basis for a low-discrepancy sample generator.

Before defining the `HammersleySampler`, we will first introduce a formal definition for discrepancy. The basic idea behind it is that the quality of a set of a set of points in an n dimensional space $[0,1]^n$ can be evaluated by looking at regions of the domain $[0,1]^n$, counting the number of points inside the region, and comparing the volume of these regions to the number of sample points inside them. In general, one fourth of the volume should have roughly one fourth of the sample points inside of it, and so forth. While it's not possible for this to always be the case, we can still try to use patterns that minimize the difference between the volume estimated by the points and the actual volume (the discrepancy.)

To compute the discrepancy of a set of points, we first pick a family of shapes B which are subsets of $[0,1]^n$. For example, boxes with one corner at the origin are often used. This corresponds to:

$$B = \{[0,v_1] \times [0,v_2] \times \cdots \times [0,v_s]\},$$

where $0 \leq v_i \leq 1$. Given a sequence of sample points $P = x_1, \ldots, x_N$, the discrepancy of P with respect to B is

$$D_N(B,P) = \sup_{b \in B} \left| \frac{\sharp\{x_i \in b\}}{N} - \lambda(b) \right|,$$

where $\lambda(b)$ is the volume of b. In other words, we're finding the maximum difference between the fraction of points inside one of the shapes and the volume of the shape. When the set of shapes B is the set of boxes with a corner at the origin (described above), this is called the star discrepancy $D_N^*(P)$. (Other popular sets of shapes to use to compute discrepancy include axis aligned boxes, where the restriction that one corner be at the origin has been removed, and hyperplanes that cut the domain into two pieces.)

For a few particular point sets, the discrepancy can be computed analytically. For example, consider the set of points in one dimension

$$x_i = \frac{i}{N}.$$

We can see that the star discrepancy of x_i is

$$D_N^*(x_1, \ldots, x_n) = \frac{1}{N}.$$

For example, take the interval $B = [0, \frac{1}{N})$. Then $\lambda(B) \approx \frac{1}{N}$, but $\sharp\{x_i \in B\} = 0$. This interval (and the intervals $[0, \frac{2}{N})$, etc.) is the interval where the largest differences between volume and fraction of points is.

We can improve on the star discrepancy of this sequence by modifying it slightly:

$$x_i = \frac{i - \frac{1}{2}}{N}.$$

Then

$$D_N^*(x_i) = \frac{1}{2N}.$$

A theorem due to H. Niederreiter provides some bounds for the star discrepancy of a sequence of points in 1D:

$$D_N^*(x_i) = \frac{1}{2N} + \max_{1 \le i \le N} \left| x_i - \frac{2i-1}{2N} \right|.$$

Thus, the second example's sequence has the lowest possible discrepancy for a sequence in 1D. In general, it is much easier to analyze and compute bounds for the discrepancy of sequences in 1D than in higher dimensions. For less simple point sequences, and for sequences in higher dimensions, the discrepancy generally must be estimated numerically, by constructing a large number of shapes B and computing their discrepancy.

Constructing low-discrepancy sequences

Given the goal of constructing a low-discrepancy sequence, we will now introduce techniques that have been developed specifically to generate sequences of points that have low discrepancy. The techniques that we will describe are all built on top of a construction called the *radical inverse*. It is based on the fact that an integer value n can be expressed in base b with a sequence of digits $a_m \dots a_2 a_1$ uniquely determined by:

$$n = \sum_{i=1}^{\infty} a_i b^{i-1}$$

Then, the radical inverse function Φ_b in base b takes an non-negative integer and converts it to a floating-point value in $[0,1)$, by reflecting these digits about the decimal point:

$$\Phi_b(n) = 0.a_1 a_2 \dots a_m$$

The function `RadicalInverse()` computes the radical inverse for a given number `n` in the base `base`. It first computes the value of a_1 by taking the remainder of the number n when divided by the base. It then divides n by the base, effectively chopping off the last digit so that the next time through the loop, it can compute a_2 by finding the remainder base b, etc. This process continues until n is zero, at which point we have found the last non-zero a_i value.

⟨*Global Inline Functions*⟩+≡

```
inline Float RadicalInverse(int n, int base) {
    Float val = 0;
    Float invBase = 1.f / base, scale = invBase;
    ++n;
    while (n > 0) {
        ⟨Compute next digit of radical inverse⟩
    }
    return val;
}
```

As we are computing the digits base b, a_i, we can incrementally construct the value of the radical inverse. The contribution of a_i to the radical inverse is

$$a_i \cdot \frac{1}{b^i},$$

so we incrementally update the value `ib`, which holds the value $1/b^i$ each time through the loop.

⟨*Compute next digit of radical inverse*⟩≡
```
int digit = (n % base);
val += digit * scale;
n /= base;
scale *= invBase;
```

Given the `RadicalInverse()` function, we can start constructing low discrepancy sequences. One of the simplest low discrepancy sequences is the Van Der Corput Sequence, which is a one-dimensional sequence given by the radical inverse function in base two.

$$x_i = \Phi_2(i)$$

n	base 2	Φ_2 (n)	
1	1	.1	$= 1/2$
2	10	.01	$= 1/4$
3	11	.11	$= 3/4$
4	100	.001	$= 3/8$
5	101	.101	$= 5/8$
⋮	⋮	⋮	

Figure 7.9: The radical inverse of the first few positive integers, computed in base 2: Φ_2 (n). Notice how successive values of Φ_2 (n) are far from all previous values of Φ_2 (n).

Figure 7.9 shows the first few values of the Van Der Corput sequence; notice how it recursively splits the intervals of the 1D line in half. The discrepancy of this sequence is

$$D_N^*(P) = O\left(\frac{\log N}{N}\right),$$

which matches the best discrepancy that has been attained for infinite sequences of n dimensions,

$$D_N^*(P) = O\left(\frac{(\log N)^s}{N}\right).$$

Two well-known low-discrepency sequences that are defined in an arbitrary number of dimensions are the *Halton* and *Hammersley* sequences. Both use the radical inverse function as well.

To generate an n dimensional Halton sequence, we use the radical inverse base b, with a different base for each dimension and where the bases used are all relatively prime to each other. (A natural choice is to use the first n prime numbers.)

$$x_i = (\Phi_2(i), \Phi_3(i), \Phi_5(i), \ldots, \Phi_{p_n}(i)).$$

One of the most useful characteristics of the Halton sequence is that it can be used even if the total number of samples needed isn't known in advance; all prefixes of a given sequence are well-distributed, so thus as additional samples are added to the sequence, the low-discrepancy property will be maintained. The discrepancy of Halton sequences is

$$D_N^*(x_i) = O\left(\frac{(\log N)^s}{N}\right),$$

which is good.

If the number of samples to be taken is known in advance, the discrepancy can be improved slightly. Hammersley point sets are defined by:

$$x_i = \left(\frac{i-\frac{1}{2}}{N}, \Phi_2(i), \Phi_3(i), \ldots, \Phi_{p_n}(i)\right),$$

where N is the total number of samples to be taken and as before all of the bases b are relatively prime.

The *folded radical inverse* function can be used to reduce the discrepancy of Hammersley and Halton sequences by substituting it for the original radical inverse function defined above. It is defined by adding the offset i to the ith digit a_i and taking the result modulus b before adding the result to the next digit to the right of the decimal point.

$$\Psi_b(n) = \sum_i ((a_i + i - 1) \mod b) \cdot \frac{1}{b^i},$$

The `FoldedRadicalInverse()` function computes Ψ_b. It is generally similar to the original `RadicalInverse()` function, with two modifications. First, it needs to track which digit is currently being processed, so that the appropriate offset can be added before the modulus; this is done in the `modOffset` variable. Second, it needs to handle the fact that Ψ_b is actually an infinite sum–even though the digits a_i are zero after a finite number of terms, the offset that is added ensures that all except $1/b$ terms beyond the point where $a_i = 0$ will be non-zero. Fortunately, the finite precision of computer floating-point numbers solves this problem: we can stop adding digits to the folded radical inverse as soon as we detect that `ib` is small enough such that adding its contribution to `val` is certain to leave `val` unchanged. The test in the `while` loop watches for this to happen.

⟨*Global Inline Functions*⟩+≡

```
inline Float FoldedRadicalInverse(int n, int base) {
    Float val = 0;
    Float invBase = 1.f/base, scale = invBase;
    ++n;
    int modOffset = 0;
    while (val + base * scale != val) {
        ⟨Compute next digit of folded radical inverse⟩
    }
    return val;
}
```

⟨*Compute next digit of folded radical inverse*⟩≡
```
int digit = ((n+modOffset) % base);
n /= base;
val += digit * scale;
scale *= invBase;
++modOffset;
```

Graphs of the first 100 Halton and Hammersley points are shown in Figure 7.10. It's possible to see that the Hammersley sequence has lower discrepancy than the Halton sequence–there are far fewer clumps of nearby sample points. Furthermore, one can see that the folded radical inverse function reduces the discrepancy of the Hammersley sequence; its effect on the Halton sequence is less visually clear, however.

The Hammersly sample generator

The `HammersleySampler` uses the folded radical inverse function to generate a Hammersley point set for image sampling. It works by mapping the first two dimensions of the Hammersley points from $[0,1]^2$ to a square region on the image plane, starting at (`xPixelStart`, `yPixelStart`) and scaled by a constant amount in both directions so that it covers the pixels up to (`xPixelEnd`, `yPixelEnd`). Any generated samples that are past (`xPixelEnd`, `yPixelEnd`) are discarded. The total number of samples generated is determined by computing the total number of pixels in the extent that is being sampled times the number of samples to be taken per-pixel.

Sampler 198

For non-square images, it's important to use the approach described above, generating extra samples and rejecting those that are outside of the image region, rather than scaling the Hammersley point set by different amounts in the x and y directions. Scaling by different amounts would effectively cause the samples to be more closely spaced in one direction than the other, which is certainly not what one expects when rendering a non-square image.

⟨*HammersleySampler Declarations*⟩≡
```
class HammersleySampler : public Sampler {
public:
    ⟨HammersleySampler Method Declarations⟩
private:
    ⟨HammersleySampler Private Data⟩
};
```

The constructor computes the length of a side of the square region samples are generated inside of, `extent`, the total number of samples to generate (and its inverse), `nSamples` and `invNSamples`, and the sample number i of the next Hammersley point x_i to be computed by `GetNextSample()`.

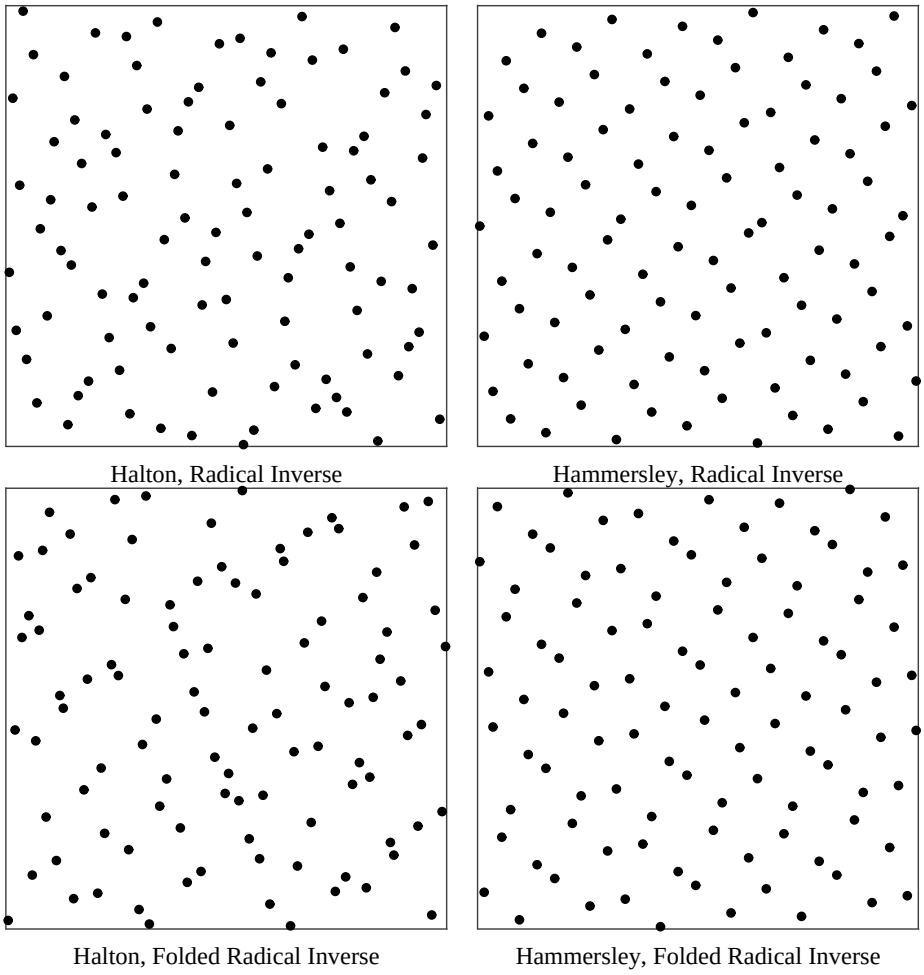


Figure 7.10: Graphs of the first 100 points in the Halton (left) and Hammersley (right) low-discrepancy point sequences, using the radical inverse Φ_b in the top row, and the folded radical inverse Ψ_b in the bottom row. The Hammersley sequence has lower discrepancy than the Halton sequence, at the cost of requiring that the number of samples to be taken be known in advance. The folded radical inverse function improves the discrepancy of both sequences.

⟨*HammersleySampler Method Definitions*⟩≡
```
HammersleySampler::HammersleySampler(int xres, int yres, int xpix,
        int ypix, const Extent2D &crop, Filter *f, bool uf)
    : Sampler(xres, yres, xpix, ypix, crop, f) {
    extent = max((xPixelEnd - xPixelStart + 1),
        (yPixelEnd - yPixelStart + 1));
    nSamples = xPixelSamples * yPixelSamples * extent * extent;
    invNSamples = 1.f / nSamples;
    curSample = 0;
    useFolded = uf;
}
```

⟨*HammersleySampler Private Data*⟩≡
```
int extent, nSamples;
int curSample;
Float invNSamples;
bool useFolded;
```

The `HammersleySampler` keeps generating points until a total of `nSamples` have been returned, after which it returns `false`, indicating that it has no more to provide.

Extent2D 27
Filter 229
GetNextSample 198
HammersleySampler 210
max 513
Sampler 198
xPixelEnd 198
xPixelSamples 198
yPixelEnd 198
yPixelSamples 198

In the implementation below, we use either the `FoldedRadicalInverse()` function (to give a Hammersley-Zaremba point set) or the `RadicalInverse()` function (to give a Hammersley point set), based on the `useFolded` parameter. With either one of these functions, it is substantially more expensive computationally to generate image samples than it is for the `StratifiedSampler` of the previous section or the `BestCandidateSampler` that will be introduced in the next section (roughly ten times as expensive computationally.)

For very simple scenes, where the cost of tracing a camera ray and computing its contribution is low, it may be more efficient to trace more rays generated by a lower-quality sample generation method to render an image of a particular quality level than it is to trace fewer rays that are "better", since the cost of generating the samples may dominate. For more complex scenes, however, where computing the contribution of a camera ray is more expensive, we can afford to spend more time to compute very good samples, since a reduction in the total number of samples that need to be taken can make up for the expense of computing the samples.

⟨*HammersleySampler Method Definitions*⟩+≡
```
bool HammersleySampler::GetNextSample(Sample *sample) {
tryAgain:
    if (curSample == nSamples) return false;
    ⟨Compute Hammersley (x,y) image sample location⟩
    ⟨Compute remaining dimensions of Hammersley sample⟩
    ++curSample;
    return true;
}
```

We start by computing the raster-space (x,y) image sample position. We immediately check to make sure that it is inside the region of pixels that need samples generated for it, so that we can skip generating the remainder of the dimensions in case it is out of bounds.

⟨*Compute Hammersley* (x,y) *image sample location*⟩≡
```
Float x = curSample * invNSamples;
Float y = useFolded ? FoldedRadicalInverse(curSample, 2) :
    RadicalInverse(curSample, 2);
sample->imagex = xPixelStart + x * extent;
sample->imagey = yPixelStart + y * extent;
if (sample->imagex > xPixelEnd || sample->imagey > yPixelEnd) {
    ++curSample;
    goto tryAgain;
}
sample->imagex -= .5f;
sample->imagey -= .5f;
```

Now that we know that we've got a valid image sample, we compute the sample points for the rest of the dimensions.

⟨*Compute remaining dimensions of Hammersley sample*⟩≡
```
static int primes[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
    41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103 };
sample->time = useFolded ? FoldedRadicalInverse(curSample, 3) :
    RadicalInverse(curSample, 3);
sample->lensx = useFolded ? FoldedRadicalInverse(curSample, 5) :
    RadicalInverse(curSample, 5);
sample->lensy = useFolded ? FoldedRadicalInverse(curSample, 7) :
    RadicalInverse(curSample, 7);
⟨Compute low-discrpeancy light and BSDF samples⟩
```

209 FoldedRadicalInverse
210 HammersleySampler
207 RadicalInverse
515 RandomFloat
494 size
198 xPixelEnd
198 yPixelEnd

For now, Cranley–Patterson stuff. Should do Köllig and Keller stuff...

⟨*Compute low-discrpeancy light and BSDF samples*⟩≡
```
for (u_int i = 0; i < sample->nLightSamples.size(); ++i)
    RotateLD2D(sample->light[i], sample->nLightSamples[i]);
for (u_int i = 0; i < sample->nBSDFSamples.size(); ++i)
    RotateLD2D(sample->bsdf[i], sample->nBSDFSamples[i]);
```

⟨*HammersleySampler Method Definitions*⟩+≡
```
void HammersleySampler::RotateLD2D(Float *samp, int nSamples) const {
#define WRAP(x) ((x) > 1 ? ((x)-1) : (x))
    Float shift = RandomFloat();
    for (u_int i = 0; i < nSamples; ++i) {
        Float s1 = (Float)i / (Float)nSamples;
        Float s2 = useFolded ? FoldedRadicalInverse(i, 2) :
            RadicalInverse(i, 2);
        samp[2*i]   = WRAP(s1 + shift);
        samp[2*i+1] = WRAP(s2 + shift);
    }
#undef WRAP
}
```

7.5 Best-Candidate Sampling Patterns

⟨*bestcandidate.cc**⟩≡
⟨*Source Code Copyright*⟩
```
#include "sampling.h"
#include "paramset.h"
```
⟨*BestCandidateSampler Declarations*⟩
⟨*BestCandidateSampler Method Definitions*⟩

Though usually better than uniform sampling, the jittered sampling pattern still has shortcomings: when choosing a sample position in one cell, we don't account for sample positions in nearby cells to try to keep adjacent samples from clumping together. Ideally, all of the sample positions across the image would be optimized so that there are as few clumps of nearby samples as possible.

For example, a *Poisson disk pattern* has been shown to be an excellent image sampling pattern. The Poisson disk pattern is a group of points such that no two of them are closer than some specified distance. Studies have shown that the rods and cones in the eye are distributed in a Poisson disk-like pattern, which suggests that this pattern might be effective for imaging.

Poisson disk patterns are usually generated by *dart throwing*: we keep generating random samples, throwing away all that are closer to a previous sample than a fixed threshold distance. This can be a very expensive process, since many darts may be necessary. Another approach is the *best candidate* algorithm. When a new sample is to be computed, a large number of random candidates are generated; all of these candidates are compared to the previous samples and the one that is farthest away from all of the previous samples is added to the pattern. Although this algorithm doesn't guarantee the Poisson disk property, it usually does quite well if enough candidates are generated. Another advantage it has is that any prefix of the final pattern is itself a well-distributed sampling pattern. Furthermore, it's easier to generate a good pattern with a pre-chosen number of samples with the best candidate algorithm than it is with a dart throwing algorithm.

In this section we will present an implementation of the best-candidate algorithm and its extension to computing sampling patterns that include good distributions of samples in additional dimensions. Because it is a computationally-intensive algorithm, we will compute a good sampling pattern once in a pre-process. The pattern can then be stored in a table and efficiently used at rendering-time.

Rather than computing a sampling pattern large enough to sample the most enormous image we'd ever render, we'll compute a pattern that can be reused by tiling it over the image plane by translating and scaling it appropriately. This means that we must consider it to have *toroidal topology*. When computing the distance between two samples, we must compute the distance between them as if the square sampling region was rolled into a torus. Thus, for these purposes points at the top of the region may have a very small distance to points at the bottom, etc.

Generating the best-candidate pattern

⟨*samplepat.cc*⟩≡
```
#include "lrt.h"
#include "sampling.h"
```
⟨*Sample Pattern Precomputation*⟩

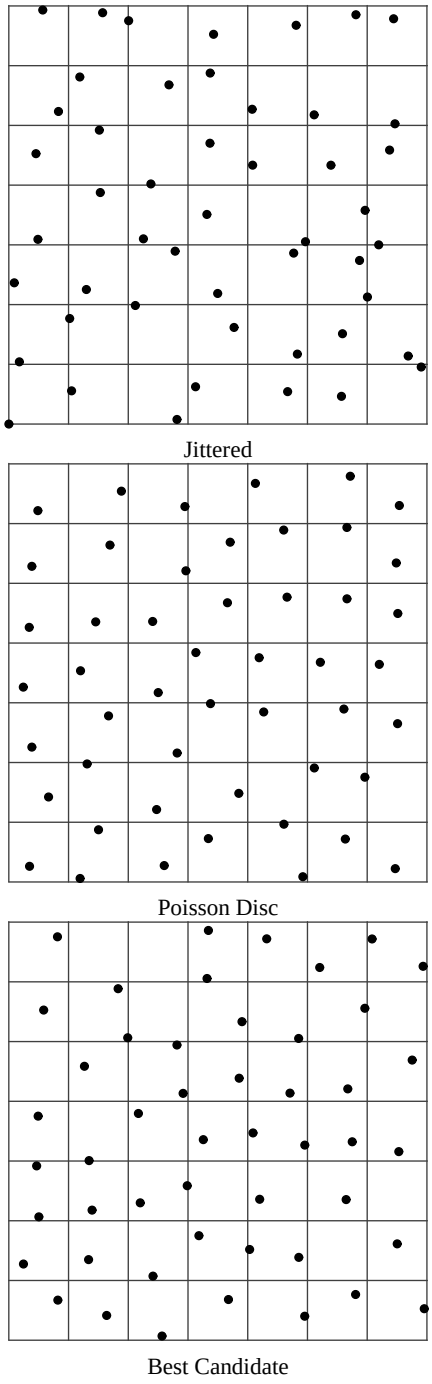


Figure 7.11: Comparison of sampling patterns. On the top is a jittered pattern: note clumping of samples and undersampling in some areas. In the middle is a Poisson disk pattern generated by dart-throwing. No two samples are closer than a fixed threshold, and although there is no guarantee that there will be one sample in each of the strata, this is usually the case. On the bottom is a pattern generated with the best-candidate algorithm; it is nearly as good as the Poisson disk pattern. (Due to its toroidal topology, the two strata at the top left with no samples have samples very close to them from the bottom left part, etc.)

We will now show the program that generates the samples in an off-line computation. First we need to define the size of the table that we will be generating.

⟨*Sampling Constants*⟩≡
```
#define SQRT_SAMPLE_TABLE_SIZE 64
#define SAMPLE_TABLE_SIZE (SQRT_SAMPLE_TABLE_SIZE * \
                           SQRT_SAMPLE_TABLE_SIZE)
```

Recall that we need to generate sample points in a nine-dimensional space. Two dimensions determine the image sample location, one determines a point in time, two more determine a point on a lens (for depth of field), and four more are potentially used when computing Monte Carlo estimates of light transport (see Chapters 14 and 15.) When generating the samples we store each of these sets of samples in a separate array.

⟨*Pattern Precomputation Local Data*⟩≡
```
static Float pixelSamples[SAMPLE_TABLE_SIZE][2];
static Float timeSamples[SAMPLE_TABLE_SIZE];
static Float lensSamples[SAMPLE_TABLE_SIZE][2];
static Float bsdfSamples[SAMPLE_TABLE_SIZE][2];
static Float lightSamples[SAMPLE_TABLE_SIZE][2];
```

Here is the `main` function for the off-line sample computation program. We compute sample values in a multi-stage process. First, we generate a well-distributed set of image sample positions. Then, given the image samples, we generate a good set of time samples. Finally, we generate good samples for the lens, BSDF and light sampling.

⟨*Sample Pattern Precomputation*⟩+≡
```
int main() {
    ⟨Compute image sample positions⟩
    ⟨Compute time samples⟩
    ⟨Compute lens, BSDF, and light samples⟩
    ⟨Output sample table⟩
    return 0;
}
```

In order to speed up the candidate evaluation, we will store the accepted samples in a grid. This allows us to only check nearby samples when computing distances. The grid splits up the 2D sample domain $[0,1]^2$ into `BC_GRID_SIZE` strata in each direction and stores a list of the integer sample numbers of the samples that overlap each cell.

⟨*Global Forward Declarations*⟩+≡
```
#define BC_GRID_SIZE 40
typedef vector<int> SampleGrid[BC_GRID_SIZE][BC_GRID_SIZE];
#define GRID(v) (int((v) * BC_GRID_SIZE))
```

To compute the image samples, we start by allocating a sample grid and calling a function to run the 2D best candidate algorithm.

⟨*Compute image sample positions*⟩≡
```
SampleGrid pixelGrid;
BestCandidate2D(pixelSamples, SAMPLE_TABLE_SIZE, &pixelGrid);
```

For the best candidate algorithm, the first image sample position is chosen completely arbitrarily and recorded in the grid. For all subsequent samples, we generate a set of candidates that are compared to the already-computed samples.

⟨*Sample Pattern Precomputation*⟩+≡
```
void BestCandidate2D(Float table[][2], int totalSamples,
        SampleGrid *grid) {
    SampleGrid localGrid;
    if (!grid) grid = &localGrid;
    cerr << "Throwing darts: ";
    ⟨Generate first 2D sample arbitrarily⟩
    for (int currentSample = 1; currentSample < totalSamples;
        ++currentSample) {
        if ((currentSample % (totalSamples/60)) == 0)
            cerr << '+';
        ⟨Generate next best 2D image sample⟩
    }
    cerr << endl;
}
```

To start off the process, we can choose any random point for the first sample; only the second sample and beyond need to be checked against previous samples.

⟨*Generate first 2D sample arbitrarily*⟩≡
```
table[0][0] = RandomFloat();
table[0][1] = RandomFloat();
addSampleToGrid(table, 0, grid);
```

A short utility function adds the `entry`th item in the given table of samples to the given `SampleGrid`.

⟨*Pattern Precomputation Utility Functions*⟩≡
```
static void addSampleToGrid(Float table[][2], int entry,
        SampleGrid *grid) {
    int u = GRID(table[entry][0]);
    int v = GRID(table[entry][1]);
    (*grid)[u][v].push_back(entry);
}
```

To generate the rest of the samples, we will use a dart throwing algorithm that throws a number of candidate darts for each needed sample. The number of darts thrown is proportional to the number of samples we have already; this ensures that the quality of the samples as we go is in some sense consistent. After throwing a dart, we see how close it is to all of the samples we've generated so far. If it's farther away from all of the accepted samples than the previous best candidate was, we keep it. At the end of the loop, the remaining candidate is kept.

216 pixelSamples
494 push_back
515 RandomFloat
216 SAMPLE_TABLE_SIZE
216 SampleGrid

⟨*Generate next best 2D image sample*⟩≡
```
Float maxDist2 = 0.;
int numCandidates = 500 * currentSample;
for (int currentCandidate = 0; currentCandidate < numCandidates;
        ++currentCandidate) {
    ⟨Generate a random candidate sample⟩
    ⟨Loop over neighboring grid cells and check distances⟩
    ⟨Keep this sample if it is the best one so far⟩
}
addSampleToGrid(table, currentSample, grid);
```

Candidate positions are chosen completely at random. Note that we're computing image sample locations in the range $[0, 1)$; it'll be up to the `Sampler` that uses the sampling pattern to scale and translate image samples into raster-space appropriately.

⟨*Generate a random candidate sample*⟩≡
```
Float candidate[2];
candidate[0] = RandomFloat();
candidate[1] = RandomFloat();
```

addSampleToGrid 217
currentSample 217
INFINITY 514
RandomFloat 515

Now that we have a candidate, we see if it's the best candidate we've come up with so far. We compute the distances to all of the already-generated samples, keeping track of the minimum of all of the distances. Whichever candidate that has the largest minimum distance is the best. For efficiency, we will actually just compute the squared distance, which gives the same result for this test and saves us a lot of expensive square root computations.

We actually only compute distances to the eight neighboring grid cells and the cell that the candidate is in; although this means that the first few samples are not optimally distributed relative to each other, this doesn't matter by the time we are done computing samples, so long as `BC_GRID_SIZE` $<$ `SQRT_SAMPLE_TABLE_SIZE`.

⟨*Loop over neighboring grid cells and check distances*⟩≡
```
Float sampleDist2 = INFINITY;
int gu = GRID(candidate[0]);
int gv = GRID(candidate[1]);
for (int du = -1; du <= 1; ++du) {
    for (int dv = -1; dv <= 1; ++dv) {
        ⟨Compute (u,v) grid cell to check⟩
        ⟨Update minimum squared distance from cell's samples⟩
    }
}
```

We do need to handle the toroidal topology here, though; if the grid cell we'd like to consider is out of bounds, we wrap around to the other end of the grid.

⟨*Compute (u,v) grid cell to check*⟩≡
```
int u = gu + du, v = gv + dv;
if (u < 0)              u += BC_GRID_SIZE;
if (u >= BC_GRID_SIZE) u -= BC_GRID_SIZE;
if (v < 0)              v += BC_GRID_SIZE;
if (v >= BC_GRID_SIZE) v -= BC_GRID_SIZE;
```

We now loop over the list of sample numbers for the samples in the grid cell we're considering. For each one, we compute the squared distance to the current candidate, recording the lowest squared distance of all the ones we check.

⟨*Update minimum squared distance from cell's samples*⟩≡
```
for (u_int g = 0; g < (*grid)[u][v].size(); ++g) {
    int s = (*grid)[u][v][g];
    Float xdist = Wrapped1DDist(candidate[0], table[s][0]);
    Float ydist = Wrapped1DDist(candidate[1], table[s][1]);
    Float d2 = xdist*xdist + ydist*ydist;
    sampleDist2 = min(sampleDist2, d2);
}
```

When we compute the 1D distance between two values in $[0,1]$, we need to handle the wrap-around issue. Consider two samples with x coordinates of .01 and .99, respectively. Direct computation will find their distance to be .98, though with wrap-around, the actual distance should be .02. Because we're only checking distances to samples in adjacent grid-cells, we can easily detect this situation when one of the distances is greater than 0.5. In that case, the true distance is just the sum of the distance from the higher sample to one plus the distance from zero to the lower sample.

⟨*Pattern Precomputation Utility Functions*⟩+≡
```
inline Float Wrapped1DDist(Float a, Float b) {
    Float d = fabsf(a - b);
    if (d < .5) return d;
    else return 1 - max(a, b) + min(a, b);
}
```

216 BC_GRID_SIZE
218 candidate
217 currentSample
513 max
218 maxDist2
513 min
218 sampleDist2
494 size

Finally, we see if this candidate has the highest squared distance to its neighbors. If so, we record its distance and tentatively put it in the output `table`.

⟨*Keep this sample if it is the best one so far*⟩≡
```
if (sampleDist2 > maxDist2) {
    maxDist2 = sampleDist2;
    table[currentSample][0] = candidate[0];
    table[currentSample][1] = candidate[1];
}
```

Now that we've got all of the image samples that we want, we turn to computing the sample positions for the rest of the dimensions. One might think that a good sample pattern could be computed by generalizing the Poisson disk concept to a higher-dimensional Poisson sphere. Interestingly enough, we can do better than this. (In the nine-dimensional case in particular, a large number of candidate samples would be needed to find good ones, anyway.)

Consider the problem of choosing time values for two nearby image samples: not only do we want the time values to not be too close together, but in fact, it's even better if the time values are as far apart as possible—in any local 2D region of the image, we'd like the best possible coverage of the complete three-dimensional sample space.

An intuition for why this is the case comes from how the sampling pattern will be used. Although we're generating a nine-dimensional pattern overall, what we're

interested in is optimizing its distribution across local areas of the two-dimensional image plane; optimizing its distribution over the nine-dimensional space is only a secondary concern.

Therefore, we'll use a two stage process for generating the sample positions. First, we will generate a well-distributed sampling pattern for the time values and for the two-dimensional lens, BSDF, and light values. Then, we will associate these samples with image samples in a way that ensures that nearby image samples have sample values for the other dimensions that are well spread-out.

As if that wasn't enough to worry about, we should also be considering *correlation*. Not only should nearby pixel samples have distant sample values for the other dimensions, but we should also make sure that, for example, the time and lens values aren't correlated: if we somehow kept choosing samples such that the time value was always similar to the lens *u* sample value, the sample pattern is not as good as it would be if the two were uncorrelated. We won't address this issue in our approach below, though at least none of our techniques are prone to introducing correlation.

For time, we generate a set of one-dimensional stratified sample values over $[0,1]$. When we're done, we will rearrange the `timeValues` array so that the i'th time sample is a good one for the i'th image sample.

Assert 498
currentSample 217
RandomFloat 515
SAMPLE_TABLE_SIZE 216
swap 513
timeSamples 216

⟨*Compute time samples*⟩≡
```
cerr << "Computing time samples: ";
for (int i = 0; i < SAMPLE_TABLE_SIZE; ++i)
    timeSamples[i] = (i + RandomFloat()) / SAMPLE_TABLE_SIZE;
for (int currentSample = 1; currentSample < SAMPLE_TABLE_SIZE;
    ++currentSample) {
    if ((currentSample % (SAMPLE_TABLE_SIZE/52)) == 0)
        cerr << '+';
    ⟨Select best time sample for current image sample⟩
}
cerr << endl;
```

⟨*Select best time sample for current image sample*⟩≡
```
int best = -1;
⟨Find best time relative to neighbors⟩
Assert(best != -1);
swap(timeSamples[best], timeSamples[currentSample]);
```

Given that we're working on finding a good time for the sample number `currentSample`, the elements of `timeSamples` from zero to `currentSample-1` have already been assigned to previous image samples and are unavailable to us. The rest of the times, from `currentSample` to `SAMPLE_TABLE_SIZE-1`, are the ones we will choose from.

⟨*Find best time relative to neighbors*⟩≡
```
Float maxMinDelta = 0.;
for (int t = currentSample; t < SAMPLE_TABLE_SIZE; ++t) {
    ⟨Compute min delta for this time⟩
    ⟨Update best if this is best time so far⟩
}
```

As when we were doing dart-throwing for image samples, we only look at the samples in the adjoining few grid cells. Of these, we will select the one that is most different than the time samples that have already been assigned to the nearby image samples.

⟨*Compute min delta for this time*⟩≡
```
int gu = GRID(pixelSamples[currentSample][0]);
int gv = GRID(pixelSamples[currentSample][1]);
Float minDelta = INFINITY;
for (int du = -1; du <= 1; ++du) {
    for (int dv = -1; dv <= 1; ++dv) {
        ⟨Check distance from times of nearby samples⟩
    }
}
```

We loop through the samples in each of the grid cells, though we need to be careful to only consider the ones that already have time samples associated with them. Therefore, we skip over the ones that have sample numbers greater than the sample we're currently working to find a time value for. For the remaining ones, we compute the distance for their time sample to the current candidate time sample, keeping track of the minimum difference.

217 currentSample
514 INFINITY
513 min
216 pixelSamples
494 size
216 timeSamples
219 Wrapped1DDist

⟨*Check distance from times of nearby samples*⟩≡
```
⟨Compute (u,v) grid cell to check⟩
for (u_int g = 0; g < pixelGrid[u][v].size(); ++g) {
    int otherSample = pixelGrid[u][v][g];
    if (otherSample < currentSample) {
        Float dt = Wrapped1DDist(timeSamples[otherSample],
            timeSamples[t]);
        minDelta = min(minDelta, dt);
    }
}
```

If the minimum distance from the current time sample is greater than the minimum distance of the previous best time sample, we update our records.

⟨*Update* `best` *if this is best time so far*⟩≡
```
if (minDelta > maxMinDelta) {
    maxMinDelta = minDelta;
    best = t;
}
```

We now go ahead and do the rest of the dimensions in turn. We generate good two-dimensional sampling patterns using dart throwing and then associate these samples with image samples in the same manner that we assigned times to image samples.

⟨*Compute lens, BSDF, and light samples*⟩≡
```
BestCandidate2D(lensSamples, SAMPLE_TABLE_SIZE);
Redistribute2D(lensSamples, pixelGrid);
BestCandidate2D(bsdfSamples, SAMPLE_TABLE_SIZE);
Redistribute2D(bsdfSamples, pixelGrid);
BestCandidate2D(lightSamples, SAMPLE_TABLE_SIZE);
Redistribute2D(lightSamples, pixelGrid);
```

After the `BestCandidate2D()` function generates a good set of 2D samples, the `Redistribute2D()` utility function takes the set of samples to assign to the image samples and reshuffles them like we reshuffled the time samples to give them a good distribution with respect to their neighbors.

⟨*Sample Pattern Precomputation*⟩+≡
```
static void Redistribute2D(Float samples[][2],
        SampleGrid &pixelGrid) {
    cerr << "Redistributing: ";
    for (int currentSample = 1;
        currentSample < SAMPLE_TABLE_SIZE; ++currentSample) {
        if ((currentSample % (SAMPLE_TABLE_SIZE/52)) == 0)
            cerr << '+';
        ⟨Select best sample for current image sample⟩
    }
    cerr << endl;
}
```

Assert 498
BestCandidate2D 217
bsdfSamples 216
currentSample 217
lensSamples 216
lightSamples 216
SAMPLE_TABLE_SIZE 216
SampleGrid 216
swap 513

⟨*Select best sample for current image sample*⟩≡
```
int best = -1;
⟨Find best 2D sample relative to neighbors⟩
Assert(best != -1);
swap(samples[best][0], samples[currentSample][0]);
swap(samples[best][1], samples[currentSample][1]);
```

As with time, we want to choose the sample from the available ones that maximizes the minimum distance to the sample values that have already been assigned to the neighboring image samples.

⟨*Find best 2D sample relative to neighbors*⟩≡
```
Float maxMinDist2 = 0.f;
for (int samp = currentSample; samp < SAMPLE_TABLE_SIZE;
        ++samp) {
    ⟨Check distance to nearby samples⟩
    ⟨Update best for 2D sample if it is best so far⟩
}
```

⟨*Check distance to nearby samples*⟩≡
```
int gu = GRID(pixelSamples[currentSample][0]);
int gv = GRID(pixelSamples[currentSample][1]);
Float minDist2 = INFINITY;
for (int du = -1; du <= 1; ++du) {
    for (int dv = -1; dv <= 1; ++dv) {
        ⟨Check 2D samples in current grid cell⟩
    }
}
```

⟨*Check 2D samples in current grid cell*⟩≡
```
⟨Compute (u,v) grid cell to check⟩
for (u_int g = 0; g < pixelGrid[u][v].size(); ++g) {
    int s2 = pixelGrid[u][v][g];
    if (s2 < currentSample) {
        Float dx = Wrapped1DDist(samples[s2][0],
            samples[samp][0]);
        Float dy = Wrapped1DDist(samples[s2][1],
            samples[samp][1]);
        Float d2 = dx*dx + dy*dy;
        minDist2 = min(d2, minDist2);
    }
}
```

217 currentSample
514 INFINITY
513 min
216 pixelSamples
494 size
219 Wrapped1DDist

⟨*Update* `best` *for 2D sample if it is best so far*⟩≡
```
if (minDist2 > maxMinDist2) {
    maxMinDist2 = minDist2;
    best = samp;
}
```

When we're all done, we open up a file and write out C++ code that initializes the table. When `lrt` is compiled, it will `#include` this file to initialize its sample table.

⟨*Output sample table*⟩≡
```
FILE *f = fopen("sampledata.cc", "w");
Assert(f);
fprintf(f, "\n/* Automatically generated %dx%d sample "
    "table (%s @ %s) */\n\n",
    SQRT_SAMPLE_TABLE_SIZE, SQRT_SAMPLE_TABLE_SIZE,
    __DATE__, __TIME__);
fprintf(f, "const Float BestCandidateSampler::sampleTable[%d][9] "
    "= {\n", SAMPLE_TABLE_SIZE);
for (int i = 0; i < SAMPLE_TABLE_SIZE; ++i) {
    fprintf(f, "  { ");
    fprintf(f, "%10.10ff, %10.10ff, ", pixelSamples[i][0],
        pixelSamples[i][1]);
    fprintf(f, "%10.10ff, ", timeSamples[i]);
    fprintf(f, "%10.10ff, %10.10ff, ", lensSamples[i][0],
        lensSamples[i][1]);
    fprintf(f, "%10.10ff, %10.10ff, ", bsdfSamples[i][0],
        bsdfSamples[i][1]);
    fprintf(f, "%10.10ff, %10.10ff, ", lightSamples[i][0],
        lightSamples[i][1]);
    fprintf(f, "},\n");
}
fprintf(f, "};\n");
```

Assert 498
bsdfSamples 216
lensSamples 216
lightSamples 216
pixelSamples 216
SAMPLE_TABLE_SIZE 216
Sampler 198
sampleTable 225
SQRT_SAMPLE_TABLE_SIZE 216
timeSamples 216

Using the best-candidate pattern

`BestCandidateSampler`, the `Sampler` that uses our sample table, is pretty straightforward. A single copy of the sample table covers

```
SQRT_SAMPLE_TABLE_SIZE / xPixelSamples
```

pixel extents in the x direction and analogously in y. As with the `StratifiedSampler`, we scan across the image from the upper left of the crop window, going left-to-right and then top-to-bottom. Here, we generate all samples inside the sample table's extent before advancing to the next region of the image that it covers.

⟨*BestCandidateSampler Declarations*⟩≡
```
class BestCandidateSampler : public Sampler {
public:
    ⟨BestCandidateSampler Method Declarations⟩
private:
    ⟨BestCandidateSampler Private Data⟩
};
```

We store our current raster-space pixel position in `XTablePos` and `YTablePos` where `XTableWidth` and `YTableWidth` are the raster-space widths in pixels that the precomputed sample table spans. `tableOffset` holds the current offset into the sample table; when it is advanced to the point where we are at the end of the table, we advance to the next region of the image that the table covers.

⟨*BestCandidateSampler Method Definitions*⟩≡
```
BestCandidateSampler::BestCandidateSampler(int xres, int yres, int xSamp,
        int ySamp, const Extent2D &crop, Filter *f)
    : Sampler(xres, yres, xSamp, ySamp, crop, f) {
    XTablePos = xPixelStart;
    YTablePos = yPixelStart;
    XTableWidth = (Float)SQRT_SAMPLE_TABLE_SIZE / xPixelSamples;
    YTableWidth = (Float)SQRT_SAMPLE_TABLE_SIZE / yPixelSamples;
    tableOffset = 0;
    ⟨Update sample shifts⟩
}
```

⟨*BestCandidateSampler Private Data*⟩≡
```
int tableOffset;
Float XTablePos, YTablePos;
Float XTableWidth, YTableWidth;
```

Here we incorporate the precomputed sample data.

⟨*BestCandidateSampler Private Data*⟩+≡
```
static const Float sampleTable[SAMPLE_TABLE_SIZE][9];
```

⟨*BestCandidateSampler Method Definitions*⟩+≡
```
#include "sampledata.cc"
```

224 BestCandidateSampler
27 Extent2D
229 Filter
515 RandomFloat
216 SAMPLE_TABLE_SIZE
198 Sampler
216 SQRT_SAMPLE_TABLE_SIZE
198 xPixelSamples
198 yPixelSamples

One problem that sometimes comes up when using replicated precomputed sample patterns is that there may be subtle image artifacts, aligned with the extent of the pattern on the image plane due to the same values being used repeatedly for time, lens position, etc. Not only are the same `SAMPLE_TABLE_SIZE` samples used and re-used (whereas the `StratifiedSampler` will at least generate different time values and so forth for each different image sample), but the upper left sample in each block of samples will always have the time value to boot.

On approach to this problem is to transform the set of sample values each time before starting to re-use the pattern. Here, we use *Cranley-Patterson rotations*, where we compute in each dimension

$$X_i' = (X_i + \xi_i) \mod 1,$$

where X_i is the sample value and ξ_i is a random number between zero and one. Because the various sampling patterns were computed with toroidal topology, the resulting pattern is still well-distributed and seamless. The table of random offsets ξ_i is updated each time we are about to reuse the table once again.

⟨*Update sample shifts*⟩≡
```
for (int i = 0; i < 9; ++i)
    sampleOffsets[i] = RandomFloat();
```

⟨*BestCandidateSampler Private Data*⟩+≡
```
Float sampleOffsets[9];
```

The `GetNextSample()` has a similar structure to the one for `StratifiedSampler`

⟨*BestCandidateSampler Method Definitions*⟩+≡
```
bool BestCandidateSampler::GetNextSample(Sample *sample) {
again:
    ⟨Return false if BestCandidateSampler is done⟩
    ⟨Compute raster sample from table⟩
    ⟨Advance to next sample table position⟩
    ⟨Check sample against crop window, goto again if outside⟩
    return true;
}
```

As with the `StratifiedSampler`, we are done generating samples when the upper *y* coordinate of the region goes below the bottom of the crop window.

⟨*Return* `false` *if* `BestCandidateSampler` *is done*⟩≡
```
if (YTablePos >= yPixelEnd)
    return false;
```

It's just some simple indexing and scaling to compute the raster-space sample position. We don't use the shifting technique on image samples: this would cause the sampling points at the borders between repeated instances of the table to have a poor distribution; preserving good image-distribution is more important than reducing correlation. The rest of the dimensions are sampled using the shifting method described above, using the `WRAP` macro that ensures that the result stays between 0 and 1.

BestCandidateSampler 224
GetNextSample 198
RadicalInverse 207
RandomFloat 515
RotateLD2D 213
sampleTable 225
size 494
tableOffset 225
XTablePos 225
XTableWidth 225
yPixelEnd 198
YTablePos 225
YTableWidth 225

⟨*Compute raster sample from table*⟩≡
```
#define WRAP(x) ((x) > 1 ? ((x)-1) : (x))
sample->imagex = XTablePos + XTableWidth * sampleTable[tableOffset][0];
sample->imagey = YTablePos + YTableWidth * sampleTable[tableOffset][1];
sample->lensx = WRAP(sampleOffsets[2] + sampleTable[tableOffset][2]);
sample->lensy = WRAP(sampleOffsets[3] + sampleTable[tableOffset][3]);
sample->time = WRAP(sampleOffsets[4] + sampleTable[tableOffset][4]);
#undef WRAP
for (u_int i = 0; i < sample->nLightSamples.size(); ++i)
    RotateLD2D(sample->light[i], sample->nLightSamples[i]);
for (u_int i = 0; i < sample->nBSDFSamples.size(); ++i)
    RotateLD2D(sample->bsdf[i], sample->nBSDFSamples[i]);
```

⟨*BestCandidateSampler Method Definitions*⟩+≡
```
void BestCandidateSampler::RotateLD2D(Float *samp, int nSamples) {
#define WRAP(x) ((x) > 1 ? ((x)-1) : (x))
    Float shift = RandomFloat();
    for (u_int i = 0; i < nSamples; ++i) {
        Float s1 = (Float)i / (Float)nSamples;
        Float s2 = RadicalInverse(i, 2);
        samp[2*i]   = WRAP(s1 + shift);
        samp[2*i+1] = WRAP(s2 + shift);
    }
#undef WRAP
}
```

We now step to the next precomputed sample value; if we've hit the end of the sample table, we try to move `XTablePos` forward. If this leaves the raster extent of the image, we move `YTablePos` ahead.

⟨*Advance to next sample table position*⟩≡
```
if (++tableOffset == SAMPLE_TABLE_SIZE) {
    ⟨Update sample shifts⟩
    tableOffset = 0;
    XTablePos += XTableWidth;
    if (XTablePos >= xPixelEnd) {
        XTablePos = xPixelStart;
        YTablePos += YTableWidth;
    }
}
```

The sample table may partially spill off the end of the image plane, so some of the samples that we generate may be outside the necessary sample region. We detect this case by checking the sample against the pixel area to be sampled and generating a new sample if it's out of bounds.

⟨*Check sample against crop window, goto* `again` *if outside*⟩≡
```
if (sample->imagex <  xPixelStart ||
    sample->imagex >= xPixelEnd   ||
    sample->imagey <  yPixelStart ||
    sample->imagey >= yPixelEnd)
    goto again;
sample->imagex -= .5f;
sample->imagey -= .5f;
```

216 SAMPLE_TABLE_SIZE
225 tableOffset
198 xPixelEnd
225 XTablePos
225 XTableWidth
198 yPixelEnd
225 YTablePos
225 YTableWidth

7.6 Image Reconstruction

Given the non-uniform set of image samples, we need to compute a final value for each of the pixels in the output image. According to the signal processing framework, we need to do three things:

1. Reconstruct a continuous image function $\tilde{L}$ from the set of image samples.
2. Prefilter the function $\tilde{L}$ to remove any frequencies past the Nyquist limit for the pixel spacing.
3. Sample $\tilde{L}$ at the pixel locations to compute the final pixel values.

Because we know that we will only be resampling the $\tilde{L}$ at the pixel locations, we don't need to construct an explicit representation of the function and can also aggregate the function of the first two steps into a single filter function.

Recall that if the original function had been uniformly sampled at a frequency greater than the Nyquist frequency and reconstructed with the sinc filter, then the reconstructed function in the first step would match the original image function perfectly–quite a feat since we were only able to point-sample it. Because the original image function has higher frequencies than we were able to sample (due to edges, etc.), we chose to sample it non-uniformly, trading off noise for aliasing.

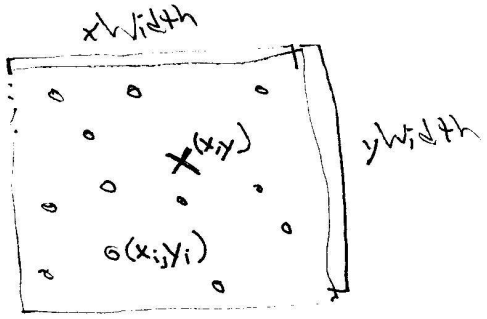


Figure 7.12: two-d image filter.

The theory behind the reconstruction equation, 7.1.1, depends on the samples being uniformly spaced; while a number of approaches have been used to try to extend it to non-uniform sampling, there is not yet as solid a footing for this. The most widely used method in graphics is based on interpolation of the samples around a pixel. To compute a final value for a pixel $p(x,y)$, this interpolation results in computing a weighted average:

$$p(x,y) = \frac{\sum_i f(x-x_i, y-y_i) L(x_i,y_i)}{\sum_i f(x-x_i, y-y_i)} \tag{7.6.2}$$

where $L(x_i,y_i)$ is the radiance value of the i'th sample, located at (x_i,y_i), and f is a filter function. See Figure 7.12, which shows a pixel at location (x,y), marked with an "x", that has a pixel filter with extent `xWidth` in the x direction and `yWidth` in the y direction. Image samples are denoted by dots, and all of the samples inside the box given by the filter extent may contribute to the pixel's value.

It turns out that the sinc filter doesn't give as good image quality as some other filters when used for filtering in this situation with non-uniform sample spacing and a sampling rate almost certainly below the Nyquist limit. For example, the sinc is prone to *ringing* artifacts, where edges in the image have faint replicated copies of the edge in nearby pixels. Furthermore, it is generally avoided for efficiency reasons because it has *infinite support*: it doesn't fall off to zero at a finite distance from its center. Otherwise all of the image sample values $L(x_i,y_i)$ would need be considered when computing a filtered value for a particular pixel. A number of other filters that have finite extent also give substantially better results in practice.

Filter Functions

First we will define the `Filter` class and an number of implementations of it. The `Filter` implements various filter functions $f(x,y)$ for use in the pixel filtering equation, 7.6.2.

⟨*Sampling Declarations*⟩ +≡
```
class Filter {
public:
    ⟨Filter Interface⟩
    ⟨Filter Data⟩
};
```

All filters have widths beyond which they have a value of zero; these may be different in the x and y directions. The constructor takes values for these and stores them for use by the sub-classes.

⟨*Filter Interface*⟩+≡
```
Filter(Float xw, Float yw)
    : xWidth(xw), yWidth(yw), halfXWidth(.5f*xw),
        halfYWidth(.5f*yw) {
}
```

⟨*Filter Data*⟩≡
```
const Float xWidth, yWidth;
const Float halfXWidth, halfYWidth;
```

The sole function that `Filter` implementations need to provide is the `Evaluate()` method. It takes an x and y argument, which are the position of the sample point *relative to the center of the filter*. The return value specifies the weight of the sample. We will never call the filter function with points outside of the filter's extent; therefore, individual filters don't need to check for this case.

⟨*Filter Interface*⟩+≡
```
virtual Float Evaluate(Float x, Float y) const = 0;
```

Box Filter

⟨*box.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "sampling.h"
#include "paramset.h"
⟨Box Filter Declarations⟩
⟨Box Filter Definitions⟩
```

One of the most commonly used filters in graphics is the *box filter* (and in fact, when filtering and reconstruction isn't addressed explicitly, the box filter is the *de facto* result. The box filter equally weights all samples within a square region of the image. Though computationally efficient, it's just about the worst filter possible. In practice, it allows high frequency sample data to leak into the output pixels, causing aliasing. The left side of Figure 7.13 shows a graph of the box filter.

⟨*Box Filter Declarations*⟩≡
```
class BoxFilter: public Filter {
public:
    BoxFilter(Float xw, Float yw) : Filter(xw, yw) { }
    Float Evaluate(Float x, Float y) const;
};
```

Because the evaluation function isn't called with (x,y) values outside of the filter's extent, we can always return 1 for the filter function's value.

⟨*Box Filter Definitions*⟩≡
```
Float BoxFilter::Evaluate(Float x, Float y) const {
    return 1.;
}
```

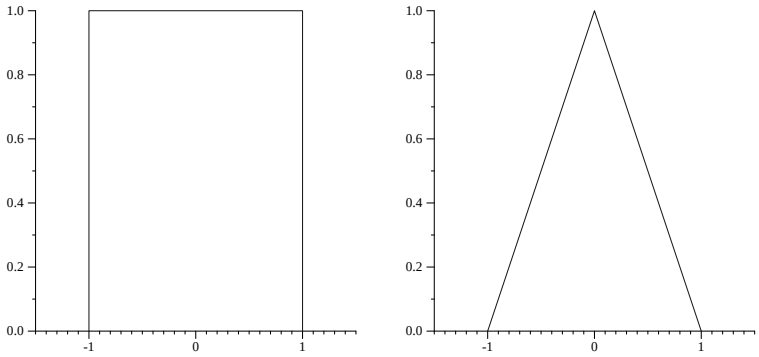


Figure 7.13: Graphs of the box filter (left) and triangle filter (right). Though neither of these is a particularly good filter, they are both computationally efficient and easy to implement.

Triangle Filter

⟨*triangle.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "sampling.h"
#include "paramset.h"
⟨Triangle Filter Declarations⟩
⟨Triangle Filter Definitions⟩
```

Filter 229
halfXWidth 229
halfYWidth 229
max 513

The triangle filter gives slightly better results than the box: samples at the output pixel have a weight of one, and the weight linearly falls off to the square extent of the filter. See the right side of Figure 7.13 for a graph of the triangle filter.

⟨*Triangle Filter Declarations*⟩≡
```
class TriangleFilter: public Filter {
public:
    TriangleFilter(Float xw, Float yw): Filter(xw, yw) { }
    Float Evaluate(Float x, Float y) const;
};
```

⟨*Triangle Filter Definitions*⟩≡
```
Float TriangleFilter::Evaluate(Float x, Float y) const {
    return max(0.f, halfXWidth - fabsf(x)) *
        max(0.f, halfYWidth - fabsf(y));
}
```

Gaussian Filter

⟨*gaussian.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "sampling.h"
#include "paramset.h"
⟨Gaussian Filter Declarations⟩
⟨Gaussian Filter Definitions⟩
```

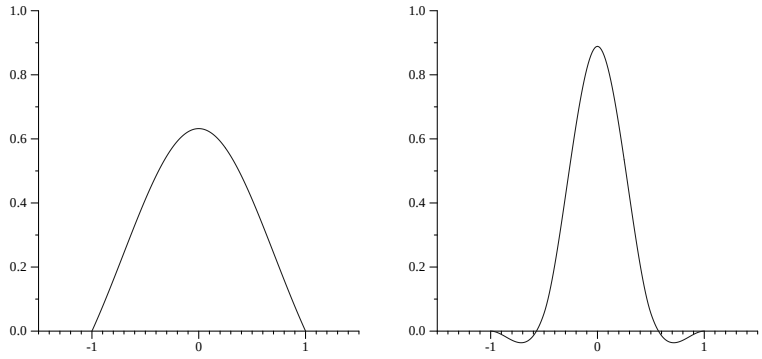


Figure 7.14: gaussian and mitchell filter graphs

The Gaussian is the first filter in `lrt` that gives good performance in practice. It applies a Gaussian shaped bump, centered at the output pixel and radially symmetric around it. We subtract the Gaussian's value at the end of its extent from the filter value; this makes the filter go to zero at its limit–see the left side of Figure 7.14. The Gaussian does tend to give blurrier images than the next two filters, however.

⟨*Gaussian Filter Declarations*⟩≡
```
class GaussianFilter : public Filter {
public:
    ⟨GaussianFilter Interface⟩
    Float Evaluate(Float x, Float y) const;
private:
    ⟨GaussianFilter Private Data⟩
    ⟨GaussianFilter Utility Functions⟩
};
```

229 Filter
229 halfXWidth
229 halfYWidth
513 max

In the constructor, we precompute a few terms that will be the same every time we evaluate the filter.

⟨*GaussianFilter Interface*⟩≡
```
GaussianFilter(Float xw, Float yw)
    : Filter(xw, yw) {
    expHalfX = expf(-halfXWidth);
    expHalfY = expf(-halfYWidth);
}
```

⟨*GaussianFilter Private Data*⟩≡
```
Float expHalfX, expHalfY;
```

⟨*Gaussian Filter Definitions*⟩≡
```
Float GaussianFilter::Evaluate(Float x, Float y) const {
    return Gaussian(x, expHalfX) * Gaussian(y, expHalfY);
}
```

⟨*GaussianFilter Utility Functions*⟩≡
```
static Float Gaussian(Float t, Float expHalf) {
    return max(0.f, expf(-t) - expHalf);
}
```

Mitchell Filter

⟨*mitchell.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "sampling.h"
#include "paramset.h"
⟨Mitchell Filter Declarations⟩
⟨Mitchell Filter Definitions⟩
```

Filter design is a difficult craft, mixing mathematical analysis and perceptual experiments: Mitchell and Netravali have developed a family of parameterized filter functions in order to be able to explore this space well. After analyzing test subjects' subjective responses to a variety of parameters, they developed a filter that tends to do a good job of trading off between *ringing*–phantom edges next to actual edges in the image–and *blurring*–overly blurred results–two common artifacts from poor reconstruction filters.

Note in the graph of this filter on the right side of Figure 7.14 that this filter function takes on negative values out by its edges; it has *negative lobes*. In practice these negative regions improve the sharpness of edges, giving crisper images (reduced blurring). If they become too large, however, ringing tends to start to enter the image.

Filter 229
halfXWidth 229
halfYWidth 229

⟨*Mitchell Filter Declarations*⟩≡
```
class MitchellFilter : public Filter {
public:
    MitchellFilter(Float xw, Float yw) : Filter(xw, yw) { }
    Float Evaluate(Float x, Float y) const;
    Float Mitchell1D(Float d) const;
};
```

Like many 2D image filtering functions, the Mitchell-Netravali filter is the product of two one-dimensional filter functions in the x and y directions. Such filters are called *separable*. (In fact, all of the filters in `lrt` are separable, though this wasn't made explicit in the previous ones.)

⟨*Mitchell Filter Definitions*⟩≡
```
Float MitchellFilter::Evaluate(Float x, Float y) const {
    return Mitchell1D(x/halfXWidth) * Mitchell1D(y/halfYWidth);
}
```

⟨*Mitchell Filter Definitions*⟩+≡
```
Float MitchellFilter::Mitchell1D(Float r) const {
    const Float B = .33333333333333333f;
    const Float C = .33333333333333333f;
    const Float ONE_SIXTH = .166666666666666666f;

    r = fabsf(r);
    if (r > 1)
        return ((-B - 6*C) * r*r*r + (6*B + 30*C) * r*r +
            (-12*B - 48*C) * r + (8*B + 24*C)) * ONE_SIXTH;
    else
        return ((12 - 9*B - 6*C) * r*r*r +
            (-18 + 12*B + 6*C) * r*r +
            (6 - 2*B)) * ONE_SIXTH;
}
```

Sinc Filter

⟨*sinc.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "sampling.h"
#include "paramset.h"
⟨Sinc Filter Declarations⟩
⟨Sinc Filter Definitions⟩
```

229 Filter
229 halfXWidth
229 halfYWidth
232 MitchellFilter

Finally, we provide the `SincFilter` class, which implements a filter based on the sinc function. In practice, the sinc filter is often multiplied by another function that goes to zero after some distance; this gives a filter function with finite extent, which is much more tractable for implementation. The function that scales the sinc down is called a windowing function; here we will use one due to Blackman. The shape of the windowed sinc is quite similar to the Mitchell-Netravali filter, so we won't graph it here.

⟨*Sinc Filter Declarations*⟩≡
```
class SincFilter : public Filter {
public:
    SincFilter(Float xw, Float yw) : Filter(xw, yw) { }
    Float Evaluate(Float x, Float y) const;
    Float Sinc1D(Float x) const;
};
```

Like the Mitchell-Netravali filter, the sinc filter is also separable.

⟨*Sinc Filter Definitions*⟩≡
```
Float SincFilter::Evaluate(Float x, Float y) const{
    return Sinc1D(x / halfXWidth) * Sinc1D(y / halfYWidth);
}
```

The implementation straightforward; we compute the value of the sinc function and then multiply it by the value of the Blackman windowing function.

⟨*Sinc Filter Definitions*⟩+≡
```
Float SincFilter::Sinc1D(Float x) const {
    if (x < 1e-5) return 1;
    if (x > 1)    return 0;
    x *= M_PI;
    Float s = sinf(x) / x;
    Float blackman = .42f + .5f * cosf(x) + .08f * cosf(2*x);
    return s * blackman;
}
```

Further Reading

One of the best books on signal processing, sampling, reconstruction, and the Fourier transform is Bracewell(Bra68). Glassner's *Principles of Digital Image Synthesis* (Gla95) has a series of chapters on the theory and application uniform and non-uniform sampling and reconstruction to computer graphics. For an extensive survey of the history of and techniques for interpolation of sampled data, including the sampling theorem, see Meijering's survey article (Mei02).

Crow first identified aliasing as a major source of artifacts in computer generated images (Cro77). Using non-uniform sampling to turn aliasing into noise was introduced by Cook et al(Coo86) and Dippé and Wold(DW85); this work was based on experiments by Yellot, who investigated the distribution of photoreceptors in the eyes of monkeys (Yel83). Dippé and Wold also first introduced the pixel filtering equation to graphics and developed a Poisson sample pattern with a minimum distance between samples. Lee et al developed a technique for adaptive sampling based on statistical tests to compute images to a given error tolerance (LRU85).

SincFilter 233

Mitchell has extensively investigated sampling patterns for ray-tracing; his 1987 and 1991 SIGGRAPH papers have many key insights, and the best candidate approach described in this chapter is based on the latter paper (Mit87; Mit91). Another efficient technique to generate Poisson disk patterns was also developed by McCool and Fiume (MF92). Hiller et al applied a technique based on relaxation that takes a random point set and improves its distribution (HDK01).

Shirley's used a concept called *discrepancy* to evaluate the quality of sample patterns (Shi91). Discrepancy gives a numeric measure of how well-distributed a set of sample points is; the better distributed it is, the lower its discrepancy. This work was built upon by Mitchell (Mit92) and Dobkin and Mitchell (DM93), Dobkin et al (DEM96).

Mitchell's first paper on discrepancy introduced the idea of using deterministic low-discrepancy sequences for sampler, removing all randomness in the interest of lower-discrepancy (Mit92). Such *quasi-random* sequences are the basis of Quasi Monte Carlo methods, which will be described in Chapter 14. More recently, Keller and collaborators have investigated quasi-random sampling patterns for a variety of applications in graphics (Kel96; Kel97). Wong et al compared numeric error with various low-discrepancy sampling schemes (WLH97), though one of Mitchell's interesting findings was that low-discrepancy sampling sequences sometimes lead to visually-objectionable artifacts in images that aren't present with other sampling patterns.

Kollig and Keller have investigated (t,m,s)-net approaches for generating sampling patterns and have paid particular attention to finding well-distributed light

source samples for a collection of pixel samples (KK02). Some of their techniques are based on algorithms developed by Friedel and Keller (FK00).

More recently, Mitchell has investigated how much better stratified sampling patterns are than random patterns in practice (Mit96); in general, the smoother the function being sampled is, the more effective they are. For very quickly-changing functions (e.g. pixels with complex geometry overlapping them), more sophisticated stratified patterns perform no better than unstratified random patterns.

Mitchell and Netravali investigated a family of filters by doing experiments with human observers to find the most effective ones; the Mitchell filter in this chapter is the one they chose as best (MN88).

Exercises

7.1 The current implementation of the `StratifiedSampler` suffers from only stratifying image samples; samples in the rest of the dimensions are just chosen randomly. Improve the stratified sampler by generating a set of samples

7.2 (t,m,s) nets, etc...

8. Image Pipeline

173 film
21 Point
198 Sampler
155 Spectrum

All the way from processing samples as they come in to make spectral image pixels, to processing the pixels for storage or display...

8.1 Processing Camera Samples

We can now put the image sampling and reconstruction theory together and write the `Sampler` function that takes image samples and filters them to compute pixel values, updating the `Film`'s pixels. Because all of the `Filter`s defined above have finite extent, we start by computing which pixels will be affected by the current sample. We then turn the pixel filtering Equation, 7.6.2, inside out, and for each pixel (x,y) that is affected by the sample, we update two running sums: one for the numerator of the sample interpolation equation and one for the denomenator. When all of the samples have been processed, final pixel values can be computed by performing the division.

⟨*Sampler Method Definitions*⟩+≡
```
void Sampler::AddSample(Film *film, const Point &Praster,
        const Spectrum &radiance, Float alpha) {
    ⟨Compute sample's raster extent⟩
    ⟨Loop over filter support and add sample to pixel arrays⟩
}
```

The first thing that we do is compute the bounds in raster-space of the pixels that will be affected by the sample. This is just half of the overall filter width in each direction from the sample locations, rounded up on the low end and rounded down on the high end so that we don't process any pixels outside of the extent where the filter is certain to be zero anyway.

⟨*Compute sample's raster extent*⟩≡
```
int x0 = Ceil2Int (Praster.x - filter->halfXWidth);
int x1 = Floor2Int(Praster.x + filter->halfXWidth);
int y0 = Ceil2Int (Praster.y - filter->halfYWidth);
int y1 = Floor2Int(Praster.y + filter->halfYWidth);
```

Now, given the extent of pixels that are affected by this sample (`x0,y0` to `x1,y1`, inclusive), we loop over all of those pixels and then filter the sample value appropriately.

⟨*Loop over filter support and add sample to pixel arrays*⟩≡
```
for (int y = y0; y <= y1; ++y)
    for (int x = x0; x <= x1; ++x) {
        ⟨Evaluate filter value⟩
        ⟨Update pixel values with filtered sample⟩
    }
```

Each integer pixel (x,y) has an instance of the filter function centered around it. To compute the filter weight for a particular sample, we compute the offset from the pixel to the sample position and evaluate the filter function at this position.

⟨*Evaluate filter value*⟩≡
```
Float fx = x - Praster.x;
Float fy = y - Praster.y;
Float filterWt = filter->Evaluate(fx, fy);
```

Now we go ahead and report the weighted sample value to the `Film`.

⟨*Update pixel values with filtered sample*⟩≡
```
if (filterWt > 0.)
    film->UpdatePixel(x, y, radiance, alpha, Praster.z, filterWt);
```

Ceil2Int 514
film 173
Floor2Int 514
halfXWidth 229
halfYWidth 229
pixels 189
UpdatePixel 189
xPixelWidth 189
yPixelWidth 189

Computing normalized pixel values

We can now define the fragment ⟨*Apply filter weights*⟩ for the `Film`'s imaging pipeline. We divide each pixel sample value by the value of `weightSum` for that pixel; this basically computes an average of all of the radiance values from all of the rays that contributed to this pixel. For efficiency, we compute one over the weight value once and then multiply by that instead of dividing by the weight value each time.

⟨*Apply filter weights*⟩≡
```
for (int o = 0; o < xPixelWidth * yPixelWidth; ++o) {
    if (pixels[o].weightSum == 0.)
        continue;
    Float invWt = 1.f / pixels[o].weightSum;
    Lout[o] *= invWt;
    AlphaOut[o] *= invWt;
}
```

Sample Crop Extents

We can now also define the fragment that etermines the range of integer pixels that must have samples generated for them in order to compute the desired image.

Samplers use these values to guide their sample generation. In particular, because the pixel interpolation filter generally extends over a number of pixels, we need to compute samples a bit outside of the range of pixels that will be output.

⟨*Initialize pixel extents from crop window*⟩≡
```
int RasterCropLeft   = Ceil2Int(xResolution * Crop.x0);
int RasterCropRight  = Ceil2Int(xResolution * Crop.x1);
int RasterCropTop    = Ceil2Int(yResolution * Crop.y0);
int RasterCropBottom = Ceil2Int(yResolution * Crop.y1);
xPixelStart = Floor2Int(RasterCropLeft   - filter->halfXWidth);
xPixelEnd   = Ceil2Int (RasterCropRight  + filter->halfXWidth);
yPixelStart = Floor2Int(RasterCropTop    - filter->halfYWidth);
yPixelEnd   = Ceil2Int (RasterCropBottom + filter->halfYWidth);
```

8.2 Spectral Image Storage

Once the camera has computed values for all of the image samples and the same values have been used to set the pixel values, we need to do something with the results. The easiest thing to do is to write out the image of floating-point SPD coefficients to disk for later processing or display by programs with knowledge of the basis functions used. More commonly, we will send the pixels through an imaging pipeline that uses information about the particular basis functions and display device being used to compute a new image suitable for display. A number of tricky issues, ranging from limitations of display devices to the behavior of the human visual system, need to be carefully addressed to do this well.

514 Ceil2Int
514 Floor2Int
229 halfXWidth
229 halfYWidth
538 TIFFWriteFloat
198 xPixelEnd
189 xPixelWidth
198 yPixelEnd
189 yPixelWidth

Saving SPD coefficients

We'll provide a Film method that takes a filename and saves out a floating-point TIFF format image that stores the coefficients of the SPDs at each pixel. In conjunction with the basis functions used for spectral representation, other programs can use this image to reconstruct the SPDs computed by the renderer.

⟨*Film Method Definitions*⟩+≡
```
void Film::WriteCoefficients(const string &filename) const {
    ⟨Compute floating-point pixel SPD coefficients⟩
    TIFFWriteFloat(filename, (Float *)Lout, AlphaOut,
        xPixelWidth, yPixelWidth, COLOR_SAMPLES,
        xResolution, yResolution);
    ⟨Release temporary image memory⟩
}
```

The first stage of this process is divided into three parts. We make a copy of the pixel values stored by the film, so that changes to them before saving them don't change the film's pixel values.

⟨*Compute floating-point pixel SPD coefficients*⟩≡
```
⟨Allocate working imaging memory and copy data⟩
⟨Apply filter weights⟩
⟨Compute premultiplied alpha color values⟩
```

⟨*Allocate working imaging memory and copy data*⟩≡
```
int nPix = xPixelWidth * yPixelWidth;
Spectrum *Lout = new Spectrum[nPix];
Float *AlphaOut = new Float[nPix];
for (int i = 0; i < nPix; ++i) {
    Lout[i] = pixels[i].L;
    AlphaOut[i] = pixels[i].alpha;
}
```

The next stage, ⟨*Apply filter weights*⟩, will be defined in the next chapter in Section 7.6 when we explain image filtering and reconstruction. Its function is to normalize the individual pixel values so that even though many samples may have contributed to each pixel, the pixel values are consistent.

Before passing the pixel values along, we multiply each by its alpha value; pixel colors scaled by alpha are known as having *premultiplied alpha* (also known as *associated alpha*). Consider a solid white object; in its center, where it has an alpha of one, its pixel color values remain white. Along the edges, its color is reduced toward black depending on how much of the pixel area the object covers–this gives softer edges against the background. This representation gives to a variety of advantages when performing compositing operations–combining multiple images together and using their alpha channels to blend them more accurately (see the further reading section for further pointers.)

DisplayInfo 241
pixels 189
Spectrum 155
xPixelWidth 189
yPixelWidth 189

⟨*Compute premultiplied alpha color values*⟩≡
```
for (int i = 0; i < xPixelWidth * yPixelWidth; ++i)
    Lout[i] *= AlphaOut[i];
```

⟨*Release temporary image memory*⟩≡
```
delete[] Lout;
delete[] AlphaOut;
```

8.3 Image Display Pipeline

To be able to convert the spectral image into a format suitable for display or printing, we'll now explain the pieces of `lrt`'s imaging pipeline in more detail. The film class has a `WriteDisplayImage()` method that applies each of a series of imaging operations in turn. A structure, `DisplayInfo`, holds parameters that describe the characteristics of a particular display device. These parameters guide the imaging process.

⟨*Film Method Definitions*⟩+≡
```
void Film::WriteDisplayImage(const DisplayInfo &dinfo) const {
    ⟨Compute floating-point pixel SPD coefficients⟩
    ⟨Apply display imaging pipeline⟩
    ⟨Release temporary image memory⟩
}
```

Figure 8.1: Display pipeline

⟨*Film Declarations*⟩+≡
```
struct DisplayInfo {
    DisplayInfo() {
        ⟨DisplayInfo Constructor Implementation⟩
    }
    ⟨DisplayInfo Data⟩
};
```

The basic display pipeline is shown in Figure 8.1. The fragment ⟨*Apply display imaging pipeline*⟩ applies each of the stages in turn; we will describe them in order here.

⟨*Apply display imaging pipeline*⟩≡
```
⟨Convert image to XYZ⟩
⟨Apply tone reproduction to image⟩
⟨Convert image to display RGB⟩
⟨Scale and handle out-of-gamut RGB values⟩
⟨Apply gamma correction⟩
⟨Map image to display range⟩
⟨Dither image⟩
⟨Save display image to disk⟩
```

Once we have properly normalized SPD coefficients at each pixel, we will take advantage of a remarkable propeorty of the human visual system that allows us to represent each pixel's color with just three floating-point numbers. The *tristimulus theory* of color perception says that all visible SPDs can be accurately represented for human observers with three values, x_λ, y_λ, and z_λ.

Given a SPD $S(\lambda)$, these values are computed by convolution with the *spectral matching curves*, $X(\lambda)$, $Y(\lambda)$ and $Z(\lambda)$ by

$$\begin{aligned} x_\lambda &= \int_\lambda S(\lambda) X(\lambda) \mathrm{d}\lambda \\ y_\lambda &= \int_\lambda S(\lambda) Y(\lambda) \mathrm{d}\lambda \\ z_\lambda &= \int_\lambda S(\lambda) Z(\lambda) \mathrm{d}\lambda. \end{aligned}$$

The three matching curves are graphed in Figure 8.2. These curves were determined by the Commission Internationale de l'Éclairge standards body after a series of experiments with human test subjects. It is believed that these matching curves are generally similar to the responses of the three types of color-sensitive cones in the human retina.

Remarkably, SPDs with substantially different distributions may have very similar x_λ, y_λ, and z_λ values. To the human observer, such SPDs actually appear the

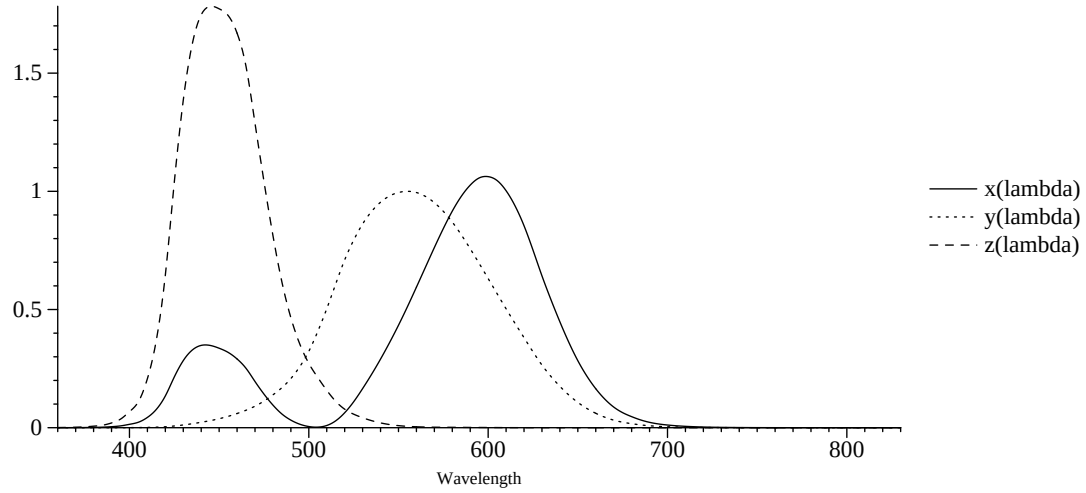


Figure 8.2: XYZ matching curves

same visually, so the XYZ representation is an accurate one. Pairs of such spectra are called *metamers*.

It is important to understand the subtlety that although XYZ works well to represent a given SPD to be displayed for a human observer, it is *not* a particularly good set of basis functions for spectral computation. For example, though XYZ values would work well to describe the perceived color of lemon-skin or a fluorescent light individually (recall Figure 5.1, which graphs these two SPDs), the product of their respective XYZ values is likely to give a noticeably different color than the XYZ value computed by multiplying a more accurate representation of their SPDs together and *then* computing XYZ values.

First, we need to add a method to the `Spectrum` class that returns the XYZ values for its SPD. It turns out that the new basis function coefficients after converting from one set of basis functions to another can be written a weighted sums of the old basis function coefficients. Here, we are converting from the original basis to the XYZ basis. For example, for x_λ,

$$\begin{aligned} x_\lambda &= \int S(\lambda)X(\lambda)\mathrm{d}\lambda \\ &\approx \int_\lambda \sum_i c_i B_i(\lambda)X(\lambda)\mathrm{d}\lambda \\ &= \sum_i c_i \left(\int_\lambda B_i(\lambda)X(\lambda)\mathrm{d}\lambda\right) \\ &= \sum_i c_i w_i^x. \end{aligned}$$

Thus, the weight values w_i^x, w_i^y and w_i^z can be precomputed and stored in an array for whatever particular basis functions are being used.

⟨*Spectrum Method Declarations*⟩+≡
```
void XYZ(Float xyz[3]) const {
    xyz[0] = xyz[1] = xyz[2] = 0.;
    for (int i = 0; i < COLOR_SAMPLES; ++i) {
        xyz[0] += XWeight[i] * c[i];
        xyz[1] += YWeight[i] * c[i];
        xyz[2] += ZWeight[i] * c[i];
    }
}
```

Also provide `Luminance()` in a separate utility function for convenience, avoid computing values for x and z when we don't need them...

⟨*Spectrum Method Declarations*⟩+≡
```
Float Luminance() const {
    Float y = 0.;
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        y += YWeight[i] * c[i];
    return y;
}
```

Therefore, we now finally need to settle on the default set of SPD basis functions for `lrt`. Though not sufficient for high-quality spectral computations, an expedient and convenient choice is to use the spectra of standard red, green, and blue phosphors for televisions and CRT display tubes. A standard set of these RGB spectra has been defined for high-definition television; the weights to convert from these RGBs to XYZ values are below:

155 Spectrum

⟨*Spectrum Method Definitions*⟩+≡
```
Float Spectrum::XWeight[COLOR_SAMPLES] = {
    0.412453f, 0.357580f, 0.180423f
};
Float Spectrum::YWeight[COLOR_SAMPLES] = {
    0.212671f, 0.715160f, 0.072169f
};
Float Spectrum::ZWeight[COLOR_SAMPLES] = {
    0.019334f, 0.119193f, 0.950227f
};
```

For convenience in computing values for `XWeight`, `YWeight` and `ZWeight` for other spectral basis functions, we will also provide the values of the standard $X(\lambda)$, $Y(\lambda)$, and $Z(\lambda)$ response curves sampled at 1nm increments from 360nm to 830nm.

⟨*Spectrum Public Data*⟩≡
```
static const int CIEstart = 360;
static const int CIEend = 830;
static const Float CIE_X[CIEend-CIEstart+1];
static const Float CIE_Y[CIEend-CIEstart+1];
static const Float CIE_Z[CIEend-CIEstart+1];
```

⟨*Spectrum Method Definitions*⟩+≡
```
const Float Spectrum::CIE_X[Spectrum::CIEend-Spectrum::CIEstart+1] = {
    ⟨CIE X function values⟩
};
const Float Spectrum::CIE_Y[Spectrum::CIEend-Spectrum::CIEstart+1] = {
    ⟨CIE Y function values⟩
};
const Float Spectrum::CIE_Z[Spectrum::CIEend-Spectrum::CIEstart+1] = {
    ⟨CIE Z function values⟩
};
```

Given the `XYZ()` method in `Spectrum`, it's easy for us to convert to an XYZ image.

⟨*Convert image to XYZ*⟩≡
```
Float *xyz = new Float[3*nPix];
for (int i = 0; i < nPix; ++i)
    Lout[i].XYZ(&xyz[3*i]);
```

We'll define some macros to clean up some of the code to come; `Y(i)` returns the y_λ value for the ith pixel, etc.

film 173
Spectrum 155
XYZ 243

⟨*ToneMap Declarations*⟩≡
```
#define X(i) (xyz[3*(i)])
#define Y(i) (xyz[3*(i)+1])
#define Z(i) (xyz[3*(i)+2])
```

8.4 Tone Mapping

⟨*tonemap.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef TONEMAP_H
#define TONEMAP_H 1
#include "lrt.h"
#include "film.h"
⟨ToneMap Declarations⟩
#endif // TONEMAP_H
```

In the early days of computer graphics, final pixel values typically had color values between zero and one, with no pretense of being associated with actual physical quantities. In the real-world, scenes often have as many as five orders of magnitude of variation from the brightest parts to the darkest parts, and the human visual system generally handles this variation well. Not only are computer displays unable to display very bright colors, they can generally display only about two orders of magnitude of brightness variation as well.

Because realisitic scenes rendered with physically-based rendering algorithms may exhibit this same mismatch between scene brightness and the display device's capabilities, it's important to address the issue of displaying the image such that it visually has as close an appearence to the actual scene as possible. It has recently been an active area of research to find good methods to compress down those extra

orders of magnitude for image display. This work has fallen under the rubric of *tone mapping* (or *tone reproduction*; it draws on research into the human visual system (*HVS*) to guide the development of techniques for image display. By exploiting properties of the HVS, tone mapping algorithms have been developed that do remarkably well at compensating for display device limitations. In this section, we will describe a few such algorithms and the principles behind them. Our coverage of this area touches on representative a subset of the possibilities, though the further reading section gives pointers to many recent papers in this field.

Luminance and photometry

Because these algorithms are generally based on human perception of brightness, tone mapping operators are usually based on the unit of *luminance*, which gives a sense of how bright a spectral power distribution appears to a human observer. For example, luminance accounts for the fact that a SPD with a particular amount of energy that is green will appear much brighter to a human than a SPD with the same amount of energy that is blue.

Luminance is closely related to radiance; given a spectral radiance value, a luminance value can be computed with a simple conversion formula. In fact, all of the radiometric quantities defined in Chapter 5 have analogs in the field of *photometry*, which is the study of visible electromagnetic radiation and its perception by the HVS. Each spectral radiometric quantity can be converted to its corresponding photometric quantity by integrating with the spectral response curve $V(\lambda)$, which describes the relative sensitivity of the human eye to various wavelengths. For example, luminance, which we will denote here by Y, is related to spectral radiance $L(\lambda)$ by

$$Y = \int_\lambda L(\lambda) V(\lambda) \mathrm{d}\lambda.$$

Fortunately, the CIE $Y(\lambda)$ tristimulus curve was chosen to be proportional to $V(\lambda)$ so that

$$Y = 683 \int_\lambda L(\lambda) Y(\lambda) \mathrm{d}\lambda.$$

Thus, we already have the luminance of each pixel in the image within a scale factor. We'll provide a macro that gives the luminance of the ith pixel:

⟨*ToneMap Declarations*⟩+≡
```
#define LUMINANCE(i) (683.f * Y(i))
```

The units of luminance are candelas per meter squared (cd/m^2), where the candela is the photometric equivalent of radiant intensity. The quantity cd/m^2 is often referred to in units of *nits*. Some representative luminance values are given in Figure 8.4.

The human eye has two types of photoreceptor responsible for detecting light: rods and cones. Rods help with perception in dark environments (*scoptic* light levels), ranging from approximately 10^{-6} to 10 cd/m^2. Rods give little information about color and are not very good at resolving fine details. Cones handle light ranging from approximately .01 to 10^8 cd/m^2 (*photopic* light levels.) There are three types of cones, with sensitivity to different wavelengths of light. (Computer displays generally display luminances from about 1 to 100 cd/m^2.)

Basic tone mapping approaches

Figure 8.3: St. Peter’s Basilica in Rome...

Luminance (cd/m^2, or nits)	
600,000	Sun at horizon
120,000	60 Watt light bulb
8,000	Clear sky
100–1000	Typical office
1–100	Typical computer display
1–10	Street lighting
0.25	Cloudy moonlight

Figure 8.4: Representative luminance values for a number of lighting settings.

The most common approach to tone reproduction is to compute a scale factor for each pixel that maps its value to the display's dynamic range. For simple tone reproduction operators, a single scale factor is often used for all pixels in the image. Such operators are called *spatially uniform* operators. They give a monotonic mapping of image luminance to display luminance. More sophisticated approaches use a scale that varies based on each pixel's brightness and the brightness of nearby pixels; these are *spatially varying* operators and they do necessarily guarantee a monotonic mapping.

That it is possible (and effective) to have a varying operator is an interesting thing. The human eye is more sensitive to relative changes in luminance locally, such that if two separate parts of the image have the same luminance, we can often get away with assigning them utterly different pixel values without the human observer noticing that anything is amiss. It turns out that it's more important that the relative pixel values compared to a pixel's neighbors are set appropriately than its absolute value be set appropriately.

The HVS's sensitivity to luminance changes varies depending on the *adaptation luminance*, Y^a. The adaptation luminance may vary over different parts of the image. In the methods below, we will use both the *display adaptation luminance* Y_d^a, which is the adaptation luminance of the human observer looking at the computer display, and the *world adaptation luminance*, Y_w^a, the adaptation luminance that the human would have if viewing the actual scene. A number of *time-dependent* tone reproduction operators have been recently developed, where the human visual system's adaptation to light over time is modeled. (e.g. when the lights are turned off in a room, over it takes a few minutes for the HVS to adjust.) In the interests of simplicity, however, we won't include the implementations of any time-dependent operators here.

One of the goals of most tone reproduction algorithms is to preserve contrast in the displayed image. Because the human visual system is more sensitive to relative contrast than it is to absolute brightness, it's more important to make sure that enough distinct colors are used in all regions of the image–bright and dim–so that different colors are seen, rather than mapping a wide range of image intensities to the same pixel values. Thus, an object that is twice as bright as another one in the scene doesn't necessarily need to be twice as bright on the display. It's the local changes in contrast that seem to be the most important thing for the human visual system.

less color perception at scoptic levels

acuity: at low luminance levels, eye isn't as good at resolving high-frequency details. 1000 nits, can resolve about 50 cycles per degree, at .001 nits, only about 2.2 cycles per degree.

Tone mapping interface

We will now define a handful of tone mapping operators. All of them inherit from the `ToneMap` base class, which specifies the interface method, `Map()`.

⟨*ToneMap Declarations*⟩+≡
```
class ToneMap {
public:
    ⟨ToneMap Interface⟩
};
```

The `Map()` function takes a pointer to the XYZ pixel values and the resolution of the image. It can also access the `DisplayInfo` structure in order to find information about the particular display device.

⟨*ToneMap Interface*⟩+≡
```
virtual void Map(Float *xyz, int xRes, int yRes,
    const DisplayInfo &di) const = 0;
```

DisplayInfo 241
xPixelWidth 189
yPixelWidth 189

The `DisplayInfo` structure holds a `ToneMap` pointer, initialized to `NULL` by default. For the tone reproduction operators that make use of information, it also holds fields that record the maximum luminance that the device is capable of displaying, `maxDisplayY`, and the adaptation luminance of the viewer, `displayAdaptationY`. These are set to common default values.

⟨*DisplayInfo Data*⟩≡
```
ToneMap *toneMap;
Float maxDisplayY, displayAdaptationY;
```

⟨*DisplayInfo Constructor Implementation*⟩≡
```
toneMap = NULL;
maxDisplayY = 100.f;
displayAdaptationY = 50.f;
```

If the tone operator is non-`NULL`, we apply it to the XYZ pixels:

⟨*Apply tone reproduction to image*⟩≡
```
if (dinfo.toneMap)
    dinfo.toneMap->Map(xyz, xPixelWidth, yPixelWidth, dinfo);
```

Maximum to white

⟨*maxwhite.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "tonemap.h"
⟨MaxWhiteOp Declarations⟩
⟨MaxWhiteOp Method Definitions⟩
```

The easiest tone reproduction operator to apply (besides hoping that the image's pixel values are already in a suitable range for the display) is the *maximum to white* operator. It loops over all of the pixels to find the one with the greatest luminance. It then scales all of the pixels so that the brightest one maps to a value of one.

There are two main disadvantages to this operator in practice. First, it doesn't account for the human visual system at all: if the lights in the scene are turned up to be 100 times brighter and the scene is re-rendered, the maximum to white operator will give the same displayed image as before. Second, a small number of very bright pixels can cause the rest of the image to be too dark to be visible. Nonetheless, it can work well for scenes without too much dynamic range in the image and serves as a baseline that can show off the improvement that smarter operators offer.

⟨*MaxWhiteOp Declarations*⟩≡
```
class MaxWhiteOp : public ToneMap {
    void Map(Float *xyz, int xRes, int yRes, const DisplayInfo &di) const;
};
```

⟨*MaxWhiteOp Method Definitions*⟩≡
```
void MaxWhiteOp::Map(Float *xyz, int xRes, int yRes,
        const DisplayInfo &di) const {
    ⟨Compute maximum luminance of all pixels⟩
    Float scale = 683.f / maxLum;
    ⟨Apply scale to all image pixels⟩
}
```

241 DisplayInfo
513 max
248 ToneMap

⟨*Compute maximum luminance of all pixels*⟩≡
```
Float maxLum = 0.;
for (int i = 0; i < xRes * yRes; ++i)
    maxLum = max(maxLum, LUMINANCE(i));
```

⟨*Apply* scale *to all image pixels*⟩≡
```
for (int i = 0; i < xRes * yRes; ++i) {
    X(i) *= scale;
    Y(i) *= scale;
    Z(i) *= scale;
}
```

Contrast-based scale factor

⟨*contrast.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "tonemap.h"
⟨ContrastOp Declarations⟩
⟨ContrastOp Method Definitions⟩
```

An early tone reproduction operator that focused on preserving contrast in the displayed image was developed by Ward. Previous researchers had developed models that describe the smallest change in luminance that is noticeable to a human observer given a particular adaptation luminance (the *just noticeable difference*, otherwise known as *JND*). For larger adaptation luminances, it takes a larger change in luminance to be noticeable.

XXX make clear that this makes dark images dim, bright images bright, etc... XXX

Figure 8.5.

Figure 8.5: contrast

Ward applied this research to derive an algorithm that computes a single spatially-uniform scale factor that attempts to preserve *contrast visibility*–given a region of the image that is just noticeably different from its neighbor, pixel values should be chosen such that the person looking at the display perceives that those two pixel values are just noticeably different. XXX Don't want to scale to more JNDs, since that's a waste of the display's dynamic range, and don't want to scale to fewer, since then we will not perceive contrast when we should. XXX

DisplayInfo 241
ToneMap 248

⟨*ContrastOp Declarations*⟩≡
```
class ContrastOp : public ToneMap {
public:
    void Map(Float *xyz, int xRes, int yRes, const DisplayInfo &di) const;
};
```

⟨*ContrastOp Method Definitions*⟩≡
```
void ContrastOp::Map(Float *xyz, int xRes, int yRes,
        const DisplayInfo &di) const {
    ⟨Compute world adaptation luminance⟩
    ⟨Compute contrast-preserving scalefactor⟩
    ⟨Apply scale to all image pixels⟩
}
```

Blackwell found that given an adaptation luminance Y^a, a reasonable model of the minimum change in luminance necessary to be visible is given by:

$$\Delta Y(Y^a) = 0.0594 \cdot (1.219 + (Y^a)^{0.4})^{2.5}.$$

We would like to scale the image in a way such that the variation in display luminances is such that the minimum discernable luminance change for the display, given the display adaptation, maps to the minimum discernable luminance change for the image being displayed, given the image adaptation. In other words, we would like to determine a scale s such that

$$\Delta Y(Y_d^a) = s \cdot \Delta Y(Y_w^a).$$

We can substitude Blackwell's model and solve this for s, giving

$$s = \left(\frac{1.219 + (Y_d^a)^{0.4}}{1.219 + (Y_w^a)^{0.4}} \right)^{2.5}.$$

Applying this scale factor to each pixel in the image maps the world luminance to display luminance. We then need to divide the result by the maximum display luminance to get pixel values in the $[0,1]$ range.

To compute the world adaptation luminance Y_w^a, we comupte a log average of the luminances in the image. This helps bright regions from overwhelming dark regions. If we knew more about the actual adaptation level (e.g. based on what part of the image the viewer was looking at), a more precise adaptation luminance could possibly be computed.

⟨*Compute world adaptation luminance*⟩≡
```
Float Ywa = 0.;
for (int i = 0; i < xRes * yRes; ++i)
    if (LUMINANCE(i) > 0) Ywa += logf(LUMINANCE(i));
Ywa = expf(Ywa / (xRes * yRes));
```

And the display adaptation luminance, Y_d^a, is available in the `DisplayAdaptionY` field of the `DisplayInfo` structure. Because the scale-factor expects luminance values, we scale it by 683 before applying it to the pixels.

⟨*Compute contrast-preserving scalefactor*⟩≡
```
Float scale = powf((1.219f + powf(di.displayAdaptationY, 0.4f)) /
    (1.219f + powf(Ywa, 0.4f)), 2.5f);
scale *= 683.f / di.maxDisplayY;
```

Varying adaptation luminance

⟨*highcontrast.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "tonemap.h"
#include "mipmap.h"
⟨HighContrastOp Declarations⟩
⟨HighContrastOp Method Definitions⟩
```

As mentioned in the introduction to this section, we can often make better use of the display's dynamic range by using a scale factor that varies over the image. Here we will implement a tone reproduction operator tailored for high-contrast scenes that computes a local adaptation luminance that varies over the image. The local adaptation luminance is then used to compute a scale-factor using a contrast-preserving tone reproduction operator, in a similar manner to the `ContrastOp` operator defined above.

The main difficulty with methods that compute local adpatation luminance is that they are prone to artifacts at boundaries between very bright and very dim parts of the image. If the tone reproduction operator scales the dim pixels using an adaptation luminance that includes the effects of the bright pixels, the dim pixels will be mapped to black, causing a halo artifact. Instead, we would like to make sure that the dim pixels have an adaptation luminance based on just nearby dim pixels.

The `HighContrastOp` operator, to be defined shortly, uses locally-linear scale-factor, based on local adaptation luminance. Local adaptation luminance is computed in a novel way that avoids halo artifacts. This approach is based on a tone reproduction operator developed by Ashikhmin (Ash02). Reinhard et al simultaneously developed an operator that uses the same technique to comupte local adaptation (ERF02). Over local regions of the image where the adaptation luminance is slowly changing, this tone reproduction operator gives a local scale factor, which is tuned to preserves contrast. However, since adaptation is allowed to vary over the image, details are preserved–bright regions aren't blown out to be white, and dark regions aren't mapped down to black pixels.

Figure 8.6

⟨*HighContrastOp Declarations*⟩≡
```
class HighContrastOp : public ToneMap {
public:
    void Map(Float *xyz, int xRes, int yRes, const DisplayInfo &di) const;
private:
    ⟨HighContrastOp Utility Methods⟩
};
```

DisplayInfo 241
ToneMap 248

The tone mapping function that `HighContrastOp` uses is based on the *threshold versus intensity* (TVI) function, which gives the just noticable luminance difference for given adaptation level $TVI(Y^a)$. This is similar to the JND function used by Ward, but is based on a more complex model of the human visual system, including response to scoptic light levels.

First, we define the *perceptual capacity*, which tells us, given a particular adaptation level, how many just-noticeable-differences a given luminance range covers:

$$\frac{Y_a - Y_b}{TVI(Y^a)}$$

To be able to quickly compute the perceptual capacity of a given pair of luminance values, the auxilary capacity function $C(Y)$ is defined as the integral

$$C(Y) = \int_0^Y \frac{\mathrm{d}Y}{TVI(Y)},$$

where the adaptation level to compute the differential perceptual capacity at a given luminance is assumed to be equal to the luminance. Then $C(Y_a) - C(Y_b)$ is the perceptual capacity from Y_a to Y_b.

Ashikhmin then made some simplifications to a widely-used TVI function in order to be able to integrate it analytically, giving the function

$$C(Y) = \begin{cases} Y/0.0014 & Y < 0.0034 \\ 2.4483 + \log(L/0.0034)/0.4027 & 0.0034 \le Y < 1 \\ 16.563 + (Y-1)/0.4027 & 1 \le Y < 7.2444 \\ 32.0693 + \log(Y/7.2444)/0.0556 & \text{otherwise} \end{cases}$$

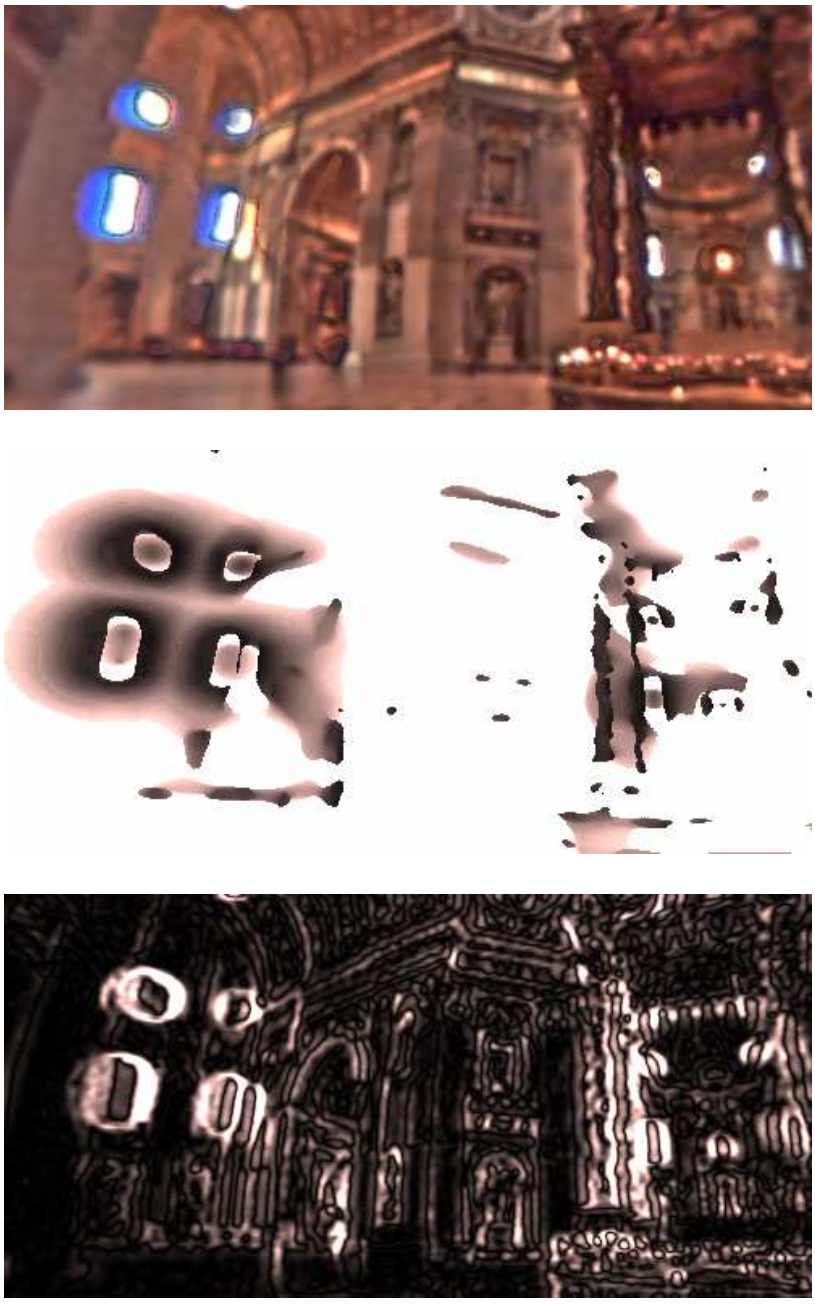


Figure 8.6: highcontrast, log widths, local contrast...

⟨*HighContrastOp Utility Methods*⟩≡
```
static Float C(Float y) {
    if (y < 0.0034f) return y / 0.0014f;
    else if (y < 1) return 2.4483f + log10f(y/0.0034f)/0.4027f;
    else if (y < 7.2444f) return 16.563f + (y - 1)/0.4027f;
    else return 32.0693f + log10f(y / 7.2444f)/0.0556f;
}
```

Given $C(Y)$, we can now take a given luminance value and determine how many JND steps it is from the minimum luminance in the image,

$$C(Y) - C(Y_{\min})$$

and we can also compute, of all of the JND steps, what fraction of the way through all of them it is:

$$\frac{C(Y) - C(Y_{\min})}{C(Y_{\max}) - C(Y_{\min})}$$

This gives us a sense of how far through the range of display luminances this world luminance should be mapped. Thus, the overall tone mapping operator, giving a result in terms of display luminance, is

$$T(Y) = Y_d^{\max} \frac{C(Y) - C(Y_{\min})}{C(Y_{\max}) - C(Y_{\min})}$$

Because we want to map final values to $[0,1]$, the display luminance value $Y_d^{\max}$ cancels out, saving us the trouble of determining it in the first place.

⟨*HighContrastOp Utility Methods*⟩+≡
```
static Float T(Float y, Float CYmin, Float CYmax) {
    return (C(y) - CYmin) / (CYmax - CYmin);
}
```

Given this tone mapping function $T(Y^a)$, the scale-factor at a given pixel (x,y) is defined by

$$s(x,y) = \frac{T(Y^a(x,y))}{Y^a(x,y))}.$$

As long as $Y^a(x,y)$ is slowly varying over the image, this is a locally-linear mapping (more or less).

We can now define the main tone reproduction function. It compues the minimum and maximum luminances of all pixels in the image so that $Y_{\max}$ and $Y_{\min}$ can be computed. In order to be able to quickly do the searches to comptue adaptation luminances, we then build an *image pyramid* data structure, where the original image is progressively filtered down into lower-resolution copies of itself. This is then used when we loop over all of the pixels and apply the tone reproduction operator.

⟨*HighContrastOp Method Definitions*⟩≡
```
void HighContrastOp::Map(Float *xyz, int xRes, int yRes,
        const DisplayInfo &di) const {
    ⟨Find minimum and maximum image luminances⟩
    ⟨Build luminance image pyramid⟩
    ⟨Apply high contrast tone mapping operator⟩
}
```

⟨*Find minimum and maximum image luminances*⟩≡
```
Float minLum = LUMINANCE(0), maxLum = LUMINANCE(0);
for (int i = 0; i < xRes * yRes; ++i) {
    minLum = min(minLum, LUMINANCE(i));
    maxLum = max(maxLum, LUMINANCE(i));
}
Float CYmin = C(minLum), CYmax = C(maxLum);
```

Most previous approaches to computing local adaptation luminance used a blurred version of the original image, though this led to the halo artifacted described previously. The insight that the developers of this approach had was that adaptation luminance shouldn't be based on a constant-sized average of luminances around the pixel (x,y), due to big changes in luminance in real-world images, but should be based on a varying average: as long as the luminance is locally roughly constant, the area can be expanded until a significant change in luminance is reached. This gives us the best of both worlds: when luminance is changing slowly, we compute adaptation luminance over a larger area, giving smooth variation of adpataion luminance when we are far from high contrast features. When contrast is quickly changing, however, we detect this and don't suffer artifacts.

A standard technique from image processing is to define the *local contrast* $lc(x,y)$ of a pixel as the magnitude of the difference between that pixel's value and its value in a blurred version of the image:

$$lc(s,x,y) = \frac{B_s(x,y) - B_{2s}(x,y)}{B_s(x,y)}.$$

241 DisplayInfo
252 HighContrastOp
513 max
513 min
336 MIPMap

Here s is the filter width used for blurring the image, expressed in pixels. We would like to find the smallest local extent around each pixel (x,y) of radius s such that $|lc(s,x,y)|$ is less than some constant value–when it becomes greater than that value, we have passed the amount of acceptable local contrast. Having found such an s, adaptation luminance is given by

$$Y^a(x,y) = B_s(x,y),$$

thus fulfilling the criteria above. The top image in Figure 8.6 shows this operator applied to the St. Peter's Basilica image, while the middle image shows the local contrast computed at each pixel for $s = 1.5$, and the bottom image shows the widths used for computing local adaptation luminance at each pixel, where the brighter the pixel, the wider a region was sampled. Notice how edges where there are large jumps in brightness in the original image are found by the local contrast function.

In order to be able to quickly find the value of the blurred image $B_s(x,y)$, we will create an image pyramid with the `MIPMap` class described in Section 11.6.

⟨*Build luminance image pyramid*⟩≡
```
Float *Yadapt = new Float[xRes * yRes];
for (int i = 0; i < xRes * yRes; ++i)
    Yadapt[i] = LUMINANCE(i);
MIPMap<Float> pyramid(xRes, yRes, Yadapt);
delete[] Yadapt;
```

Now we loop over all of the pixels in the image, computate the adaptation luminance, and can then directly apply the tone reproduction operator.

⟨*Apply high contrast tone mapping operator*⟩≡
```
ProgressReporter progress(xRes*yRes, "Tone Mapping");
for (int y = 0; y < yRes; ++y) {
    Float yc = Float(y) / Float(yRes-1);
    for (int x = 0; x < xRes; ++x) {
        Float xc = Float(x) / Float(xRes-1);
        ⟨Compute adaptation luminance⟩
        ⟨Apply tone mapping based on adaptation luminance⟩
        progress(stderr);
    }
}
fprintf(stderr, "\n");
```

To compute the adaptation luminance, we get look up the value of the pixel with a given blur amount and with four times that blur amount to compute the local contrast function. If it's above the value .25 (an arbitrary constant, chosen after some experimentation), we set the adaptation luminance by the average of a slightly region around the pixel. Otherwise, we increase the blur radius a bit and try again.

⟨*Compute adaptation luminance*⟩≡
```
Float dwidth = 1.f / Float(max(xRes, yRes));
Float maxWidth = 32.f / Float(max(xRes, yRes));
Float width = 2.f * dwidth, prevWidth = 0.f;
Float Yadapt;
Float prevlc = 0.f;
const Float maxLocalContrast = .5f;
while (1) {
    ⟨Compute local contrast at (x,y)⟩
    ⟨If maximum contrast is exceeded, compute adaptation luminance⟩
    ⟨Increase search region and prepare to compute contrast again⟩
}
```

Lerp 512
Lookup 335
max 513
ProgressReporter 498

⟨*Compute local contrast at* (x,y)⟩≡
```
Float b0 = pyramid.Lookup(xc, yc, width, 0.f, 0.f, width);
Float b1 = pyramid.Lookup(xc, yc, 2.f*width, 0.f, 0.f, 2.f*width);
Float lc = fabsf((b0 - b1) / b0);
```

⟨*If maximum contrast is exceeded, compute adaptation luminance*⟩≡
```
if (lc > maxLocalContrast) {
    Float t = (lc - prevlc) / maxLocalContrast;
    Float w = Lerp(t, prevWidth, width);
    Yadapt = pyramid.Lookup(xc, yc, w, 0.f, 0.f, w);
    break;
}
```

⟨*Increase search region and prepare to compute contrast again*⟩≡
```
prevlc = lc;
prevWidth = width;
width += dwidth;
if (width >= maxWidth) {
    Yadapt = pyramid.Lookup(xc, yc, maxWidth, 0.f, 0.f, maxWidth);
    break;
}
```

⟨*Apply tone mapping based on adaptation luminance*⟩≡
```
Float scale = 683.f * T(Yadapt, CYmin, CYmax) / Yadapt;
int off = x + y*xRes;
X(off) *= scale;
Y(off) *= scale;
Z(off) *= scale;
```

Spatially-varying non-linear scale

⟨*nonlinear.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "tonemap.h"
⟨NonLinearOp Declarations⟩
⟨NonLinearOp Method Definitions⟩
```

241 DisplayInfo
335 Lookup
248 ToneMap

One last approach is less well-grounded in the perception literature, though it works remarkably well in practice. As with the `MaxWhiteOp` operator, we start by computing the maximum luminance of all pixels in the image. We then scale the (x,y)th pixel by the factor

$$s(x,y) = \frac{\left(1+\frac{Y(x,y)}{Y_{\max}^2}\right)}{1+Y(x,y)}$$

This maps black pixels to zero and the brightest pixels to white. In between, darker pixels require relatively less change in brightness to cause a given change in output pixel value than bright pixels do. This matches the human visual system, which has a generally logarithmic response curve, rather than a linear one.

Figure 8.7.

⟨*NonLinearOp Declarations*⟩≡
```
class NonLinearOp : public ToneMap {
    void Map(Float *xyz, int xRes, int yRes, const DisplayInfo &di) const;
};
```

Figure 8.7: nonlinear

⟨*NonLinearOp Method Definitions*⟩≡
```
void NonLinearOp::Map(Float *xyz, int xRes, int yRes,
        const DisplayInfo &di) const {
    ⟨Compute world adaptation luminance⟩
    Float invLum2 = 1.f / (Ywa * Ywa);
    for (int i = 0; i < xRes * yRes; ++i) {
        Float scale = (1.f + Y(i) * invLum2) /
            (1.f + Y(i));
        X(i) *= scale;
        Y(i) *= scale;
        Z(i) *= scale;
    }
}
```

DisplayInfo 241
NonLinearOp 257

8.5 Device RGB Conversion and Output

After the tone reproduction step, we should have pixel XYZ values with brightness between zero and one. (Some tone reproduction operators don't guarantee this, so we'll clamp the values to this range later in the pipeline just to be sure.) We will now use information about the particular display device being used to convert the device-independent XYZ pixel values to device-dependent RGB values. This is another change of spectral basis, where the new basis is determined by the spectral response curves of the red, green, and blue elements of the display device. As before, weights to convert from XYZ to the device RGB can be precomputed. The `DisplayInfo` structure holds the weights for the particular display being used.

⟨*DisplayInfo Data*⟩+≡
```
Float rWeight[3], gWeight[3], bWeight[3];
```

By default, these are initialized to the appropriate weights for the RGB primaries as specified by the HDTV standard.

⟨*DisplayInfo Constructor Implementation*⟩+≡
```
rWeight[0] =  3.240479f; rWeight[1] = -1.537150f; rWeight[2] = -0.498535f;
gWeight[0] = -0.969256f; gWeight[1] =  1.875991f; gWeight[2] = 0.041556f;
bWeight[0] =  0.055648f; bWeight[1] = -0.204043f; bWeight[2] = 1.057311f;
```

⟨*Convert image to display RGB*⟩≡
```
Float *rgb = new Float[3*nPix];
⟨Define RGB access macros⟩
for (int i = 0; i < nPix; ++i) {
    R(i) = dinfo.rWeight[0]*X(i) + dinfo.rWeight[1]*Y(i) +
        dinfo.rWeight[2]*Z(i);
    G(i) = dinfo.gWeight[0]*X(i) + dinfo.gWeight[1]*Y(i) +
        dinfo.gWeight[2]*Z(i);
    B(i) = dinfo.bWeight[0]*X(i) + dinfo.bWeight[1]*Y(i) +
        dinfo.bWeight[2]*Z(i);
}
delete[] xyz;
xyz = NULL;
```

⟨*Define RGB access macros*⟩≡
```
#define R(i) (rgb[3*(i)])
#define G(i) (rgb[3*(i)+1])
#define B(i) (rgb[3*(i)+2])
```

Unfortunately, there are many colors that modern displays cannot reproduce; such colors are called *out of gammut*. (For example, XXX oranges and purples XXX.) Such colors will have RGB values outside the range $[0,1]$. There aren't any completely satisfactory solutions to this problem; it's all a matter of trading off different kinds of error. We will just clamp out of gammut colors to the range $[0,1]$. This works well for colors that aren't too far out of that range, though it does break down in cases like a color with RGB values $(2,1,1)$. This method clamps it to $(1,1,1)$, turning what was a reddish color into white.

513 Clamp

⟨*Scale and handle out-of-gamut RGB values*⟩≡
```
for (int i = 0; i < nPix; ++i) {
    R(i) = Clamp(dinfo.gain * R(i), 0., 1.);
    G(i) = Clamp(dinfo.gain * G(i), 0., 1.);
    B(i) = Clamp(dinfo.gain * B(i), 0., 1.);
}
```

Now we need to adjust the color values for the non-linear change in displayed brightness that displays based on cathode ray tubes (CRTs) exhibit. With these kinds of displays, the displayed brightness doesn't vary linearly with the pixel values: a pixel with value 100 isn't usually twice as bright as a pixel with value 50. (Although newer display technologies, such as LCD screens don't naturally have non-linear response like this, they are generally built with logic that mimics this characteristic of CRTs. CHECK THIS.)

This non-linear response is generally modeled with a power function

$$d = v^{1/\gamma},$$

where d is the display brightness, v is the voltage applied to the display's electron gun, and the *gamma value* γ is generally 2.2.

⟨*DisplayInfo Data*⟩+≡
```
Float gain, invGamma;
```

⟨*DisplayInfo Constructor Implementation*⟩+≡
```
invGamma = 1.f / 2.2f;
```

⟨*Apply gamma correction*⟩≡
```
for (int i = 0; i < nPix; ++i) {
    R(i) = powf(R(i), dinfo.invGamma);
    G(i) = powf(G(i), dinfo.invGamma);
    B(i) = powf(B(i), dinfo.invGamma);
}
```

Once we have gamma corrected pixel values between 0 and 1, we may need to map them to the range of values that the display expects (e.g. 0 to 255.) Some image file formats can store floating-point pixel values, so we don't always need to perform this step.

⟨*DisplayInfo Data*⟩+≡
```
bool integerFormat;
int maxDisplayValue;
```

⟨*DisplayInfo Constructor Implementation*⟩+≡
```
integerFormat = true;
maxDisplayValue = 255;
```

⟨*Map image to display range*⟩≡
```
if (dinfo.integerFormat) {
    for (int i = 0; i < nPix; ++i) {
        R(i) *= dinfo.maxDisplayValue;
        G(i) *= dinfo.maxDisplayValue;
        B(i) *= dinfo.maxDisplayValue;
        AlphaOut[i] *= dinfo.maxDisplayValue;
    }
}
```

Finally, we may *dither* the pixel values before converting them to integer values for display. Dithering involves adding a small random noise value to each pixel's color component. It improves the visual quality of displayed images by making the transition between areas with one pixel to another less well-delineated.

⟨*DisplayInfo Data*⟩+≡
```
Float ditherAmount;
```

⟨*DisplayInfo Constructor Implementation*⟩+≡
```
ditherAmount = 0.5f;
```

⟨*Dither image*⟩≡
```
if (dinfo.ditherAmount > 0) {
    for (int i = 0; i < nPix; ++i) {
        R(i) += RandomFloat(-dinfo.ditherAmount,
            dinfo.ditherAmount);
        G(i) += RandomFloat(-dinfo.ditherAmount,
            dinfo.ditherAmount);
        B(i) += RandomFloat(-dinfo.ditherAmount,
            dinfo.ditherAmount);
    }
}
```

Finally now, we can save the image out to disk.

⟨*DisplayInfo Data*⟩+≡
```
string filename;
```

⟨*DisplayInfo Constructor Implementation*⟩+≡
```
filename = "lrt.tiff";
```

⟨*Save display image to disk*⟩≡
```
if (dinfo.integerFormat)
    TIFFWrite8Bit(dinfo.filename, rgb, AlphaOut, xPixelWidth,
        yPixelWidth, 3, xResolution, yResolution);
else
    TIFFWriteFloat(dinfo.filename, rgb, AlphaOut, xPixelWidth,
        yPixelWidth, 3, xResolution, yResolution);
```

515 RandomFloat
536 TIFFWrite8Bit
538 TIFFWriteFloat
189 xPixelWidth
189 yPixelWidth

Further Reading

Tumblin and Rushmeier first introduced a the first tone mapping algorithms to computer graphics and sparked the recent focus on tone reproduction (TR93). Other early work included Chiu et al's spatially-varying scale (CHS+93), and Ward's contrast-preservation scale (War94a), which we have implemented in Section 8.4.

misc: (THG99), (LRP97)

boundary preservation (TT99)

Wandell's book on vision has excellent coverage of properties of the human visual system (Wan95).

Reinhard et al photographic (ERF02), followup (Rei02)

Ashikhmin contrast boundary stuff (Ash02)

Durand and Dorsey (DD02)

vision overview (Fer01)

Interactive Durand and Dorsey (DD00)

complex/sophisticated (PFFG98)

drive rendering (RPG99) (also cite the meyer paper)

adaptation and masking (FPSG96) extended Ward's contrast-based method to handle scoptic lighting levels, including reduced color sensitivity and spatial acuity. (FPSG97)

Time dependence (PTYG00)

Frankle and McCann 83 retinex paper

Spencer et al and Nishita glare papers!

Survey article (DCWP02).

Perceptually-driven rendering: Bolin and Meyer (BM98), Ramasubramanian et al (RPG99).

Exercises

8.1 exercise

9. Reflection Models

This chapter defines a set of classes that implement various models for describing light scattering at surfaces. Recall that the BRDF abstraction was introduced in Chapter 5 to describe light scattering at surfaces. We will define generic interface to these surface reflection and transmission functions, which are known as BRDFs (bidirectional reflectance distribution functions) and BTDFs (bidirectional transmittance distribution functions. Scattering from surfaces is often most easily described as a mixture of a set of BRDFs and BTDFs; in Chapter 10, we will introduce a BSDF object that holds multiple BRDFs and BTDFs to represent overall scattering from the surface.

Specific reflection models come from a number of sources:

1. Real world data: reflection distribution properties of a number of real-world surfaces have been measured. This data may then be tabularized or a set of basis functions may be fit to it.

2. Phenomenological: equations that attempt to describe the qualitative properties of real-world surfaces can be remarkably effective at mimicking them. These BSDFs can be particularly easy to use, since they tend to have intuitive parameters (e.g. "roughness") that modify their behavior.

3. Simulation: if low-level information is known about the composition of a surface (e.g. that a paint is comprised of colored particles of some average size suspended in a medium, or that a particular fabric is comprised from two types of thread, each with known reflectance properties), light scattering inside that surface may be simulated to generate data that can then be fit to basis functions.

4. Geometric optics: similar to simulation approaches, if the surface's lower-level scattering and geometric properties are known, then models can often be derived directly from these descriptions. This approach is much more tractable if geometric optics is used to model light's interaction with the surface–this is a much simpler model, not taking into account wave effects like polarization, etc.

5. Physical (wave) optics: some reflection models have been derived using a detailed model of light, treating it as a wave and computing the solution to Maxwell's equations to find how it scatters from a surface with known properties. These models tend to be computationally expensive however, and usually aren't substantially more accurate than models based on Geometric optics.

In this chapter, we will define implementations of reflection models based on measured data, phenomenological models, and geometric optics.

Before we define the reflection and transmission interfaces and classes, a brief overview of how they fit into the overall system and are used in the process of computing outgoing radiance at a point being shaded: the integrator classes, defined in Chapter 15, are responsible for determining which surface is first visible along a ray and computing the scattered radiance at that point. One the hit point is found, the integrator runs the surface shader that was bound to the surface. The surface shader is a short procedure that is responsible for deciding what the BSDF is at a particular point on the surface (see Chapter 10); it returns a BSDF object that holds BRDFs and BTDFs for that point. The integrators then use the BSDF to compute the scattered light at the point, based on the incoming illumination from the light sources in the scene.

Basic terminology

In order to be able to compare the visual appearence of different reflection models, we will introduce some basic terminology for describing reflection from surfaces. Reflection from real surfaces often doesn't cleanly fit into the categorization below, though it offers a general framework to start out with.

Reflection from surfaces can be split into three categories: *diffuse*, *glossy*, and *specular*. Diffuse surfaces scatter light equally in all directions–although a perfectly diffuse surface isn't physically reliazible, examples of near-diffuse surfaces include dull chalkboards and matte paint. Glossy surfaces (for example, gloss paint, or plastic) scatter light preferentially in a set of reflected directions–they show blurry reflections of other objects. Specular surfaces are in a sense a limiting case of glossy surfaces, reflecting incident light in a single direction. Mirrors and glass are examples of specular surfaces.

Isotropic vs anisotropic (3D vs 4D)...

XXX image showing these differences.

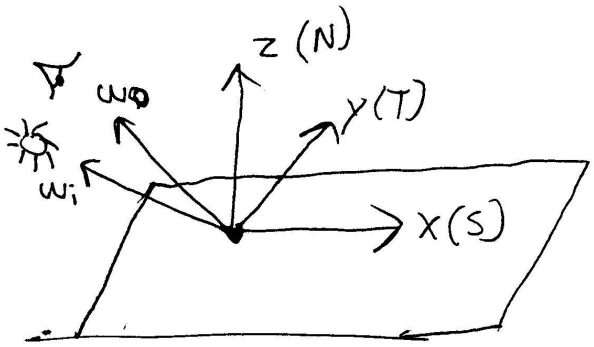


Figure 9.1: basic BSDF interface setting

9.1 Basic Interface

⟨*reflection.h**⟩≡
⟨*Source Code Copyright*⟩
```
#ifndef REFLECTION_H
#define REFLECTION_H
#include "lrt.h"
#include "geometry.h"
```
⟨*BSDF Declarations*⟩
⟨*BxDF Declarations*⟩
⟨*BSDF Inline Methods*⟩
```
#endif // REFLECTION_H
```

⟨*reflection.cc**⟩≡
⟨*Source Code Copyright*⟩
```
#include "lrt.h"
#include "color.h"
#include "reflection.h"
#include "shapes.h"
#include "mc.h"
#include <stdarg.h>
```
⟨*BxDF Utility Functions*⟩
⟨*BxDF Method Definitions*⟩
⟨*BSDF Method Definitions*⟩
⟨*BSDF MC Methods*⟩

We will first define the interface for the individual BRDF and BTDF functions. BRDFs and BTDFs share a common base-class, `BxDF`, which defines the basic interface that they adhere to. There are a few important conventions to keep in mind when reading and implementing them:

1. They are all defined with respect to a standard reflection coordinate system, aligned such that the surface normal lies along the $+z$ axis and the S and T directions lie along the $+x$ and $+y$ axes, respectively. (See Figure 9.1.) All direction vectors passed to and from these routines should be defined

with respect to this coordinate system. An important implication of this convention is that the dot product of a direction vector in this coordinate system with the surface normal is just the vector's z component. We will make use of this fact extensively throughout this chapter.

2. The incident light direction, $\vec{\omega}_i$, and the outgoing viewing direction, $\vec{\omega}_o$, will be normalized and outward facing after being transformed into the the local coordinate system at the surface. The surface normal should always point to the "outside" of the object, which helps us determine if light is entering or exiting transmissive objects. (Note that the local surface coordinate system may not be exactly the same as the coordinate system returned by the shapes' `Intersect()` routines presented in Chapter 3; They can be modified slightly to achieve effects like bump-mapping.)

3. The `BxDF`s should *not* concern themselves with whether $\vec{\omega}_i$ and $\vec{\omega}_o$ lie in the same hemisphere. For example, although a reflective `BxDF` should in principle return zero reflection if the incident direction is above the surface and the outgoing direction below it, here we will expect them to compute and return the amount of light reflected as if they were in the same hemisphere anyway. Higher-level code in the system will handle making sure that only reflective or transmissive scattering routines are evaluated as appropriate. (The need for this convention will be described in Section 10.1.)

4. We assume that light in different wavelengths is *decoupled*; energy at one wavelength will not be reflected at a different wavelength. Thus, flourescent materials are not supported.

Both BRDFs and BTDFs inherit from a base `BxDF` class that specifies their common interface. Because both have the exact same interface, this reduces repeated code and allows some parts of the system to work with `BxDF`s generially without distinguishing between BRDFs and BTDFs.

⟨*BxDF Declarations*⟩≡
```
class BxDF {
public:
    ⟨BxDF Interface Declarations⟩
};
```

By creating new and separate byes for BRDFs and BTDFs, we can improve type-safety and be able to distinguish between them in the parts of the system that need to keep them separate.

⟨*BxDF Declarations*⟩+≡
```
class BRDF : public BxDF {
};
```

⟨*BxDF Declarations*⟩+≡
```
class BTDF : public BxDF {
};
```

Both `BRDF`s and `BTDF`s will provide an `f` function that returns the value of the distribution function for the given pair of directions.

⟨*BxDF Interface Declarations*⟩+≡
```
virtual Spectrum f(const Vector &wo, const Vector &wi) const = 0;
```

In contrast to most BxDFs, which scattering incident light from a single direction in many outgoing directions, perfectly specular objects, like a mirror, glass, or water, only scatter light from a single incident direction in a single outgoing direction. Such BxDFs are best described with *delta distributions* that are zero except for the single direction where light is scattered.

These BxDFs need special handling in lrt, so we will also provide a method called f_delta() to handle scattering that involves the use of delta functions. BxDFs that implement it return true from IsSpecular(). The f_delta() method returns not only the amount of light scattered, but also along what direction; it is necessary for the BxDF to choose the direction in this case, since the caller has no chance of generating the appropriate $\vec{\omega}_i$ direction on their own.

Delta functions have some subtle implications for the light transport algorithms in Chapter 15; Section 15.1 describes the issues in detail.

⟨*BxDF Interface Declarations*⟩+≡
```
virtual bool IsSpecular() const { return false; }
```

⟨*BxDF Interface Declarations*⟩+≡
```
virtual Spectrum f_delta(const Vector &wi, Vector *wo) const {
    return 0.;
}
```
155 Spectrum
16 Vector

Reflectance

It can be useful to take the aggregate behavior of the 4D BxDF, defined as a function over pairs of directions, and reduce it to a 2D function over a single direction, or even to a constant value that describes its overall scattering behavior.

The *hemispherical-directional reflectance* is a 2D function that gives the total reflection in a given direction due to constant illumination over the hemisphere, or, equivalently, total reflection over the hemisphere due to light from a given direction. It is defined as:

$$\rho_{dh}(\omega) = \frac{1}{\pi}\int_{\Omega} f_r(\omega,\omega')|cos\theta|d\omega'.$$

We will add a method to the BxDF class to compute this value.

⟨*BxDF Interface Declarations*⟩+≡
```
virtual Spectrum rho(const Vector &w) const;
```

For some BxDFs, this integral can be computed analytically. For the rest, we will provide a method to estimate the value of ρ_{dh} in Section 14.3 the chapter on Monte Carlo integration.

The *hemispherical-hemispherical reflectance* of a surface is denoted by ρ_{hh} and is a constant spectral value that gives the fraction of incident light reflected by a surface when the incident light is the same from all directions. It is:

$$\rho_{hh} = \frac{1}{\pi}\int_{\Omega}\int_{\Omega} f_r(\omega_i,\omega_r)\,|\cos\theta_i\cos\theta_r|d\omega_i d\omega_r$$

⟨*BxDF Interface Declarations*⟩+≡
```
virtual Spectrum rho() const;
```

BRDF/BTDF Adapter

It's handy to be define an adapter class that lets us re-use an already-defined `BRDF` class as a `BTDF`, especially for phenomenological models that may be equally plausible models of transmission. The `BRDFToBTDF` class takes a `BRDF` pointer in the constructor and uses it to implement the `BTDF` interface. In particular, this means forwarding method calls on to the `BRDF`, possible switching the $\vec{\omega}_i$ direction to lie in the same hemisphere as $\vec{\omega}_o$, as the `BRDF` expects.

⟨*BxDF Declarations*⟩+≡
```
class BRDFToBTDF : public BTDF {
public:
    BRDFToBTDF(BRDF *b) { brdf = b; }
    ~BRDFToBTDF() { delete brdf; }
    ⟨BRDFToBTDF Method Declarations⟩
private:
    BRDF *brdf;
};
```

BRDF 266
BTDF 266
Spectrum 155
Vector 16

⟨*BRDFToBTDF Method Declarations*⟩≡
```
static Vector otherHemisphere(const Vector &w) {
    return Vector(w.x, w.y, -w.z);
}
```

⟨*BRDFToBTDF Method Declarations*⟩+≡
```
bool IsSpecular() { return brdf->IsSpecular(); }
Spectrum rho(const Vector &w) const {
    return brdf->rho(otherHemisphere(w));
}
Spectrum rho() const { return brdf->rho(); }
```

⟨*BxDF Method Definitions*⟩≡
```
Spectrum BRDFToBTDF::f(const Vector &wo, const Vector &wi) const {
    return brdf->f(wo, otherHemisphere(wi));
}
```

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum BRDFToBTDF::f_delta(const Vector &wo, Vector *wi) const {
    Spectrum f = brdf->f_delta(wo, wi);
    *wi = otherHemisphere(*wi);
    return f;
}
```

9.2 Specular Reflection and Transmission

The behavior of light at perfectly smooth surfaces is relatively easy to characterize analytically, in both the physical and geometric optics models. These surfaces exhibit *perfect specular reflection and transmission* of incident light: for a given $\vec{\omega}_i$ direction, all light is scattered in a single outgoing direction. For specular reflection, this direction is the outgoing direction that makes the same angle with the normal that the incoming direction does; see Figure 9.2.

For transmission, this direction is given by *Snell's law*, which relates the angle of the transmitted direction θ_t to the angle of the incident ray, θ_i. Snell's law depends on the *index of refraction* for the medium the incident ray is in and the index of refraction of the medium it is entering. The index of refraction describes how much more slowly light travels in a particular medium compared to the speed of light in a vacuum. We will use the "eta" symbol, η, to denote the index of refraction. Although its value is usually dependent on the wavelength of light, we will make the usual simplification in graphics by representing it by a single average floating-point value. Snell's law is:

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

XXX mention that wavelength-dependence is what gives *dispersion*: incident white light split into spectral components, e.g. by a prism.

In addition to knowing in which direction light is reflected and transmitted by a smooth surface, we also need to compute how much light is reflected and transmitted. The *Fresnel equations* tell us just this: they are the solution to Maxwell's equations at smooth surfaces. There are two sets of Fresnel equations; one for *dielectric media*–objects that don't conduct electricity, like glass–and one for *conductors*, like metals.

For each of these cases, the respective Fresnel equations have two forms, depending on the polarization of the incident light. If we assume that light is *circularly polarized*–that it is randomly oriented with respect to the light wave–then the Fresnel equations for light polarized parallel to the wave direction and light polarized perpendicular to the wave direction are squared and added together to give the Fresnel reflectance.

To compute the Fresnel reflectance of a dielectric, we need to know the indices of refraction for the two media; see Figure 9.2. Figure 9.3 has the indices of refraction for a number of dielectric materials.

The Fresnel formulae for dielectrics are:

$$\begin{aligned} r_{\parallel} &= \frac{\eta_t(N\cdot\vec{\omega}_i)+\eta_i(N\cdot\vec{\omega}_t)}{\eta_t(N\cdot\vec{\omega}_i)-\eta_i(N\cdot\vec{\omega}_t)} \\ r_{\perp} &= \frac{\eta_i(N\cdot\vec{\omega}_i)+\eta_t(N\cdot\vec{\omega}_t)}{\eta_i(N\cdot\vec{\omega}_i)-\eta_t(N\cdot\vec{\omega}_t)} \end{aligned}$$

where $r_{\parallel}$ is the Fresnel reflectance for parallel polarized light and $r_{\perp}$ is the reflectance for perpendicular polarized light. η_i and η_t are the indices of refraction for the incident and transmitted media, and $\vec{\omega}_i$ and $\vec{\omega}_t$ are the incident and transmitted directions, where $\vec{\omega}_t$ was computed with Snell's law. For light with random polarization (the usual assumption in graphics),

$$r = \frac{1}{2}(r_{\parallel}^2 + r_{\perp}^2).$$

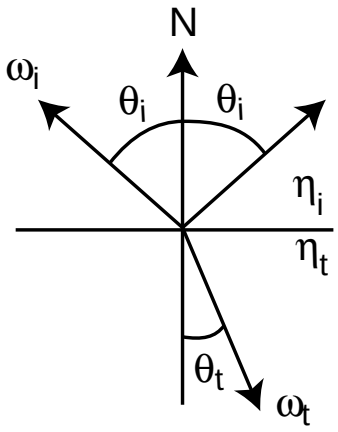


Figure 9.2: Basic setting for specular reflection and transmission. The reflected direction is the direction $\vec{\omega}_o$ opposite the incident direction $\vec{\omega}_i$ that makes the same angle θ_i with the surface normal as the incident ray. The transmitted direction makes an angle θ_t with the negated surface normal, where θ_t is given by Snell's law, which depends on the indices of refraction of the incident and transmitted media, η_i and η_t, respectively.

Medium	Index of refraction η
Vacuum	1.0
Air at sea level	1.00029
Ice	1.31
Water (20° C)	1.333
Fused Quartz	1.46
Glass	1.5 - 1.6
Sapphire	1.77
Diamond	2.42

Figure 9.3: Indices of refraction for a variety of objects, giving the ratio of the speed of light in a vacuum to the speed of light in the medium. Though this is a generally a wavelength-dependent quantity, these values are just averages over the visible wavelengths.

Object	η	k
Gold	0.37	2.82
Silver	0.177	3.638
Copper	0.617	2.63
Steel	2.485	3.433

Figure 9.4: Representative measured values of η and k for a few conductors (data from Hall.)

⟨*BxDF Utility Functions*⟩≡
```
Spectrum FrDiel(Float cosi, Float cost, const Spectrum &etai,
        const Spectrum &etat) {
    Spectrum Rparl = ((etat * cosi) - (etai * cost)) /
        ((etat * cosi) + (etai * cost));
    Spectrum Rperp = ((etai * cosi) - (etat * cost)) /
        ((etai * cosi) + (etat * cost));
    return (Rparl*Rparl + Rperp*Rperp) / 2.f;
}
```

Due to conservation of energy, the energy transmitted by a dielectric is $1 - F_r$, if F_r is the Fresnel reflectance.

155 Spectrum

Conductors don't transmit light. Some of the incident light is absorbed by the material and turned into heat; the Fresnel formula for conductors tells how much is reflected. In addition to depending on $\cos\theta_i$, it depends on η, the index of refraction of the conductor, and k, its *absorption coefficient*. Values for η and k for a few conductors are given in Figure 9.4. As with the index of refraction for dielectrics, these quantities are in general wavelength-dependent though are represented as averages here.

A widely used approximation to the Fresnel reflectance for conductors is

$$\begin{aligned} r_\parallel^2 &= \frac{(\eta^2+k^2)(N\cdot\vec{\omega}_i)^2 - 2\eta(N\cdot\vec{\omega}_i)+1}{(\eta^2+k^2)(N\cdot\vec{\omega}_i)^2 + 2\eta(N\cdot\vec{\omega}_i)+1} \\ r_\perp^2 &= \frac{(\eta^2+k^2) - 2\eta(N\cdot\vec{\omega}_i)+(N\cdot\vec{\omega}_i)^2}{(\eta^2+k^2) + 2\eta(N\cdot\vec{\omega}_i)+(N\cdot\vec{\omega}_i)^2} \end{aligned}$$

⟨*BxDF Utility Functions*⟩+≡
```
Spectrum FrCond(Float cosi, const Spectrum &eta, const Spectrum &k) {
    Spectrum tmp = (eta*eta + k*k) * cosi*cosi;
    Spectrum Rparl2 = (tmp - (2.f * eta * cosi) + 1) /
        (tmp + (2.f * eta * cosi) + 1);
    Spectrum tmp_f = eta*eta + k*k;
    Spectrum Rperp2 = (tmp_f - (2.f * eta * cosi) + cosi*cosi) /
       (tmp_f + (2.f * eta * cosi) + cosi*cosi);
    return (Rparl2 + Rperp2) / 2.f;
}
```

For many conductors, values for η and/or k aren't known–less work has gone into measuring these values for real surfaces than has been done for measuring η for dielectrics. Two approximation methods have been applied in graphics to

finding plausible values for these quantities. Both assume that the reflectance of the object has been measured at normal incidence: the viewer and the light are both looking directly down on the surface. By fixing the value of one of η or k and then substituting into the Fresnel conductor formula, a value for the other parameter can be computed so that the proper reflectance is computed for normal incidence.

The first method computes an approximate value of η, assuming that the absorption coefficient is equal to zero.

⟨*BxDF Utility Functions*⟩+≡
```
Spectrum FresnelApproxEta(const Spectrum &intensity) {
    return (Spectrum(1.) + intensity.Sqrt()) /
        (Spectrum(1.) - intensity.Sqrt());
}
```

And the second technique computes an approximate value of k assuming that $\eta = 1$.

⟨*BxDF Utility Functions*⟩+≡
```
Spectrum FresnelApproxK(const Spectrum &intensity) {
    return 2.f * (intensity / (Spectrum(1.) - intensity)).Sqrt();
}
```

Spectrum 155

For convenience, we will define an abstract `Fresnel` class that defines an interface for computing Fresnel reflection coefficients for given directions, and will write `FresnelConductor` and `FresnelDielectric` instances of it for those two cases. This helps to simplify the implementation of subsequent BRDFs that may need to support both forms.

⟨*BxDF Declarations*⟩+≡
```
class Fresnel {
public:
    ⟨Fresnel Interface⟩
};
```

⟨*Fresnel Interface*⟩+≡
```
virtual Spectrum evaluate(Float cosi) const = 0;
```

⟨*BxDF Declarations*⟩+≡
```
class FresnelConductor : public Fresnel {
public:
    ⟨FresnelConductor Interface⟩
private:
    ⟨FresnelConductor Private Data⟩
};
```

⟨*FresnelConductor Interface*⟩+≡
```
FresnelConductor(const Spectrum &e, const Spectrum &kk)
    : eta(e), k(kk) {
}
```

⟨*FresnelConductor Private Data*⟩≡
```
Spectrum eta, k;
```

The evaluation routine for `FresnelConductor` is simple; it just calls the appropriate utility function.

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum FresnelConductor::evaluate(Float cosi) const {
    return FrCond(fabsf(cosi), eta, k);
}
```

⟨*BxDF Declarations*⟩+≡
```
class FresnelDielectric : public Fresnel {
public:
    ⟨FresnelDielectric Interface⟩
private:
    ⟨FresnelDielectric Private Data⟩
};
```

⟨*FresnelDielectric Interface*⟩+≡
```
FresnelDielectric(Float ei, Float et) {
    eta_i = ei;
    eta_t = et;
}
```

513 Clamp
271 FrCond
271 FrDiel
272 Fresnel
513 max
155 Spectrum

⟨*FresnelDielectric Private Data*⟩≡
```
Float eta_i, eta_t;
```

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum FresnelDielectric::evaluate(Float cosi) const {
    ⟨Compute Fresnel reflectance for dielectric⟩
}
```

For dielectric media, things are a bit more complicated. First, we need to determine if the incident direction is on the outside of the medium or in the inside of it. Next, we apply Snell's law to compute the sine of the angle the transmitted direction makes with the surface normal. We can then compute the cosine of this angle using the identity $\sin^2 x + \cos^2 x = 1$.

⟨*Compute Fresnel reflectance for dielectric*⟩≡
```
cosi = Clamp(cosi, -1.f, 1.f);
⟨Compute indices of refraction for dielectric⟩
Float sint = ei/et * sqrtf(max(0.f, 1.f - cosi*cosi));
if (sint > 1.) {
    ⟨Handle total internal reflection⟩
}
else {
    Float cost = sqrtf(max(0.f, 1.f - sint*sint));
    return FrDiel(fabsf(cosi), cost, ei, et);
}
```

The sign of the cosine of the incident direction indicates on which side of the medium the direction lies; see Figure 9.5. If the cosine is between 0 and 1, the direction is on the outside, and if it is between -1 and 0, it's on the inside. We set

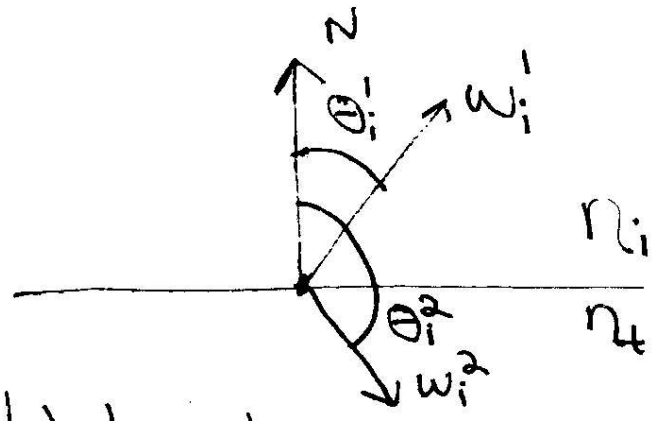


Figure 9.5: The cosine of the angle θ that a direction $\vec{\omega}_i$ makes with the surface normal tells us if the direction is pointing outside the surface (in the same hemisphere as the normal), or inside the surface. In the standard BxDF coordinate system, this test just requires checking the z component of the direction vector. Here, $\vec{\omega}_i^1$ is in the upper hemisphere, with a positive-valued cosine, while $\vec{\omega}_i^2$ is in the lower hemisphere.

Fresnel 272
Spectrum 155
swap 513

the variables `ei` and `et` such that `ei` has the index of refraction of the medium the incident ray is in.

⟨*Compute indices of refraction for dielectric*⟩≡
```
bool entering = cosi > 0.;
Float ei = eta_i, et = eta_t;
if (!entering)
    swap(ei, et);
```

When light is traveling from one medium to another with a lower index of refraction, incident angles near grazing have no transmission into the other medium. The angle at which this happens is called the *critical angle*; when θ_i is greater than the critical angle, *total internal reflection* occurs–all of the light is just reflected. That case is detected here by a value of $\sin\theta_t$ greater than one; we just set `F` to 1, rather than using the Fresnel equations.

⟨*Handle total internal reflection*⟩≡
```
return 1.;
```

⟨*BxDF Declarations*⟩+≡
```
class FresnelNoOp : public Fresnel {
public:
    Spectrum evaluate(Float cosi) const;
};
```

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum FresnelNoOp::evaluate(Float cosi) const {
    return 1.;
}
```

Specular reflection

We can now implement the `SpecularReflection` class, our first specular `BxDF`. It describes perfect reflection. First, we will derive the BRDF for a specular reflector. If the Fresnel equations say that the fraction of light reflected is $F_r(\vec{\omega}_o)$, then we need a BRDF such that

$$L_o(\vec{\omega}_o) = F_r(\vec{\omega}_o)\, L_i(\vec{\omega}_i)$$

where $\vec{\omega}_i$ is the reflection vector for $\vec{\omega}_o$ about the surface normal.

Such a BRDF can be constructed using the *Dirac delta distribution*, a special distribution $\delta(x)$ defined such that

$$\delta(x) = 0$$

for all $x \neq 0$, but where

$$\int_{-\infty}^{\infty} \delta(x)\, dx = 1.$$

This means that

$$\int f(x)\, \delta(x - x_0)\, dx = f(x_0) \tag{9.2.1}$$

The delta distribution requires special handling compared to standard functions. In particular, integrals with delta distributions must be evaluated by sampling the delta distribution; their values cannot be properly computed without doing so. Consider the delta distribution equation 9.2.1: if we tried to evaluate it using the trapezoid rule or some other numerical integration technique, there would be zero probability that any of the evaluation points x_i would have a non-zero value of $\delta(x_i)$. Rather, we must allow the delta distribution to determine the evaluation point itself. We will see this issue in practice for both specular BxDFs as well as some of the light sources to be defined in Chapter 12.

266 BRDF

Using the definition of the delta distribiton in conjunction with the scattering equation, 5.4.8, we can find that the BRDF for perfect specular reflection is

$$f_r(\vec{\omega}_i, \vec{\omega}_o) = F_r(\vec{\omega}_o) \frac{\delta(\vec{\omega}_i - \mathrm{R}(\vec{\omega}_o, N))}{|\cos\theta_i|}$$

if $\mathrm{R}(\vec{\omega}_o, N)$ is the specular reflection vector for $\vec{\omega}_o$ reflected about the surface normal N.

⟨*BxDF Declarations*⟩+≡
```
class SpecularReflection : public BRDF {
public:
    ⟨SpecularReflection Methods⟩
private:
    ⟨SpecularReflection Private Data⟩
};
```

The `SpecularReflection` BxDF takes a `Fresnel` object to describe dielectric or conductor Fresnel properties and an additional spectrum, which is further used to scale the reflected color.

⟨*SpecularReflection Methods*⟩+≡
```
SpecularReflection(const Spectrum &r, Fresnel *f)
    : R(r), fresnel(f) {
}
```

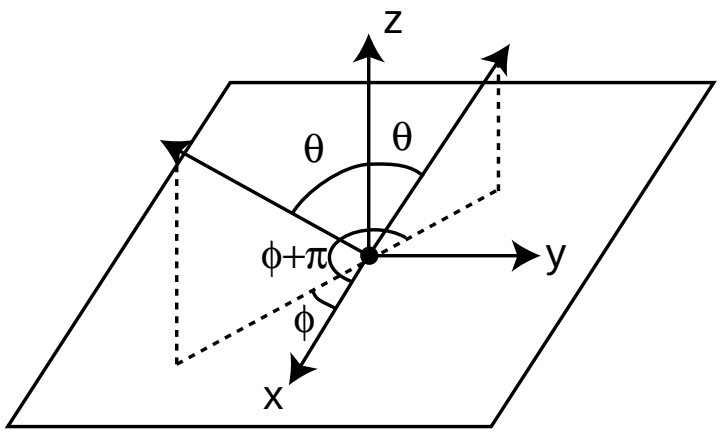


Figure 9.6: Given an incident direction that makes an angle θ with the surface normal and an angle ϕ with the x axis, the reflected ray about the normal makes an angle θ with the normal and $\phi+\pi$ with the x axis. The (x,y,z) coordinates of this direction can be found by scaling the incident direction by $(-1,-1,1)$.

⟨*SpecularReflection Private Data*⟩≡
```
Spectrum R;
Fresnel *fresnel;
```

Fresnel 272
Spectrum 155
SpecularReflection 275
Vector 16

The rest of the implementation is completely straightforward; we return no scattering from `f`, since for an arbitrary pair of directions, the delta function returns no scattering.

⟨*SpecularReflection Methods*⟩+≡
```
Spectrum f(const Vector &, const Vector &) const {
    return Spectrum(0.);
}
```

⟨*SpecularReflection Methods*⟩+≡
```
bool IsSpecular() const { return true; }
```

However, we do implement the `f_delta()` method, which selects an appropriate direction according to the delta function.

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum SpecularReflection::f_delta(const Vector &wo,
        Vector *wi) const {
    ⟨Compute perfect specular reflection direction⟩
    return fresnel->evaluate(wo.z) * R /
        fabsf(wi->z);
}
```

To compute the reflection direction, we need to compute the reflection of $\vec{\omega}_o$ around the surface normal. Because we're doing all these computations in a canonical shading coordinate system where the surface normal is defined to be $(0,0,1)$, the computation is quite simple–all we need to do is to rotate $\vec{\omega}_o$ by π radians about N–see Figure 9.6.

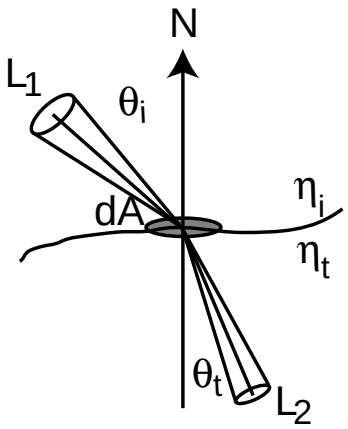


Figure 9.7: The amount of transmitted radiance at the boundary between media with different indices of refraction is scaled by the squared ratio of the two indices of refraction. Intuitively, this can be understood to be the result of the radiance's differential solid angle being squeezed down or expanded as a result of transmission.

Recall the transformation matrix from Chapter 2 for a rotation around the z axis; if the angle of rotation is π radians, it is:

16 Vector

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

When a vector is multiplied by this matrix, the effect is just to negate the x and y components and thus it's easy to compute the reflection direction.

⟨*Compute perfect specular reflection direction*⟩≡

```
*wi = Vector(-wo.x, -wo.y, wo.z);
```

Specular Transmission

We will now derive the BTDF for specular transmission. Snell's law does more than give us the direction for the transmitted ray–interestingly enough it also shows that radiance along a ray changes as the ray goes between media of different indices of refraction.

Consider radiance incident at the boundary between two media, with indices of refraction η_i and η_t for the incident and transmitted media, respectively–see Figure 9.7. We will denote by τ the fraction of incident energy that is transmitted (τ will generally be given by the Fresnel equations.) The amount of transmitted differential flux, then, is:

$$d^2\Phi_t = \tau d^2\Phi_i.$$

If we use the definition of radiance, Equation 5.2.4, we have

$$(L_t \cos\theta_t \, dA \, d\vec{\omega}_t) = \tau(L_i \cos\theta_i \, dA \, d\vec{\omega}_i).$$

Expanding the solid angles to spherical angles, we have

$$(L_t \cos\theta_t \, dA \sin\theta_t \, d\theta_t \, d\phi_t) = \tau(L_i \cos\theta_i \, dA \sin\theta_t \, d\theta_i \, d\phi_i). \tag{9.2.2}$$

We can now differentiate Snell's law with respect to θ, which gives the relation

$$\eta_i \cos\theta_i \, d\theta_i = \eta_t \cos\theta_t \, d\theta_t.$$

And thus,

$$\frac{\cos\theta_i \, d\theta_i}{\cos\theta_t \, d\theta_t} = \frac{\eta_t}{\eta_i}.$$

Substituting this and Snell's law into Equation 9.2.2 and simplifying, we have

$$L_t \eta_i^2 d\phi_t = \tau L_i \eta_t^2 d\phi_i.$$

Because $\phi_t = \phi_i + \pi$, $d\phi_t = d\phi_i$, so

$$L_t = \tau L_i \frac{\eta_t^2}{\eta_i^2}. \tag{9.2.3}$$

The BTDF for specular transmission is thus

$$f_t(\vec{\omega}_o, \vec{\omega}_t) = \frac{\eta_t^2}{\eta_i^2}(1 - F_r(\vec{\omega}_o)) \frac{\delta(\vec{\omega}_o - \mathrm{T}(\vec{\omega}_t))}{|\cos\theta_t|}$$

BTDF 266
Spectrum 155
Vector 16

The `SpecularTransmission` class is almost exactly the same as `SpecularReflection` except that the sampled direction is the direction for perfect specular transmission.

⟨*BxDF Declarations*⟩+≡
```
class SpecularTransmission : public BTDF {
public:
    ⟨SpecularTransmission Methods⟩
private:
    ⟨SpecularTransmission Private Data⟩
};
```

⟨*SpecularTransmission Methods*⟩≡
```
SpecularTransmission(const Spectrum &t, Float ei, Float et)
        : fresnel(ei, et) {
    T = t;
    etai = ei;
    etat = et;
}
```

Because conductors do not transmit light, we always use a `FresnelDielectric` object to do the Fresnel computations.

⟨*SpecularTransmission Private Data*⟩≡
```
Spectrum T;
Float etai, etat;
FresnelDielectric fresnel;
```

⟨*SpecularTransmission Methods*⟩+≡
```
Spectrum f(const Vector &, const Vector &) const {
    return Spectrum(0.);
}
```

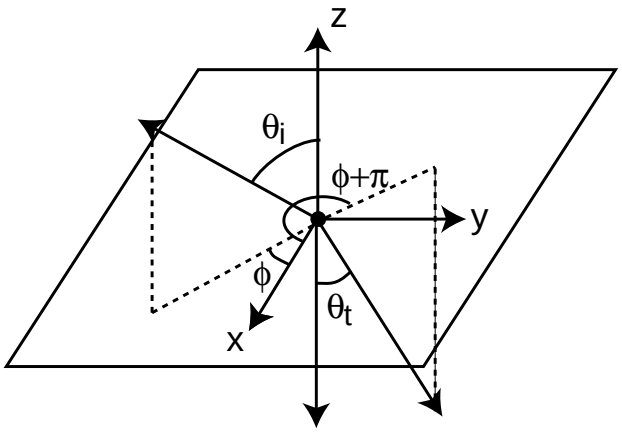


Figure 9.8: The specularly transmitted direction make an angle θ_t with the negated surface normal, $-z$. Like specular reflection, the angle it makes with the x axis is π greater than the incident ray's angle.

⟨*SpecularTransmission Methods*⟩+≡
```
bool IsSpecular() const { return true; }
```

Figure 9.8 shows the basic setting for specular transmission. The incident ray is refracted about the surface normal, with the angle θ_t given by Snell's law.

```
155 Spectrum
278 SpecularTransmission
278 SpecularTransmission::T
513 swap
 16 Vector
```

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum SpecularTransmission::f_delta(const Vector &wo,
        Vector *wi) const {
    ⟨Figure out which η is for incident and which transmitted⟩
    ⟨Compute transmitted ray direction⟩
    Float cosi = wo.z;
    Spectrum F = fresnel.evaluate(cosi);
    return (et*et)/(ei*ei) * (Spectrum(1.)-F) * T / fabsf(wi->z);
}
```

We start by seeing if the incident ray is entering or exiting the refractive medium; we use the convention that the surface normal (and thus the $(0,0,1)$ direction in our local reflection space) is oriented such that it points outside of the object. Therefore, if the z component of the $\vec{\omega}_o$ direction is greater than zero, the incident ray is coming from outside of the object.

⟨*Figure out which* η *is for incident and which transmitted*⟩≡
```
bool entering = wo.z > 0.;
Float ei = etai, et = etat;
if (!entering)
    swap(ei, et);
```

Figure 9.9 shows the basic setting for computing the transmitted ray direction.

We next compute `sini2` and `sint2`, which are the squares of $\sin\theta_i$ and $\sin\theta_t$, respectively. In the reflection coordinate system, $\sin\theta_i$ is equal to the sum of the squares of the x and y components of $\vec{\omega}_o$. $(\sin\theta_t)^2$ can be computed directly from $(\sin\theta_i)^2$ using Snell's law.

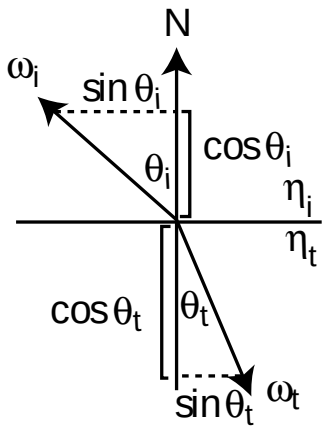


Figure 9.9: Basic geometry for computing the transmitted direction $\vec{\omega}_t$ from the incident direction $\vec{\omega}_i$. The $\cos\theta$ terms are equal to the z components of the direction vectors and the $\sin\theta$ terms are equal to the xy lengths of the corresponding direction vectors.

max 513
Vector 16

We then apply the trigonometric identity $\sin^2\theta + \cos^2\theta = 1$ to compute $\cos\theta_t$ from $\sin\theta_t$; this directly gives us the z component of the transmitted direction. To compute the x and y components, we first mirror $\vec{\omega}_o$ about the normal, as we did for specular reflection, but then scale it by the ratio $\sin\theta_t / \sin\theta_i$ to give it the proper magnitude. From Snell's law, this ratio is just η_i/η_t, though, which we happen to have computed previously.

⟨*Compute transmitted ray direction*⟩≡
```
Float sini2 = wo.x*wo.x + wo.y*wo.y;
Float eta = ei / et;
Float sint2 = eta * eta * sini2;
⟨Handle total internal reflection for transmission⟩
Float cost = sqrtf(max(0.f, 1.f - sint2));
if (entering) cost = -cost;
Float sintOverSini = eta;
*wi = Vector(sintOverSini * -wo.x, sintOverSini * -wo.y, cost);
```

We need to handle the case of total internal reflection here as well; if the squared value of $\sin\theta_t$ is greater than one, no transmission is possible, so we just return black.

⟨*Handle total internal reflection for transmission*⟩≡
```
if (sint2 > 1.) return 0.;
```

9.3 Lambertian Reflection

One of the simplest BRDFs is the Lambertian model; it models a diffuse surface that scatters incident illumination equally in all directions. That is, the particular directions of the incident and outgoing directions make no difference for how much light is scattered. Our Lambertian scattering implementation just takes a reflectance SPD which gives the fraction of incident light that is scattered.

⟨*BxDF Declarations*⟩+≡
```
class Lambertian : public BRDF {
public:
    ⟨Lambertian Methods⟩
private:
    ⟨Lambertian Private Data⟩
};
```

⟨*Lambertian Methods*⟩≡
```
Lambertian(const Spectrum &reflectance)
    : R(reflectance), RoverPI(reflectance * INV_PI) {
}
```

⟨*Lambertian Private Data*⟩≡
```
Spectrum R, RoverPI;
```

The reflection distribution function for `Lambertian` is quite straightforward, since its value is constant; we just return the overall reflectance divided by π. The need for the normalization by $1/\pi$ can be understood with the scattering equation. If a point with Lambertian reflectance ρ is being uniformly illuminated by a constant amount of radiance from all directions, $L_i(\vec{\omega}_o) = c$, then outgoing radiance in any direction should be ρ*c*. Substituting into the scattering equation and simplifying, we find that $f_r = c/\pi$.

266 BRDF
514 INV_PI
155 Spectrum
16 Vector

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum Lambertian::f(const Vector &wo,
        const Vector &wi) const {
    return RoverPI;
}
```

The directional-hemispherical and hemispherical-hemispherical reflectance values for a Lambertian BRDF can be computed analytically.

⟨*Lambertian Methods*⟩+≡
```
Spectrum rho(const Vector &w) const { return R; }
```

⟨*Lambertian Methods*⟩+≡
```
Spectrum rho() const { return R; }
```

9.4 Microfacet Models

Most geometric optics approaches to modeling surface reflection are based on the idea that rough surfaces can be modeled as a collection of small *microfacets*. A surface comprised of microfacets is essentially a heightfield, where the distribution of faces is described statistically. For example, the left half of Figure 9.10 shows the cross-section of a relatively rough surface. On the right is a much smoother microfacet surface.

Microfacet-based BRDF models work by statistically modeling the scattering of light from a large collection of such microfacets. If we assume that the differential area being illuminated, d*A*, is relatively large compared to the size of individual microfacets, then a large number of microfacets are illuminated, so their aggregate behavior can be modeled.

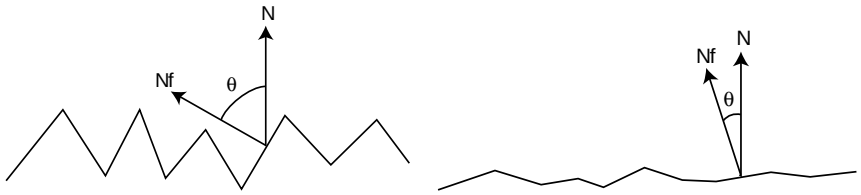


Figure 9.10: Microfacet surface models are often described by a function that describes the distribution of microfacet normals Nf with respect to the surface normal N. The greater the variation of microfacet normals, the rougher the surface is (left). Smooth surfaces have relatively little variation of microfacet normals (right).

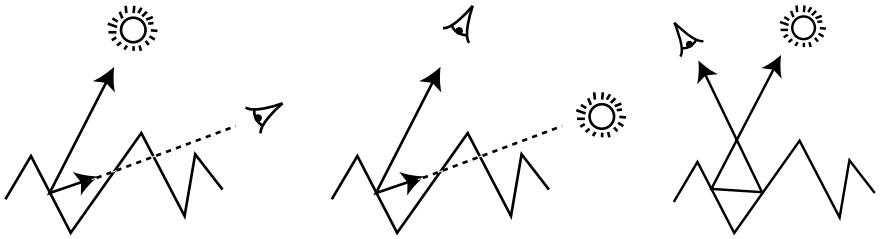


Figure 9.11: There are three important geometric effects to consider with microfacet reflection models. On the left is *masking*, where the microfacet of interest isn't visible to the viewer due to occlusion by another microfacet. In the middle is *shadowing*, where analogously light doesn't reach the microfacet. On the right is *inter-reflection*, where light bounces among the microfacets before reaching the viewer.

The two main components of microfacet normals are an expression for the distribution of facets and a BRDF that describes how light scatters from individual microfacets. Given these, the hard part is to derive a closed form expression that gives the BRDF that describes scattering from such a surface. Perfect mirror reflection is typically used for the microfacet BRDF, though the Oren–Nayar model (described below) treats them as Lambertian reflectors.

Finally, local lighting effects at the microfacet level need to be considered–see Figure 9.11. Consider an individual microfacet of interest, indicated by a heavy line in the figure: on the left, we can see that the viewer may not be able to see it, due to it being masked by another microfacet. As in the middle, it may be in shadow, due to shadowing from a neighboring microfacet. Finally, on the right, inter-reflection among the microfacets may cause the microfacet to receive illumination even though it can't see the light directly (or, it may receive more light than expected). A common simplification is to assume that all of the microfacets are symmetric V-shaped grooves. If this assumption is made, then interreflection with most of the other microfacets can be ignored; only the neighboring microfacet in the groove needs to be considered.

Particular microfacet-based BRDFs consider each of these effects with varying degrees of accuracy–the general approach is to make the best approximations to these effects possible, given the desire of wrapping up with a relatively-easily evaluated expression at the end.

Oren–Nayar diffuse reflection

Oren and Nayar observed that real-world objects don't match Lambertian reflection very well. Specifically, rough surfaces generally appear brighter as the illumination direction approaches the viewing direction. They developed a model that started with a description rough surfaces in terms of symmetric V-microfacets. They assumed that each microfacet exhibited Lambertian reflection individually, and derived a BRDF that models the aggregate reflection of the collection of microfacets. The distribution of microfacets was assumed to be Gaussian, where the parameter σ described the standard deviation of the orientation angle.

The resulting model, which accounted for shadowing, masking, and inter-reflection among the microfacets didn't have a closed-form solution, so they set out to find a functional approximation that fit it well. The result is:

$$f_r(\vec{\omega}_i, \vec{\omega}_o) = \frac{\rho}{\pi}\left(A + B\max(0, \cos(\phi_i - \phi_o))\sin\vec{\omega}_+ \tan\vec{\omega}_-\right)$$

where if σ is in radians,

$$\begin{aligned} A &= 1 - \frac{\sigma^2}{2(\sigma^2 + 0.33)} \\ B &= \frac{0.45\sigma^2}{\sigma^2 + 0.09} \\ \vec{\omega}_+ &= \max(\theta_i, \theta_o) \\ \vec{\omega}_- &= \min(\theta_i, \theta_o) \end{aligned}$$

514 Radians
155 Spectrum

We can precompute the values of the A and B parameters to the model and store them away in the constructor; this will save us work in evaluating the BRDF later.

⟨*OrenNayar Methods*⟩+≡
```
OrenNayar(const Spectrum &reflectance, Float sig)
    : R(reflectance) {
    Float sigma2 = Radians(sig*sig);
    A = 1.f - (sigma2 / (2.f * (sigma2 + 0.33f)));
    B = 0.45f * sigma2 / (sigma2 + 0.09f);
}
```

⟨*OrenNayar Private Data*⟩≡
```
Spectrum R;
Float A, B;
```

Evaluating the model is relatively straightforward; just a matter of applying some trigonometry to computing the values for the terms in the model. We start by computing and storing $\sin\theta_i$ and $\sin\theta_o$; recall from to the section on specular transmission and Figure 9.12 that the *xy* magnitude of the direction vectors gives these values.

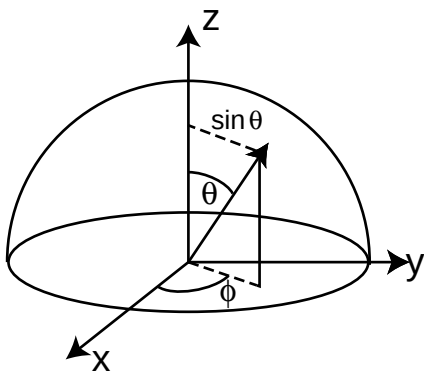


Figure 9.12: As was done for the SpecularTransmission BTDF, the $\sin\theta$ term is found by computing the length of the dashed line, which is the magnitude of the *xy* components of the vector. The $\sin\phi$ and $\cos\phi$ terms can be computed using the circular coordinate equations $x = r\cos\phi$ and $y = r\sin\phi$, where r, the length of the dashed line, was already computed for $\sin\theta$.

⟨*BxDF Method Definitions*⟩+≡

```
Spectrum OrenNayar::f(const Vector &wo,
        const Vector &wi) const {
    Float sinthetai = sqrtf(wi.x*wi.x + wi.y*wi.y);
    Float sinthetao = sqrtf(wo.x*wo.x + wo.y*wo.y);
    ⟨Compute cosine term of Oren–Nayar model⟩
    ⟨Compute sine and tangent terms of Oren–Nayar model⟩
    return R * INV_PI * (A + B * maxcos * sinalpha * tanbeta);
}
```

INV_PI 514
OrenNayar::R 283
Spectrum 155
Vector 16

We now need to compute the $\max(0, \cos(\phi_i - \phi_o))$ term. We can apply the trigonometric identity

$$\cos(a-b) = \cos a \cos b + \sin a \sin b,$$

such that we just need to compute the sines and cosines of ϕ_i and ϕ_o. The geometric setting for this is shown in Figure 9.12. In the plane of the point being shaded, the vector $\vec{\omega}$ has coordinates (x,y), which are given by $r\cos\phi$ and $r\sin\phi$, respectively. The radius r is just $\sin\theta$, so

$$\begin{aligned} \cos\phi &= \frac{x}{r} = \frac{x}{\sin\theta} \\ \sin\phi &= \frac{y}{r} = \frac{y}{\sin\theta}. \end{aligned}$$

⟨*Compute cosine term of Oren–Nayar model*⟩≡

```
Float sinphii = wi.y / sinthetai;
Float cosphii = wi.x / sinthetai;
Float sinphio = wo.y / sinthetao;
Float cosphio = wo.x / sinthetao;
Float dcos = cosphii * cosphio + sinphii * sinphio;
Float maxcos = max(0.f, dcos);
```

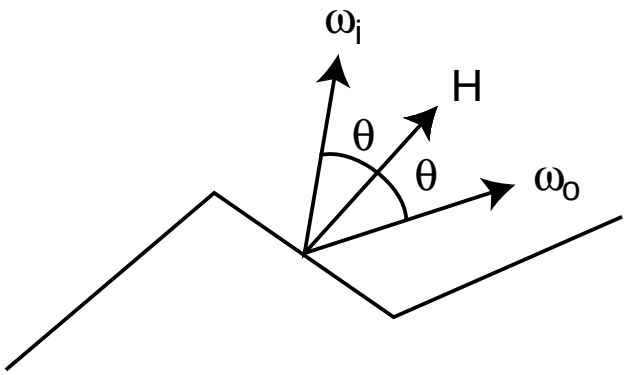


Figure 9.13: For perfectly specular microfacets and a given pair of directions $\vec{\omega}_i$ and $\vec{\omega}_o$, only those microfacets with normal $\vec{\omega}_h = \widehat{\vec{\omega}_i + \vec{\omega}_o}$ will reflect any light from $\vec{\omega}_i$ to $\vec{\omega}_o$.

Finally, we compute the $\sin\alpha$ and $\tan\beta$ terms. Note that whichever of $\vec{\omega}_i$ or $\vec{\omega}_o$ has a larger value for $\cos\theta$ (i.e. a larger value of its z component), has a *smaller* value for θ. Given the knowledge of which angle is smaller, we can set $\sin\alpha$ directly from the appropriate $\sin\theta$ value already computed. The tangent can then just be computed using the identity $\tan a = \sin a / \cos a$.

513 max

⟨*Compute sine and tangent terms of Oren–Nayar model*⟩≡

```
Float sinalpha, tanbeta;
if (fabsf(wi.z) > fabsf(wo.z)) {
    sinalpha = sinthetao;
    tanbeta = sinthetai / fabsf(wi.z);
}
else {
    sinalpha = sinthetai;
    tanbeta = sinthetao / fabsf(wo.z);
}
```

Torrance–Sparrow model

The first(?) microfacet model was developed by Torrance and Sparrow to model metallic surfaces. The modeled surfaces as collections of perfectly smooth planar microfacets; because they are smooth, the microfacets have perfect specular reflection. The surface is statistically described by a distribution function $D(\theta)$ that gives the probability that a microfacet has orientation θ (recall Figure 9.10 which shows how roughness and the microfacet normal distribution function are related).

Because the microfacets are perfectly specular, only those that are oriented exactly so that they reflect the incident direction $\vec{\omega}_i$ to the outgoing direction $\vec{\omega}_o$ give any reflection for that pair of directions. It can be shown that only those microfacets with a normal equal to the *half-angle vector*,

$$\vec{\omega}_h = \widehat{\vec{\omega}_i + \vec{\omega}_o}$$

cause perfect specular reflection from $\vec{\omega}_i$ to $\vec{\omega}_o$ (and vice-versa). (See Figure 9.13.)

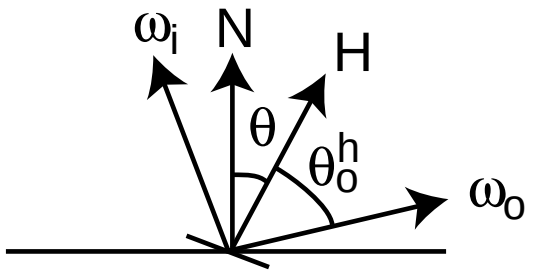


Figure 9.14: Setting for the derivation of the Torrance–Sparrow model. For directions $\vec{\omega}_i$ and $\vec{\omega}_i$, only microfacets with normal $\vec{\omega}_h$ reflect light. The angle between $\vec{\omega}_h$ and N is denoted by θ and the angle between $\vec{\omega}_h$ and $\vec{\omega}_o$ is denoted by θ_h. (The angle between $\vec{\omega}_h$ and $\vec{\omega}_i$ is also necessarily θ_h.)

The derivation of the Torrance–Sparrow has a number of interesting steps; we'll go through it in some detail here.

Consider the differential flux incident on the microfacets oriented with half-angle $\vec{\omega}_h$ for directions $\vec{\omega}_i$ and $\vec{\omega}_o$, $\mathrm{d}^2\Phi_h$. From the definition of radiance, Equation 5.2.4, it is

$$\mathrm{d}^2\Phi_h = L_i(\vec{\omega}_i)\,\mathrm{d}\vec{\omega}\,\mathrm{d}A^{\perp}(\vec{\omega}_h) = L_i(\vec{\omega}_i)\,\mathrm{d}\vec{\omega}\cos\theta_h\,\mathrm{d}A(\vec{\omega}_h),$$

where we have written $\mathrm{d}A\,(\vec{\omega}_h)$ for the area measure of the microfacets with orientation $\vec{\omega}_h$ and $\cos\theta_h$ for the cosine of the angle between $\vec{\omega}_i$ and $\vec{\omega}_h$ (see Figure 9.14.)

The differential area of microfacets with orientation $\vec{\omega}_h$ is just

$$\mathrm{d}A(\vec{\omega}_h) = D(\vec{\omega}_h)\mathrm{d}\vec{\omega}_h\mathrm{d}A.$$

The first two terms describe the differential area of facets per unit area that have the proper orientation, and the $\mathrm{d}A$ term converts this to differential area.

Therefore,

$$\mathrm{d}^2\Phi_h = L_i(\vec{\omega}_i)\,\mathrm{d}\vec{\omega}\cos\theta_h D(\vec{\omega}_h)\,\mathrm{d}\vec{\omega}_h\,\mathrm{d}A(\vec{\omega}_h). \tag{9.4.4}$$

If we assume that the microfacets individually reflect light according to Fresnel's law, the outgoing flux is

$$\mathrm{d}^2\Phi_o = F(\vec{\omega}_i,\vec{\omega}_o)\mathrm{d}^2\Phi_h. \tag{9.4.5}$$

Again using the definition of radiance, the reflected outgoing radiance is

$$L(\vec{\omega}_o) = \frac{\mathrm{d}^2\Phi_o}{\mathrm{d}\vec{\omega}_o\cos\theta_o\mathrm{d}A}.$$

If we substitute Equation 9.4.5 into this and then Equation 9.4.4 into the result, we have

$$L(\vec{\omega}_o) = \frac{F(\vec{\omega}_i,\vec{\omega}_o)\,L_i(\vec{\omega}_o)\,\mathrm{d}\vec{\omega}_i D(\vec{\omega}_h)\,\mathrm{d}\vec{\omega}_h\,\mathrm{d}A\cos\theta_h}{\mathrm{d}\vec{\omega}_o\,\mathrm{d}A\cos\theta_o}$$

In Section 14.3, we will derive an important relation between $\mathrm{d}\vec{\omega}_h$ and $\mathrm{d}\vec{\omega}_o$; it is

$$\mathrm{d}\vec{\omega}_h = \frac{\mathrm{d}\vec{\omega}_o}{4\cos\theta_h}.$$

We can substitute this into the previous equation and simplify, giving

$$L(\vec{\omega}_o) = \frac{F(\vec{\omega}_i, \vec{\omega}_o)\, L_i(\vec{\omega}_o)\, D(\vec{\omega}_h)\, \mathrm{d}\vec{\omega}_i}{4 \cos\theta_o}.$$

We can now apply the definition of the BRDF, Equation 5.4.7, giving us the Torrance–Sparrow BRDF:

$$f_r(\vec{\omega}_i, \vec{\omega}_o) = \frac{D(\vec{\omega}_h)\, F(\vec{\omega}_i, \vec{\omega}_o)}{4 \cos\theta_i \cos\theta_o}$$

Note that this obeys reciprocity.

The Torrance–Sparrow model also includes a *geometric attenuation* term, which describes the fraction of microfacets that are masked or shadowed, given directions $\vec{\omega}_i$ and $\vec{\omega}_o$. This G term can just be included in the derivation as the Fresnel term was above. The full model, then, is

$$f_r(\vec{\omega}_i, \vec{\omega}_o) = \frac{D(\vec{\omega}_h)\, G(\vec{\omega}_i, \vec{\omega}_o)\, F(\vec{\omega}_i, \vec{\omega}_o)}{4 \cos\theta_i \cos\theta_o}. \tag{9.4.6}$$

One of the nice things about the Torrance–Sparrow model is that the derivation doesn't depend on the particular microfacet distribution being used. Furthermore, because it doesn't depend on a particular Fresnel function, it can be used for both conductors and dielectrics. However, reflection functions besides perfect specular reflection can *not* be easily substituted: the relationship between $\mathrm{d}\vec{\omega}_h$ and $\mathrm{d}\vec{\omega}_o$ used in its derivation depends on the specular reflection assumption.

266 BRDF
16 Vector

We can now define a general microfacet-based BRDF. It takes a pointer to an abstract `MicrofacetDistribution` class, which provides routines to compute the D term of the Torrance–Sparrow model. Here is the pure virtual function that `MicrofacetDistribution`s must implement; it gives the probability density for microfacets to be oriented with normal $\vec{\omega}_h$.

⟨*BxDF Declarations*⟩+≡
```
class MicrofacetDistribution {
public:
    ⟨MicrofacetDistribution Interface⟩
};
```

⟨*MicrofacetDistribution Interface*⟩+≡
```
virtual Float D(const Vector &wh) const = 0;
```

The `Microfacet` BRDF, then, just takes a pointer to a distribution, the reflectance of the object, and a Fresnel function.

⟨*BxDF Declarations*⟩+≡
```
class Microfacet : public BRDF {
public:
    ⟨Microfacet Methods⟩
private:
    ⟨Microfacet Private Data⟩
};
```

⟨*BxDF Method Definitions*⟩+≡
```
Microfacet::Microfacet(const Spectrum &reflectance, Fresnel *f,
        MicrofacetDistribution *d)
        : R(reflectance) {
    fresnel = f;
    distribution = d;
}
```

⟨*Microfacet Private Data*⟩≡
```
Spectrum R;
MicrofacetDistribution *distribution;
Fresnel *fresnel;
```

Evaluating the terms of the BRDF is straightforward. For the Fresnel term, recall that the angle θ_h is the same between $\vec{\omega}_h$ and both $\vec{\omega}_i$ and $\vec{\omega}_o$, so it doesn't matter which of them we use to compute the cosine of the angle between them.

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum Microfacet::f(const Vector &wo, const Vector &wi) const {
    Float cosThetaO = fabsf(wo.z);
    Float cosThetaI = fabsf(wi.z);
    Vector wh = (wi + wo).Hat();
    Spectrum F = fresnel->evaluate(Dot(wi, wh));
    return R * distribution->D(wh) * G(wi, wo, wh) * F /
         (4.f * cosThetaI * cosThetaO);
}
```

Fresnel 272
Hat 19
Microfacet 287
MicrofacetDistribution 287
min 513
Spectrum 155
Vector 16

Torrance and Sparrow derived a geometric attenuation term assuming that the microfacets were made of infinitely long V-shaped grooves. This assumption is a more restricted one than was used to derive the reflection model from the general microfacet distribution, but it made it possible for them to derive a closed form result. Furthermore, their attenuation factor doesn't account for the roughness of the surface, which naturally affects the amount of shadowing and masking. That said, the result is easy to evaluate and the overall model matches real-world surfaces well.

⟨*Microfacet Methods*⟩+≡
```
Float G(const Vector &wo, const Vector &wi,
        const Vector &wh) const {
    Float NdotH = fabsf(wh.z);
    Float NdotWO = fabsf(wo.z), NdotWI = fabsf(wi.z);
    Float WOdotH = fabsf(Dot(wo, wh));
    return min(1.f, min((2.f * NdotH * NdotWO / WOdotH),
                (2.f * NdotH * NdotWI / WOdotH)));
}
```

Blinn Microfacet Distribution

The Blinn microfacet model models an geometric falloff of distribution of microfacet normal orientations with respect to the underlying surface normal. The most likely microfacet orientation is in the surface normal direction, falling off

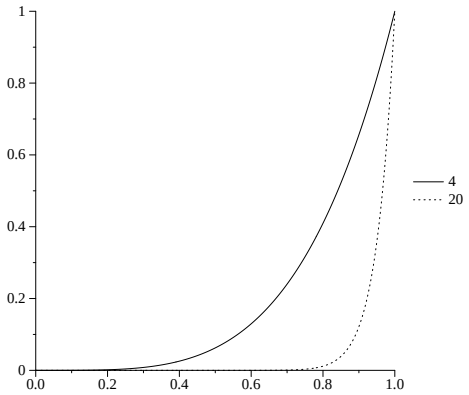


Figure 9.15: Graph showing the effect of varying the exponent for the Blinn microfacet distribution model. The solid line shows the graph of the non-normalized distribution function x^4, and the dotted line shows the graph of x^{20}. The larger the exponent, the more likely it is that a microfacet will be oriented close to the surface normal, as would be the case for a smooth surface.

to no microfacets oriented perpendicular to the normal. For smooth surfaces, this falloff happens very quickly, and for rough surfaces, it is more gradual.

287 MicrofacetDistribution

⟨*BxDF Declarations*⟩+≡
```
class Blinn : public MicrofacetDistribution {
public:
    Blinn(Float e) { exponent = e; }
    ⟨Blinn Method Declarations⟩
private:
    Float exponent;
};
```

The Blinn model is

$$D(\vec{\omega}_h) = c(\vec{\omega}_h \cdot N)^e$$

where e is a user-supplied exponent that controls the rate and c is a constant term that normalizes the distribution so that it is a valid (normalized) probability distribution function. It is

$$c = e + 1.$$

Figure 9.15 gives a sense of how varying the exponent changes the distribution. The solid line shows the distribution of cosines of the angle between the surface normal and the microfacet normal with an exponent of 4, corresponding to a rough surface. As such there is a fair probability of microfacets being oriented in a direction substantially far away from the normal. The dashed line shows the effect of a higher exponent of 20, corresponding to a smoother surface. For this case, there is low probability that any microfacets will be very far off from the surface normal direction.

⟨*Blinn Method Declarations*⟩≡
```
Float D(const Vector &wh) const {
    Float costhetah = fabsf(wh.z);
    return (exponent+1) * powf(max(0.f, costhetah), exponent);
}
```

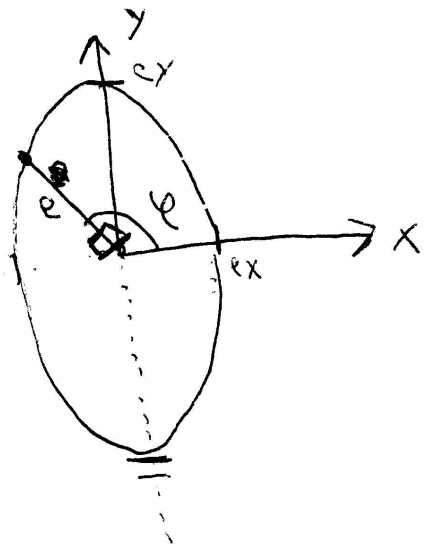


Figure 9.16: The two exponents e_x and e_y for the anisotropic microfacet distribution function give specular exponents for microfacets facing exactly along the x and y axes, respectively. For microfacets with other orientations, the exponent e is comptued by finding the radius e of the super-ellipse for the actual orientation angle ϕ.

max 513
Vector 16

Anisotropic microfacet model

Ashikhmin and Shirley have developed a microfacet distribution function for modeling the appearence of anisotropic surfaces. Recall that an anisotropic BRDF is one where the reflection characteristics at a point vary as the surface is rotated about that point in the plane perpindicular to the surface normal. Brushed metals and some types of fabric exhibit anisotropy.

Their model is physicall-based, has intuitive parameters, is efficient, and fits well into the Monte Carlo integration techniques that will be introduced in later chapters. We won't derive their model in detail here, but refer the interested reader to their original paper and technical report (AS02; AS00). Their model takes two pareneters: e_x, which gives an exponent for the distribution function for half-angle vectors with an azimuthal angle that orients them exactly along the $\pm x$ axis, and e_y, an exponents for microfacets oriented along the $\pm y$ axis. Exponents for intemediate orientations are found by considering these two values as es the lengths of the axis of a (super-quadric?) ellipse and finding the appropriate value for the actual microfacet orientation–see Figure 9.16.

The resulting microfacet distribution function is

$$D(\vec{\omega}_h) = \sqrt{(e_x+1)\cdot(e_y+1)}(\vec{\omega}_h \cdot N)^{e_x \cos^2\phi + e_y \sin^2\phi}.$$

⟨*BxDF Declarations*⟩+≡
```
class Anisotropic : public MicrofacetDistribution {
public:
    Anisotropic(Float x, Float y) { ex = x; ey = y; }
    ⟨Anisotropic Method Declarations⟩
private:
    Float ex, ey;
};
```

The terms of the distribution function can be computed quite efficiently. Recall from the Oren–Nayar BRDF that $\cos\phi = x/\sin\theta$ aqnd $\sin\phi = y/\sin\theta$. Since we want to compute $\cos^2\phi$ and $\sin^2\phi$, however, we can use the substitution $\sin^2\theta + \cos^2\theta = 1$, so that

$$\cos^2\phi = \frac{x^2}{1-z^2}$$
$$\sin^2\phi = \frac{y^2}{1-z^2}.$$

Thus, the implementation is:

⟨*Anisotropic Method Declarations*⟩≡
```
Float D(const Vector &wh) const {
    Float costhetah = fabsf(wh.z);
    Float e = (ex * wh.x * wh.x + ey * wh.y * wh.y) /
        (1.f - costhetah * costhetah);
    return sqrtf((ex+1)*(ey+1)) * powf(costhetah, e);
}
```

287 MicrofacetDistribution
16 Vector

9.5 Lafortune Model

Lafortune, Foo, Torrance, and Greenberg have developed a BRDF Model tailored for fitting measured BRDF data to a parameterized model with a relatively small number of parameters. As a bonus, their model is easy to implement and quite efficient. The genesis of their model is the *Phong model*–one of the first BRDF models developed for graphics. The original Phong model has a number of shortcomings–most glaring that it is not reciprical or energy-conserving–that the Lafortune model avoids.

The *modified Phong BRDF*, which is reciprocal, is

$$f_r(\vec{\omega}_i, \vec{\omega}_o) = (\vec{\omega}_i \cdot R(\vec{\omega}_o, N))^e = (\vec{\omega}_o \cdot R(\vec{\omega}_i, N))^e,$$

where $R(\vec{\omega}, N)$ is the operator that reflects the vector $\vec{\omega}$ about the surface normal N. Like the Blinn microfacet distribution model, the cosine of the angle between the two vectors is raised to a given power. In the canonical BRDF coordinate system, the Phong model can be equivalently written as

$$f_r(\vec{\omega}_i, \vec{\omega}_o) = (\vec{\omega}_i \cdot (\vec{\omega}_o \times (-1, -11)))^e = (\vec{\omega}_o \cdot (\vec{\omega}_i \times (-1, -11)))^e.$$

The Lafortune model uses the key observation that the vector $(-1,-1,1)$ in the modified Phong model can itself be a parameter to the BRDF. We will call this

vector the *orientation vector*, since it orients the direction of maximum reflection. For example, if the orientation vector was $(-1,-1,0.5)$, the main reflection vector would be lowered from the perfect specular direction to be closer to the surface. (Many glossy surfaces in fact have such *off-specular* reflective behavior. The Blinn microfacet model is maximally reflective in the specular direction, however.)

If the orientation vector was $(1,1,1)$, the surface would be *retro-reflective*–light would be primarily reflected back along the direction it arrived along. The moon is an example of a retro-reflective surface.

Given the generalization of expressing the Phong model in terms of an orientation vector, the Lafortune model expresses the BRDF as the sum of multiple *lobes*, each one specified in terms of an orientation vector and a specular exponent plus a Lambertian diffuse term. The contribution of each lobe is determined by the magnitude of the orientation vector–the reo-riented incident vector is no longer necessarily of unit length and its lenght affects the magnitude of the dot product. (This makes for an unintuitive control for manual adjustment of the BRDF's characteristics, though it is less troublesome if the BRDF is being automatically fit to measured data.) Thus, we have:

$$f_r(\vec{\omega}_i,\vec{\omega}_o) = \frac{\rho_d}{\pi} + \sum_i^{\text{nlobes}} (\vec{\omega}_i \cdot (\vec{\omega}_o \times o_i))_i^e,$$

BRDF 266

where ρ_d is the diffuse reflectance, o_i are the orientation vectors, and e_i are the specular exponents.

As a further generalization, each orientation vector and specular exponent is allowed to vary as a funciton of wavelength; we represent each of them with `Spectrum` objects in the implementation below. This gives a natural way to express wavelength-dependent reflection variation in the model.

⟨*BxDF Declarations*⟩+≡
```
class Lafortune : public BRDF {
public:
    ⟨Lafortune Methods⟩
private:
    ⟨Lafortune Private Data⟩
};
```

Our implementation limits the number of separate lobes to the compile-time constant `MAX_LOBES`, in order to avoid additional run-time memory allocation.

⟨*BxDF Method Definitions*⟩+≡
```
Lafortune::Lafortune(const Spectrum &r, int nl, const Spectrum *xx,
    const Spectrum *yy, const Spectrum *zz, const Spectrum *e)
    : R(r) {
    nLobes = nl;
    Assert(nLobes <= MAX_LOBES);
    for (int i = 0; i < nLobes; ++i) {
        x[i] = xx[i];
        y[i] = yy[i];
        z[i] = zz[i];
        exponent[i] = e[i];
    }
}
```

⟨*Lafortune Private Data*⟩≡
```
Spectrum R;
#define MAX_LOBES 3
Spectrum x[MAX_LOBES], y[MAX_LOBES], z[MAX_LOBES];
Spectrum exponent[MAX_LOBES];
int nLobes;
```

498 Assert
292 Lafortune
155 Spectrum
16 Vector

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum Lafortune::f(const Vector &wo, const Vector &wi) const {
    Spectrum ret = R / M_PI;
    for (int i = 0; i < nLobes; ++i) {
        ⟨Evaluate Lafortune model for ith lobe⟩
    }
    return ret;
}
```

The paper the introduced this model originally defined the orientation vector so that the vector $(1,1,1)$ would give the classic Phong model. However, we will use the different convention that $(-1,-1,1)$ gives the Phong model, in order to be consistent with our specular reflection BRDF.

Evaluating each lobe is straightforward. We simultaneously compute the re-oriented $\vec{\omega}_o$ vector by multiplying its *x*, *y*, and *z* coefficients with the appropriate spectral orientation coefficients and compute the dot product of the result with $\vec{\omega}_i$, giving a spectral result which is itself then raised to the spectral exponent provided.

⟨*Evaluate Lafortune model for ith lobe*⟩≡
```
Spectrum v = x[i] * wo.x * wi.x + y[i] * wo.y * wi.y +
    z[i] * wo.z * wi.z;
ret += v.Pow(exponent[i]);
```

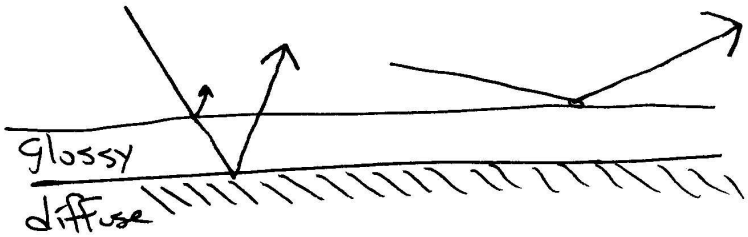


Figure 9.17: The `FresnelBlend` BRDF models the effect of a surface with a glossy layer on top of a diffuse substrate. As on the angle of incidence of the direction vectors $\vec{\omega}_i$ and $\vec{\omega}_o$ heads toward glancing (right), the amount of light that reaches the diffuse substrate is reduced by Fresnel effects and the diffuse layer becomes less visibly apparent.

9.6 Fresnel Incidence Effects

BRDF 266

Shirley and collaborators have often made the observation that most BRDF Models in graphics do not account for the effect of Fresnel reflection reducing the amount of light reaching the bottom level of layered objects. Consider a polished wood table or a wall with glossy paint: if you look at their surfaces head-on, you primarily see the wood or the paint pigment color. As you move your viewpoint toward a glancing angle, you see less of the underlying color as it is overwhelmed by increasing glossy reflection due to Fresnel effects. The images in Figure XXX show this effect.

In this section, we will implement a BRDF model due to Ashikhmin and Shirley that models a diffuse underlying surface with a glossy specular surface abobve it. The effect of reflection from the diffuse surface is modulated according to how much energy is left after Fresnel effects have been considered. Figure 9.17. shows this: on the left, the incident direction is close to the normal, so most light is transmitted to the diffuse layer and the diffuse term dominates. On the right, the incident direction is close to glancing, so glossy reflection is the primary mode of reflection.

⟨*BxDF Declarations*⟩+≡
```
class FresnelBlend : public BRDF {
public:
    ⟨FresnelBlend Methods⟩
private:
    ⟨FresnelBlend Private Data⟩
};
```

The model takes two spectra, representing diffuse and specular reflectance, and a microfacet distribution function for the glossy layer.

⟨*BxDF Method Definitions*⟩+≡
```
FresnelBlend::FresnelBlend(const Spectrum &d, const Spectrum &s,
        MicrofacetDistribution *dist)
    : Rd(d), Rs(s) {
    distribution = dist;
}
```

⟨*FresnelBlend Private Data*⟩≡
```
Spectrum Rd, Rs;
MicrofacetDistribution *distribution;
```

This model is based on the weighted sum of a glossy specular term and a diffuse term. Accounting for reciprocity and energy conservation, the glossy specular term is derived as

$$f_r(\vec{\omega}_i,\vec{\omega}_o) = \frac{D(\vec{\omega}_h)}{8\pi(\vec{\omega}_h\cdot\vec{\omega}_i)\left(\max((N\cdot\vec{\omega}_i),(N\cdot\vec{\omega}_o))\right)}F(\vec{\omega}_i,\vec{\omega}_o),$$

where $D(\vec{\omega}_h)$ is a microfacet distribution term and $F(\vec{\omega}_i,\vec{\omega}_o)$ represents Fresnel reflectance. Note that this is quite similar to the Torrance–Sparrow model.

The key to Ashikhmin and Shirley's model was deriving a diffuse term such that the model still obeyed reciprocity and conserved energy. One key to making the derivation practical was using an approximation to the Fresnel reflection equations due to Schlick, who computed Fresnel reflection as

$$F(\cos\theta) = R + (1-R)(1-\cos\theta)^5,$$

where R is the reflectance of the surface at normal incidence.

Given this Fresnel term, they showed that the diffuse term below successfully modeled Fresnel-based reduced diffuse reflection in a physically plausible manner:

$$f_r(\vec{\omega}_i,\vec{\omega}_o) = \frac{28R_d}{23\pi}(1-R_s)\left(1-\left(1-\frac{(N\cdot\vec{\omega}_i)}{2}\right)^5\right)\left(1-\left(1-\frac{(N\cdot\vec{\omega}_o)}{2}\right)^5\right)$$

⟨*FresnelBlend Methods*⟩+≡
```
Spectrum SchlickFresnel(Float costheta) const {
    return Rs + powf(1 - costheta, 5.f) * (Spectrum(1.) - Rs);
}
```

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum FresnelBlend::f(const Vector &wo, const Vector &wi) const {
    Spectrum diffuse = (28.f/(23.f*M_PI)) * Rd *
        (Spectrum(1.) - Rs) *
        (1 - powf(1 - .5f * fabsf(wi.z), 5)) *
        (1 - powf(1 - .5f * fabsf(wo.z), 5));
    Vector H = (wi + wo).Hat();
    Spectrum specular = distribution->D(H) /
        (8.f * M_PI * fabsf(Dot(wi, H)) * max(fabsf(wi.z), fabsf(wo.z))) *
        SchlickFresnel(Dot(wi, H));
    return diffuse + specular;
}
```

294 FresnelBlend
19 Hat
513 max
287 MicrofacetDistribution
155 Spectrum
16 Vector

Further Reading

Phong developed and early empirical reflection model for glossy surfaces in computer graphics (Pho75). Though not reciprocal or energy-conserving, it was a cornerstone of the first synthetic images of non-Lambertian objects. The Torrance–Sparrow microfacet model is described in (TS67); a variant of it was applied to computer graphics by Cook and Torrance (CT81; CT82).

Hall's book collected and described the state of the art in physically-based surface reflection models for graphics in 1989; it remains a seminal reference (Hal89). It discussed the physics of surface reflection in detail, with many pointers to the original literature and with many tables of useful measured data about reflection from real surfaces.

Cite wavelength-dependent IOR work? Incl Glassner DIS (Gla95, Section 11.8), Delvin et al survey? Smits, musgrave stuff

Beckman developed an early physical optics model of surface reflection XXX, which Kajiya used to derive an anisotropic model for computer graphics (Kaj85). Beckman's work was built upon more recently by He et al (HTSG91). However, Nayar et al have shown that some reflection models based on physical (wave) optics have substantially the same characteristics as those based on geometric optics–the geometric optics approximations don't seem to cause too much error (except for very smooth surfaces) (NIK91). This is a helpful result, giving experimental basis to the general belief that wave optics models aren't usually worth their computational expense for computer graphics applications.

The Oren–Nayar Lambertian model is described in their 1994 SIGGRAPH paper (ON94). Other notable BRDF models recently developed in computer graphics include Ward's anisotropic model (War92), Hanrahan and Krueger's model of subsurface reflection (HK93), and Schlick (Sch93). Ashikhmin et al recently developed techniques for computing self-shadowing terms for arbitrary microfacet distributions, without needing to make the assumptions that Torrance and Sparrow did (APS00).

Lafortune et al (LFTG97).

Ashikhmin and Shirley anisotropic model (AS02; AS00)

A number of researchers have investigated how to find BRDFs based on modeling the small-scale geometric features of a reflective surface. This work includes Cabral et al's computing BRDFs from bump maps (CMS87), Fournier's normal distribution functions (Fou92), and Westin et al (WAT92).

Exercises

9.1 simulation: geom and brdf, fire rays at it, tabularize BRDF. isotropic a big win–3d table θ_o, θ_i, $d\phi$...

9.2 Hanrahan–Krueger subsurface stuff.

9.3 Derive Snell's law, using Fermat's principle, give basic setup for it...

10. Materials

The low-level BRDFs and BTDFs introduced in Chapter 9 solve only part of the problem of describing how a surface scatters light. Although they describe how light is scattered at a particular point on the surface, but we still need to know *which* BRDFs and BTDFs describe the scattering at a point, and what the parameters to these scattering functions are.

In this chapter, we provide a general procedural shading mechanism to generate BRDFs and BTDFs for points on surfaces. The basic idea is that a *surface shader* is bound to each primitive in the scene. The surface shader is a small procedure that is executed at a point to be shaded; it returns the `BSDF`, which holds a collection of `BRDF`s and `BTDF`s that describes the scattering at the point. This is a somewhat different shading paradigm than many rendering systems use—most combine the function of the surface shader and the lighting integrator (see Chapter 15) into a single shader. By separating these two pieces, a more flexible system results that is better able to handle new light transport algorithms.

10.1 BSDFs

We now present the implementation of our the general `BSDF` class. It represents a weighted mixture of `BRDF`s and `BTDF`s, allowing the rest of the system to work with composite `BSDF`s directly, rather than having to consider all of the components they are built from.

Equally important, the `BSDF` class hides the mechanics of shading normals from the rest of the system. Shading normals, either from per-vertex normals on polygonal meshes, or from bump mapping, can substantially improve the visual richness of scenes. However, because they are an *ad hoc* construct, they are tricky to incorporate into a physically-based renderer. Those issues will all be handled in the `BSDF`, simplifying other parts of the system.

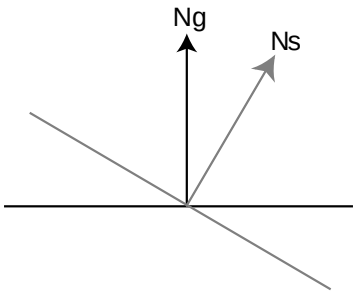


Figure 10.1: The geometric normal, N_g, defined by the surface geometry, and the shading normal, N_s, given by per-vertex normals and/or bump mapping will generally specify different hemispheres for integrating incident illumination to compute surface reflection. This inconsistency is important to handle carefully.

⟨*BSDF Declarations*⟩≡
```
class BSDF {
public:
    ⟨BSDF Method Declarations⟩
private:
    ⟨BSDF Member Variables⟩
};
```

DifferentialGeometry 47

The BSDF constructor takes two pieces of DifferentialGeometry: dgS, is the *shading* differential geometry, where the normal, *S*, and *T* vectors may have been modified from the true geometric normal and tangent vectors of the original surface and dgG, which represents the true geometric characteristics at the point being shaded–see Figure 10.1. Throughout this section, we will use the convention where N_s is the shading normal and N_g is the geometric normal.

⟨*BSDF Method Definitions*⟩≡
```
BSDF::BSDF(const DifferentialGeometry &dgS,
        const DifferentialGeometry &dgG) {
    Ng = dgG.Nn;
    ⟨Orient shading normal to match geometric normal⟩
}
```

The constructor stores the geometric normal as given. It then flips the shading coordinate frame if needed, so that the shading normal lies in the hemisphere around geometric normal–the assumption is that the shading normal represents a relatively small perturbation of the geometric normal, so should be in the same hemisphere.

⟨*Orient shading normal to match geometric normal*⟩≡
```
if (Dot(Ng, dgS.Nn) < 0) {
    Ns = -dgS.Nn;
    Ss = -dgS.S;
    Ts = -dgS.T;
}
else {
    Ns = dgS.Nn;
    Ss = dgS.S;
    Ts = dgS.T;
}
```

⟨*BSDF Member Variables*⟩≡
```
Normal Ns, Ng;
Vector Ss, Ts;
```

BRDFs and BTDFs are stored with associated weight values, provided by the caller when they are added to the BSDF.

⟨*BSDF Inline Methods*⟩≡
```
inline void BSDF::Add(BRDF *b, Float w) {
    brdfs.push_back(b);
    rWeights.push_back(w);
}
```

266 BRDF
298 BSDF
266 BTDF
23 Normal
494 push_back
494 size
16 Vector

⟨*BSDF Member Variables*⟩+≡
```
vector<BRDF *> brdfs;
vector<BTDF *> btdfs;
vector<Float> rWeights, tWeights;
```

⟨*BSDF Inline Methods*⟩+≡
```
inline void BSDF::Add(BTDF *b, Float w) {
    btdfs.push_back(b);
    tWeights.push_back(w);
}
```

⟨*BSDF Method Declarations*⟩+≡
```
int NumComponents() const {
    return (int) (brdfs.size() + btdfs.size());
}
```

Because a BSDF can contain more than one specular component (glass, for instance is a specular reflector *and* transmitter, we need to be able to compute the number of specular components in any given BSDF.

⟨*BSDF Inline Methods*⟩+≡
```
inline int BSDF::NumSpecular() const {
    int n = 0;
    u_int i;
    for (i = 0; i < brdfs.size(); ++i)
        if (brdfs[i]->IsSpecular()) ++n;
    for (i = 0; i < btdfs.size(); ++i)
        if (btdfs[i]->IsSpecular()) ++n;
    return n;
}
```

We also provide a transformation to and from the local coordinate system expected by BxDFs (as described in Section 9.1). In this coordinate system, the surface normal is along $(0,0,1)$, the primary tangent is $(1,0,0)$ and the secondary tangent is $(0,1,0)$. This transformation into "shading space" simplified many of the BxDF equations in Chapter 9. These transformations are computed in the same way as the DifferentialGeometry methords for transforming to and from the differential geometry's frame; see Section 2.7 for more information.

The transformation to shading space normalizes the resulting vector, since many BxDF implementations depend on this. However, we don't normalize directions in world space, since there's not a corresponding assumption for world-space rays.

brdfs 299
BSDF 298
btdfs 299
BxDF::IsSpecular 267
Hat 19
size 494
Vector 16

⟨*BSDF Method Declarations*⟩+≡
```
Vector WorldToLocal(const Vector &v) const {
    return Vector(Dot(v, Ss), Dot(v, Ts), Dot(v, Ns)).Hat();
}
```

⟨*BSDF Method Declarations*⟩+≡
```
Vector LocalToWorld(const Vector &v) const {
    return Vector(Ss.x * v.x + Ts.x * v.y + Ns.x * v.z,
                  Ss.y * v.x + Ts.y * v.y + Ns.y * v.z,
                  Ss.z * v.x + Ts.z * v.y + Ns.z * v.z);
}
```

Shading normals can cause a variety of undesirable artifacts in practice–see Figure 10.2. On the left is a *light leak*: the geometric normal indicates that $\vec{\omega}_i$ and $\vec{\omega}_o$ lie on opposide sides of the surface, so if the surface is not transmissive, the light should have no contribution. However, if we directly evaluate the scattering equation 5.4.8 about the hemisphere centered around the shading normal, we will incorrectly incorporate the light from $\vec{\omega}_i$. Thus, we can see that N_s can't just be used as a direct replacement for N_g in rendering computations.

The right side of Figure 10.2 shows a similar situation: the shading normal indicates that no light should be reflected to the viewer, since it is not in the same hemisphere as the illumination, while the geometric normal incidates that they are in the same hemisphere. Direct use of N_s would cause ugly black spots on the surface where this situation happens.

Fortinately, there is an elegant solution to these problems. When evaluating the BSDF, we use the geometric normal to decide if we should be evaluating reflection or transmission: if $\vec{\omega}_i$ and $\vec{\omega}_o$ lie in the same hemisphere with respect to N_g, we evaluate the BRDFs, and otherwise we evaluate the BTDFs.

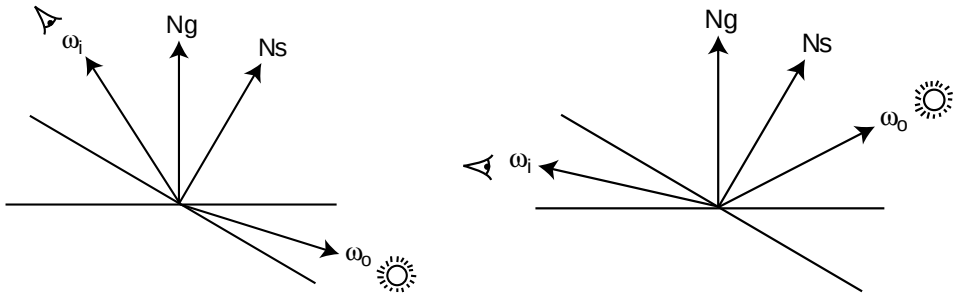


Figure 10.2: The two types of error that result from using shading normals: on the left, a light leak, where the geometric normal indicates that the light is on the back-side of the surface, but the shading normal indicates the light is visible (assuming a reflective and not transmissive surface.) On the right is a dark spot, where the geometric normal indicates that the surface is illuminated but the shading normal indicates that the viewer is behind the lit side of the surface.

Given that convention, recall from Section 9.1 that BxDFs in lrt should evaluate their values without regard to whether $\vec{\omega}_i$ and $\vec{\omega}_o$ are in the same or are in different hemispheres. Thus, light leaks are avoided, since we only evaluate the BTDFs for the situation in the left side of Figure 10.2, giving us no reflection for a purely reflective surface. Similarly, black spots are avoided since we would evaluate the BRDFs for the situation on the right side of the figure, even though the shading normal thinks that the directions are in different hemispheres. Because the BRDFs evaluate their values in this case, we get a reasonable result.

299 brdfs
298 BSDF
300 BSDF::WorldToLocal
299 btdfs
267 BxDF::f
299 rWeights
494 size
155 Spectrum
299 tWeights
16 Vector

Given all that, evaluating the BSDF is easy. We just transform the world-space direction vectors to local BSDF space, determine whether we should be using the BRDFs or the BTDFs, and loop over the appropriate set, evaluating a weighted sum of their contributions.

⟨*BSDF Inline Methods*⟩+≡

```
inline Spectrum BSDF::f(const Vector &woW, const Vector &wiW) const {
    Vector wi = WorldToLocal(wiW), wo = WorldToLocal(woW);
    Spectrum f = 0.;
    if (Dot(wiW, Ng) * Dot(woW, Ng) > 0)
        for (u_int i = 0; i < brdfs.size(); ++i)
            f += rWeights[i] * brdfs[i]->f(wo, wi);
    else
        for (u_int i = 0; i < btdfs.size(); ++i)
            f += tWeights[i] * btdfs[i]->f(wo, wi);
    return f;
}
```

The f_delta function of a BSDF is slightly different from the BxDF. It takes an additional argument, specifying which component to query. Typically the caller of this function will determine the number of specular components and loop over them, repeatedly calling this function.

⟨*BSDF Inline Methods*⟩+≡
```
inline Spectrum BSDF::f_delta(int component, const Vector &w,
        Vector *wi) const {
    BxDF *spec = NULL;
    Float weight = 0.;
    ⟨Find the componentth specular component⟩
    Vector wo = WorldToLocal(w);
    Spectrum f = weight * spec->f_delta(wo, wi);
    *wi = LocalToWorld(*wi);
    return f;
}
```

⟨*Find the* componentth *specular component*⟩≡
```
for (u_int i = 0; i < brdfs.size(); ++i) {
    if (brdfs[i]->IsSpecular()) {
        if (component-- == 0) {
            spec = brdfs[i];
            weight = rWeights[i];
            break;
        }
    }
}
if (spec == NULL) {
    ⟨Look for specular reflection in BTDFs⟩
}
if (spec == NULL) return 0.;
```

We'll also provide BSDF methods that sum up the reflectance values of their individual BxDFs; the implementation of these methods is straightforward and won't be shown here.

⟨*BSDF Method Declarations*⟩+≡
```
Spectrum rho() const;
Spectrum rho(const Vector &wo) const;
```


brdfs 299
BSDF 298
BSDF::LocalToWorld 300
BSDF::WorldToLocal 300
BxDF 266
BxDF::f_delta 267
BxDF::IsSpecular 267
rWeights 299
size 494
Spectrum 155
Vector 16

10.2 Material Interface and Bump Mapping

⟨*materials.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef MATERIALS_H
#define MATERIALS_H
#include "lrt.h"
#include "primitives.h"
#include "texture.h"
#include "color.h"
#include "reflection.h"
⟨Material Class Declarations⟩
⟨Material creation macros⟩
#endif // MATERIALS_H
```

⟨*materials.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "materials.h"
#include "color.h"
#include "reflection.h"
#include "texture.h"
#include "shapes.h"
⟨Material Method Definitions⟩
```

⟨*Material Class Declarations*⟩≡
```
class Material : public ReferenceCounted<Material> {
public:
    ⟨Material Interface⟩
private:
    ⟨Material Private Data⟩
};
```

There are two main functions that `Material`s are responsible for implementing. The first is a pure virtual function that returns the `BSDF` for a point on a surface represented by a `Surf`. The material is responsible for synthesizing relevant information about the texture and geometric surface properties at the point to generate the scattering function at the point.

⟨*Material Interface*⟩≡
```
virtual BSDF *GetBSDF(const Surf *surf) const = 0;
```

Since our usual interface to the hit point is through a `Surf`, we will also add a convenience method to `Surf` that returns the `BSDF` at the hit point. It just forwards the request on to the `Material`.

⟨*Surf Method Definitions*⟩≡
```
BSDF *Surf::GetBSDF(const RayDifferential &ray) const {
    ⟨Update statistics for number of points shaded⟩
    ⟨Compute filter region for anti-aliasing⟩
    primitive->Bump(dgGeom, &dgShading);
    return primitive->material->GetBSDF(this);
}
```

⟨*Update statistics for number of points shaded*⟩≡
```
static StatsCounter pointsShaded("Shading", "Number of points shaded");
++pointsShaded;
```

⟨*Compute filter region for anti-aliasing*⟩≡
```
dgGeom.ComputeDifferentials(ray);
```

Filter Regions for Anti-Aliasing

The various `Material`s below will evaluate texture functions (described in the next chapter) to compute spatially-varying reflectance properties of surfaces. These functions may have high frequency variation in screen space, so it is worth-while for them to try to remove frequencies beyond the Nyquist limit. The trick, then, is figuring out how the 2D screen sampling frequency translates to some frequency in the texture space.

298 BSDF
309 Bump
306 ComputeDifferentials
10 dgGeom
10 dgShading
579 primitives
26 RayDifferential
508 ReferenceCounted
501 StatsCounter
10 Surf

Figure 10.3: two rays on an object gives the local sampling frequency...

Because the bump-mapping code in the next section also needs to be able to reason about sampling frequency, we'll do the computations here...

This is a difficult problem to solve correctly in all cases; some assumptions and approximations need to be made in order to make the problem tractible. However, it's far better to make these assumptionsand do some form of anti-aliasing rhan to give up and just increase the image sampling rate to reduce aliasing. Tracing more camera rays is an expensive computational cost to bear.

Figure 10.3 shows the basic setting–the density of rays on the image plane implicitly determines a sampling rate at points in the scene. Given two adjacent rays from the camera, here we are computing shaded values on the object at two nearby points. If the variation of the texture function on the object has a higher frequency content than the point sampling rate can capture, the final image will have aliasing.

However, if the texture function is aware of the local sampling frequency of rays intersecting the object, it can try to remove higher-frequency variations in its value. The key to tracking this information is the `RayDifferential` structure, which was defined in Section 2.4 and is initialized in the `Scene::Render()` function in Section 1.5. In addition toe the ray actually being traced through the scene, it records two offset rays, one offset horizontally one pixel from the camera ray and the other offset vertically by one pixel.

All of the ray intersection routines only use the main camera ray for their computations; the auxiliary rays are ignored. Once we've found an intersection and are evaluating textures, however, we use the auxiliary rays to estimate local sampling frequency (note ignoring higher pixel sampling rate–oh, well...)

The key to this estimate is that we make the approximation that the surface is locally flat with respect to the sampling frequency at the point being shaded. This is a reasonably approximation to make in practice. Furthermore, it is hard to do much better, since ray-tracing is by nature a point-sampling method–we have no additional information about the scene in between the rays we traced, anyway.

Given this approximation, we compute the plane through the point intersected by the main ray and tangent to the surface there. This plane is given by the implicit plane equation

$$ax + by + cz + d = 0,$$

where $a = N_x$, $b = N_y$, $c = N_z$, and $d = -(N \cdot P)$.

Figure 10.4: rays intersecting the tangent plane lets us approximate the relevant variations...

Next, we intersect the auxiliary rays r_x and r_y with this plane (Figure 10.4). Given their hit positions, we would like the find the amount of variation in position on the surface and variation in parametric (u, v) coordinates between adjacent camera ray smaples; these give us the sampling rate in texture parameter space, which individual textures can use to determine their maximum allowed frequency conetnt.

306 ComputeDifferentials
47 DifferentialGeometry
21 Point
26 RayDifferential
16 Vector

⟨*DifferentialGeometry Data*⟩+≡
```
mutable Vector dPdx, dPdy;
mutable Float dudx, dvdx, dudy, dvdy;
```

⟨*Initialize* DifferentialGeometry *from parameters*⟩+≡
```
dudx = dvdx = dudy = dvdy = 0;
```

⟨*DifferentialGeometry Method Declarations*⟩+≡
```
DifferentialGeometry(const Point &p, const Vector &dpdu,
        const Vector &dpdv, const Vector &dndu,
        const Vector &dndv, Float uu, Float vv,
        const Shape *sh, Float dux, Float dvx,
        Float duy, Float dvy)
    : P(p), dPdu(dpdu), dPdv(dpdv), dNdu(dndu), dNdv(dndv) {
    ⟨Initialize DifferentialGeometry from parameters⟩
    dudx = dux;
    dvdx = dvx;
    dudy = duy;
    dvdy = dvy;
}
```

⟨*DifferentialGeometry Method Declarations*⟩+≡
```
void ComputeDifferentials(const RayDifferential &r) const;
```

⟨*DifferentialGeometry Method Definitions*⟩≡
```
void DifferentialGeometry::ComputeDifferentials(const RayDifferential &ray) con
    if (ray.hasDifferentials) {
        ⟨Estimate screen-space change in P and (u,v)⟩
    }
    else {
        dudx = dvdx = 0.;
        dudy = dvdy = 0.;
        dPdx = dPdy = Vector(0,0,0);
    }
}
```

⟨*Estimate screen-space change in P and* (u,v)⟩≡
```
⟨Compute auxiliary intersection points with plane⟩
dPdx = Px - P;
dPdy = Py - P;
⟨Compute (u,v) offsets at auxiliary points⟩
```

Given their hit positions, we approximate the positions P_x and P_y on the surface with the intersection locations on the tangent plane.

Ray-plane intersection: if origin is P and direction is D, then:

$$t = \frac{-((a,b,c)\cdot P) + d}{(a,b,c)\cdot D}$$

Don't compute plane a, b, and c, since they're just in `dgGeom.Nn`.

⟨*Compute auxiliary intersection points with plane*⟩≡
```
Float D = -Dot(Nn, Vector(P.x, P.y, P.z));
Float tx = -(Dot(Nn, Vector(ray.rx.O.x, ray.rx.O.y, ray.rx.O.z)) + D) /
    Dot(Nn, ray.rx.D);
Point Px = ray.rx.O + tx * ray.rx.D;
Float ty = -(Dot(Nn, Vector(ray.ry.O.x, ray.ry.O.y, ray.ry.O.z)) + D) /
    Dot(Nn, ray.ry.D);
Point Py = ray.ry.O + ty * ray.ry.D;
```

compute their parametric (u,v) coordinates by taking advantage of the face thatthe surface's $\partial P/\partial u$ and $\partial P/\partial v$ form a (not-necessarily orthogonal) coordinate system on the plane and that the coordinates of the auxiliary intersection points in terms of this coordinate system are their coordinates with respect to the (u,v) parameterization. Given a position P' on the plane, we can compute its position with respect to the coordinate system by

XXX

$$(P' - P) = (\partial P/\partial u \partial P/\partial v)\begin{pmatrix} du \\ dv \end{pmatrix}$$

or

$$\begin{pmatrix} \mathbf{P}' - \mathbf{P}_x \\ \mathbf{P}' - \mathbf{P}_y \\ \mathbf{P}' - \mathbf{P}_z \end{pmatrix} = \begin{pmatrix} \partial\mathbf{P}/\partial\mathbf{u}_x & \partial\mathbf{P}/\partial\mathbf{v}_x \\ \partial\mathbf{P}/\partial\mathbf{u}_y & \partial\mathbf{P}/\partial\mathbf{v}_y \\ \partial\mathbf{P}/\partial\mathbf{u}_z & \partial\mathbf{P}/\partial\mathbf{v}_z \end{pmatrix}\begin{pmatrix} du \\ dv \end{pmatrix}$$

This is a linear system in three equations of two unknowns–i.e. it's over-constrained. However, we need to be careful since one of the equations may be degenerate–e.g.

DifferentialGeometry 47
Point 21
RayDifferential 26
Vector 16

if $\partial P/\partial u$ and $\partial P/\partial v$ are in the xy plane such that their z components are both zero, then the third equation will be degenerate. To deal with this, since we only need two equations to solve the system, we'd like to choose two that won't have degeneracies. Easy way to do this is to take the cross product of $\partial P/\partial u$ and $\partial P/\partial v$ and see which coordinate of the result has the largest magnitude; throw away that coordinate and use the other two. But that cross product is already available in `Nn`...

⟨*Compute* (u,v) *offsets at auxiliary points*⟩≡
```
⟨Initialize A, Bx, and By matrices for offset computation⟩
SolveLinearSystem2x2(A, Bx, x);
dudx = x[0];
dvdx = x[1];
SolveLinearSystem2x2(A, By, x);
dudy = x[0];
dvdy = x[1];
```

⟨*Initialize* `A`, `Bx`, *and* `By` *matrices for offset computation*⟩≡
```
Float A[2][2], Bx[2], By[2], x[2];
if (fabsf(Nn.x) > fabsf(Nn.y) &&
    fabsf(Nn.x) > fabsf(Nn.z)) {
    ⟨Project onto yz plane to initialize matrices⟩
}
else if (fabsf(Nn.y) > fabsf(Nn.z)) {
    ⟨Project onto xz plane to initialize matrices⟩
}
else {
    ⟨Project onto xy plane to initialize matrices⟩
}
```

⟨*Project onto yz plane to initialize matrices*⟩≡
```
#define C1 y
#define C2 z
⟨Initialize matrices for chosen projection plane⟩
#undef C1
#undef C2
```

⟨*Initialize matrices for chosen projection plane*⟩≡
```
A[0][0] = dPdu.C1;
A[0][1] = dPdv.C1;
A[1][0] = dPdu.C2;
A[1][1] = dPdv.C2;
Bx[0] = Px.C1 - P.C1;
Bx[1] = Px.C2 - P.C2;
By[0] = Py.C1 - P.C1;
By[1] = Py.C2 - P.C2;
```

Bump mapping

All materials take an optional `Float` texture map that defines a displacement at each point on the surface: each point x has a displaced point x' associated with

510 SolveLinearSystem2x2

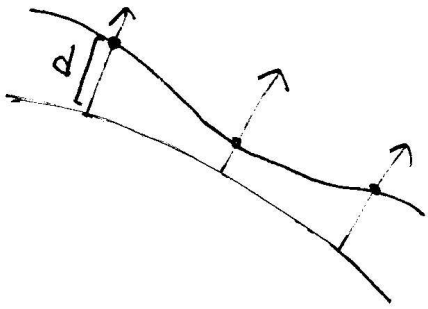


Figure 10.5: The displacement texture associated with each material defines a new surface based on the old one, offset by the displacement amount along the normal at each point. `lrt` doesn't compute a geometric representation of this displaced surface, though it does use it to compute shading normals for bump-mapping.

it, defined by $x' = x + dN(x)$, where d is the offset returned by the displacement texture at x and $N(x)$ is the surface normal at x–see Figure 10.5. We will use this texture to compute bump-mapped shading normals below, though it could also be used in an implementation of displacement mapping.

Material 303
Texture 323

⟨*Material Interface*⟩+≡
```
Material(Texture<Float> *disp) {
    displace = disp;
}
```

⟨*Material Private Data*⟩≡
```
Texture<Float> *displace;
```

The second important `Material` method, `bump`, is responsible for computing the effect of bump mapping at the point being shaded. Two instances of `DifferentialGeometry` are stored in `Surfs`; the first, `dgGeom`, represents the *geometric* differential geometry at the hit point–the true geometry of the intersection. The second, `dgShading`, represents the *shading* geometry; by default, it is the same as `dgGeom`, but the `Material` also may perturb its normal or tangent vectors in order to simulate the effect of rough surfaces or modify the mapping for anisotropy, respectively.

To compute a shading normal at a point, we will evaluate the displacement texture at two auxiliary points next to the current point x: see Figure 10.6. We move a distance du along the $\partial P/\partial u$ vector and dv along the $\partial P/\partial v$ vector to the two auxiliary points x_u and x_v. By evaluating the displacement at these three points, we compute three points on the displaced surface, x', x'_u and x'_v. The cross product of new tangent vectors $\vec{\mathbf{v}}_\mathbf{u} = x'_u - x$ and $\vec{\mathbf{v}}_\mathbf{v} = x'_v - x$ gives the shading normal N_s.

This approach is based on the assumption that the surface is locally flat around x: if it has a large curvature, then $x + \partial P/\partial u * du$ may be far from the actual surface. However, as long as du and dv are chosen so that they move a relatively small distance about x, this isn't a problem in practice.

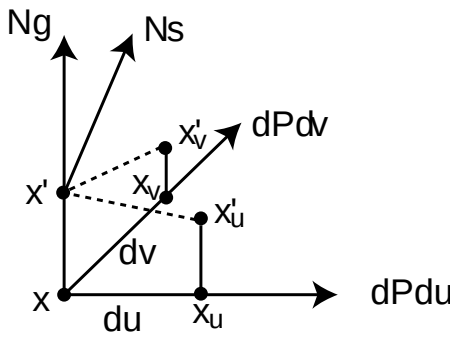


Figure 10.6: To compute the shading normal at a point, we evaluate the displacement texture at that point and at two auxiliary points. By taking the cross product of the vectors from the main point to the auxiliary point, we find the shading normal. The auxiliary points are found by offsetting by the parametric distances *du* and *dv* along the $\partial P/\partial u$ and $\partial P/\partial v$ vectors.

⟨*Material Method Definitions*⟩+≡
```
void Material::Bump(const DifferentialGeometry &dgg,
        Texture<Normal> *Ns, Texture<Vector> *Ss,
        DifferentialGeometry *dgs) const {
    if (!displace && !Ns && !Ss) {
        *dgs = dgg;
        return;
    }
    ⟨Evaluate texture for shading normal and tangent⟩
    *dgs = DifferentialGeometry(dgg.P, Svert, Tvert,
        Vector(0,0,0), Vector(0,0,0), dgg.u, dgg.v,
        dgg.shape, dgg.dudx, dgg.dvdx, dgg.dudy,
        dgg.dvdy);
    if (displace) {
        ⟨Evaluate displacment and compute bumped values⟩
    }
}
```

20 Cross
47 DifferentialGeometry
19 Hat
303 Material
308 Material::displace
23 Normal
323 Texture
16 Vector

Will expect these guys to be in world space already...

⟨*Evaluate texture for shading normal and tangent*⟩≡
```
Normal Nvert = Ns ? Ns->Evaluate(dgg).Hat() : dgg.Nn;
Vector Svert, Tvert;
Svert = Ss ? Ss->Evaluate(dgg).Hat() : dgg.S;
if (Ns || Ss) {
    Tvert = Cross(Svert, Nvert);
    Svert = Cross(Tvert, Nvert);
}
else
    Tvert = dgg.T;
```

⟨*Evaluate displacment and compute bumped values*⟩≡
⟨*Compute offset positions and evaluate displacement texture*⟩
⟨*Return bump-mapped differential geometry*⟩

Given the offset distances, we use the `DifferentialGeometry::Shift()` method to compute the differential geometry at the auxiliary points. We can then evaluate the displacement texture at the three points and compute the three displaced positions.

⟨*Compute offset positions and evaluate displacement texture*⟩≡
```
DifferentialGeometry dgdx, dgdy;
dgs->ShiftX(&dgdx);
dgs->ShiftY(&dgdy);
Point P = dgs->P + Vector(dgs->Nn) *
    displace->Evaluate(*dgs);
Point Px = dgdx.P + Vector(dgdx.Nn) * displace->Evaluate(dgdx);
Point Py = dgdy.P + Vector(dgdy.Nn) * displace->Evaluate(dgdy);
```

The `Shift*()` methods use the local-flatness assumption mentioned above. New (u,v) coordinates are easily computed based on the offset the caller provided. We assume that the partial derivatives, tangents, and surface normal are the same at the shifted point due to the flatness assumption. The new point P, then, is just computed by moving the appropriate distances along $\partial P/\partial u$ and $\partial P/\partial v$.

Cross 20
DifferentialGeometry 47
Hat 19
Material::displace 308
Normal 23
Point 21
Texture::evaluate 323
Vector 16

⟨*DifferentialGeometry Method Declarations*⟩+≡
```
void ShiftX(DifferentialGeometry *g) const {
    *g = *this;
    g->u += dudx;
    g->v += dvdx;
    g->P += dudx * dPdu + dvdx * dPdv;
    g->Nb += Normal(dudx * dNdu + dvdx * dNdv);
    g->Nn = g->Nb.Hat();
    g->T = Cross(g->S, g->Nb);
    g->S = Cross(g->T, g->Nb);
}
```

Given the new positions, we compute partial derivatives with forward differences. This is all that we need to do here; the `DifferentialGeometry` constructor takes care of computing the resulting normal, etc.

XXX Actually this is wasteful; mostly just need to recalculate N values, etc...

⟨*Return bump-mapped differential geometry*⟩≡
```
Float dx = sqrtf(dgs->dudx*dgs->dudx + dgs->dvdx*dgs->dvdx);
Float dy = sqrtf(dgs->dudy*dgs->dudy + dgs->dvdy*dgs->dvdy);
*dgs = DifferentialGeometry(dgs->P, (P-Px) / dx, (P-Py) / dy,
    Vector(0,0,0), Vector(0,0,0), dgs->u, dgs->v, dgs->shape);
```

10.3 Matte

⟨*matte.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "materials.h"
⟨Matte Class Declarations⟩
⟨Matte Method Definitions⟩
```

⟨*Matte Class Declarations*⟩≡
```
class Matte : public Material {
public:
    ⟨Matte Interface⟩
private:
    ⟨Matte Private Data⟩
};
```

The simplest surface is `Matte`. It describes a diffusely-reflecting surface. A `Matte::Kd` texture parameter gives the overall reflectivity of the surface at each point.

⟨*Matte Interface*⟩≡
```
Matte(Texture<Spectrum> *kd, Texture<Float> *sig,
        Texture<Float> *disp)
    : Material(disp) {
    Kd = kd;
    sigma = sig;
}
```

303 Material
155 Spectrum
323 Texture

⟨*Matte Private Data*⟩≡
```
Texture<Spectrum> *Kd;
Texture<Float> *sigma;
```

We need to destroy the `Texture` when the material is deleted. For brevity, we won't include the destructors for the rest of the materials in this chapter.

⟨*Matte Method Definitions*⟩≡
```
Matte::~Matte() {
    delete Kd;
    delete sigma;
}
```

The `BSDF` method just puts the pieces together. The `Matte::Kd` texture is evaluated to compute the `Kd` color at the point being shaded. This is then passed on to create a `Lambertian BxDF`, which is returned inside a `BSDF` object.

⟨*Matte Method Definitions*⟩+≡
```
BSDF *Matte::GetBSDF(const Surf *surf) const {
    Spectrum r = Kd->Evaluate(surf->dgShading);
    Float sig = sigma->Evaluate(surf->dgShading);
    BSDF *ret = new BSDF(surf->dgShading, surf->dgGeom);
    if (sig == 0.)
        ret->Add(new Lambertian(r));
    else
        ret->Add(new OrenNayar(r, sig));
    return ret;
}
```

10.4 Plastic

⟨*plastic.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "materials.h"
⟨Plastic Class Declarations⟩
⟨Plastic Method Definitions⟩
```


BSDF 298
dgGeom 10
dgShading 10
Lambertian 281
Material 303
Matte 311
Matte::Kd 311
Matte::sigma 311
Spectrum 155
Surf 10
Texture 323

⟨*Plastic Class Declarations*⟩≡
```
class Plastic : public Material {
public:
    ⟨Plastic Interface⟩
private:
    ⟨Plastic Private Data⟩
};
```

A more interesting surface is plastic. Plastic can be modelled as a mixture of a diffuse and glossy scattering function, with appropriate parameters controlling the particular colors and glossiness. The parameters to `Plastic` are two reflectivities, `Kd` and `Ks`, which control how much diffuse reflection there is and how much glossy specular reflection there is. Next is a roughness parameter (which should range from zero to one) that determines the size of the specular highlight; the higher it is, the rougher the surface and the smaller the highlight.

⟨*Plastic Interface*⟩+≡
```
Plastic(Texture<Spectrum> *kd, Texture<Spectrum> *ks,
        Texture<Float> *rough, Texture<Float> *disp)
        : Material(disp) {
    Kd = kd;
    Ks = ks;
    roughness = rough;
}
```

⟨*Plastic Private Data*⟩≡
```
Texture<Spectrum> *Kd, *Ks;
Texture<Float> *roughness;
```

⟨*Plastic Method Definitions*⟩+≡
```
BSDF *Plastic::GetBSDF(const Surf *surf) const {
    Spectrum kd = Kd->Evaluate(surf->dgShading);
    BRDF *diff = new Lambertian(kd);
    Fresnel *fresnel = new FresnelDielectric(1.5f, 1.f);
    Spectrum ks = Ks->Evaluate(surf->dgShading);
    Float rough = roughness->Evaluate(surf->dgShading);
    BRDF *spec = new Microfacet(ks, fresnel, new Blinn(1.f / rough));
    BSDF *ret = new BSDF(surf->dgShading, surf->dgGeom);
    ret->Add(diff);
    ret->Add(spec);
    return ret;
}
```

10.5 Translucent

⟨*translucent.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "materials.h"
⟨Translucent Class Declarations⟩
⟨Translucent Method Definitions⟩
```

⟨*Translucent Class Declarations*⟩≡
```
class Translucent : public Material {
public:
    ⟨Translucent Interface⟩
private:
    ⟨Translucent Private Data⟩
};
```


289 Blinn
266 BRDF
298 BSDF
10 dgGeom
10 dgShading
272 Fresnel
281 Lambertian
303 Material
287 Microfacet
312 Plastic
312 Plastic::Kd
312 Plastic::Ks
312 Plastic::roughness
155 Spectrum
10 Surf

⟨*Translucent Method Definitions*⟩+≡
```
BSDF *Translucent::GetBSDF(const Surf *surf) const {
    BSDF *ret = new BSDF(surf->dgShading, surf->dgGeom);

    Float r = reflect->Evaluate(surf->dgShading);
    Float t = transmit->Evaluate(surf->dgShading);
    if (r == 0. && t == 0.) return ret;

    Spectrum kd = Kd->Evaluate(surf->dgShading);
    if (!kd.Black()) {
        if (r > 0.) ret->Add(new Lambertian(r * kd));
        if (t > 0.) ret->Add(new BRDFToBTDF(new Lambertian(t * kd)));
    }
    Spectrum ks = Ks->Evaluate(surf->dgShading);
    if (!ks.Black()) {
        Float rough = roughness->Evaluate(surf->dgShading);
        if (r > 0.) {
            Fresnel *fresnel = new FresnelDielectric(1.5f, 1.f);
            ret->Add(new Microfacet(r * ks, fresnel,
                new Blinn(1.f / rough)));
        }
        if (t > 0.) {
            Fresnel *fresnel = new FresnelDielectric(1.5f, 1.f);
            ret->Add(new BRDFToBTDF(new Microfacet(t * ks, fresnel,
                new Blinn(1.f / rough))));
        }
    }
    return ret;
}
```


Blinn 289
BRDFToBTDF 268
BSDF 298
dgGeom 10
dgShading 10
Fresnel 272
Lambertian 281
Material 303
Microfacet 287
Spectrum 155
Surf 10
Translucent 313

10.6 Glass

⟨*glass.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "materials.h"
⟨Glass Class Declarations⟩
⟨Glass Method Definitions⟩
```

⟨*Glass Class Declarations*⟩≡
```
class Glass : public Material {
public:
    ⟨Glass Interface⟩
private:
    ⟨Glass Private Data⟩
};
```

Another surface shader simulates glass (poorly, since Fresnel effects aren't yet included.) Nevertheless, a combination of specular reflection and refraction brings

us to the heart of recursive ray-tracing and can lead to some nifty images. Our parameters include reflection and transmission coefficients as well as the index of refraction of the object.

⟨*Glass Interface*⟩+≡
```
Glass(Texture<Spectrum> *r, Texture<Spectrum> *t,
        Texture<Float> *i, Texture<Float> *disp)
        : Material(disp) {
    Kr = r;
    Kt = t;
    index = i;
}
```

⟨*Glass Private Data*⟩≡
```
Texture<Spectrum> *Kr, *Kt;
Texture<Float> *index;
```

As usual, we start by computing new parameters from the primitive's user-supplied values. We then generate a new `BSDF` that holds reflective and transmissive `BRDFs` as appropriate given the parameter values.

⟨*Glass Method Definitions*⟩+≡
```
BSDF *Glass::GetBSDF(const Surf *surf) const {
    Spectrum R = Kr->Evaluate(surf->dgShading);
    Spectrum T = Kt->Evaluate(surf->dgShading);
    Float ior = index->Evaluate(surf->dgShading);
    BSDF *ret = new BSDF(surf->dgShading, surf->dgGeom);
    if (!R.Black())
        ret->Add(new SpecularReflection(R,
            new FresnelDielectric(1., ior)));
    if (!T.Black())
        ret->Add(new SpecularTransmission(T, 1., ior));
    return ret;
}
```


298 BSDF
10 dgGeom
10 dgShading
314 Glass
303 Material
155 Spectrum
275 SpecularReflection
278 SpecularTransmission
10 Surf
323 Texture

10.7 Shiny Metal

⟨*shinymetal.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "materials.h"
⟨ShinyMetal Class Declarations⟩
⟨ShinyMetal Method Definitions⟩
```

⟨*ShinyMetal Class Declarations*⟩≡
```
class ShinyMetal : public Material {
public:
    ⟨ShinyMetal Interface⟩
private:
    ⟨ShinyMetal Private Data⟩
};
```

Another basic combination of scattering functions gives us something that looks like a shiny metal surface. We have both a glossy specular reflection, with reflectance Ks, and perfect mirror specular reflection, with reflectance Kr.

⟨*ShinyMetal Interface*⟩+≡
```
ShinyMetal(Texture<Spectrum> *ks, Texture<Float> *rough,
        Texture<Spectrum> *kr, Texture<Float> *disp)
        : Material(disp) {
    Ks = ks;
    roughness = rough;
    Kr = kr;
}
```

⟨*ShinyMetal Private Data*⟩≡
```
Texture<Spectrum> *Ks, *Kr;
Texture<Float> *roughness;
```

⟨*ShinyMetal Method Definitions*⟩+≡
```
BSDF *ShinyMetal::GetBSDF(const Surf *surf) const {
    Spectrum spec = Ks->Evaluate(surf->dgShading);
    Float rough = roughness->Evaluate(surf->dgShading);
    Spectrum R = Kr->Evaluate(surf->dgShading);

    MicrofacetDistribution *md = new Blinn(1.f / rough);
    Spectrum k = 0.;
    Fresnel *frMf = new FresnelConductor(FresnelApproxEta(spec), k);
    Fresnel *frSr = new FresnelConductor(FresnelApproxEta(R), k);
    BSDF *ret = new BSDF(surf->dgShading, surf->dgGeom);
    ret->Add(new Microfacet(1., frMf, md));
    ret->Add(new SpecularReflection(1., frSr));
    return ret;
}
```


Blinn 289
BSDF 298
dgGeom 10
dgShading 10
Fresnel 272
FresnelApproxEta 272
Material 303
Microfacet 287
MicrofacetDistribution 287
Spectrum 155
SpecularReflection 275
Surf 10
Texture 323

10.8 Diffuse Substrate

XXX need a better name

⟨*substrate.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "materials.h"
⟨Substrate Class Declarations⟩
⟨Substrate Method Definitions⟩
```

⟨*Substrate Class Declarations*⟩≡
```
class Substrate : public Material {
public:
    ⟨Substrate Interface⟩
private:
    ⟨Substrate Private Data⟩
};
```

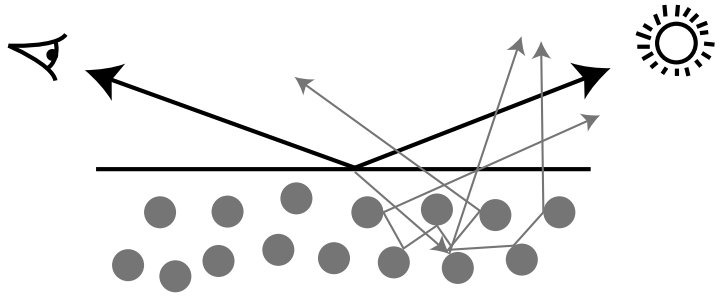


Figure 10.7:

A reasonably good model of glossy paint can be constructed using some of the pieces we have put together so far. There are two main types of light reflection with glossy paint: some of the incident light is specularly reflected at the surface, and the rest is transmitted into a substrate with suspended colored particles–see Figure 10.7. The transmitted light is interacts with the particles, and some wavelengths of light are absorbed, based on the particle color. The remaining light eventually exits.

If we make the assumption that the exiting light exits in random directions, reflection from the substrate can be modeled with a Lambertian BRDF. We will use the Fresnel formula for dielectrics to determine how much light is reflected and how much is transmitted, giving us weighting terms for the specular reflection and the body reflection BRDFs.

⟨*Substrate Interface*⟩≡
```
Substrate(Texture<Spectrum> *kd, Texture<Spectrum> *ks,
          Texture<Float> *u, Texture<Float> *v, Texture<Float> *disp)
    : Material(disp) {
    Kd = kd;
    Ks = ks;
    nu = u;
    nv = v;
}
```

⟨*Substrate Private Data*⟩≡
```
Texture<Spectrum> *Kd, *Ks;
Texture<Float> *nu, *nv;
```

⟨*Substrate Method Definitions*⟩+≡
```
BSDF *Substrate::GetBSDF(const Surf *surf) const {
    Spectrum d = Kd->Evaluate(surf->dgShading);
    Spectrum s = Ks->Evaluate(surf->dgShading);
    Float u = nu->Evaluate(surf->dgShading);
    Float v = nv->Evaluate(surf->dgShading);

    BSDF *ret = new BSDF(surf->dgShading, surf->dgGeom);
    ret->Add(new FresnelBlend(d, s, new Anisotropic(1.f/u, 1.f/v)));
    return ret;
}
```

291 Anisotropic
298 BSDF
10 dgGeom
10 dgShading
294 FresnelBlend
303 Material
155 Spectrum
10 Surf
323 Texture

10.9 Measured Data

⟨*clay.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "materials.h"
⟨Clay Class Declarations⟩
⟨Clay Method Definitions⟩
```

Cornell Program of Computer Graphics...

⟨*Clay Class Declarations*⟩≡
```
class Clay : public Material {
public:
    Clay(Texture<Float> *disp) : Material(disp) { }
    BSDF *GetBSDF(const Surf *surf) const;
};
```

⟨*Clay Method Definitions*⟩≡
```
BSDF *Clay::GetBSDF(const Surf *surf) const {
    ⟨Declare clay coefficients⟩
    BSDF *ret = new BSDF(surf->dgShading, surf->dgGeom);
    ret->Add(new Lafortune(Spectrum(diffuse), 3, xy, xy, z, e));
    return ret;
}
```

BSDF 298
dgGeom 10
dgShading 10
Lafortune 292
Material 303
Spectrum 155
Surf 10
Texture 323

⟨*Declare clay coefficients*⟩≡
```
static Float diffuse[3] = {   0.383626f,   0.260749f,   0.274207f };
static Float xy0[3] =     {  -1.089701f,  -1.102701f,  -1.107603f };
static Float z0[3] =      {  -1.354682f,  -2.714801f,  -1.569866f };
static Float e0[3] =      {  17.968505f,  11.024489f,  21.270282f };
static Float xy1[3] =     {  -0.733381f,  -0.793320f,  -0.848206f };
static Float z1[3] =      {   0.676108f,   0.679314f,   0.726031f };
static Float e1[3] =      {   8.219745f,   9.055139f,  11.261951f };
static Float xy2[3] =     {  -1.010548f,  -1.012378f,  -1.011263f };
static Float z2[3] =      {   0.910783f,   0.885239f,   0.892451f };
static Float e2[3] =      { 152.912795f, 141.937171f, 201.046802f };
static Spectrum xy[3] = { Spectrum(xy0), Spectrum(xy1), Spectrum(xy2) };
static Spectrum z[3] = { Spectrum(z0), Spectrum(z1), Spectrum(z2) };
static Spectrum e[3] = { Spectrum(e0), Spectrum(e1), Spectrum(e2) };
```

Will ifdraft felt, primer, skin...

Further Reading

Amanatides's cone tracing method (Ama84) and Heckbert and Hanrahan (HH84) were the first to extend ray tracing to incorporate an area associated with each image sample, rather than just an infinitessimal ray.

Ray differentials (Ige99). Extended by Suykens and Willems to handle glossy reflection as well (SW01). See also Turkowski's technical report (Tur93). Also Shinya et al (STN87), and Mitchell and Hanrahan (MH92). Gritz and Hahn (GH96), though theirs doesn't get good anisotropic filter regions and doesn't account for the variation in angle that a pixel area subtends as you go from the center to the edges of the image plane. Collins estimated ray footprint by keeping tree of all rays traced from a given eye ray, examining corresponding rays at the same level and position (Col94).

Phong and Crow first introduced the idea of interpolating per-vertex shading normals to give the appearence of smooth surfaces from polygonal meshes (PC75). Blinn later developed the bump-mapping technique to give the appearence of geometric complexity on coarse meshes (Bli78).

Snyder and Barr noted the light leak problem from per-vertex shading normals and proposed a number of work-arounds (SB87). The method we have used in this chapter is from Veach's thesis (Vea97, Section 5.3); it is a more robust solution than those of Snyder and Barr.

Kajiya generalized the idea of bump mapping the normal to *frame mapping* (Kaj85).

Shading normals introduce a number of subtle problems to physically-based light transport algorithms that we have not addressed here. For example, they can easily lead to surfaces that reflect more energy than was incident upon them, which can wreak havoc with light transport algorithms. Veach has investigated this issue in depth and proposed a number of solutions (Vea96).

Gondek et al investigated reflection from glossy painted surfaces (GMN94); some of the observations from their paper influenced the *ad-hoc* paint material model introduced here.

Lafortune coefficients from measurements taken for Marschner at al paper (MWL+99).

Exercises

10.1 texture plus specular reflection anti-aliasing by point sampling and averaging BSDF under filter kernel: no need for more ray tracing, just increases texturing and BSDF evaluation time.

10.2 Specular aliasing detection and elimination. Amanatides has discussed the issue (Ama92), though here we'll suggest that...

11. Texture

We will now describe a set of interfaces and classes that allow us to incorporate *texture* into our material models. All of the various materials that were described in Chapter 10 have a few parameters that describe their respective materials–e.g. what its diffuse reflectance is, how glossy a surface is, etc. Realistic materials are generally *spatially varying*–their properties vary over different positions on the surface. The texture code in this chapter computes values for these parameters as they vary over surfaces being shaded (e.g. varying colors in a wood grain pattern, etc.).

In graphics, the techniques used to compute these varying parameters fall under the area of *texturing*. In `lrt`, a texture is simply a function that evaluates to a floating point or spectral value. It may be a zero-dimensional function (e.g. it returns a constant); it may be a two-dimensional function of (u, v) surface parameter values; or it may be a three-dimensional function (e.g. of position in the scene). This chapter will include all three types of textures. Two-dimensional image maps are a well-known type of texturing–they are incorporated into our texturing framework in Section 11.6.

Texture functions may themselves be a source of high-frequency variation in the image function–see Figure 11.1, which shows an aliases image of a checkerboard on a plane. At the horizon, the number of checks inside a given pixel area is very large–the middle of theat figure shows a blow-up of one pixel's area at the horizon. Although this aliasing is reduced with the non-uniform sampling techniques from Chapter 7, a better solution is to implement texture functions that are aware of their frequency content and remove higher-frequency components. For many texture functions, doing so isn't too difficult and is substantially more efficient than increasing the image sampling rate by tracing more rays. The first section of this chapter will describe general approaches to texture anti-aliasing and the interface

Figure 11.1: texture aliasing example

that will be used in `lrt` throughout the rest of this chapter for implementing various anti-aliasing approaches.

⟨*texture.h**⟩≡
  ⟨*Source Code Copyright*⟩
  `#ifndef TEXTURE_H`
  `#define TEXTURE_H`
  `#include "lrt.h"`
  `#include "color.h"`
  `#include "geometry.h"`
  `#include "transform.h"`
  `#include "shapes.h"`
  `#include "primitives.h"`
  `#include "mipmap.h"`
  `#include "camera.h"`
  `#include "shapes/trianglemesh.h"`
  `#include <string>`
  `using std::string;`
  ⟨*Texture Class Declarations*⟩
  ⟨*Texture Template Method Definitions*⟩
  `#endif // TEXTURE_H`

primitives 579

⟨*texture.cc**⟩≡
  ⟨*Source Code Copyright*⟩
  `#include "texture.h"`
  ⟨*Texture Forward Declarations*⟩
  ⟨*Texture Cache Data*⟩
  ⟨*Texture Cache Methods*⟩
  ⟨*Perlin Noise Data*⟩
  ⟨*Texture Method Definitions*⟩

11.1 Sampling and Anti-Aliasing

Three main options, in order of preference...

1. filter out high frequency stuff before sampling (e.g. image maps)
2. frequency clamping: don't introduce high frequency stuff in the first place (e.g. sums of noise).
3. super-sampling in texture space: evaluate the texture function at a bunch of points and average.

First is theoretically best. Second is usually pretty good but not necessarily the same result. Third is likely to be inefficient, but at least less so than tracing more camera rays.

In that chapter, we had to throw in the towel and accept the fact that the image function being sampled by rays will have infinite frequency content (e.g. from edges), and thus suffer from aliasing. However, this isn't to say that we should give up completely on removing high-frequencies from the image that we're sampling: if it's relatively cheap to do so (as it is to deal with texture aliasing), it's worth doing so, so that the user doesn't need to increase pixel sampling (and trace many more rays) solely in order to reduce texture aliasing.

47 DifferentialGeometry

XXX need a figure showing image samples and their extent as well as texture image samples for 2D image mapping. Projection of circular region on image plane into ellipse on texture XXX

How the differential quantities in `DifferentialGeometry` are used to drive this process, etc...

11.2 Texture Interface

`Texture` is a template class based on the return type of its evaluation function. This allows us to reuse almost all of the texturing code between textures that return floating point values and textures that return spectra.

⟨*Texture Class Declarations*⟩≡
```
template <class T> class Texture {
public:
    ⟨Texture Interface⟩
};
```

The key to `Texture`'s interface is its evaluation function; it returns a value of the template type `T`, usually either `Float` or `Spectrum`. It has access to the `DifferentialGeometry` at the point being shaded; various textures below will use different parts of this structure to do their work. Textures that anti-alias themselves will use the differential values `dPdx`, `dPdy`, `dudx`, `dvdx`, `dudy`, and `dtdy` to do so...

⟨*Texture Interface*⟩≡
```
virtual T Evaluate(const DifferentialGeometry &) const = 0;
```

11.3 Basic Textures

`ConstantTexture` returns the same value no matter where it is evaluated. It has no frequency content, and needs no anti-aliasing. This may not seem that useful, but having this ability simplifies material creation. For example, all of the materials presented in the next chapter are textured. A diffuse object that is a solid color will have one of these `ConstantTexture`s associated with it. This way, the shading system will always evaluate a texture to get the surface color at a point, avoiding the need for separate textured and non-textured versions of materials.

⟨*Texture Class Declarations*⟩+≡
```
template <class T>
class ConstantTexture: public Texture<T> {
public:
    ConstantTexture(const T &v) { value = v; }
    T Evaluate(const DifferentialGeometry &) const;
private:
    T value;
};
```

⟨*Texture Template Method Definitions*⟩≡
```
template <class T>
T ConstantTexture<T>::Evaluate(const DifferentialGeometry &) const {
    return value;
}
```

DifferentialGeometry 47
Texture 323

Scale

One of the most useful things that can be done with the `Texture`s in this chapter is to compose them together, feeding the output of one texture into the input of another. The `ScaleTexure` takes two textures, a base map and a scale, and returns their product when evaluated. This texture can also ignore anti-aliasing, leaving it to its members to handle.

⟨*Texture Class Declarations*⟩+≡
```
template <class T1, class T2>
class ScaleTexture : public Texture<T2> {
public:
    ScaleTexture(Texture<T1> *s, Texture<T2> *v) {
        scale = s;
        value = v;
    }
    ⟨ScaleTexture Methods⟩
private:
    Texture<T1> *scale;
    Texture<T2> *value;
};
```

⟨*Texture Template Method Definitions*⟩+≡
```
template <class T1, class T2>
T2 ScaleTexture<T1, T2>::Evaluate(const DifferentialGeometry &dg) const {
    return scale->Evaluate(dg) * value->Evaluate(dg);
}
```

We need to delete the child textures used by `ScaleTexture`s when they are deleted. We won't show the destructors for the rest of the textures in this chapter; if they hold pointers to other textures they will delete them in their destructors.

⟨*ScaleTexture Methods*⟩+≡
```
~ScaleTexture() {
    delete scale;
    delete value;
}
```

Mixtures

The `MixTexture` class is a more general variation of `ScaleTexture`. It takes three textures as input: two may be of any type, and the third must return a floating point value. The floating point texture is then used to blend between the two other textures. Note that we can use a `ConstantTexture` for the floating point values to achieve a uniform blend, or a more complex `Texture` to blend in a more creative way.

⟨*Texture Class Declarations*⟩+≡
```
template <class T>
class MixTexture : public Texture<T> {
public:
    MixTexture(Texture<T> *t1, Texture<T> *t2,
               Texture<Float> *amt) {
        tex1 = t1;
        tex2 = t2;
        amount = amt;
    }
    ⟨MixTexture Interface⟩
private:
    Texture<T> *tex1, *tex2;
    Texture<Float> *amount;
};
```

47 DifferentialGeometry
324 ScaleTexture
323 Texture

To evaluate the mixture, we just evaluate the three textures and use the floating point value to linearly interpolate between the two; when the blend amount `amt` is zero, the first texture's value is returned and when it is one, the second one's value is returned. We will generally assume that `amt` will be between zero and one, ensuring that we always interpolate, rather than sometimes extrapolating. However, this behavior is not enforced, and texture extrapolation is possible.

⟨*Texture Template Method Definitions*⟩+≡
```
template <class T>
T MixTexture<T>::Evaluate(const DifferentialGeometry &dg) const {
    T t1 = tex1->Evaluate(dg), t2 = tex2->Evaluate(dg);
    Float amt = amount->Evaluate(dg);
    return (1. - amt) * t1 + amt * t2;
}
```

11.4 2D Mappings

The rest of the textures in this chapter are functions that take a two-dimensional (s,t) coordinate or a three-dimensional (x,y,z) coordinate and compute a texture value at the given position. Sometimes there are obvious ways to choose these *texture coordinates*: for parametric surfaces, such as the quadrics in Chapter 3, there is a natural two-dimensional parameterization of the surface, and for all surfaces the shading point P is a natural choice for a three-dimensional coordinate. In `lrt`, we will use the convention that 2D texture coordinates are denoted by (s,t); this helps make clear the distinction between the intrinsic (u,v) parameterization of the underlying surface and the possibly-different coordinate values used for texturing.

In general, however, there is often not a natural parameterization of complex surfaces. For instance, given an arbitrary subdivision surface, there is no simple and robust way to assign (s,t) texture values to the whole thing so that the entire $[0,1]^2$ (s,t) space is covered continuously and without distortion. Indeed, how to generate smooth and not-distorted parameterizations of complex meshes is currently an active area of research. This section will introdoce two abstract base classes–`TextureMapping2D` and `TextureMapping3D`–that provide an interface for computing 2D and 3D texture coordinates. We will then implement a number of standard mappings using them.

DifferentialGeometry 47

The `TextureMapping2D` base class has a single method, `map`, which is given the `DifferentialGeometry` at the shading point and returns the (s,t) texture coordinates via `Float *`s. Furthermore, it returns estimates for the change in s and t with respect to pixel x and y coordinates in `dsdx`, `dtdx`, `dsty`, and `dtdy`.

⟨*Texture Class Declarations*⟩+≡
```
class TextureMapping2D {
public:
    virtual ~TextureMapping2D() { }
    virtual void Map(const DifferentialGeometry &dg,
        Float *s, Float *t, Float *dsdx, Float *dtdx,
        Float *dsdy, Float *dtdy) const = 0;
};
```

2D Identity Mapping

The simplest texture mapping uses the 2D parametric (u,v) coordinates in the `DifferentialGeometry` to compute the texture coordinates. These can be offset and scaled with user-supplied values in each dimension.

⟨*Texture Class Declarations*⟩+≡
```
class IdentityMapping2D : public TextureMapping2D {
public:
    IdentityMapping2D(Float su = 1, Float sv = 1,
        Float du = 0, Float dv = 0);
    void Map(const DifferentialGeometry &dg, Float *s, Float *t,
        Float *dsdx, Float *dtdx,
        Float *dsdy, Float *dtdy) const;
private:
    Float su, sv, du, dv;
};
```

⟨*Texture Method Definitions*⟩≡
```
IdentityMapping2D::IdentityMapping2D(Float _su, Float _sv,
        Float _du, Float _dv) {
    su = _su; sv = _sv;
    du = _du; dv = _dv;
}
```

⟨*Texture Method Definitions*⟩+≡
```
void IdentityMapping2D::Map(const DifferentialGeometry &dg,
        Float *s, Float *t, Float *dsdx, Float *dtdx,
        Float *dsdy, Float *dtdy) const {
    *s = su * dg.u + du;
    *t = sv * dg.v + dv;
    *dsdx = su * dg.dudx;
    *dtdx = sv * dg.dvdx;
    *dsdy = su * dg.dudy;
    *dtdy = sv * dg.dvdy;
}
```

Spherical Mapping

Another useful mapping effectively wraps a sphere around the object. Each point is projected along the vector from the sphere's center through the point, up to the sphere's surface. There, the same (u, v) mapping as was used for the sphere shape is used.

47 DifferentialGeometry
21 Point
326 TextureMapping2D
32 Transform

The SphericalMapping2D object stores a transformation that is applied to points before htis mapping is performed; this effectively allows the sphere to be positioned and oriented with respect to the object.

⟨*Texture Class Declarations*⟩+≡
```
class SphericalMapping2D : public TextureMapping2D {
public:
    SphericalMapping2D(const Transform &toSph)
        : toSphere(toSph) {
    }
    void Map(const DifferentialGeometry &dg, Float *s, Float *t,
        Float *dsdx, Float *dtdx,
        Float *dsdy, Float *dtdy) const;
private:
    void sphere(const Point &P, Float *s, Float *t) const;
    Transform toSphere;
};
```

XXX compute the differentials by just applying the mapping to all three points and taking the differences...

⟨*Texture Method Definitions*⟩+≡
```
void SphericalMapping2D::Map(const DifferentialGeometry &dg,
        Float *s, Float *t, Float *dsdx, Float *dtdx,
        Float *dsdy, Float *dtdy) const {
    Float sx, tx, sy, ty;
    sphere(dg.P, s, t);
    sphere(dg.P + dg.dPdx, &sx, &tx);
    *dsdx = sx - *s;
    *dtdx = tx - *t;
    if (*dtdx > .5) *dtdx = 1. - *dtdx;
    sphere(dg.P + dg.dPdy, &sy, &ty);
    *dsdy = sy - *s;
    *dtdy = ty - *t;
    if (*dtdy > .5) *dtdy = 1. - *dtdy;
}
```

⟨*Texture Method Definitions*⟩+≡
```
void SphericalMapping2D::sphere(const Point &P, Float *s, Float *t) const {
    Vector vec = (toSphere(P) - Point(0,0,0)).Hat();
    Float theta = SphericalTheta(vec);
    Float phi = SphericalPhi(vec);
    *s = theta / M_PI;
    *t = phi / (2.f * M_PI);
}
```


DifferentialGeometry 47
Hat 19
Point 21
SphericalMapping2D 327
SphericalPhi 164
SphericalTheta 164
TextureMapping2D 326
Transform 32
Vector 16

Cylindrical Mapping

Like the spherical mapping, the cylindrical mapping effectively wraps a cylinder around the object having texture coordinates computed for it. It also supports a transformation to orient the mapping cylinder.

⟨*Texture Class Declarations*⟩+≡
```
class CylindricalMapping2D : public TextureMapping2D {
public:
    CylindricalMapping2D(const Transform &toCyl)
        : toCylinder(toCyl) {
    }
    void Map(const DifferentialGeometry &dg, Float *s, Float *t,
        Float *dsdx, Float *dtdx,
        Float *dsdy, Float *dtdy) const;
private:
    void cylinder(const Point &P, Float *s, Float *t) const;
    Transform toCylinder;
};
```

⟨*Texture Method Definitions*⟩+≡
```
void CylindricalMapping2D::Map(const DifferentialGeometry &dg,
        Float *s, Float *t, Float *dsdx, Float *dtdx,
        Float *dsdy, Float *dtdy) const {
    Float sx, tx, sy, ty;
    cylinder(dg.P, s, t);
    cylinder(dg.P + dg.dPdx, &sx, &tx);
    *dsdx = sx - *s;
    *dtdx = tx - *t;
    if (*dtdx > .5) *dtdx = 1. - *dtdx;
    cylinder(dg.P + dg.dPdy, &sy, &ty);
    *dsdy = sy - *s;
    *dtdy = ty - *t;
    if (*dtdy > .5) *dtdy = 1. - *dtdy;
}
```

⟨*Texture Method Definitions*⟩+≡
```
void CylindricalMapping2D::cylinder(const Point &P, Float *s,
        Float *t) const {
    Vector vec = (toCylinder(P) - Point(0,0,0)).Hat();
    *s = (M_PI + atan2f(vec.y, vec.x)) / (2.f * M_PI);
    *t = (vec.z + 1.f) * 0.5f;
}
```

328 CylindricalMapping2D
47 DifferentialGeometry
19 Hat
21 Point
326 TextureMapping2D
16 Vector

Planar Mapping

Another classing mapping method is the planar mapping. The point to have texture coordinates computed is effectively projected onto a plane; a 2D parameterization of the plane gives texture coordinates for the point. For example, a point P could be projected on the $z = 0$ plane to yield texture coordinates given by $u = P_x$ and $v = P_y$.

More generally, we can define such a parameterized plane with two non-parallel vectors $\vec{\mathbf{v}}_\mathbf{u}$ and $\vec{\mathbf{v}}_\mathbf{v}$ and offsets du and dv. The texture coordinates are given by taking the dot product of the vector from the point to the origin with each vector $\vec{\mathbf{v}}_\mathbf{u}$ and $\vec{\mathbf{v}}_\mathbf{v}$ and then adding the offset. For the example in the previous paragraph, we'd have $\vec{\mathbf{v}}_\mathbf{u} = (1,0,0)$, $\vec{\mathbf{v}}_\mathbf{v} = (0,1,0)$, and $du = dv = 0$.

⟨*Texture Class Declarations*⟩+≡
```
class PlanarMapping2D : public TextureMapping2D {
public:
    PlanarMapping2D(const Vector &v1, const Vector &v2, Float du = 0,
        Float dv = 0);
    void Map(const DifferentialGeometry &dg, Float *s, Float *t,
        Float *dsdx, Float *dtdx,
        Float *dsdy, Float *dtdy) const;
private:
    Vector vs, vt;
    Float ds, dt;
};
```

⟨*Texture Method Definitions*⟩+≡
```
PlanarMapping2D::PlanarMapping2D(const Vector &_v1,
        const Vector &_v2, Float _ds, Float _dt) {
    vs = _v1;
    vt = _v2;
    ds = _ds;
    dt = _dt;
}
```

⟨*Texture Method Definitions*⟩+≡
```
void PlanarMapping2D::Map(const DifferentialGeometry &dg,
        Float *s, Float *t, Float *dsdx, Float *dtdx,
        Float *dsdy, Float *dtdy) const {
    Vector vec = dg.P - Point(0,0,0);
    *s = ds + Dot(vec, vs);
    *t = dt + Dot(vec, vt);
    *dsdx = Dot(dg.dPdx, vs);
    *dtdx = Dot(dg.dPdx, vt);
    *dsdy = Dot(dg.dPdy, vs);
    *dtdy = Dot(dg.dPdy, vt);
}
```

DifferentialGeometry 47
Point 21
Texture 323
Vector 16

11.5 Interpolated Textures

Two simple textures interpolate between constant values based on the relation of the (s,t) coordinates of the point being shaded to values at the four corners of $[0,1]^2$ or at the vertices of a triangle mesh. These textures also don't consider anti-aliasing, since they don't tend to be the source of high frequency variations, and because in any case, if they used a box filter to remove high frequencies, the result is the same as just evaluating at the main point for those cases.

Bilinear Interpolation

⟨*Texture Class Declarations*⟩+≡
```
template <class T>
class BilerpTexture : public Texture<T> {
public:
    ⟨BilerpTexture Interface⟩
    T Evaluate(const DifferentialGeometry &) const;
private:
    ⟨BilerpTexture Private Data⟩
};
```

An even more general class is the `BilerpTexture` class. It provides bilinear interpolation between four constant values. Figure 11.2 shows the idea: values are defined at $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$ in (s,t) parameter space. The value at a particular (s,t) position is found by interpolating between them.

Figure 11.2: basic bilerp idea

⟨*Texture Template Method Definitions*⟩+≡
```
template <class T>
BilerpTexture<T>::BilerpTexture(TextureMapping2D *m,
        const T &t00, const T &t01, const T &t10, const T &t11) {
    mapping = m;
    tex00 = t00;
    tex01 = t01;
    tex10 = t10;
    tex11 = t11;
}
```

330 BilerpTexture
47 DifferentialGeometry
326 TextureMapping2D

⟨*BilerpTexture Private Data*⟩≡
```
TextureMapping2D *mapping;
T tex00, tex01, tex10, tex11;
```

The interpolated value of the four values at a (s,t) position can be computed by three linear interpolations. For example, we can first interpolate *u* of the way between the values at $(0,0)$ and $(1,0)$ and store that in a temporary `tmp1`. We can then interpolate *u* of the way between the $(0,1)$ and $(1,1)$ values and store the result in `tmp2`. Finally, by interpolating *v* of the way between `tmp1` and `tmp2` gives us our final result. (We get the same result if we first interpolate between $(0,0)$ and $(0,1)$ in *v*, etc.)

Rather than doing all this work and storing the intermediate values explicitly, an appropriately weighted average of the four corner values gives us the same value. The result of this is in the `return` statement in the evaluation routine below.

⟨*Texture Template Method Definitions*⟩+≡
```
template <class T>
T BilerpTexture<T>::Evaluate(const DifferentialGeometry &dg) const {
    Float u, v, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &u, &v, &dsdx, &dtdx, &dsdy, &dtdy);
    return (1-u)*(1-v) * tex00 + (1-u)*v * tex01 + u*(1-v) * tex10 +
        u*v * tex11;
}
```

Barycentric Interpolation

A generalization of the bilinear interpolation texture, `VertexTexture`, stores values at the vertices of a triangle mesh and interpolates the three surrounding vertex values to compute the value at a particular point on a particular face.

⟨*Texture Class Declarations*⟩+≡
```
template <class T>
class VertexTexture : public Texture<T> {
public:
    ⟨VertexTexture Interface⟩
    T Evaluate(const DifferentialGeometry &) const;
private:
    ⟨VertexTexture Private Data⟩
};
```

⟨*Texture Template Method Definitions*⟩+≡
```
template <class T>
VertexTexture<T>::VertexTexture(const T *vs, int nv) {
    nVertices = nv;
    vals = new T[nv];
    for (int i = 0; i < nv; ++i)
        vals[i] = vs[i];
}
```
DifferentialGeometry 47
Texture 323

⟨*VertexTexture Private Data*⟩≡
```
int nVertices;
T *vals;
```

The `VertexTexture` can only be assigned to `Triangle` shapes; we will depend on the rest of the system to enforce this. Therefore, here we can find the vertex indices for the triangle vertices from the `Shape` pointer in the `DifferentialGeometry`, giving us the indices to use into the per-vertex data here.

⟨*Texture Template Method Definitions*⟩+≡
```
template <class T>
T VertexTexture<T>::Evaluate(const DifferentialGeometry &dg) const {
    ⟨Find three vertex texture values, v0, v1, and v2⟩
    ⟨Compute barycentric coordinates for point⟩
    return b[0] * v0 + b[1] * v1 + b[2] * v2;
}
```

⟨*Find three vertex texture values,* `v0`, `v1`, *and* `v2`⟩≡
```
Triangle *tri = (Triangle *)(dg.shape);
int *v = tri->v;
const T &v0 = vals[v[0]], &v1 = vals[v[1]], &v2 = vals[v[2]];
```

Recall that the (u,v) parametric coordinates in the `DifferentialGeometry` for a triangle are computed with barycentric interpolation of parametric coordinates at the triangle vertices.

$$\begin{aligned} u &= b_0u_0 + b_1u_1 + b_2u_2 \\ v &= b_0v_0 + b_1v_1 + b_2v_2 \end{aligned}$$

Because b_i are barycentric coordinates, $b_0 = 1 - b_1 - b_2$. Here, u, v, u_i and v_i are all known, u and v from the `DifferentialGeometry` and u_i and v_i from the `Triangle`. We can substitue for the b_0 term and rewrite the above equations, giving a linear system in two unknowns b_1 and b_2.

$$\begin{pmatrix} u_1 - u_0 & u_2 - u_1 \\ v_1 - v_0 & v_2 - v_1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} u - u_0 \\ v - v_0 \end{pmatrix}$$

This is a linear system of the basic form $AX = B$. We can solve for X to give us the two barycentric coordinates by inverting A

$$X = A^{-1}B.$$

The closed form solution for this is implemented in the utility routine `SolveLinearSystem2x2()`.

⟨*Compute barycentric coordinates for point*⟩≡
```
Float b[3];
⟨Initialize A and B matrices for barycentrics⟩
if (!SolveLinearSystem2x2(A, B, &b[1])) {
    ⟨Handle degenerate parametric mapping⟩
}
else
    b[0] = 1.f - b[1] - b[2];
```

78 GetUVs
510 SolveLinearSystem2x2

⟨*Initialize A and B matrices for barycentrics*⟩≡
```
Float uv[3][2];
tri->GetUVs(uv);
Float A[2][2] = { { uv[1][0] - uv[0][0], uv[2][0] - uv[0][0] },
                  { uv[1][1] - uv[0][1], uv[2][1] - uv[0][1] } };
Float B[2] = { dg.u - uv[0][0], dg.v - uv[0][1] };
```

If the determinant of A is zero, the solution is undefined. This could happen if all three triangle vertices had the same texture coordinates, for example. In this case, we just set the barycentric coordinates arbitrarily.

⟨*Handle degenerate parametric mapping*⟩≡
```
b[0] = b[1] = b[2] = .3333333333f;
```

11.6 Image Maps

The class `ImageMap` handles basic operations for 2D image maps stored on disk. It will be key to the implementation of `ImageTexture`s, in a few pages. The caller provides the filename of a TIFF texture, and we read it into an array of `Spectrum`s. This can actually be somewhat wasteful, since most TIFFs are stored with 8-bit values in their red, green, and blue channels, while our `Spectrum` class stores spectra with 32-bit floating point values for each color component. Because we do want to support general image maps with floating-point values though, it's easiest to just store textures in `Spectrum` objects.

`ImageTexture`s implement textures from 2D bitmaps stored on disk. Careful filtering of the bitmap values is essential for anti-aliasing...

⟨*Texture Class Declarations*⟩+≡
```
class ImageMap {
public:
    ImageMap(const string &filename);
    ⟨ImageMap Method Declarations⟩
private:
    ⟨ImageMap Private Data⟩
};
```

⟨*Texture Method Definitions*⟩+≡
```
ImageMap::ImageMap(const string &filename) {
    int width, height;
    Spectrum *texels = TIFFRead(filename, &width, &height);
    if (texels) {
        mipmap = new MIPMap<Spectrum>(width, height, texels);
        delete[] texels;
    }
    else mipmap = NULL;
}
```

MIPMap 336
Spectrum 155
TIFFRead 532

⟨*ImageMap Private Data*⟩≡
```
MIPMap<Spectrum> *mipmap;
```

Texture image is a 2D set of point samples of a presumably continuous function. Consider projection of textured object onto the viewing screen: the rate at which the texture is sampled, given by the image sampling rate, the texture map function, the size the object projects to on the screen, may be much higher or much lower than the rate at which there are 2D texture samples. And the points at which we're sampling the texture will be different than the ones at which it is defined.

Recall from Section 7.1 sampling and signal processing theory. We will have aliasing if the texture function isn't sampled by screen samples at a sufficiently high rate. There are a few key differences from the image sampling issues discussed in Chapter 7, however: first, it's cheap to get the value of a sample–just an array lookup (as opposed to having to trace a ray). Second, we can find out anything we want to about the behavior of the texture image function–it's fully defined by the set of samples that we have.

We'd like to take this opportunity to reduce aliasing in the final image by pre-filtering the texture image according to the rate at which we're sampling it in the final image. This sampling rate may in general change from pixel to pixel–*space variant*–since it's determined by scene geometry, stuff like that that is varying in unusual ways. Thus, efficiently pre-filtering the texture function, reconstructing a new function and then resampling it at a particular location has received a bit of attention in graphics.)

Figure 11.3: may need to look at many texels to filter a texture over a large area...

⟨*Texture Method Definitions*⟩+≡
```
void ImageMap::Lookup(Float s, Float t, Float dsdx, Float dtdx,
        Float dsdy, Float dtdy, Spectrum *val) const {
    if (!mipmap)
        *val = Spectrum(1.f);
    else
        *val = mipmap->Lookup(s, t, dsdx, dtdx, dsdy, dtdy);
}
```
334 ImageMap
243 Luminance
155 Spectrum

⟨*Texture Method Definitions*⟩+≡
```
void ImageMap::Lookup(Float s, Float t, Float dsdx, Float dtdx,
        Float dsdy, Float dtdy, Float *val) const {
    Spectrum sp;
    Lookup(s, t, dsdx, dtdx, dsdy, dtdy, &sp);
    *val = sp.Luminance();
}
```

MIP Maps

The `MIPMap` class implements two different methods for efficient texture filtering. For a high-resolution bitmap, naive filtering of the texture samples may be extremely inefficient–Figure 11.3 shows a bitmap, with texels indicated by dots and the filter region indicated by the rectangular dashed region. When the texture projects to a very small area on the screen, a large number of texels may need to be filtered.

XXX spatially invariant filter, image pyramid, tri-linear interpolation/Gaussian...

⟨*mipmap.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef MIPMAP_H
#define MIPMAP_H 1
#include "lrt.h"
⟨MIPMap Declarations⟩
⟨MIPMap Method Definitions⟩
#endif // MIPMAP_H
```

⟨*MIPMap Declarations*⟩≡
```
template <class T> class MIPMap {
public:
    ⟨MIPMap Public Interface⟩
private:
    ⟨MIPMap Private Methods⟩
    ⟨MIPMap Private Data⟩
};
```

⟨*MIPMap Method Definitions*⟩≡
```
#define BLOCK_SIZE 4
#define LOG_BLOCK_SIZE 2
#define UP(x) (((x)+BLOCK_SIZE-1) & (!BLOCK_SIZE-1))
```

⟨*MIPMap Method Definitions*⟩+≡
```
template <class T>
MIPMap<T>::MIPMap(int ur, int vr, T *d) {
    nLevels = Float2Int(log(max(ur, vr)) / log(2.));
    uRes = new int[nLevels];
    vRes = new int[nLevels];
    data = new T *[nLevels];
    ⟨Initialize most detailed level of MIPMap⟩
    for (int i = 1; i < nLevels; ++i) {
        ⟨Initialize ith MIPMap level from i − 1st level⟩
    }
    ⟨Initialize MIPMap filter weights if needed⟩
}
```

AllocL2CacheAligned 507
Float2Int 514
max 513

⟨*MIPMap Private Data*⟩≡
```
T **data;
int nLevels;
int *uRes, *vRes;
```

⟨*Initialize most detailed level of MIPMap*⟩≡
```
uRes[0] = ur;
vRes[0] = vr;
data[0] = (T *)AllocL2CacheAligned(UP(ur) * UP(vr) * sizeof(T));
for (int v = 0; v < vr; ++v)
    for (int u = 0; u < ur; ++u)
        texel(0, u, v) = d[u + v * ur];
```

⟨*Initialize ith MIPMap level from i − 1st level*⟩≡
```
uRes[i] = max(1, uRes[i-1]/2);
vRes[i] = max(1, vRes[i-1]/2);
data[i] = (T *)AllocL2CacheAligned(UP(uRes[i]) * UP(vRes[i]) * sizeof(T));
```
⟨*Filter four texels from finer level of pyramid*⟩

⟨*Filter four texels from finer level of pyramid*⟩≡
```
for (int v = 0; v < vRes[i]; ++v)
    for (int u = 0; u < uRes[i]; ++u)
        texel(i, u, v) = .25f * (texel(i-1, 2*u, 2*v) +
            texel(i-1, 2*u+1, 2*v) + texel(i-1, 2*u, 2*v+1) +
            texel(i-1, 2*u+1, 2*v));
```

The `texel()` utility function returns a reference to a `Spectrum` for the given texel. If an out-of-range texel coordinate is passed in, we clamp it to the range of valid texel coordinates, such that the edge texels are repeated throughout space. Depending on the object being rendered and how the texture map is being used, other strategies such as returning a texels with a fixed color, mirroring the texture map across the edges, or repeating the texture map continuously across the (s,t) plane are often useful.

⟨*MIPMap Method Definitions*⟩+≡
```
template <class T>
const T &MIPMap<T>::texel(int level, int u, int v) const {
    ⟨Return (u,v) texel from level⟩
}
```

⟨*Return* (u,v) *texel from* `level`⟩≡
```
u = Clamp(u, 0, uRes[level]-1);
v = Clamp(v, 0, vRes[level]-1);
int bu = (u >> LOG_BLOCK_SIZE), bv = (v >> LOG_BLOCK_SIZE);
int ou = u & (BLOCK_SIZE-1), ov = v & (BLOCK_SIZE-1);
int offset = BLOCK_SIZE * BLOCK_SIZE *
    ((UP(uRes[level]) >> LOG_BLOCK_SIZE) * bv + bu);
offset += BLOCK_SIZE * ov + ou;
return data[level][offset];
```

513 Clamp
336 MIPMap

elliptically weighted average(Hec86).

⟨*MIPMap Private Data*⟩+≡
```
#define WEIGHT_LUT_SIZE 256
static Float weightLut[WEIGHT_LUT_SIZE];
static bool weightsInitialized;
```

⟨*MIPMap Method Definitions*⟩+≡
```
template <class T> Float MIPMap<T>::weightLut[WEIGHT_LUT_SIZE];
template <class T> bool MIPMap<T>::weightsInitialized = false;
```

⟨*Initialize MIPMap filter weights if needed*⟩≡
```
if (!weightsInitialized) {
    for (int i = 0; i < WEIGHT_LUT_SIZE; ++i) {
        Float alpha = 2;
        Float r2 = float(i) / float(WEIGHT_LUT_SIZE - 1);
        weightLut[i] = expf(-alpha * r2);
    }
    weightsInitialized = true;
}
```

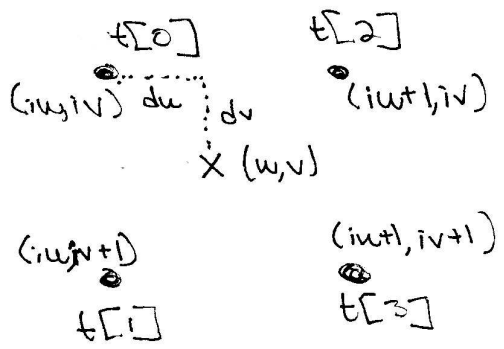


Figure 11.4: bilinear interpolation of texels

⟨*MIPMap Method Definitions*⟩+≡
```
template <class T>
MIPMap<T>::~MIPMap() {
    for (int i = 0; i < nLevels; ++i)
        FreeCacheAligned(data[i]);
    delete[] data;
    delete[] uRes;
    delete[] vRes;
}
```

FreeCacheAligned 507
MIPMap 336

The texture map covers the region in texture coordinates from $(0,0)$ to $(1,1)$. We need to first take the coordinates given by the user and handle the case where they're outside this range. We then compute texture coordinates in the space spanned from $(0,0)$ to $(\text{width},\text{height})$, where width and height are the number of texels in each direction. Finally, we compute the integer texture coordinates of the upper left of the four texels that we'll be using.

XXX Figure 11.4.

There is an important subtlety in how the texture coordinates are computed that is a result of what convention we use for where the texels are positioned. Figure 11.5 shows the two possibilities for how an image map with two texels in each direction might be laid out over $[0,1]^2$. The natural choice is to place the four texels at integer locations $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$; these points are marked with circles in the figure.

A better choice, however, is to place them at the points marked with "x"s; this convention is used by most graphics APIs, including OpenGL. Advantages of using this convention include XXX. It is easy to implement this convention; the texture coordinates just need to be offset by 0.5 after they are scaled by the image resolution in each direction.

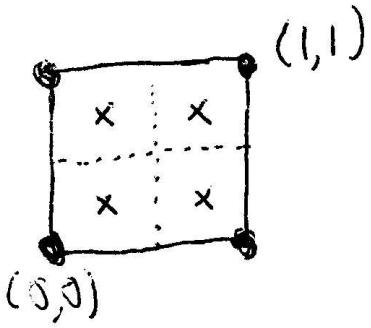


Figure 11.5: Those 0.5 texel offsets...

⟨*MIPMap Method Definitions*⟩+≡
```
template <class T>
T MIPMap<T>::bilerp(int level, Float u, Float v) const {
    if (level >= nLevels) level = nLevels-1;
    u = u * uRes[level] - 0.5f;
    v = v * vRes[level] - 0.5f;
    int u0 = Floor2Int(u), v0 = Floor2Int(v);
    ⟨Compute bilinear interpolation weights⟩
    return weights[0] * texel(level, u0, v0) +
        weights[1] * texel(level, u0, v0+1) +
        weights[2] * texel(level, u0+1, v0) +
        weights[3] * texel(level, u0+1, v0+1);
}
```

514 Floor2Int
336 MIPMap

⟨*Compute bilinear interpolation weights*⟩≡
```
Float du = u - u0, dv = v - v0;
Float weights[4];
weights[0] = (1.-du)*(1.-dv);
weights[1] = (1.-du)*dv;
weights[2] = du*(1.-dv);
weights[3] = du*dv;
```

⟨*MIPMap Method Definitions*⟩+≡
```
template <class T>
T MIPMap<T>::Lookup(Float u, Float v, Float width) const {
    static StatsCounter mipTrilerps("Texture", "Trilinear MIPMap lookups");
    ++mipTrilerps;
    Float level = -logf(max(width, 1e-8f)) / logf(2.);
    if (level >= nLevels-1)
        return bilerp(0, u, v);
    else {
        int level = nLevels-1-level;
        int l0 = Floor2Int(level);
        Float delta = level - l0;
        return (1.f-delta) * bilerp(l0, u, v) +
            delta * bilerp(l0+1, u, v);
    }
}
```

⟨*MIPMap Method Definitions*⟩+≡
```
template <class T>
T MIPMap<T>::Lookup(Float u, Float v, Float du0, Float dv0,
        Float du1, Float dv1) const {
    static StatsCounter ewaLookups("Texture", "EWA filter lookups");
    ++ewaLookups;
    ⟨Compute ellipse minor and major axes⟩
    ⟨Clamp ellipse eccentricity if too large⟩
    ⟨Choose level of detail for EWA lookup⟩
    if (lod >= nLevels)
        return texel(nLevels-1, 0, 0);
    else
        return ewaLod(u, v, du0, dv0, du1, dv1, lod);
}
```

Floor2Int 514
Lookup 335
max 513
MIPMap 336
StatsCounter 501
Texture 323

⟨*Compute ellipse minor and major axes*⟩≡
```
Float major[2], minor[2];
if (du0*du0 + dv0*dv0 < du1*du1 + dv1*dv1) {
    major[0] = du1;
    major[1] = dv1;
    minor[0] = du0;
    minor[1] = dv0;
}
else {
    major[0] = du0;
    major[1] = dv0;
    minor[0] = du1;
    minor[1] = dv1;
}
Float majorLength = sqrtf(major[0]*major[0] + major[1]*major[1]);
Float minorLength = sqrtf(minor[0]*minor[0] + minor[1]*minor[1]);
```

if the eccentricity of the ellipse is looking to be too big, scale up the shorter of

the two vectors so that it's a little more reasonable. This lets us avoid spending inordinate amounts of time filtering very long and skinny regions (which take a lot of time), at the expense of some blurring...

⟨*Clamp ellipse eccentricity if too large*⟩≡
```
const Float maxEccentricity = 10;
Float invMinorLength = 1.f / minorLength;
Float e = majorLength * invMinorLength;
if (e > maxEccentricity) {
    Float scale = e / maxEccentricity;
    minor[0] *= scale;
    minor[1] *= scale;
    minorLength *= scale;
}
```

Pick a lod such that we're looking at somewhere around 3-9 texels in the minor axis direction.

⟨*Choose level of detail for EWA lookup*⟩≡
```
int lod = max(0, nLevels - 1 - Log2Int(5.f * invMinorLength));
```

⟨*MIPMap Method Definitions*⟩+≡
```
template <class T>
T MIPMap<T>::ewaLod(Float u, Float v, Float du0, Float dv0,
        Float du1, Float dv1, int level) const {
    ⟨Convert EWA coordinates to appropriate scale for level⟩
    ⟨Compute ellipse coefficients to bound EWA filter region⟩
    ⟨Compute the ellipse's (s,t) bounding box in texture space⟩
    ⟨Scan over ellipse bound and compute quadratic equation⟩
}
```

513 max
336 MIPMap

⟨*Convert EWA coordinates to appropriate scale for level*⟩≡
```
u = u * uRes[level]; // - 0.5f;
v = v * vRes[level]; // - 0.5f;
du0 *= uRes[level];
dv0 *= vRes[level];
du1 *= uRes[level];
dv1 *= vRes[level];
```

compute ellipse coefficients to bound the region: `A*x*x + B*x*y + C*y*y = F`.

⟨*Compute ellipse coefficients to bound EWA filter region*⟩≡
```
Float A = dv0*dv0 + dv1*dv1 + 1;
Float B = -2.f * (du0*dv0 + du1*dv1);
Float C = du0*du0 + du1*du1 + 1;
Float F = A*C - B*B*0.25f;
Float invF = 1.f / F;
A *= invF;
B *= invF;
C *= invF;
```

⟨*Compute the ellipse's (s,t) bounding box in texture space*⟩≡
```
Float det = -B*B + 4.f*A*C;
Float invDet = 1.f / det;
Float uSqrt = sqrtf(det * C), vSqrt = sqrtf(A * det);
int u0 = Ceil2Int (u - 2.f * invDet * uSqrt);
int u1 = Floor2Int(u + 2.f * invDet * uSqrt);
int v0 = Ceil2Int (v - 2.f * invDet * vSqrt);
int v1 = Floor2Int(v + 2.f * invDet * vSqrt);
static StatsRatio ewaTexels("Texture", "Texels per EWA lookup", false);
ewaTexels.add((1+u1-u0) * (1+v1-v0), 1);
```

⟨*Scan over ellipse bound and compute quadratic equation*⟩≡
```
T num(0.);
Float den = 0;
for (int iv = v0; iv <= v1; ++iv) {
    Float V = iv - v;
    for (int iu = u0; iu <= u1; ++iu) {
        Float U = iu - u;
        Float r2 = A*U*U + B*U*V + C*V*V;
        if (r2 < 1.) {
            ⟨Add EWA sample for current texel⟩
        }
    }
}
return num / den;
```


Ceil2Int 514
Float2Int 514
Floor2Int 514
StatsRatio 501
Texture 323

⟨*Add EWA sample for current texel*⟩≡
```
Float weight = weightLut[Float2Int(r2 * (WEIGHT_LUT_SIZE - 1))];
num += texel(level, iu, iv) * weight;
den += weight;
```

Texture caching

Because the user may re-use a texture many times within a scene, and because we may have to look up a texture at shading time, we provide a global hash table of texture maps, so that they are only loaded once, even if used multiple times.

⟨*Texture Cache Data*⟩≡
```
static StringHashTable textures;
```

⟨*Texture Cache Methods*⟩≡
```
ImageMap *GetTexture(const string &filename) {
    ImageMap *ret = (ImageMap *)textures.Search(filename);
    if (!ret) {
        static StatsCounter texLoaded("Texture",
            "Number of image maps loaded");
        ++texLoaded;
        ret = new ImageMap(filename);
        textures.Add(filename, ret);
    }
    return ret;
}
```

Image Texture Maps

We now provide the `Texture` subclass that uses a `ImageMap` for image mapping.

⟨*Texture Class Declarations*⟩+≡
```
template <class T>
class ImageTexture : public Texture<T> {
public:
    ⟨ImageTexture Interface⟩
private:
    ⟨ImageTexture Private Data⟩
};
```

⟨*Texture Template Method Definitions*⟩+≡
```
template <class T>
ImageTexture<T>::ImageTexture(TextureMapping2D *m,
        const string &filename) {
    mapping = m;
    imageMap = GetTexture(filename);
}
```

⟨*ImageTexture Private Data*⟩≡
```
ImageMap *imageMap;
TextureMapping2D *mapping;
```

The evaluation routine is a straightforward of texture coordinate computation and image map lookup. The lookup is written in a slightly tortuous manner so that we can overload the `ImageMap` lookup method based on the pointer type passed in for the return value. This in turn was necessary since it's not possible to overload functions by return type in C++.

⟨*Texture Template Method Definitions*⟩+≡
```
template <class T>
T ImageTexture<T>::Evaluate(const DifferentialGeometry &dg) const {
    Float s, t, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
    T val;
    imageMap->Lookup(s, t, dsdx, dtdx, dsdy, dtdy, &val);
    return val;
}
```


47 DifferentialGeometry
334 ImageMap
335 Lookup
516 Search
501 StatsCounter
323 Texture
326 TextureMapping2D

11.7 Solid and Procedural Texturing

Once one starts to think of (s,t) texture coordinates as quantities that can be computed in a number of ways–not just from the parametric coordinates of the surface, the next step is to consider textures defined over a three-dimensional domain (often called *solid textures*.) The nice thing about solid textures is that all objects have a natural three-dimensional texture mapping–the object-space position. This is a substantial advantage for texturing objects that don't have a natural two-dimensional parameterization (e.g. triangle meshes and implicit surfaces), and for objects that have a distorted parameterization (e.g. the poles of a sphere.)

We will define a `TextureMapping3D` class that defines the interface for generating three-dimensional texture coordinates.

Note that this isn't true for procedural textures, where in general, it's expensive to compute what is going on at a particular point, and where those point samples don't fully characterize the function. Therefore, what we'd like to do there is to remove high-frequency stuff in the signal before we take samples from it. Thus, the thing computing the procedural value needs to be aware of the frequency content of the various things that it does along the way so that it can stop/remove stuff that is too high-frequency and will alias. Though this sounds daunting, fortunately, there are a handful of techniques that work well to handle this.

⟨*Texture Class Declarations*⟩+≡
```
class TextureMapping3D {
public:
    virtual ~TextureMapping3D() { }
    virtual Point Map(const DifferentialGeometry &dg, Point *dPdx,
        Point *dPdy) const = 0;
};
```

The natural three dimensional mapping just takes the world-space coordinate of the point being shaded and applies a linear transformation to it.

DifferentialGeometry 47
Point 21
Transform 32

⟨*Texture Class Declarations*⟩+≡
```
class IdentityMapping3D : public TextureMapping3D {
public:
    IdentityMapping3D(const Transform &x) : xform(x) { }
    Point Map(const DifferentialGeometry &dg, Point *dPdx,
        Point *dPdy) const {
        return xform(dg.P);
    }
private:
    Transform xform;
};
```

The problem that solid textures introduce is texture representation; a three-dimensional bitmap takes up a fair amount of storage space, and is much harder to acquire than a two-dimensional texture map (which can come from a digital photograph, a rendered image, a texture painted by an artists, etc.) Therefore, simultaneous to the invention of solid texturing was the invention of *procedural texturing*–the idea that short programs could be used to generate texture values at arbitrary positions on surfaces in the scene.

A simple instance of this idea is a procedural sine wave. If one wanted to use a sine wave for bump-mapping to simulate waves in water, for example, one might as well just evaluate the `sin` function at points on the surface as needed. It's inefficient (and inaccurate) to precompute values of the function at a grid of points and then store them in an image map. If one can invent a three-dimensional function that describes the colors of wood-grain in a solid block of wood, for instance, then one can generate images of complex objects that appear to be carved from wood. Over the years, procedural texturing has grown in application considerably as techniques have been developed to describe more and more complex surfaces procedurally.

Procedural texturing has a number of other interesting implications. First, it can be used to reduce overall memory requirements for rendering, by avoiding the storage of large, high-resolution texture maps. In addition, procedural shading gives the promise of potentially infinite detail; as the viewer approaches an object, the texturing function is evaluated at the points being shaded, which naturally leads to the right amount of detail being visible. In contrast, image texture maps typically become blurry when the viewer is too close to them. However, procedural textures are much more difficult to control and make localized changes to than image maps.

UV texture

A trivial procedural texture, mostly useful for debugging the parameterization of Shapes, converts the surface's (u,v) coordinates into the first two components of a Spectrum.

⟨*Texture Class Declarations*⟩+≡
```
class UVTexture : public Texture<Spectrum> {
public:
    UVTexture(TextureMapping2D *m) {
        mapping = m;
    }
    ⟨UVTexture Interface⟩
private:
    TextureMapping2D *mapping;
};
```

47 DifferentialGeometry
155 Spectrum
323 Texture
326 TextureMapping2D

⟨*Texture Method Definitions*⟩+≡
```
Spectrum UVTexture::Evaluate(const DifferentialGeometry &dg) const {
    Float u, v, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &u, &v, &dsdx, &dtdx, &dsdy, &dtdy);
    Float cs[COLOR_SAMPLES];
    memset(cs, 0, COLOR_SAMPLES * sizeof(Float));
    cs[0] = u;
    cs[1] = v;
    return Spectrum(cs);
}
```

Checkerboard

The checkerboard is the canonical basic procedural texture. The (s,t) texture coordinates are used to break parameter space up into square parametric regions which are shaded with alternating patterns. Rather than just supporting checkerboards that switch between two fixed colors, we allow the user to pass in two texture maps.

⟨*Texture Class Declarations*⟩+≡
```
template <class T> class UVCheckerboard : public Texture<T> {
public:
    ⟨UVCheckerboard Interface⟩
private:
    ⟨UVCheckerboard Private Data⟩
};
```

⟨*UVCheckerboard Interface*⟩≡
```
UVCheckerboard(TextureMapping2D *m, Texture<T> *c1,
        Texture<T> *c2) {
    mapping = m;
    tex1 = c1;
    tex2 = c2;
}
```

⟨*UVCheckerboard Private Data*⟩≡
```
Texture<T> *tex1, *tex2;
TextureMapping2D *mapping;
```

After getting the (s,t) texture coordinates from the `TextureMapping2D`, we round the texture coordinates to the nearest integers, add them together, and see if the result has odd or even parity; this determines which of the two texture maps we evaluate.

⟨*Texture Template Method Definitions*⟩+≡
```
template <class T>
T UVCheckerboard<T>::Evaluate(
        const DifferentialGeometry &dg) const {
    Float u, v, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &u, &v, &dsdx, &dtdx, &dsdy, &dtdy);
    if ((Round(u) + Round(v)) % 2 == 0)
        return tex1->Evaluate(dg);
    return tex2->Evaluate(dg);
}
```

DifferentialGeometry 47
Texture 323
TextureMapping2D 326

Solid Checkerboard

The previous `Checkerboard` class wraps a checkerboard pattern *around* the object in parameter space. We can also define a solid checkerboard pattern based on three-dimensional texture coordinates. As such that the object is effectively carved out of 3D checker cubes with the parameter space checkerboard, we provide two texture maps to choose between. Note that these two textures need not be solid textures themselves; we are merely going to choose between them based on the 3D position of the hit point.

⟨*Texture Class Declarations*⟩+≡
```
template <class T> class SolidCheckerboard : public Texture<T> {
public:
    ⟨SolidCheckerboard Interface⟩
private:
    ⟨SolidCheckerboard Private Data⟩
};
```

⟨*SolidCheckerboard Interface*⟩≡
```
SolidCheckerboard(TextureMapping3D *m, Texture<T> *c1,
        Texture<T> *c2) {
    mapping = m;
    tex1 = c1;
    tex2 = c2;
}
```

⟨*Texture Template Method Definitions*⟩+≡
```
template <class T>
T SolidCheckerboard<T>::Evaluate(
        const DifferentialGeometry &dg) const {
    Point dPdx, dPdy;
    Point P = mapping->Map(dg, &dPdx, &dPdy);
    if ((Round(P.x) + Round(P.y) + Round(P.z)) % 2 == 0)
        return tex1->Evaluate(dg);
    return tex2->Evaluate(dg);
}
```

11.8 Noise

In order to write solid textures for complex surface appearances, it is helpful to be able to introduce some controlled variation to the process. Consider a wood plank floor, for example: each plank's color is likely to be slightly different than the others. Or consider a windswept lake; we might want to have waves of similar amplitude across the entire like, but we don't want them to be the same (as they would be if they were constructed from a of sine waves, for example.)

The solution to these sorts of problems came in the form of what has been called a *noise function*. In general, a noise function should be smoothly-varying function defined over $\mathbb{R}^3$, ranging between -1 and 1, but without obviously repeating patterns to it. Equally important, it should be *band limited*, with a maximum frequency of roughly 1, which makes it possible to control their frequency content, so that the patterns it generates don't have any frequencies higher than the pixel sample spacing when the object is projected onto the screen.

47 DifferentialGeometry
21 Point
346 SolidCheckerboard
323 Texture

Many of the noise functions that have been developed are built on the idea of an integer lattice throughout $\mathbb{R}^3$. Some value is associated with each (x,y,z) position in space, where each of *x*, *y*, and *z* are integers. Then, given an arbitrary position in space, the eight adjoining lattice values are found. They are then interpolated in some manner to compute the noise value at the particular point.

A simple example of this is *value noise*. Pseudo-random numbers between -1 and 1 are associated with each lattice point, and actual noise values are computed with trilinear interpolation or with a more complex spline interpolant, which can give a smoother result (by avoiding derivative discontinuities when moving from one lattice cell to another.)

For such a noise function, given an integer (x,y,z) lattice point, we must be able to efficiently compute its parameter value. Because it's infeasible to store values for all possible (x,y,z) points, some cleverness is needed. One option is to use a hash function, where the coordinates are hashed and then used to look up parameters from a fixed-size table of precomputed pseudo-random parameter values.

Here we will implement a noise function introduced by Ken Perlin; as such, it is known as *Perlin noise*. It has a value of zero at all (x,y,z) integer lattice points. Its variation comes from varying gradient vectors at each lattice point that guide the interpolation of a function in between the points. This noise function has many of the desired characteristics of a noise function described above and is reasonably computationally efficient and easy to implement. See Figure 11.7 for a graph of Perlin noise.

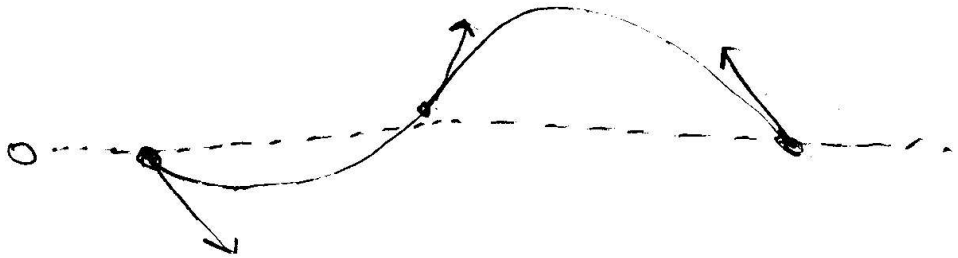


Figure 11.6: generating noise from gradients at integer lattice points

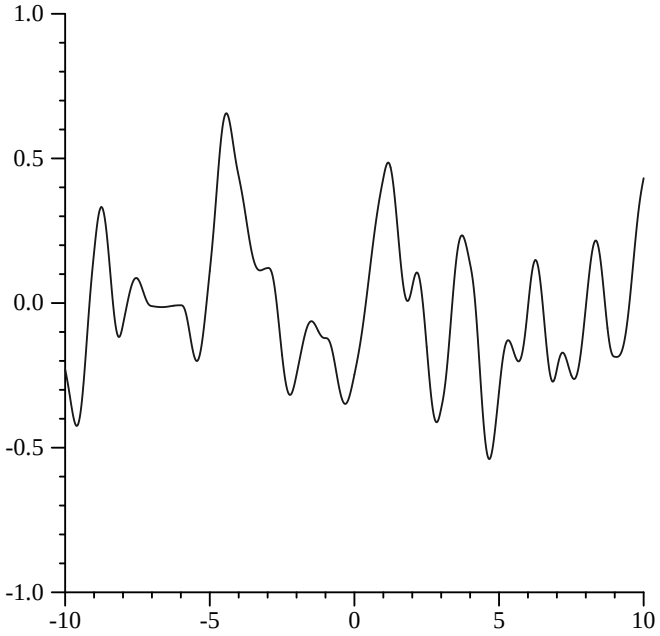


Figure 11.7: Graph of noise function; note that it is smoothly varying, doesn't have unexpected high-frequencies, and ranges between -1 and 1.

A *permutation table* is used to map integer lattice coordinates into an index array of interpolant values. In a pre-process we fill an array of size NOISE_PERM_SIZE with numbers from 0 to NOISE_PERM_SIZE-1 and then randomly shuffle its order. We then make an array of size 2*NOISE_PERM_SIZE that holds the resulting table two times in succession.

Given an integer (x,y,z) lattive coordinate, then, we look up a value in the permutation table as:

```
NoisePerm[NoisePerm[NoisePerm[ix]+iy]+iz];
```

where ix = x % NOISE_PERM_SIZE, and so forth. By doing three permutations in this way, we avoid regularity that might be present if we used NoisePerm[ix+iy+iz], where we'd get the same result if ix and iy were interchanged, etc. By replicating the table twice, we avoid the need to compute modulus values after lookups, like [[(NoisePerm[ix]+iy)

⟨*Perlin Noise Data*⟩≡
```
#define NOISE_PERM_SIZE 256
static int NoisePerm[2 * NOISE_PERM_SIZE] = {
    151, 160, 137, 91, 90, 15, 131, 13, 201, 95, 96,
    53, 194, 233, 7, 225, 140, 36, 103, 30, 69, 142,
    ⟨Noise permutation table⟩
};
```

514 Floor2Int
350 Gradient
21 Point

To evaluate the noise function, we first need to find the eight gradient vectors for the cell the (x,y,z) point is in. Then we just need to do the 3D interpolation.

⟨*Texture Method Definitions*⟩+≡
```
Float Noise(Float x, Float y, Float z) {
    ⟨Initialize eight gradients for this cell⟩
    ⟨Compute gradient weights⟩
    ⟨Compute trilinear interpolation of weights⟩
}
```

We first get the eight gradients for the cell of the point being shaded.

⟨*Initialize eight gradients for this cell*⟩≡
```
int ix = Floor2Int(x);
int iy = Floor2Int(y);
int iz = Floor2Int(z);
const Point &g000 = Gradient(ix,   iy,   iz);
const Point &g100 = Gradient(ix+1, iy,   iz);
const Point &g010 = Gradient(ix,   iy+1, iz);
const Point &g110 = Gradient(ix+1, iy+1, iz);
const Point &g001 = Gradient(ix,   iy,   iz+1);
const Point &g101 = Gradient(ix+1, iy,   iz+1);
const Point &g011 = Gradient(ix,   iy+1, iz+1);
const Point &g111 = Gradient(ix+1, iy+1, iz+1);
```

Given an integer lattice point, we use the permutation table to compute an offset value between 0 and NOISE_PERM_SIZE. We then take the low-order bits of this to get an offset into the table of precomputed gradient directions (so long as

NUM_GRADIENTS and NOISE_PERM_SIZE are powers of two, we can use efficient *and* operations rather than expensive modulus functions.)

⟨*Texture Method Definitions*⟩+≡
```
inline const Point &Gradient(int x, int y, int z) {
    x &= NOISE_PERM_SIZE-1;
    y &= NOISE_PERM_SIZE-1;
    z &= NOISE_PERM_SIZE-1;
    int offset = NoisePerm[NoisePerm[NoisePerm[x]+y]+z];
    return NoiseDirs[offset & (NUM_GRADIENTS-1)];
}
```

The set of gradient vectors is just the twelve vectors from the center of a cube to its edges. The original formulation of Perlin noise also had a precomputed table of pseudo-random gradient directions, though Perlin has more recently suggested that the randomness from the permutation table is enough to remove regularity from the noise function. As a bonus, fewer multiplications are needed in the remainder of the implementation if all gradients have coordinates -1, 0, or 0. Here, we pad the 12 vector table out to 16 entries by repeating a few of them; the savings from being able to do an *and* rather than a modulus to compute which gradient to use makes this a worthwhile trade-off.

NOISE_PERM_SIZE 349
NoisePerm 349
Point 21

⟨*Perlin Noise Data*⟩+≡
```
#define NUM_GRADIENTS 16
static Point NoiseDirs[NUM_GRADIENTS] = {
    Point(1, 1, 0),   Point(-1, 1, 0),  Point(1, -1, 0),
    Point(-1, -1, 0), Point(1, 0, 1),   Point(-1, 0, 1),
    Point(1, 0, -1),  Point(-1, 0, -1), Point(0, 1, 1),
    Point(0, -1, 1),  Point(0, 1, -1),  Point(0, -1, -1),
    Point(1, 1, 0),   Point(-1, 1, 0),  Point(0, -1, 1),
    Point(0, -1, -1)
};
```

Given the eight gradients, we compute each ones contribution at the point being shaded. This is just the dot product of the gradient with the offset vector from the respective integer lattice point to the point being shaded.

⟨*Texture Method Definitions*⟩+≡
```
inline Float NoiseDot(const Point &P, Float x, Float y, Float z) {
    return P.x*x + P.y*y + P.z*z;
}
```

⟨*Compute gradient weights*⟩≡
```
Float dx = x - ix, dy = y - iy, dz = z - iz;
Float w000 = NoiseDot(g000, dx,   dy,   dz);
Float w100 = NoiseDot(g100, dx-1, dy,   dz);
Float w010 = NoiseDot(g010, dx,   dy-1, dz);
Float w110 = NoiseDot(g110, dx-1, dy-1, dz);
Float w001 = NoiseDot(g001, dx,   dy,   dz-1);
Float w101 = NoiseDot(g101, dx-1, dy,   dz-1);
Float w011 = NoiseDot(g011, dx,   dy-1, dz-1);
Float w111 = NoiseDot(g111, dx-1, dy-1, dz-1);
```

Finally, given these eight weights, we want to trilinearly interpolate between them at the point. Rather than interpolating with `dx`, `dy`, and `dz` directly, though, we run each of them through a smoothing function. This ensures that the noise function has first and second derivative continuity as we move from lattice cell to lattice cell.

⟨*Texture Method Definitions*⟩+≡
```
inline Float NoiseWeight(Float t) {
    Float t3 = t*t*t;
    Float t4 = t3*t;
    return 6*t4*t - 15*t4 + 10*t3;
}
```

⟨*Compute trilinear interpolation of weights*⟩≡
```
Float wx = NoiseWeight(dx);
Float wy = NoiseWeight(dy);
Float wz = NoiseWeight(dz);
Float x00 = Lerp(wx, w000, w100);
Float x10 = Lerp(wx, w010, w110);
Float x01 = Lerp(wx, w001, w101);
Float x11 = Lerp(wx, w011, w111);
Float y0 = Lerp(wy, x00, x10);
Float y1 = Lerp(wy, x01, x11);
return Lerp(wz, y0, y1);
```

512 Lerp
350 NoiseDot
323 Texture
326 TextureMapping2D

Random Polka Dots

To show a basic use of the noise function, we'll write a polka-dot texture. This texture divides (s,t) texture space into rectangular cells. Each cell has a 50% chance of having a dot inside of it, where the dot is randomly placed inside the cell.

`PolkaDots` takes the usual 2D mapping function, as well as two `Texture`s, one for the regions of the surface outside of the dots and one for the regions inside.

⟨*PolkaDots Interface*⟩≡
```
PolkaDots(TextureMapping2D *m, Texture<T> *c1, Texture<T> *c2) {
    mapping = m;
    outsideDot = c1;
    insideDot = c2;
}
```

⟨*PolkaDots Private Data*⟩≡
```
Texture<T> *outsideDot, *insideDot;
TextureMapping2D *mapping;
```

The evaluation function is pretty straightforward. We start by taking the (s,t) texture coordinates and computing integer `uCell` and `vCell` values, which give us the coordinates of the cell that we're in. (See Figure 11.8.)

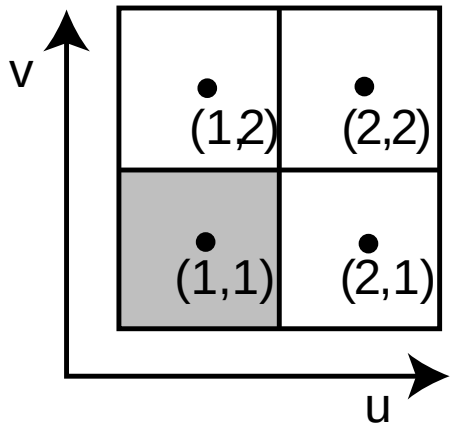


Figure 11.8:

⟨*Texture Template Method Definitions*⟩+≡
```
template <class T>
T PolkaDots<T>::Evaluate(const DifferentialGeometry &dg) const {
    ⟨Compute cell incides for dots⟩
    ⟨Return insideDot result if point is inside dot⟩
    return outsideDot->Evaluate(dg);
}
```

DifferentialGeometry 47

⟨*Compute cell incides for dots*⟩≡
```
Float u, v, dsdx, dtdx, dsdy, dtdy;
mapping->Map(dg, &u, &v, &dsdx, &dtdx, &dsdy, &dtdy);
int uCell = Round(u), vCell = Round(v);
```

Once we know the cell indices, we need to decide if there is a polka dot in the cell. Obviously, this computation needs to be consistent, so that for all times that this routine runs for points in a particular cell, it always returns the same result. On the other hand, we'd like the result to not be regular. Enter noise: we evaluate the noise function at a position that is the same for all points inside this cell– `uCell+.5, vCell+.5`. If this is greater than zero, we decide that there is a dot in the cell and continue processing.

Recall that out noise function always returns zero at integer (x,y,z) coordinates, so we don't want to just evaluate it at `uCell, vCell`. Although the 3D noise function would actually be evaluating noise at `uCell, vCell, .5`, slices through noise with integer values for any of the are not as good as with all of them offset.

If there is a dot in the cell, we use the same trick to randomly shift the center of the dot around; we compute a new dot position using noise to offset it from the center of the cell.

Finally, we just need to decide if the (s,t) coordinates are within distance `radius` of the shifted center. We compute their squared distance to the center and compare it to the squared radius.

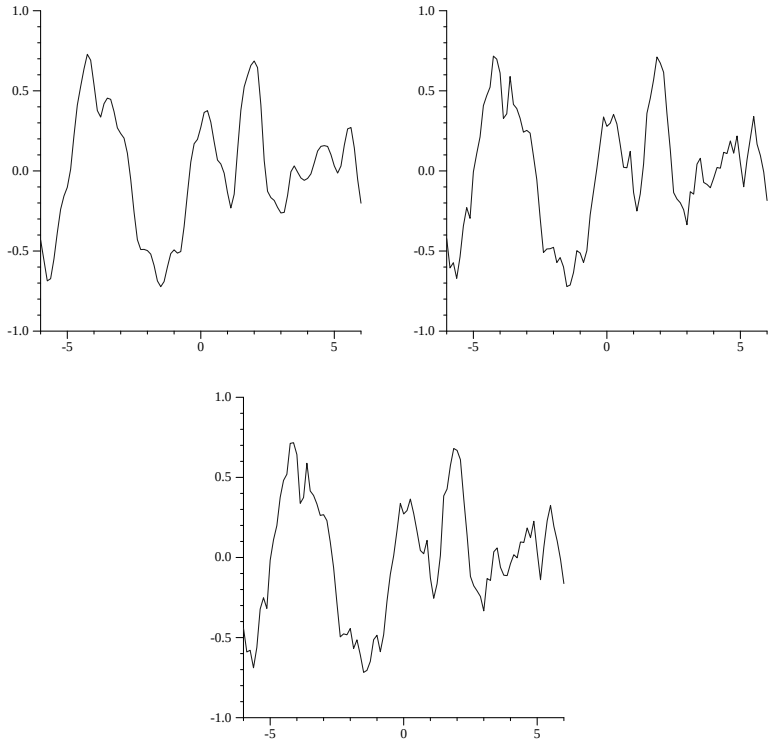


Figure 11.9: 2, 4, 6 ocatves

⟨*Return* `insideDot` *result if point is inside dot*⟩≡
```
if (Noise(uCell+.5, vCell+.5) > 0) {
    Float radius = .35;
    Float maxShift = 0.5 - radius;
    Float uCenter = uCell + maxShift *
        Noise(uCell + 1.5, vCell + 2.8);
    Float vCenter = vCell + maxShift *
        Noise(uCell + 4.5, vCell + 9.8);
    Float du = fabsf(u - uCenter), dv = fabsf(v - vCenter);
    if (du*du + dv*dv < radius*radius)
        return insideDot->Evaluate(dg);
}
```

This texture, like all procedural textures in this chapter, is an *implicit texture*; in other words, the texture function is written to be able to describe the texture at any particular point being shaded–because it does so in a way such that it squares

FBm

⟨*FBmTexture Interface*⟩+≡
```
FBmTexture(int oct, Float roughness, TextureMapping3D *map) {
    omega = roughness;
    octaves = oct;
    mapping = map;
}
```

⟨*FBmTexture Private Data*⟩≡
```
int octaves;
Float omega;
TextureMapping3D *mapping;
```

⟨*Texture Method Definitions*⟩+≡
```
Float FBm(const Point &P, Float omega, int octaves) {
    Float sum = 0., lambda = 1., o = 1.;
    for (int i = 0; i < octaves; ++i) {
        sum += o * Noise(lambda*P.x, lambda*P.y, lambda*P.z);
        lambda *= 1.99f;
        o *= omega;
    }
    return sum;
}
```

⟨*Texture Method Definitions*⟩+≡
```
Float FBmTexture::Evaluate(const DifferentialGeometry &dg) const {
    Point dPdx, dPdy;
    Point P = mapping->Map(dg, &dPdx, &dPdy);
    return FBm(P, omega, octaves);
}
```

DifferentialGeometry 47
Noise 349
Point 21

Windy Waves

A simple application of FBm can give a reasonably convincing representation of ocean waves. This `Texture` is based on two observations. First, that across the surface of a wind-swept lake (for example), some areas are relatively smooth and some are more choppy; this comes from the natural variation of wind's strength from area to area. Second, that the overall form of individual waves on the surface can be well described by FBm.

⟨*Windy Interface*⟩+≡
```
Windy(TextureMapping3D *map) {
    mapping = mapping;
}
```

⟨*Windy Private Data*⟩≡
```
TextureMapping3D *mapping;
```

The evaluation function uses two calls to the `FBm` function. The first scales down the point `P` by a factor of 10; as a result, the first call to `FBm` returns relatively low-frequency variation over the object being shaded. We use this to determine the local strength of the wind. The second call figures out the amplitude of the wave at the particular point, independent of the amount of wind there. The product of these two values gives the actual wave offset for the particular location. Figure 11.10 shows the result.

Figure 11.10: windy waves

⟨*Texture Method Definitions*⟩+≡
```
Float Windy::Evaluate(const DifferentialGeometry &dg) const {
    Point dPdx, dPdy;
    Point P = mapping->Map(dg, &dPdx, &dPdy);
    Point Ps = Point(.1f*P.x, .1f*P.y, .1f*P.z);
    Float windStrength = FBm(Ps, .5f, 3);
    Float waveHeight = FBm(P, .5f, 6);
    return fabsf(windStrength) * waveHeight;
}
```

```
 47 DifferentialGeometry
 21 Point
354 Windy
```

Marble

Further Reading

2D texture mapping with images was first introduced to graphics by Blinn and Newell (BN76).

Feibush et al were the first graphics researchers to investigate a spatially-varing filter function, instead of using a box filter (FLC80).

Norton et al (NRS82).

Crow summed area tables (Cro84).

Williams mipmap (Wil83). Greene and Heckbert EWA (GH86b), Heckbert texture mapping survey (Hec86). Heckbert's MS thesis (Hec89). Landsdale MS thesis (Lan91).

Fournier and Fiume filtering method (FF88). McCormack et al fast and anisotropic (MPFJ99).

Procedural texturing was introduced by Cook (Coo84), Perlin (Per85), and Peachey (Pea85).

Shading languages: Hanrahan and Lawson(HL90), Cook (Coo84), Perlin (Per85). See Ebert et al (EMP$^+$03) and Apodaca and Gritz (AG00) for techniques for writing procedural shaders. The stuff here is similar to the shade tree approach.

Solid texture developed by Gardner (Gar84), Perlin (Per85), Peachey (Pea85).

Peachey's chapter in Texturing and Modeling has a great summary of approaches to noise functions (**?**). Worley developed a wacky new noise function (Wor96). Perlin's paper on the revised noise function (Per02).

Windy shader here based on Musgrave's in texturing and modeling.

Exercises

11.1 texture memory: tiling, not blowing 8-bit TIFFs into full-blown `Spectrum` objects unless necessary.

11.2 feline texture filtering

11.3 Implement plugin shading language to allow user-written programs to compute texture values.

11.4 detect specular highlight aliasing: gauss map, then find maximum value of $\vec{\omega}_h$ inside the spherical triangle–either $(0,0,1)$, at a vertex, or along an edge? Can we be sure that all $\vec{\omega}_h$ will be inside the spherical triangle given by the three points, or is that just going to be good enough?

11.5 shading with closures, multi-point-sample textures and BSDFs..

12. Light Sources

In order to be able to see the scene we're rendering, it's necessary that some of the objects in the scene emit light into the scene. In this chapter, we'll describe the abstract light class, which defines the basic abstractions and interfaces used for light sources in `lrt`. We'll then describe the implementations of a number of useful light sources.

12.1 Light Interface

⟨*light.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef LIGHT_H
#define LIGHT_H
#include "lrt.h"
#include "geometry.h"
#include "transform.h"
#include "color.h"
#include "paramset.h"
#include "mc.h"
⟨Light Classes⟩
⟨AreaLight Classes⟩
#endif // LIGHT_H
```

⟨*light.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "light.h"
#include "scene.h"
⟨Light Method Definitions⟩
```

All lights share three common parameters. First is a boolean that records whether the light should cast shadows or whether it should just illuminate all objects in the scene regardless of whether other objects may block the light. This clearly has no basis in reality, though it can be useful for artistic effect. Second is a transformation that defines the light's coordinate system in terms of the scene's world coordinate system. Just like shapes, it's often handy to be able to write a light's implementation assuming a particular coordinate system (e.g. a spotlight located at the origin of its light space, shining down the $+z$ axis.) Finally, we store the total power emitted by the light.

⟨*Light Classes*⟩≡
```
class Light {
public:
    ⟨Light Interface⟩
protected:
    ⟨Light Protected Data⟩
};
```

⟨*Light Interface*⟩+≡
```
Light(bool shadows, const Transform &l2w,
      const Spectrum &power) {
    CastsShadows = shadows;
    LightToWorld = l2w;
    WorldToLight = LightToWorld.GetInverse();
    Power = power;
}
```

GetInverse 43
Spectrum 155
Transform 32

⟨*Light Protected Data*⟩≡
```
bool CastsShadows;
Transform WorldToLight, LightToWorld;
Spectrum Power;
```

So that the `Integrator`s can compute light reflection at a point on a surface, `Light`s must be able to compute the differential irradiance arriving at a location in the scene due to their illumination. Recall from Section 5.2 that irradiance, E, is the flux density per area; from a point source of flux, it falls off proportionally to the cosine of the angle of incident light with the surface normal of the receiver, and inversely proportional to the squared distance between the two.

There are two versions of this method: one is for area light sources–lights that are defined in terms of emission from a piece of geometry; the other is for delta-function lights–lights that don't have geometry associated with them but are defined in terms of emission from a single point, a single direction, etc. Delta lights are a useful mathematical abstraction, though they don't strictly reflect reality.

Here is the differential irradiance function for area lights; the caller passes in the `Scene` so that the light is able to trace shadow rays if necessary, the local differential geometry of the point being illuminated, and the direction of possible incident illumination that the caller is interested in. If the light isn't visible along that direction from that point, no differential irradiance should be returned. See Figure 12.1.

Figure 12.1: differential irradiance setting

⟨*Light Interface*⟩+≡
```
virtual Spectrum L(const Point &Ps, const Point &Pl,
    VisibilityTester *) const;
```

The second version of the differential irradiance function is for delta light sources. Here, it's necessary that the light source be able to choose the incident direction $\vec{\omega}$; therefore, the caller passes a pointer to the direction vector, which the light must fill in.

⟨*Light Interface*⟩+≡
```
virtual Spectrum dE(const Point &P, const Normal &N,
    Vector *w, VisibilityTester *vis) const = 0;
```

Visibility Testing

23	Normal
21	Point
26	Ray
5	Scene
155	Spectrum
16	Vector

⟨*Light Classes*⟩+≡
```
struct VisibilityTester {
    void SetSegment(const Point &p1, const Point &p2, bool castsShadows) {
        r = Ray(p1, p2-p1, RAY_EPSILON, 1.f - RAY_EPSILON);
        traceRay = castsShadows;
    }
    void SetRay(const Point &p, const Vector &d, bool castsShadows) {
        r = Ray(p, d, RAY_EPSILON);
        traceRay = castsShadows;
    }
    bool Unoccluded(const Scene *scene) const;
    Spectrum Transmittance(const Scene *scene) const;

    Ray r;
    bool traceRay;
};
```

⟨*Light Method Definitions*⟩+≡
```
bool VisibilityTester::Unoccluded(const Scene *scene) const {
    ⟨Update shadow ray statistics⟩
    return !scene->IntersectP(r);
}
```

⟨*Light Method Definitions*⟩+≡
```
Spectrum VisibilityTester::Transmittance(const Scene *scene) const {
    return scene->Transmittance(r);
}
```

Since shadow rays may repreent a significant fraction of overall rendering time, it's useful to keep track of the total number of shadow rays traced.

⟨*Update shadow ray statistics*⟩≡
```
static StatsCounter nShadowRays("Lights",
    "Number of shadow rays traced");
++nShadowRays;
```

We'll provide two methods in the base `Light` class for light implementations to use for tracing shadow rays. The first such method is `Light::visible`; it takes two points in the scene and returns `true` if the two are mutually visible to each other. We start out by skipping the ray test if this light doesn't cast shadows and otherwise construct an appropriate ray and trace it.

The second visibility method takes a point and a direction and checks to see if there is any occlusion along the corresponding ray. This is useful for light sources that are modeled as being infinitely far away from the scene.

12.2 Point Lights

⟨*pointlight.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "light.h"
#include "shapes.h"
⟨PointLight Classes⟩
⟨PointLight Method Definitions⟩
```

Light 358
Scene 5
Spectrum 155
StatsCounter 501
VisibilityTester 359

Now we can present some light source implementations. The `PointLight` base class implements an isotropic point light source; it shines the same amount of light in all directions. However, new types of point lights with more complex light distributions may be derived from this subclass. For example, spotlights will be defined as a sub-class of `PointLight`.

⟨*PointLight Classes*⟩≡
```
class PointLight : public Light {
public:
    ⟨PointLight Methods⟩
private:
    ⟨PointLight Private Data⟩
};
```

`PointLight`s are positioned at the origin in light space; to place them elsewhere, the world-to-light transform should be adjusted with an additional translation as appropriate. We precompute the world-space position of the light in the constructor by transforming $(0,0,0)$ from light space to world space and precompute the intensity for an isotropic point source as well.

⟨*PointLight Method Definitions*⟩≡
```
PointLight::PointLight(bool shadows, const Transform &light2world,
        const Spectrum &intensity)
    : Light(shadows, light2world, intensity * 4. * M_PI) {
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
}
```

⟨*PointLight Private Data*⟩≡
```
Point lightPos;
Spectrum Intensity;
```

Point lights are defined in terms of their radiant intensity. For an isotropic point light, the radiant intensity is constant and independent of direction.

For point lights, the differential irradiance is not defined as a function of direction; only a version of that method that returns a direction is allowed. We start by computing the incident direction $\vec{\omega}$ and normalizing it. Next we check for occlusion between the light and the point being illuminated; if the two are visible to each other, we compute incident differential irradiance.

⟨*PointLight Method Definitions*⟩+≡
```
Spectrum PointLight::dE(const Point &P, const Normal &N,
        Vector *w, VisibilityTester *visibility) const {
    *w = (lightPos - P).Hat();
    visibility->SetSegment(P, lightPos, CastsShadows);
    return Intensity * fabs(Dot(*w, N)) /
        DistanceSquared(lightPos, P);
}
```

358 CastsShadows
23 DistanceSquared
19 Hat
358 Light
358 LightToWorld
23 Normal
21 Point
360 PointLight
155 Spectrum
32 Transform
16 Vector
359 VisibilityTester

Spot Light

⟨*spotlight.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "light.h"
#include "shapes.h"
⟨SpotLight Classes⟩
⟨SpotLight Method Definitions⟩
```

⟨*SpotLight Classes*⟩≡
```
class SpotLight : public Light {
public:
    SpotLight(bool shadows, const Transform &light2world,
        const Spectrum &, Float width, Float fall);
    Spectrum dE(const Point &P, const Normal &N, Vector *w,
        VisibilityTester *vis) const;
    ⟨SpotLight Member Functions⟩
private:
    ⟨SpotLight Private Data⟩
};
```

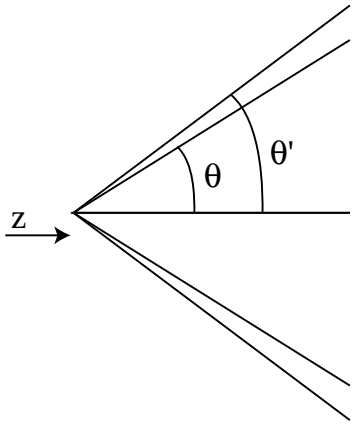


Figure 12.2: Spotlights are defined by two angles, *falloffStart* and *totalWidth*. Objects inside the inner cone of angles, up to *falloffStart* are fully illuminated by the light. The directions between *falloffStart* and *totalWidth* are a transition zone that ramps down from full illumination to no illumination, such that points outside the *totalWidth* cone aren't illuminated at all. The cosine of the angle between the vector to a point p and the spotlight axis, $\cos\theta$, can easily be computed with a dot product.

Light 358
LightToWorld 358
Point 21
Radians 514
Spectrum 155
SpotLight 361
Transform 32

Spot lights are a handy variation on point lights; rather than shining illumination in all directions, they light objects in a cone of directions from their position. For simplicity, we will define the spotlight in the light coordinate system to always be at the position $(0,0,0)$, pointing down the $+z$ axis. To place or orient it elsewhere in the scene, the `WorldToLight` matrix can be set appropriately.

Two angles are passed in the constructor to set the extent of the `SpotLight`'s cone: the overall angular width of the cone, and the angle at which fall-off from full illumination to no illumination starts; see Figure 12.2. We precompute and store the cosines of these angles in the spotlight object, for efficiency when computing illumination later.

⟨*SpotLight Method Definitions*⟩≡
```
SpotLight::SpotLight(bool shadows, const Transform &light2world,
        const Spectrum &intensity, Float width, Float fall)
    : Light(shadows, light2world, intensity) {
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
    cosTotalWidth = cosf(Radians(width));
    cosFalloffStart = cosf(Radians(fall));
}
```

⟨*SpotLight Private Data*⟩≡
```
Float cosTotalWidth, cosFalloffStart;
Point lightPos;
Spectrum Intensity;
```

`SpotLight` is a sub-class of `PointLight`, so the only method we need to implement is `SpotLight::I`, which gives the intensity in a particular direction `w`. We start by computing the cosine of the angle between the outgoing direction and the

$+z$ axis; we can compare this to the cosines of the falloff and overall width angles to see where the point lies with respect to the spot light cone. To compute the cosine of the offset angle to a point p, we have (see Figure 12.2):

$$\begin{aligned}\cos\theta &= \widehat{p-(0,0,0)}\cdot(0,0,1)\\ &= \mathbf{p}_z/\|\vec{\mathbf{p}}\|\end{aligned}$$

We can trivally determine that points with a cosine greater than the cosine of the falloff angle are inside the cone receiving full illumination, and points with cosine less than the width angle's cosine are completely outside the cone.

⟨*SpotLight Method Definitions*⟩+≡
```
Spectrum SpotLight::dE(const Point &P, const Normal &N, Vector *w,
        VisibilityTester *visibility) const {
    *w = (lightPos - P).Hat();
    visibility->SetSegment(P, lightPos, CastsShadows);
    return Intensity * Falloff(-*w) * fabs(Dot(*w, N)) /
        DistanceSquared(lightPos, P);
}
```

⟨*SpotLight Method Definitions*⟩+≡
```
Float SpotLight::Falloff(const Vector &w) const {
    Vector wl = WorldToLight(w).Hat();
    Float costheta = wl.z;
    if (costheta < cosTotalWidth)
        return 0.;
    if (costheta > cosFalloffStart)
        return 1.;
    ⟨Compute falloff inside spotlight cone⟩
}
```

358 CastsShadows
362 cosFalloffStart
362 cosTotalWidth
23 DistanceSquared
19 Hat
23 Normal
21 Point
155 Spectrum
361 SpotLight
16 Vector
359 VisibilityTester
358 WorldToLight

For points inside the transition range, we determine how far it is along between the start of falloff and the end, and arbitrarily scale the intensity accordingly.

⟨*Compute falloff inside spotlight cone*⟩≡
```
Float delta = (costheta - cosTotalWidth) /
    (cosFalloffStart - cosTotalWidth);
return delta*delta*delta*delta;
```

Texture Projection Light

⟨*projectionlight.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "light.h"
#include "shapes.h"
#include "texture.h"
⟨ProjectionLight Classes⟩
⟨ProjectionLight Method Definitions⟩
```

⟨*ProjectionLight Classes*⟩≡
```
class ProjectionLight : public Light {
public:
    ProjectionLight(bool shadows, const Transform &light2world,
        const Spectrum &intensity, Texture<Spectrum> *tex,
        Float fov);
    ~ProjectionLight();
    Spectrum dE(const Point &P, const Normal &N, Vector *w,
        VisibilityTester *vis) const;
    ⟨ProjectionLight Member Functions⟩
private:
    ⟨ProjectionLight Private Data⟩
};
```

Another useful light source acts like a slide projector: it takes a texture map and projects its image out into the scene. We use a projective transformation to project lights in the scene onto a projection plane; see Figure 12.3. A field of view value is given with the light so that the constructor can compute an appropriate matrix.

⟨*ProjectionLight Method Definitions*⟩+≡
```
ProjectionLight::ProjectionLight(bool shadows,
        const Transform &light2world,
        const Spectrum &intensity, Texture<Spectrum> *tex,
        Float fov)
    : Light(shadows, light2world, intensity) {
    projectionMap = tex;
    hither = RAY_EPSILON;
    yon = 1e10;
    lightProjection = Perspective(fov, hither, yon);
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
    ⟨Cache ProjectionLight diagonal fov for sampling⟩
}
```

Light 358
LightToWorld 358
Normal 23
Perspective 181
Point 21
Spectrum 155
Texture 323
Transform 32
Vector 16
VisibilityTester 359

⟨*ProjectionLight Private Data*⟩≡
```
Texture<Spectrum> *projectionMap;
Transform lightProjection;
Float hither, yon;
Point lightPos;
Spectrum Intensity;
```

Because `ProjectionLight` also inherits from `PointLight`, we just need to override its `I` method.

⟨*ProjectionLight Method Definitions*⟩+≡
```
Spectrum ProjectionLight::dE(const Point &P, const Normal &N, Vector *w,
        VisibilityTester *visibility) const {
    *w = (lightPos - P).Hat();
    visibility->SetSegment(P, lightPos, CastsShadows);
    return Intensity * Projection(-*w) * fabs(Dot(*w, N)) /
        DistanceSquared(lightPos, P);
}
```

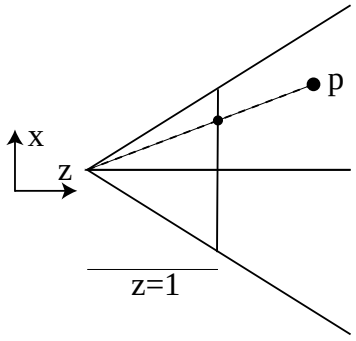


Figure 12.3: The basic setting for projection light sources. A point p in the scene can be projected on to the plane of the projected image by dividing each of its coordinates by its z coordinate, giving a point with $z = 1$. We can then use the x and y coordinates to index into a texture.

⟨*ProjectionLight Method Definitions*⟩+≡
```
Spectrum ProjectionLight::Projection(const Vector &w) const {
    Vector wl = WorldToLight(w);
    ⟨Discard directions behind projection light⟩
    ⟨Project point on to projection plane⟩
    ⟨Compute projected light for direction⟩
}
```

We immediately discard projection points that are behind the hither and plane for the projection. Because the projective transformation has the unfortunate property that it projects points behind the center of projection to points in front of it, is important in particular to discard points with a negative z value. Otherwise, given a projected point, we wouldn't be able to know if it was originally behind the light (and not illuminated) or in front of it.

⟨*Discard directions behind projection light*⟩≡
```
if (wl.z < hither) return 0.;
```

After projecting the point to the projection plane, points with coordinate values between ± 1 are inside the projection window. We then offset and scale them to get (s,t) texture coordinates inside $[0,1]^2$ to use when evaluating the projection texture map.

⟨*Project point on to projection plane*⟩≡
```
Point Pl = lightProjection(Point(wl.x, wl.y, wl.z));
if (Pl.x < -1 || Pl.x > 1 || Pl.y < -1 || Pl.y > 1) return 0.;
Float s = (Pl.x + 1) * 0.5f;
Float t = (Pl.y + 1) * 0.5f;
```

We can now just go ahead and evaluate the texture map, setting up a fake `DifferentialGeometry` for it to use.

⟨*Compute projected light for direction*⟩≡
```
DifferentialGeometry dgMap(Point(Pl.x, Pl.y, 0), Vector(1,0,0),
    Vector(0,1,0), Vector(0,0,0), Vector(0,0,0), s, t, NULL);
return projectionMap->Evaluate(dgMap);
```

358 CastsShadows
47 DifferentialGeometry
23 DistanceSquared
19 Hat
364 lightProjection
23 Normal
21 Point
361 PointLight::lightPos
364 ProjectionLight
364 projectionMap
155 Spectrum
16 Vector
359 VisibilityTester
358 WorldToLight

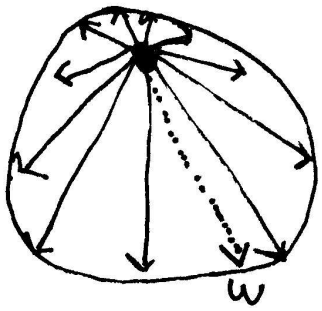


Figure 12.4: An example of a goniometric diagram specifying an outgoing light distribution from a point light source (in 2D). The intensity for a given outgoing direction $\vec{\omega}$ is found by interpolating the intensities of the adjacent samples.

Goniometric diagram lights

⟨*goniometric.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "light.h"
#include "shapes.h"
#include "scene.h"
#include "texture.h"
⟨GoniometricLight Classes⟩
⟨GoniometricLight Method Definitions⟩
```

Light 358
Spectrum 155
Transform 32

The *goniometric diagram* describes the distribution of luminance rom a point light source; widely used in illumination engineering to characterize lights. Here, we'll implement a light source that uses goniometric diagrams encoded in texture maps to describe the emission distribution of the light.

The implementation is very similar to the point light sources defined previously in this section; we just scale the intensity based on outgoing direction according to the goniometric diagram's values. Figure 12.4 shows an example in two dimensions.

⟨*GoniometricLight Classes*⟩≡
```
class GoniometricLight : public Light {
public:
    GoniometricLight(bool shadows, const Transform &light2world,
        const Spectrum &, const string &texname);
    ⟨GoniometricLight Member Functions⟩
private:
    ⟨GoniometricLight Private Data⟩

};
```

⟨*GoniometricLight Private Data*⟩≡
```
Point lightPos;
Spectrum Intensity;
ImageMap *imageMap;
```

Goniometric diagrams are usually defined in a coordinate space where the *y* axis is up, so we'll swap *y* and *z* before using the spherical coordiantes functions...

⟨*GoniometricLight Member Functions*⟩+≡
```
Float Scale(const Vector &w) const {
    Vector wp = w;
    swap(wp.y, wp.z);
    Float theta = SphericalTheta(wp);
    Float phi = SphericalPhi(wp);
    Float val = 1;
    if (imageMap) imageMap->Lookup(theta / M_PI, phi / (2.f*M_PI),
        0, 0, 0, 0, &val);
    return val;
}
```

12.3 Infinite Point Lights

⟨*distantlight.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "light.h"
#include "shapes.h"
#include "scene.h"
⟨InfinitePointLight Classes⟩
⟨InfinitePointLight Method Definitions⟩
```

⟨*InfinitePointLight Classes*⟩≡
```
class InfinitePointLight : public Light {
public:
    ⟨InfinitePointLight Methods⟩
private:
    ⟨InfinitePointLight Private Data⟩
};
```


19 Hat
334 ImageMap
368 InfinitePointLight::lightDir
358 Light
358 LightToWorld
335 Lookup
21 Point
36 Scale
155 Spectrum
164 SphericalPhi
164 SphericalTheta
513 swap
32 Transform
16 Vector

Another light source type is a *directional light*. It describes an emitter where at every point in space, illumination arrives from the same direction. Light sources like the sun (as considered from earth) can be thought of as directional light sources—though they are actually point or area light sources, because they're so far away, the illumination effectively arrives in parallel beams.

⟨*InfinitePointLight Method Definitions*⟩≡
```
InfinitePointLight::InfinitePointLight(bool shadows,
        const Transform &light2world, const Spectrum &radiance,
        const Vector &dir)
    : Light(shadows, light2world, radiance) {
    lightDir = -LightToWorld(dir).Hat();
    L = radiance;
}
```

⟨*InfinitePointLight Private Data*⟩≡
```
Vector lightDir;
Spectrum L;
```

Now the method for differential irradiance. Directional lights don't quite fit in with our previous decision to characterize lights in terms of their total power. Interestingly enough, the total power emitted by a directional light is proportional to the area of the scene receiving light. (Alternatively, a directional light could be thought of as having *infinite* power, since a scene of infinite extent would receive infinite energy, though this mostly illustrates a break down of the abstraction.) Therefore, we interpret the power value as the amount of emitted radiance along a ray from the directional light. Note that that we are using the `Unoccluded()` method instead of the `visible` method for tracing shadow rays.

⟨*InfinitePointLight Method Definitions*⟩+≡
```
Spectrum InfinitePointLight::dE(const Point &P,
        const Normal &N, Vector *w, VisibilityTester *visibility) const {
    *w = lightDir;
    visibility->SetRay(P, *w, CastsShadows);
    return L * fabs(Dot(*w, N));
}
```

CastsShadows 358
InfinitePointLight 367
Light 358
Normal 21
Point 21
primitives 579
Spectrum 155
Vector 16
VisibilityTester 359

12.4 Area Lights

⟨*area.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "light.h"
#include "primitives.h"
⟨AreaLight Function Definitions⟩
```

⟨*AreaLight Classes*⟩≡
```
class AreaLight : public Light {
public:
    ⟨AreaLight Interface⟩
protected:
    ⟨AreaLight Protected Data⟩
};
```

We'll now start to provide some functionality for *area lights*; these are associated with geometry in the scene that emit light. We calculate and store the area of the light source when it is defined, because these area calculations are quite expensive. Computation methods for surface area are described in Chapter 3.

⟨*AreaLight Function Definitions*⟩≡
```
AreaLight::AreaLight(bool shadows, const Transform &light2world,
        const Spectrum &power, const Reference<Shape> &s)
    : Light(shadows, light2world, power) {
    shape = s;
    area = shape->Area();
}
```

⟨*AreaLight Protected Data*⟩≡
```
Reference<Shape> shape;
Float area;
```

We provide two methods unique to area lights. The first evaluates the area light's emitted radiance (usually denoted in formulae by L) at a point on the surface of the light for a given direction. We assume that the given point is on the surface of the light.

For the basic area lights here, the amount of radiance emitted is the same at all points on the light and the same for all outgoing directions. (More generally, emission may vary depending on both of these values.) We can compute emitted irradiance by dividing flux by the surface area (because emission is constant over the surface); dividing this by π, the area of the hemisphere with projected solid angle measure, gives radiance in a particular direction.

⟨*AreaLight Interface*⟩+≡
```
virtual Spectrum L(const Point &x, const Vector &w) const {
    return Power/(area * M_PI);
}
```

It's also handy to be able to compute emitted irradiance (often called *radiosity*) at a point on the light. Here we also assume that the point `x` is on the light's surface and that the light's emission doesn't vary by location.

⟨*AreaLight Interface*⟩+≡
```
virtual Spectrum B(const Point &x) const {
    return Power/area;
}
```

Finally, we give the most important interface for area lights: the differential irradiance.

⟨*AreaLight Function Definitions*⟩+≡
```
Spectrum AreaLight::L(const Point &Ps, const Point &Pl,
        VisibilityTester *visibility) const {
    DifferentialGeometry hit;
    Float thit;
    Vector w = Pl - Ps;
    Ray ray(Ps, w);
    if (shape->Intersect(ray, &thit, &hit)) {
        visibility->SetSegment(Ps, Pl, CastsShadows);
        return L(Pl, -w);
    }
    return 0.;
}
```

We will add a method to the `Surf` class that makes it easy to compute the emitted radiance at a surface point. It returns the value from the `AreaLight::L` method if the primitive is an area light, or 0 otherwise.

368 AreaLight
359 AreaLight::L
358 CastsShadows
47 DifferentialGeometry
358 Light
21 Point
358 Power
26 Ray
509 Reference
155 Spectrum
32 Transform
16 Vector
359 VisibilityTester

⟨*Surf Method Definitions*⟩+≡
```
Spectrum Surf::Le(const Vector &wo) const {
    if (primitive->areaLight)
        return primitive->areaLight->L(dgGeom.P, wo);
    else return Spectrum(0.);
}
```

Multi-Sample Area Lights

It's often useful to be able to flag area lights for extra sampling; e.g. a large light might need many samples to get smooth soft shadows, while a small one would need just a few.

A convenient way to make this possible while keeping the implementation of the `Integrator`s simple

⟨*AreaLight Classes*⟩+≡
```
class MultiAreaLight : public Light {
public:
    ⟨MultiAreaLight Methods⟩
private:
    ⟨MultiAreaLight Private Data⟩
};
```

⟨*AreaLight Protected Data*⟩+≡
```
friend class MultiAreaLight;
```

⟨*MultiAreaLight Methods*⟩≡
```
MultiAreaLight(AreaLight *p, int nSamples)
    : Light(p->CastsShadows, p->LightToWorld, p->Power) {
    parent = p;
    scale = 1.f / (Float)nSamples;
}
```

⟨*MultiAreaLight Private Data*⟩≡
```
AreaLight *parent;
Float scale;
```

⟨*MultiAreaLight Methods*⟩+≡
```
Spectrum Le(const RayDifferential &r) const;
Spectrum L(const Point &x, const Vector &w) const;
Spectrum B(const Point &x) const;
Spectrum L(const Point &Ps, const Point &Pl,
    VisibilityTester *visibility) const;
Spectrum dE(const Point &P, const Normal &N,
    Vector *w, VisibilityTester *vis) const;
```

areaLight 581
AreaLight 368
CastsShadows 358
dgGeom 10
Light 358
LightToWorld 358
Normal 23
Point 21
Power 358
RayDifferential 26
Spectrum 155
Surf 10
Vector 16
VisibilityTester 359

Figure 12.5: Uffizi latlong map

12.5 Infinite Area Lights

⟨*infinitelight.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "light.h"
#include "texture.h"
#include "shapes.h"
#include "scene.h"
⟨InfiniteAreaLight Classes⟩
⟨InfiniteAreaLight Function Definitions⟩
```
358 Light

⟨*InfiniteAreaLight Classes*⟩≡
```
class InfiniteAreaLight : public Light {
public:
    ⟨InfiniteAreaLight Interface⟩
    ~InfiniteAreaLight();
private:
    ⟨InfiniteAreaLight Private Data⟩
};
```

Another useful kind of light is the infinite area light. This is an area light source at infinity that surrounds the entire scene; one good way to visualize it is as an enormous sphere that casts light into the scene from every direction. One use of infinite area lights is *environment lighting*: given a representation of illumination in some environment, synthetic objects can be lit as if they were in that environment. A widely-used representation for light for this application is the latitude-longitude radiance map; it stores emitted radiance as a function of direction. A lat-long environment map of the Uffizi Gallery in Florence is shown in Figure 12.5; a teapot illuminated by the illumination from this map is shown in Figure 12.6.

Like the other lights, the `InfiniteAreaLight` takes a transformation matrix; here its use is to orient the texture map. We use spherical coordinates to map from directions on the sphere to (θ, ϕ) directions from from there to (u, v) texture coordinates; the transformation describes which direction is "up".

Figure 12.6: Teapot in Uffizi environment

⟨*InfiniteAreaLight Function Definitions*⟩+≡
```
InfiniteAreaLight::InfiniteAreaLight(bool shadows,
        const Transform &light2world,
        const Spectrum &power, Texture<Spectrum> *tex)
    : Light(shadows, light2world, power) {
    radianceMap = tex;
}
```

⟨*InfiniteAreaLight Private Data*⟩≡
```
Texture<Spectrum> *radianceMap;
```

Like directional lights, the total power from the infinite area light is related to the surface area of the scene. Therefore, here we also treat the power as the radiance.

⟨*Compute infinite light radiance for this direction*⟩≡
```
Spectrum L = Power;
if (radianceMap != NULL) {
    Vector wh = WorldToLight(w).Hat();
    Vector S, T;
    CoordinateSystem(wh, &S, &T);
    Float s = SphericalPhi(wh) / (2.f*M_PI);
    Float t = SphericalTheta(wh) / M_PI;
    DifferentialGeometry dgLight(Point(wh.x, wh.y, wh.z),
        S, T, Vector(0,0,0), Vector(0,0,0), s, t, NULL);
//  dgLight.ComputeDifferentials(r);
    L *= radianceMap->Evaluate(dgLight);
}
```

Because infinite area lights need to be able to contribute radiance to rays that don't hit any geometry in the scene, we'll add a method to the base `Light` class that returns emitted radiance due to that light along a ray that didn't hit anything in the scene.

XXX how does this change/become better integrated when we have support for volumetric stuff?? XXX

ComputeDifferentials 306
CoordinateSystem 21
DifferentialGeometry 47
Hat 19
InfiniteAreaLight 371
Light 358
Point 21
Spectrum 155
SphericalPhi 164
SphericalTheta 164
Texture 323
Transform 32
Vector 16
WorldToLight 358

⟨*Light Method Definitions*⟩+≡
```
Spectrum Light::Le(const RayDifferential &) const {
    return Spectrum(0.);
}
```

The `InfiniteAreaLight`'s implementation of this can reuse the fragment from its `dE` method.

⟨*InfiniteAreaLight Function Definitions*⟩+≡
```
Spectrum InfiniteAreaLight::Le(const RayDifferential &r) const {
    Vector w = r.D;
    ⟨Compute infinite light radiance for this direction⟩
    return L;
}
```

Further Reading

Warn developed early models of light sources with non-isotropic emission distributions (War83). More recently, Barzel has described a highly parameterized model for light sources, including many controls for controlling rate of falloff, the area of space that is illuminated, etc. Bjorke has developed flexible controls for controlling illuminaton for artistic effect (Bjorke 01 renderman course notes). (The Barzel and Bjorke approaches are not physically based, however.)

371 InfiniteAreaLight
358 Light
26 RayDifferential
155 Spectrum
16 Vector

Blinn and Newell first introduced the idea of environment maps and their use for simulating illumination (BN76), though they only considered illumination of specular objects. Greene also developed these ideas, considering anti-aliasing and different representations for environment maps (Gre86).

Miller and Hoffman first considered using environment maps to illuminate objects with diffuse BRDFs (MH84). Debevec later extended this work (Deb98).

As for efficient ray-tracing, lights are special in that we don't care about the geometric details of intersection, just whether or not there is one along a given ray. Beyond the `IntersectP()` stuff we already do, light buffer (HG86), shaft culling (HW94). Minkowski sum to effectively expand primitives (or bounds of primitives) in scene so that intersecting one ray against primitives tells if any of a collection of rays might have intersected the actual primitives (Luk01).

XXX Mention ways of gathering up bundles of rays XXX

Hart et al generalize light shadow cache, find blockers and clip light source geometry against them (HDG99).

Exercises

12.1 depth-mapped shadows for lights. Williams (Wil78), Reeves et al (RSC87).

12.2 light cache to accelerate shadow rays

12.3 Volumetric ambient light that varies with `x` or `w`

13. Volume Scattering

Until now, `lrt` has been described under the assumption that the scene is a collection of surfaces in a vacuum; this allowed us to assume that radiance was unchanging along rays between surfaces. There are many situations where this assumption is inaccurate: fog and smoke attenuate and scatter light that passes through them, for example, and scattering by particles in the atmosphere is what makes the sky blue and sunsets red.

This chapter introduces the mathematical description of the effects that operate on light as it passes through *participating media*; simulating these effects allows us to render images with effects including atmospheric haze, beams of light through clouds, light passing through cloudy water, and subsurface scattering, which describes scattering from objects where light exits the object at a different place than it enters (accounting for this effect is important for realistic rendering of translucent materials, marble, and human skin). These effects are all caused by particles suspended in a 3D region of space. Their effect on radiance depends on both the density of particles and their composition.

This chapter first describes the basic physical processes that change radiance along rays passing through participating media. We will then describe a basic interface for modeling different types of media, the `VolumeRegion` base-class, and provide implementations of a number of useful representations. These `VolumeRegion`s will be used by `VolumeIntegrator`s, which account for light interactions in participating media and will be described and implemented in Section 15.9.

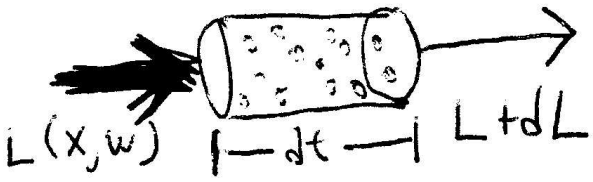


Figure 13.1:

13.1 Volume Scattering Processes

There three main processes that affect the distribution of radiance in an environment passing through participating media.

- The first is absorption, which describes the reduction in radiance passing from one point to another due to the absorption of energy (i.e. its conversion to another form of energy, such as heat).
- Second is emission, which describes energy that is added to the environment from luminous particles.
- The last is scattering, which describes how light heading in one direction scattered to different directions due to collisions with particles.

The charactersitics of all of these properties may be *homogeneous* or *inhomogeneous*. Homogeneous properties are constant throughout the medium being considered, while inhomogeneous properties may vary arbitrarily throughout it.

Absorption

The first type of interaction that we will discuss is *absorption*. Consider thick black smoke from a fire: the smoke obscures the light from objects behind it due to absorption by the black smoke particles. The thicker the smoke, the less one can see of what is behind it.

Absorption is described by the *absorption cross-section*, σ_a; it is the probability that light is absorbed per unit distance travelled in the medium. In general, it may vary by both position x and direction $\vec{\omega}$, though it is normally just a function of position. It is also in general a spectrally-varying quantity. Figure 13.1 shows the effect of absorption along a differential length of a ray. The ray is carrying an amount of radiance L as it enters a differential volume. Particles in the volume absorb some of the radiance and $\mathrm{L} + \mathrm{dL}$ is the amount that exits.

The change in radiance along differential ray length dt is described by the differential equation

$$\mathrm{dL}(x, \vec{\omega}) = -\sigma_a(x, \vec{\omega})\mathrm{L}(x, \vec{\omega})\mathrm{d}t.$$

For a ray passing along a non-differential distance between two points x and x' in direction $\vec{\omega} = x' \hat{-} x$ with distance d between x and x', the differential equation can be solved to give the integral equation

$$e^{-\int_0^d \sigma_a(x+t\vec{\omega},\vec{\omega})\mathrm{d}t},$$

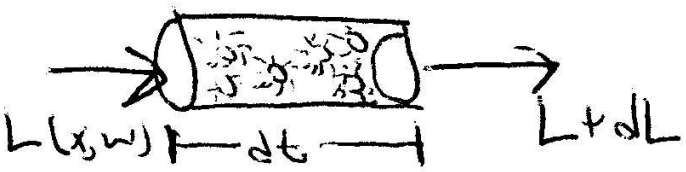


Figure 13.2: emission

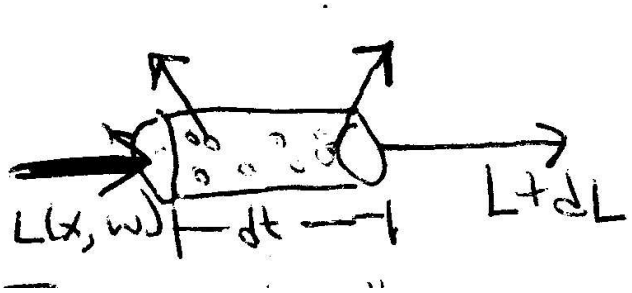


Figure 13.3: outscatter

Emission

In `lrt`, we will assume that emission in the volume is a given property of the scene and that we can easily compute emitted radiance at any point in any direction. Various chemical and thermal processes (or nuclear processes, e.g. in the case of the sun), convert energy into visible wavelengths of light which illuminate the environment. Figure 13.2 shows emission in a differential volume. If we denote emitted radiance at a point in a volume x in a direction $\vec{\omega}$ by $\mathrm{L_e}(x,\vec{\omega})$, then the change in radiance due to emission is

$$\mathrm{dL}(x,\vec{\omega}) = -\sigma_s(x,\vec{\omega})\mathrm{L}(x,\vec{\omega})\mathrm{d}t.$$

Out-scattering and extinction

The third basic light interaction is *scattering*. As a beam of radiance propagates through a medium, it may collide with particles in the medium and be scattered into different directions (see Figure 13.3). The probability of such a scattering event occurring per unit distance is given by the scattering coefficient, σ_s. Similarly to the attenuation coefficient, the change in radiance along a differential length d$t$is given by

$$\mathrm{dL}(x,\vec{\omega}) = -\sigma_s(x,\vec{\omega})\mathrm{L}(x,\vec{\omega})\mathrm{d}t.$$

The total reduction in radiance due to absorption and out-scattering, then, is given by $\sigma_a + \sigma_s$,

$$\mathrm{dL}(x,\vec{\omega}) = -(\sigma_a(x,\vec{\omega}) + \sigma_s(x,\vec{\omega}))\mathrm{L}(x,\vec{\omega})\mathrm{d}t.$$

The sum of these terms is denoted by the attenuation coefficient σ_t,

$$\sigma_t(x,\vec{\omega}) = \sigma_a(x,\vec{\omega}) + \sigma_s x\vec{\omega}$$

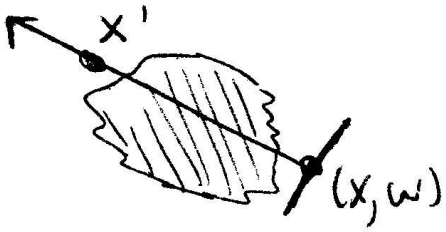


Figure 13.4: beam transmittance

Figure 13.5: mult beam transmittance

Given the attenuation coefficient σ_t, the differential equation can be solved to find the *beam transmittance*

$$T_r(x \to x') = e^{-\int_0^d \sigma_t(x+t\vec{\omega},\vec{\omega})\mathrm{d}t},$$

where T_r denotes the beam transmittance between x and x'. (This quantity is also often called the extinction.)

Thus, if reflected radiance from a point on a surface in a given direction is given by $\mathrm{L}(x,\vec{\omega})$, after accounting for extinction, the radiance at another point $x' = x + d\vec{\omega}$ in direction $\vec{\omega}$ is

$$T_r(x \to x')\mathrm{L}(x,\vec{\omega}).$$

(See Figure 13.4.)

Two useful basic properties of beam transmittance are $T_r(x \to x) = 1$, and in a vacuum, $T_r(x \to x') = 1$ for all x'. Another important property, true in all media, is

$$T_r(x \to x'') = T_r(x \to x')T_r(x' \to x''),$$

for all points x' between x and x''. (See Figure 13.5.) This is a useful property for volume scattering implementations, since it allows us to incrementally compute transmittance at many points along a ray while only needing to find the product of the previously-computed transmittance with the additional reduction.

The exponentiated term in T_r is called the *optical thickness* between the two points. It is denoted by the symbol τ:

$$\tau(x \to x') = \int_0^d \sigma_t(x + t\vec{\omega})\mathrm{d}t.$$

In a homogeneous medium, σ_t, is a constant, and *Beer's law* describes the attenuation.

$$T_r = e^{-\sigma_t d},$$

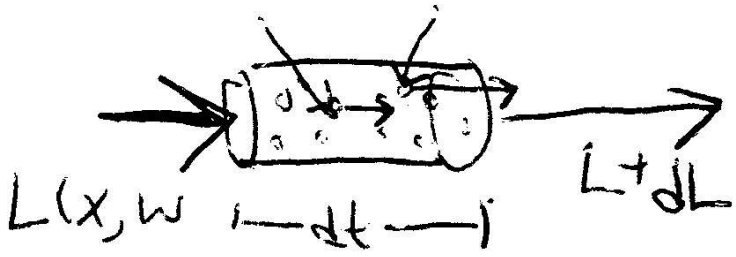


Figure 13.6: inscatter

follows directly.

It is often useful to be able to characterize the fraction of light attenuated due to scattering with respect to the total attenuation. This is given by the *albedo* α, which ranges from zero to one.

$$\alpha = \frac{\sigma_s}{\sigma_a + \sigma_s} = \frac{\sigma_s}{\sigma_t}$$

It gives the fraction of light that is redistributed at each scattering event.

In media with low albedos, roughly $\alpha < 0.5$, absorption is the dominant process, while in high albedo media, roughly $\alpha > 0.5$, scattering is the main determinant of the final radiance distribution.

In-scattering

While out-scattering reduces radiance along a ray due to scattering in different directions, *in-scattering* accounts for increased radiance due to radiance from other directions; see Figure 13.6. Under the assumption that the individual particles that cause these scattering events are separated by a few times the lengths of their radii, it is possible to ignore interactions between these particles when describing about scattering at some location (van81). Under these assumptions, the *phase function*, $p(\vec{\omega} \to \vec{\omega}')$, is a function of the two directions and describes the angular distribution of scattered radiation at a point. It is the volumetric analog to the BSDF.

Phase functions are defined so that they are normalized so that for all $\vec{\omega}$,

$$\frac{1}{4\pi} \int_{S^2} p(\vec{\omega} \to \vec{\omega}')\, d\vec{\omega}' = 1. \tag{13.1.1}$$

In most naturally-occuring media, the phase function is a function of the angle between the two directions $\vec{\omega}$ and $\vec{\omega}'$; such media are called isotropic and these phase functions are often written as $p(\cos\theta)$. In exotic media, such as those with crystalline-type structure, the phase function is a function of the values of each of the two angles, though this is much less common. An important property of naturally-occurring phase functions is that they are *reciprocal*: the two directions can be interchanged and the phase function's value remains unchanged.

Phase functions can be isotropic or anisotropic as well. The isotropic phase function describes equal scattering in all directions and is thus is independent of either of the two angles and always has a value of $1/4\pi$. Anisotropic phase functions depend on either the angle between the two directions or the two directions themselves, depending on if the medium is isotropic or anisotropic, respectively.

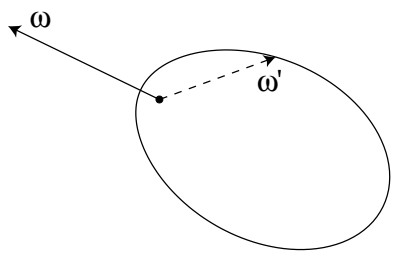


Figure 13.7: The phase function describes the distribution of scattered radiance in directions $\vec{\omega}'$ at a point, given incident radiance along the direction $\vec{\omega}$. Here we have plotted the Henyey-Greenstein phase function with an asymmetry parameter g equal to 0.5.

The change in radiance due to in-scattering is given by the *source term*,

$$S(x,\vec{\omega}) = \sigma_s(x,\vec{\omega}) \int_{S^2} p(\vec{\omega}' \rightarrow \vec{\omega}) L_i(x,\vec{\omega}') d\vec{\omega}'.$$

Vector 1[illegible]

13.2 Phase Functions

Just as there are a wide variety of BSDF models to describe scattering from surfaces, a variety of phase functions have been developed, ranging from parameterized models, which can be used to fit a function with a small number of parameters to measured data, and the analytic, which are derived by directly deriving the scattered radiance distribution that results from scattering from particles with known shape and material (e.g. scattering from spherical water droplets.)

In this section, we will describe some commonly-used phase functions and provide their implementations. The simplest of them is the isotropic phase function, which describes equal scattering in all directions. From the normalization constraint of Equation 13.1.1, it follows that

$$p_{isotropic}(\vec{\omega} \rightarrow \vec{\omega}') = \frac{1}{4\pi}.$$

⟨*Volume Scattering Definitions*⟩≡
```
Float PhaseIsotropic(const Vector &w, const Vector &wp) {
    return 1.f / (4. * M_PI);
}
```

A number of anisotropic phase functions, describing angularly-varying scattering, are also useful. All of the anisotropic phase functions in the remainder of this section are described in terms of the cosine of the angle between the two-directions–see Figure 13.8. Note that we are using a different convention for the direction of vectors at a scattering event in a volume than we used for scattering at a surface, where both vectors faced away from the surface. This matches the usual convention used for phase functions.

XXX $p(\vec{\omega} \rightarrow \vec{\omega}')$ gives scattering direction probability density at a point in the volume, analogous to BRDF at a surface.

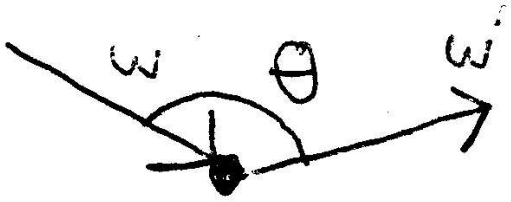


Figure 13.8: phase costheta convention

If the particles have radius r that is smaller than the wavelength of light λ, the Rayleigh model is a good fit if $r/\lambda < 0.05$.

$$p_{rayleigh}(\vec{\omega} \to \vec{\omega}') = p_{rayleigh}(\vec{\omega} \cdot \vec{\omega}') = p_{rayleigh}(\cos\theta) = \frac{3}{16\pi}\left(1+\cos^2\theta\right)$$

Wavelength-dependent Rayleigh scattering is what makes the sky blue and sunsets red.

⟨*Volume Scattering Definitions*⟩+≡

```
Float PhaseRayleigh(const Vector &w, const Vector &wp) {
    Float costheta = Dot(w, wp);
    return  3.f/(16.f*M_PI) * (1 + costheta * costheta);
}
```

16 Vector

Mie scattering is when $r \approx \lambda$. Water droplets, fog. Nishita et al suggest two approximations for hazy and murky atmospheric conditions

$$p_{Mie-hazy}(\cos\theta) = 1 + \frac{9}{4\pi}\left(\frac{1+\cos\theta}{2}\right)^8$$

and

$$p_{Mie-murky}(\cos\theta) = 1 + \frac{50}{4\pi}\left(\frac{1+\cos\theta}{2}\right)^{32}$$

⟨*Volume Scattering Definitions*⟩+≡

```
Float PhaseMieHazy(const Vector &w, const Vector &wp) {
    Float costheta = Dot(w, wp);
    return 9.f/(4.f*M_PI) * powf(1.f + costheta*costheta, 8.f);
}
```

⟨*Volume Scattering Definitions*⟩+≡

```
Float PhaseMieMurky(const Vector &w, const Vector &wp) {
    Float costheta = Dot(w, wp);
    return 50.f/(4.f*M_PI) * powf(1.f + costheta*costheta, 32.f);
}
```

Henyey-Greenstein an empirical parameterized phase function, developed for fitting to measured data. It takes a parameter g, $-1 < g < 1$ that controls the amount and direction of anisotropy.

$$p_{HG}(\cos\theta) = \frac{1}{4\pi}\frac{1-g^2}{(1+g^2-2g(\cos\theta))^{3/2}}$$

Negative values of g correspond to *back-scattering*, where light is mostly scattered back toward the incident direction, and positive values correspond to forward scattering. The greater the magnitude of g, the more scattering is scattered close to the $-\vec{\omega}$ or $\vec{\omega}$ directions (for back-scattering and forward scattering, respectively.)

XXX Figure to show this.

⟨*Volume Scattering Definitions*⟩+≡
```
Float PhaseHG(const Vector &w, const Vector &wp, Float g) {
    Float costheta = Dot(w, wp);
    return 1.f / (4.f * M_PI) * (1.f - g*g) /
        powf(1.f + g*g - 2.f * g * costheta, 1.5f);
}
```

Phase functions are often defined by an asymmetry parameter, g, that is the average value of the product of the phase function with the cosine of the angle between $\vec{\omega}'$ and $\vec{\omega}$. The range of g is from -1 to 1, corresponding to total back-scattering to total forward scattering, respectively. Given an arbitrary phase function, its g value can be computed by:

$$g = \frac{1}{2}\int_{S^2} \mathrm{p}(\vec{\omega} \to \vec{\omega}')(\vec{\omega}\cdot\vec{\omega}')\mathrm{d}\vec{\omega}'$$

Vector 16

Thus, isotropic scattering corresponds to a g of zero. Any number of phase functions can satisfy this quation; the g value alone is not enough to uniquely describe a scattering distribution. Nevertheless, the convenience of being able to easily convert a complex scattering distribution into a simple parameterized model often outweighs the loss in accuracy.

More complex phase functions that aren't described well with a single asymmetry parameter are often modeled with a weighted sum of phase functions like Henyey-Greenstein, each with different parameter values:

$$\mathrm{p}(\vec{\omega} \to \vec{\omega}') = \sum_{i=1}^{n} w_i \, \mathrm{p}_i(\vec{\omega} \to \vec{\omega}')$$

where the weights, w_i necessarily sum to one so that the normalization condition, Equation 13.1.1, holds.

One final phase function was developed by Schlick as an efficient approximation to the Henyey–Greenstein function. It has been widely used in computer graphics due to its computational efficiency. It is

$$\mathrm{p}_{Schlick}(\cos\theta) = \frac{1}{4\pi}\frac{1-g^2}{(1-g\cos\theta)^2}$$

⟨*Volume Scattering Definitions*⟩+≡
```
Float PhaseSchlick(const Vector &w, const Vector &wp, Float g) {
    Float gcostheta = g * Dot(w, wp);
    return 1.f / (4. * M_PI) * (1 - g*g) /
        ((1 - gcostheta) * (1 - gcostheta));
}
```

13.3 Volume Description

As part of the scene description, the volume-varying scattering information can be defined by the user. The abstract `VolumeRegion` class provides the basic interface that describes volume scattering in a region of the scene. Multiple `VolumeRegion`s of different types can be used to describe different types of scattering in different parts of the scene. In this section, we will describe the basic interface as well as a handful of useful implementations.

⟨*volume.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef VOLUME_H
#define VOLUME_H 1
#include "lrt.h"
#include "color.h"
#include "geometry.h"
#include "paramset.h"
#include "transform.h"
⟨Volume Scattering Declarations⟩
#endif // VOLUME_H
```

⟨*volume.cc**⟩≡ 26 Ray
```
⟨Source Code Copyright⟩
#include "volume.h"
⟨Volume Scattering Definitions⟩
```

⟨*Volume Scattering Declarations*⟩+≡
```
class VolumeRegion {
public:
    ⟨VolumeRegion Methods⟩
};
```

All `VolumeRegion`s must be able to compute their axis-aligned world-space bounding box and to check to see if a given ray intersects the region. If the ray does intersect it, the `Intersect()` routine should return the parametric t range of the segment that overlaps the volume in

⟨*VolumeRegion Methods*⟩+≡
```
virtual BBox WorldBound() const = 0;
virtual bool Intersect(const Ray &ray, Float *t0, Float *t1) const = 0;
```

There are four basic functions that allow `VolumeRegion`s to describe their possibly spatially-varying scattering properties. Given a world-space point and direction, `sigma_a()`, `sigma_s()`, and `Le()` return the corresponding values for the given position and direction. The `phase()` method returns the value of the phase function for the given pair of directions for the given point.

⟨*VolumeRegion Methods*⟩+≡
```
virtual Spectrum sigma_a(const Point &, const Vector &) const = 0;
virtual Spectrum sigma_s(const Point &, const Vector &) const = 0;
virtual Spectrum Le(const Point &, const Vector &) const = 0;
virtual Float phase(const Point &, const Vector &,
    const Vector &) const = 0;
```

For convenience. Some implementations may be able to do this more efficiently if we know both are needed...

⟨*Volume Scattering Definitions*⟩+≡
```
Spectrum VolumeRegion::sigma_t(const Point &P, const Vector &w) const {
    return sigma_a(P, w) + sigma_s(P, w);
}
```

Finally, the `tau()` method computes the optical thickness that the ray passes through in the volume from `ray(ray.mint)` and `ray(ray.maxt)`. The `HomogeneousRegion` below can compute this value exactly, while more complex regions will be Monte Carlo integration to compute it (see Section 14.5.)

⟨*VolumeRegion Methods*⟩+≡
```
virtual Spectrum tau(const Ray &ray) const = 0;
```

Homogeneous Region

The simplest volume representation, `HomogeneousRegion`, describes a region of space bounded by a `BBox` with homogeneous scattering properties throughout it. Values for σ_a, σ_s, the phase function's g value, and the amount of emission $L_e$are passed to the constructor. In conjunction with a transformation from world to volume space and an axis-aligned volume space bound, this suffices to describe the region.

GetInverse 43
Point 21
Ray 26
Spectrum 155
Transform 32
Vector 16
VolumeRegion 383

⟨*homogeneous.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "volume.h"
⟨HomogeneousRegion Declarations⟩
⟨HomogeneousRegion Definitions⟩
```

⟨*HomogeneousRegion Declarations*⟩≡
```
class HomogeneousRegion : public VolumeRegion {
public:
    ⟨HomogeneousRegion Methods⟩
private:
    ⟨HomogeneousRegion Private Data⟩
};
```

⟨*HomogeneousRegion Methods*⟩≡
```
HomogeneousRegion(const Spectrum &sa, const Spectrum &ss, Float gg,
        const Spectrum &emit, const BBox &e,
        const Transform &v2w) {
    WorldToVolume = v2w.GetInverse();
    sig_a = sa;
    sig_s = ss;
    g = gg;
    le = emit;
    extent = e;
}
```

⟨*HomogeneousRegion Private Data*⟩≡
```
Spectrum sig_a, sig_s, le;
Float g;
BBox extent;
Transform WorldToVolume;
```

Because the bound is maintained internally in the volume's object space, we need to transform it for the `WorldBound()` method.

⟨*HomogeneousRegion Methods*⟩+≡
```
BBox WorldBound() const { return WorldToVolume.GetInverse()(extent); }
```

⟨*HomogeneousRegion Methods*⟩+≡
```
bool Intersect(const Ray &r, Float *t0, Float *t1) const {
    Ray ray = WorldToVolume(r);
    return extent.IntersectP(ray, t0, t1);
}
```

Implementation of the rest of the `VolumeRegion` interface methods is straightforward; we just verify that the given point is inside the region's extent and return the appropriate value if so.

⟨*HomogeneousRegion Methods*⟩+≡
```
Spectrum sigma_a(const Point &p, const Vector &) const {
    return extent.Inside(WorldToVolume(p)) ? sig_a : 0.;
}
```

43	GetInverse
30	Inside
382	PhaseHG
21	Point
26	Ray
155	Spectrum
32	Transform
16	Vector

⟨*HomogeneousRegion Methods*⟩+≡
```
Spectrum sigma_s(const Point &p, const Vector &) const {
    return extent.Inside(WorldToVolume(p)) ? sig_s : 0.;
}
```

⟨*HomogeneousRegion Methods*⟩+≡
```
Spectrum sigma_t(const Point &p, const Vector &) const {
    return extent.Inside(WorldToVolume(p)) ? (sig_a + sig_s) : 0.;
}
```

⟨*HomogeneousRegion Methods*⟩+≡
```
Spectrum Le(const Point &p, const Vector &) const {
    return extent.Inside(WorldToVolume(p)) ? le : 0.;
}
```

⟨*HomogeneousRegion Methods*⟩+≡
```
Float phase(const Point &p, const Vector &wi, const Vector &wo) const {
    if (!extent.Inside(WorldToVolume(p))) return 0.;
    return PhaseHG(wi, wo, g);
}
```

⟨*HomogeneousRegion Methods*⟩+≡
```
Spectrum tau(const Ray &r) const {
    Ray ray = WorldToVolume(r);
    Float t0, t1;
    if (!extent.IntersectP(ray, &t0, &t1)) return 0.;
    return Distance(ray(t0), ray(t1)) * (sig_a + sig_s);
}
```

Varying-Density Volumes

A number of the volume representations to come are based on the assumption that the underlying particles throughout the medium all have the same scattering properties, but that their density changes spatially at different points in the medium. In order to reduce duplicated code and so that the various representations can just focus on varying the density of the particles, we will define a `DensityRegion` class that implements many of the `VolumeRegion` interface functions.

The `DensityRegion` adds a new virtual function, `density()` that its sub-classes must implement. However, the sub-classes are freed from needing to implement `sigma_a()`, `sigma_s()`, etc., since default implementations of those methods just scale the given scattering properties with the local density at the point.

⟨*Volume Scattering Declarations*⟩+≡
```
class DensityRegion : public VolumeRegion {
public:
    ⟨DensityRegion Methods⟩
protected:
    ⟨DensityRegion Protected Data⟩
};
```


Distance 23
GetInverse 43
Point 21
Ray 26
Spectrum 155
Transform 32
Vector 16
VolumeRegion 383

⟨*DensityRegion Methods*⟩≡
```
DensityRegion::DensityRegion(const Spectrum &sa, const Spectrum &ss, Float gg,
        const Spectrum &emit, const Transform &v2w) {
    WorldToVolume = v2w.GetInverse();
    sig_a = sa;
    sig_s = ss;
    g = gg;
    le = emit;
}
```

⟨*DensityRegion Protected Data*⟩≡
```
Transform WorldToVolume;
Spectrum sig_a, sig_s, le;
Float g;
```

⟨*DensityRegion Methods*⟩+≡
```
virtual Float density(const Point &Pobj) const = 0;
```

⟨*DensityRegion Methods*⟩+≡
```
Spectrum sigma_a(const Point &p, const Vector &) const {
    return density(WorldToVolume(p)) * sig_a;
}
```

⟨*DensityRegion Methods*⟩+≡
```
Spectrum sigma_s(const Point &p, const Vector &) const {
    return density(WorldToVolume(p)) * sig_s;
}
```

⟨*DensityRegion Methods*⟩+≡
```
Spectrum sigma_t(const Point &p, const Vector &) const {
    return density(WorldToVolume(p)) * (sig_a + sig_s);
}
```

⟨*DensityRegion Methods*⟩+≡
```
Spectrum Le(const Point &p, const Vector &) const {
    return density(WorldToVolume(p)) * le;
}
```

⟨*DensityRegion Methods*⟩+≡
```
Float phase(const Point &p, const Vector &wi, const Vector &wo) const {
    return PhaseHG(wi, wo, g);
}
```

3D Grids

Point-sampled data, kind of like an imagemap.

In the `VolumeGrid` representation, the density is stored at a regular 3D grid of positions and is interpolated to compute the density at positions between the sample points. Here, we read the density values from disk, thus allowing a variety of sources of data (e.g. physical simulation in a pre-process, acquiring data from a real object, as from a medical CT scan, etc.) The user supplies baseline values of σ_a, σ_s, etc., all of which are just scaled by the local density at the point of interest.

⟨*volumegrid.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "volume.h"
⟨VolumeGrid Declarations⟩
⟨VolumeGrid Definitions⟩
```

386 DensityRegion
382 PhaseHG
21 Point
155 Spectrum
16 Vector

⟨*VolumeGrid Declarations*⟩≡
```
class VolumeGrid : public DensityRegion {
public:
    ⟨VolumeGrid Methods⟩
private:
    ⟨VolumeGrid Private Data⟩
};
```

⟨*VolumeGrid Declarations*⟩+≡
```
#define SAMP(x,y,z) (d[(z)*nx*ny + (y)*nx + (x)])
```

⟨*VolumeGrid Definitions*⟩≡
```
VolumeGrid::VolumeGrid(const Spectrum &sa, const Spectrum &ss, Float gg,
        const Spectrum &emit, const BBox &e, const Transform &v2w,
        const string &filename)
    : DensityRegion(sa, ss, gg, emit, v2w) {
    extent = e;
    FILE *f = fopen(filename.c_str(), "r");
    if (!f) {
        fprintf(stderr, "Unable to open volume file %s\n",
            filename.c_str());
        d = new Float[1];
        d[0] = 0;
        nx = ny = nz = 1;
    }
    ⟨Process volume data from file⟩
    fclose(f);
}
```

We support a very simple volume file format, with three integers at the start to encode the dimensions in each direction and then an 8-bit character for each volume sample.

XXX bad error handling, not platform independent, 16-bit or float would probably be better, etc... XXX

DensityRegion 386
GetInverse 43
Spectrum 155
Transform 32
VolumeGrid 387

⟨*Process volume data from file*⟩≡
```
fread(&nx, sizeof(int), 1, f);
fread(&ny, sizeof(int), 1, f);
fread(&nz, sizeof(int), 1, f);
d = new Float[nx*ny*nz];
for (int i = 0; i < nx*ny*nz; ++i) {
    unsigned char c;
    fread(&c, sizeof(unsigned char), 1, f);
    d[i] = c * (1.f/255.f);
}
```

⟨*VolumeGrid Private Data*⟩≡
```
Float *d;
int nx, ny, nz;
BBox extent;
```

⟨*VolumeGrid Methods*⟩+≡
```
~VolumeGrid() { delete[] d; }
```

⟨*VolumeGrid Methods*⟩+≡
```
BBox WorldBound() const { return WorldToVolume.GetInverse()(extent); }
```

⟨*VolumeGrid Methods*⟩+≡
```
bool Intersect(const Ray &r, Float *t0, Float *t1) const {
    Ray ray = WorldToVolume(r);
    return extent.IntersectP(ray, t0, t1);
}
```

⟨*VolumeGrid Definitions*⟩+≡
```
Float VolumeGrid::density(const Point &Pobj) const {
    if (!extent.Inside(Pobj)) return 0;
    ⟨Compute voxel coordinates and offsets for Pobj⟩
    ⟨Trilinearly interpolate density values to compute local density⟩
}
```

⟨*Compute voxel coordinates and offsets for* Pobj⟩≡
```
Float voxx = (Pobj.x - extent.pMin.x) /
    (extent.pMax.x - extent.pMin.x) * (nx-1);
Float voxy = (Pobj.y - extent.pMin.y) /
    (extent.pMax.y - extent.pMin.y) * (ny-1);
Float voxz = (Pobj.z - extent.pMin.z) /
    (extent.pMax.z - extent.pMin.z) * (nz-1);
int vx = Clamp(Floor2Int(voxx), 0, nx - 2);
int vy = Clamp(Floor2Int(voxy), 0, ny - 2);
int vz = Clamp(Floor2Int(voxz), 0, nz - 2);
Float dx = voxx - vx;
Float dy = voxy - vy;
Float dz = voxz - vz;
```

⟨*Trilinearly interpolate density values to compute local density*⟩≡
```
Float d00 = Lerp(dx, SAMP(vx, vy, vz),     SAMP(vx+1, vy, vz));
Float d10 = Lerp(dx, SAMP(vx, vy+1, vz),   SAMP(vx+1, vy+1, vz));
Float d01 = Lerp(dx, SAMP(vx, vy, vz+1),   SAMP(vx+1, vy, vz+1));
Float d11 = Lerp(dx, SAMP(vx, vy+1, vz+1), SAMP(vx+1, vy+1, vz+1));
Float d0 = Lerp(dy, d00, d10);
Float d1 = Lerp(dy, d01, d11);
return Lerp(dz, d0, d1);
```

513 Clamp
386 DensityRegion
514 Floor2Int
30 Inside
512 Lerp
28 pMax
28 pMin
21 Point
26 Ray
387 VolumeGrid

Exponential Mist

Density varies linearly as a function of z

$$d = ae^{-bP_z}$$

⟨*exponential.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "volume.h"
⟨ExponentialMist Declarations⟩
⟨ExponentialMist Definitions⟩
```

⟨*ExponentialMist Declarations*⟩≡
```
class ExponentialMist : public DensityRegion {
public:
    ⟨ExponentialMist Methods⟩
private:
    ⟨ExponentialMist Private Data⟩
};
```

⟨*ExponentialMist Methods*⟩≡
```
ExponentialMist(const Spectrum &sa, const Spectrum &ss, Float gg,
        const Spectrum &emit, const BBox &e, const Transform &v2w,
        Float a, Float b)
    : DensityRegion(sa, ss, gg, emit, v2w) {
    extent = e;
    A = a;
    B = b;
}
```

⟨*ExponentialMist Private Data*⟩≡
```
BBox extent;
Float A, B;
```

⟨*ExponentialMist Methods*⟩+≡
```
BBox WorldBound() const { return WorldToVolume.GetInverse()(extent); }
```

⟨*ExponentialMist Methods*⟩+≡
```
bool Intersect(const Ray &r, Float *t0, Float *t1) const {
    Ray ray = WorldToVolume(r);
    return extent.IntersectP(ray, t0, t1);
}
```

DensityRegion 386
Distance 23
ExponentialMist 389
GetInverse 43
Point 21
Ray 26
Spectrum 155
Transform 32

⟨*ExponentialMist Methods*⟩+≡
```
Float density(const Point &Pobj) const {
    return A * exp(-B * Pobj.z);
}
```

$$d_{\text{total}} = \int_0^t d(t')\,\mathrm{d}t' = |P_1 - P_0| \int_{z_0}^{z_1} e^{-z}\,\mathrm{d}z = \frac{A|P_1 - P_0|}{B\tilde{\mathbf{d}}(\mathbf{r})_z}\left(e^{-o(\mathbf{r})_z + \tilde{\mathbf{d}}(\mathbf{r})_z |P_1 - P_0|} - e^{-o(\mathbf{r})_z}\right)$$

⟨*ExponentialMist Methods*⟩+≡
```
Spectrum tau(const Ray &r) const {
    Ray ray = WorldToVolume(r);
    Float t0, t1;
    if (!extent.IntersectP(ray, &t0, &t1)) return 0.;
    Float dist = Distance(ray(t0), ray(t1));
    return (A * dist)/(B * ray.D.z) *
        (exp(-ray(t1).z) - exp(-ray(t0).z)) *
        (sig_a + sig_s);
}
```

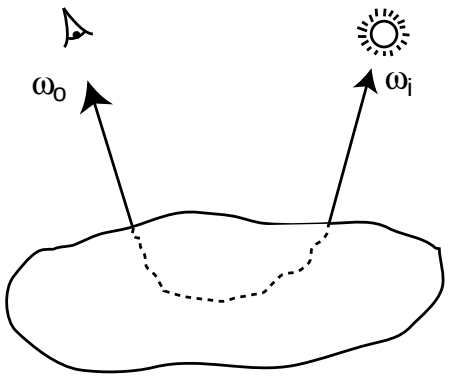


Figure 13.9: The bidirectional scattering-surface reflectance distribution function generalizes the BRDF to account for light that exits the surface at a point other than where it enters. It is more difficult to evaluate in practice, though subsurface light transport can be responsible for a substantial part of the appearance of many real-world objects.

13.4 Subsurface Scattering

Foo.

The BSSRDF

There is an important assumption implicit in the BSDF and the scattering equation: that the only incident light that has an effect on the outgoing radiance at x is also incident on the surface at x–light that hits the surface at other points x' is assumed to not affect outgoing radiance at x.

Equivalently, the BSDF assumes that the distribution of incident radiance on the surface is uniform over a relatively large area of the surface with respect to the amount of scattering that goes on beneath the surface.

For many types of surfaces–human skin, marble, etc.–there is a significant amount of *subsurface light transport*, however. Light that enters a surface at one location may travel for some distance underneath the surface, undergoing scattering there, before exiting at another position–see Figure 13.9. (Chapter 13 describes the mechanics for describing light transport and scattering through volumetric media such as these.)

The *bidirectional scattering-surface reflectance distribution function* (BSSRDF) is the formalism that describes this. It is a distribution function $S(x', \vec{\omega}_i, x, \vec{\omega}_o)$ that describes the proportion of outgoing differential radiance at point x in direction $\vec{\omega}_o$ due to differential irradiance at x' from direction $\vec{\omega}_i$.

The scattering equation for the BSSRDF requires integration over surface area *and* incoming direction; it is substantially more complex than Equation 5.4.8.

$$L_o(x, \vec{\omega}_o) = \int_A \int_{S^2} S(x', \vec{\omega}_i, x, \vec{\omega}_o) \cos\theta_i \, d\vec{\omega}_i \, dA$$

Fortunately, points x' that are far away from x generally contribute little to $L_o(x, \vec{\omega}_o)$. This fact can be a substantial help in implementations.

Further Reading

The books written by van de Hulst (van80) and Preisendorfer (Pre65; Pre76) are excellent introductions to volume light transport. Chandrasekhar's seminal book is another excellent resource (Cha60).

The Henyey–Greenstein phase function was originally described in Henyey and Greenstein's 1941 paper (HG41). Detailed discussion of scattering and phase functions and derivations of phase functions that describe scattering from independent spheres, cylinders, and other simple shapes can be found in van de Hulst (van81). In particular, extensive discussion of the commonly-used Mie and Rayleigh scattering models (which describe scattering from particles approximately the size of or larger than the wavelength of incident radiation and particles much smaller than the wavelength of incident radiation, respectively) is available there. Hansen and Travis's survey article is also a good introduction to the variety of commonly-used phase functions (**?**).

Blinn first introduced basic volume scattering algorithms to graphics (Bli82b). Other important early work includes Kajiya and von Herzen (KH84), Max (Max86), and Nishita et al (NMN87). Glassner's book has a thorough overview of this topic and previous applications of it in graphics (Gla95), and Max's survey article also concisely summarizes the topic (Max95).

Volume scattering has been applied to simulating atmospheric scattering. Work on this topic includes Klassen (Kla87) and Nishita et al (NMN87). More recently, Preetham et al's SIGGRAPH paper introduced a physically rigorous and computationally efficient atmospheric and sky-lighting model (PSS99).

Subsurface scattering was first introduced to graphics by Hanrahan and Krueger (HK93), though their approach did not accurately simualte light that entered the object at points other than at the point being shaded. Dorsey et al applied photon maps to simulating true subsurface scattering (DEL$^+$99). Other work in this area includes papers by Pharr and Hanrahan (PH00) and Jensen et al (JMLH01; JB02).

There are a number of important applications of visualizing volumetric datasets for medical and engineering applications–this area is called *volume rendering*. In many of these applications, radiometric accuracy is substantially less important then developing techniques that help make structure in the data apparent (e.g. where the bones are in CT scan data.) Early papers in this area include Levoy's (Lev88; Lev90b; Lev90a) and Drebin et al (DCH88).

Volume datasets from Stanford. `http://www.volvis.org`.

In this chapter, we have ignored all issues related to sampling and anti-aliasing of volumes, though in principle this issues should be considered, e.g. for the case of a volume that occupies just a few pixels on the screen. Marschner and Lobb present the theory and practice of sampling and reconstruction for three-dimensional datasets, applying ideas similar to those in Chapter 7 (ML94).

Binary volume octree to speed up traversal of empty regions (Lev88). Classification... Danskin and Hanrahan various techniques based on 3D pyramid of volume data to speed traversal, use lower precision computations when contribution to final result is low (DH92).

Rushmeier and Torrance finite element stuff (RT87).

Schramm et al?? (SGM97).

Exercises

13.1 use depth-mapped shadowmap stuff for fast light beams through atmosphere

13.2 pass radiance into `attenuation()`/`L()` functions of `VolumeIntegrator`, use their magnitudes to guide how many MC samples to take, etc...

14. Monte Carlo Integration

Monte Carlo is a flexible way of using random sampling to estimate the values of integrals. One of its key features is that it one only needs to be able to evaluate a function to be integrated $f(x)$ at arbitrary points x in order to generate estimates of the value of $\int f(x)\mathrm{d}x$; this makes Monte Carlo relatively easy to implement. In contrast to techniques like the trapezoid rule or more complex quadrature methods for estimating the value of integrals, Monte Carlo works especially well with integrals over many dimensions.

In graphics, we often have difficult integrals that need to be estimated, e.g. to compute the amount of light reflected by the BSDF at a point with the reflection Equation 5.4.8, it is necessary integrate the incident light at a point over all directions over the hemisphere. Unless one somehow has both a closed form expression for the incident lighting distribution *and* can compute the convolution of the incident light with the BSDF analytically, some other method must be used. Monte Carlo integration makes it possible to compute an estimate for the reflected radiance simply by sampling a set of directions over the hemisphere, computing incident radiance along them, multiplying by the BSDF's value, and applying a weighting term.

The main disadvantage of Monte Carlo is that it converges at a rate of just $O(n^{-1/2})$, where n is the number of samples taken; four times more samples are needed to reduce the error by half. In images, the artifacts from insufficient Monte Carlo sampling generally show up as noise–some pixels are much too bright and some are much too dark. This is visually unappealing! Most of the effort involved in implementing Monte Carlo routines is in choosing the best possible Monte Carlo techniques to keep this error as low as possible.

⟨*mc.h**⟩≡
⟨*Source Code Copyright*⟩
```
#ifndef MC_H
#define MC_H
```
⟨*MC Utility Declarations*⟩
⟨*MC Class Declarations*⟩
```
#endif
```

⟨*mc.cc**⟩≡
⟨*Source Code Copyright*⟩
```
#include "lrt.h"
#include "geometry.h"
#include "shapes.h"
#include "mc.h"
```
⟨*MC Function Definitions*⟩

14.1 Background

We will start by defining some basic terms and covering background concepts from probability. A *random variable x* is a value from some domain that has some distribution of values. The domain may be discrete (e.g. a fixed set of possibilities) or continuous (e.g. the real numbers $\mathbb{R}$).

For example, the result of a roll of a die is a discrete random variable sampled from the set of events $X_i = \{1,2,3,4,5,6\}$. Each event has a probability $p_i = 1/6$ and the sum of probabilities $\sum p_i$ is necessarily one. We can take a continuous random variable ξ that is uniformly distributed among the real numbers between zero and one and map it to a discrete random variable, choosing X_i if:

$$\sum_{j=1}^{i-1} p_j < \xi \leq \sum_{j=1}^{i} p_j.$$

For lighting applications, we might want to define a probability of sampling illumination from each of a set of light sources, based on the power from each source relative to the total power from all sources.

$$p_i = \frac{\Phi_i}{\sum_j \Phi_j}.$$

The *cumulative distribution function* (cdf) $P(x_i)$ of a random variable is the probability that a value from the variable's domain is less than x_i:

$$P(x) = Pr\{X \leq x\}.$$

For the die example, $P(2) = 1/3$, for instance.

The random variable that takes on all values between zero and one with equal probability is an example of a continuous random variable. Because it has equal probability of taking on all values within that range, it is called a *uniform random variable*. We will denote it by the special symbol ξ, since we will often be interested in using it to generate samples from other distributions.

Another example of a continuous random variable is one that ranges over the real numbers between 0 and 2 where the probability of it taking on any particular value x is related to the value $2-x$: it is twice as likely for it to take on a value around zero as it is to take one around one, etc. The *probability density function* (pdf) formalizes this idea: it describes the relative probability of a random variable taking on a particular value. The pdf $p(x)$ is just the derivative of the random variable's cdf.

$$p(x) = \frac{dP(x)}{\mathrm{d}x}$$

For uniform random variables, $p(x)$ is a constant.

Pdfs are necessarily non-negative and integrate to one over their domains. For the uniform random variable ξ, $P(x) = x$ and $p(x) = 1$. We will use the notation $x \sim p$ to denote that x is a random variable with the pdf p.

Given an arbitrary interval $[a,b]$ in the domain, the pdf can give the probability that a random variable lies inside the interval.

$$P(x \in [a,b]) = \int_a^b p(x)\mathrm{d}x$$

The Monte Carlo Estimator

We can now define the Monte Carlo estimator, which gives a method for estimating the value of an integral. First, we define the *expected value* $E[f(x)]$ of a function f, which is the average value that f takes on over some density.

$$E[f(x)] = \int f(x)p(x)\mathrm{d}x \tag{14.1.1}$$

Consider finding the expected value of the cosine function between 0 and π, where p is uniform. Because $p(x)$ must integrate to one over the domain, we have $p(x) = 1/\pi$ and

$$\begin{aligned} E[\cos(x)] &= \int_0^\pi \frac{\cos x}{\pi}\,\mathrm{d}x \\ &= \frac{1}{\pi}(-\sin\pi + \sin 0) \\ &= 0 \end{aligned}$$

Which is precisely what we expect.

The expected value can be estimated with the sum

$$E[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i), \tag{14.1.2}$$

where $x_i \sim p$.

We can use Equations 14.1.1 and 14.1.2 to derive the basic Monte Carlo estimator. If we want to estimate the integral of some function $f(x)$ (rather than the integral of $f(x)p(x)$) then we can set $g(x) = f(x)p(x)$ and apply Equation 14.1.1 to see that:

$$\int f(x) \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \tag{14.1.3}$$

This is the basic Monte Carlo estimator. One way of understanding it on an intuitive level is to see that it is necessary to compensate for samples x_i that are taken with higher probability than others by reducing their relative contribution to give less weight in the estimate. It can be equivalently written as:

$$E[\int f(x)] = \frac{1}{N}\sum_{i=1}^{N} f(x_i)w(x_i) \tag{14.1.4}$$

where $w(x_i)$ is a weight that ensures that the expected value of the sum is equal to $\int f(x)$. Thus, in Equation 14.1.3, $w(x) = 1/p(x)$. We will use this form of the Monte Carlo estimator for the remainder of the book.

As an example of Monte Carlo in action, to compute the integral of some one-dimensional function $f(x)$ over the domain $[0,1]$, if we randomly sample uniform random variables ξ_i over the domain, the estimate is

$$E[\int f(x)] = \frac{1}{N}\sum_{i=1}^{N} f(x_i),$$

since $p(\xi) = 1$.

For multi-dimensional integration, the extension of these ideas is straightforward. Samples x_i are taken from a multi-dimensional density and the estimator is applied as usual. (Here is a key difference between MC and quadrature methods for integration in higher dimensions: the number of samples N can be chosen completely independently from the number of dimensions.)

Sampling Random Variables

Given a random variable distributed according to some distribution, we need a way to generate samples according to the distribution in order to use Monte Carlo.

One is *rejection sampling*. This is a method that first uniformly samples a value from the domain but then rejects it with some probability that ensures that the accepted values have the desired distribution. For example, in the 1D case, we can generate samples with density proportional to any function $f(x)$ where we know its upper bound, M. We choose two uniform random numbers, ξ_1 and ξ_2. If $\xi_2 < f(\xi_1)/M$, then we accept ξ_1 as a sample. Otherwise we reject it and choose two new random numbers. Rejection sampling is easy to implement, though it does require that we be able to compute the upper bound of the function. Its efficiency is closely tied to how close the bound is to the function's value over the domain.

For pdfs that can be integrated analytically, the *inversion method* (also known as the *transformation method* can be applied. The idea behind this is that uniform random variables are transformed to random variables from the desired distribution. To generate a sample from an arbitrary one-dimensional pdf given a uniform random number ξ, we need to solve the equation

$$\xi = \int_{-\infty}^{x} p(x)\,\mathrm{d}x$$

for x. It is best if this can be done analytically, though numerical techniques can be applied as well.

For example, uniform distributions are easy to sample this way: to sample a uniform 1D distribution of reals from $[-5,5]$, we compute $-5+10\xi$. To sample uniformly in higher-dimensional domains, additional ξ_i are used.

This transformation is an instance of a more general XXX. If we have a transformation from a random variable with one distribution to one with another given by some function $x' = f(x)$, then the relationship between their pdfs is given by the original pdf $p(x)$ and the *Jacobian* of the function $f(x)$. In one-dimension, this means that

$$p'(x') = \left| \frac{\partial x}{\partial x'} \right| p(x).$$

For the example above for sampling from $[-5,5]$,

$$\frac{\partial x}{\partial x'} = \frac{1}{10}$$

and the pdf is $1/10$, which matches the value for the pdf that direct integration and normalization of $-5+10\xi$ would give us.

Consider the sampling the density of the power function

$$p(x) = (n+1)x^n$$

over the domain $[0,1]$. To generate samples from the pdf, we take a uniform random number ξ and determine which value of $x \in [0,1]$ it maps to:

$$\begin{aligned} \xi &= (n+1)\int_0^x x^n \,\mathrm{d}x \\ &= x^{n+1} \\ x &= \sqrt[n+1]{\xi} \end{aligned}$$

In higher dimensions, the inversion method is more tricky. We need to sample one dimension at a time. For example, to sample from a 2D distribution $p(x,y)$, we define two new 1D distributions:

$$\begin{aligned} p_x(x) &= \int_{-\infty}^{\infty} p(x,y)\,dy \\ p_y(y|x) &= \frac{p(x,y)}{\int_{-\infty}^{\infty} p(x,y')\,dy'} \end{aligned} \tag{14.1.5}$$

The first is a distribution on x, which says that the probability density for sampling a particular x value is given by the density over y values for that x. The second distribution is a conditional distribution that says, given that some x value has been sampled, the distribution to sample from for y values is given by the 1D density of x values for that x, normalized to be a valid pdf.

It is often the case that the density $p(x,y)$ is separable such that $p(x,y) = p_1(x)p_2(y)$. Then we can sample each dimension independently and compute the final pdf as the product of the pdfs for each dimension.

Sampling Piecewise Constant 1D Functions

An interesting exercise is to work out how to sample from one-dimensional piecewise-constant functions (step functions). We will first consider one-dimensional piecewise-constant functions defined over $[0,1]$ and will then extend the approach to sampling two-dimensional piecewise-constant functions.

Assume that the one-dimensional function's domain is split into N equal-sized pieces of size $\Delta = 1/N$. These regions start and end at points $x_i = i * \Delta$, where i

Figure 14.1:

ranges from 0 to N inclusive, and within each region, the value of the function $f(x)$ is a constant–see the left side of Figure 14.1. The value of $f(x)$ is

$$f(x) = \begin{cases} v_0 & : \quad x_0 \le x < x_1 \\ v_1 & : \quad x_1 \le x < x_2 \\ \dots & : \quad \dots \end{cases}$$

The integral of $\int f(x)\mathrm{d}x$ is

$$c = \int_0^1 f(x)\mathrm{d}x = \sum_{i=0}^{N-1} \Delta f_i = \sum_{i=0}^{N-1} \frac{f_i}{N}, \tag{14.1.6}$$

and so it is easy to construct the pdf $p(x)$ for $f(x)$ by $f(x)/c$. By direct application of the relevant formulas, the cdf $F(x)$ is a piecewise linear function defined at the points x_i by

$$\begin{aligned} F(x_0) &= 0 \\ F(x_1) &= \int_{x_0}^{x_1} p(x)\mathrm{d}x = v_0/(N*c) = F(x_0) + v_0/(N*c) \\ F(x_2) &= \int_{x_0}^{x_2} p(x)\mathrm{d}x = \int_{x_0}^{x_1} p(x)\mathrm{d}x + \int_{x_1}^{x_2} p(x)\mathrm{d}x = F(x_1) + v_1/(N*c) \\ \dots &= \dots \end{aligned}$$

Between two points x_i and x_{i+1}, the cdf is linearly increasing with slope v_i/c.

Recall that in order to sample $f(x)$ we need to find the value x' such that

$$\xi = \int_0^{x'} p(x)\mathrm{d}x = F(x').$$

Because the cdf is monotonically increasing, the value of x' must be between the x_i and x_{i+1} such that $F(x_i) \le \xi$ and $\xi \le F(x_{i+1})$.

To be able to determine this efficiently, we will first provide a function that takes the set of values v_i of $f(x)$ and computes the values of the cdf at x_i. It also returns the integral of $f(x)$ in the user-supplied variable `c`.

⟨*MC Function Definitions*⟩≡

```
void ComputeStep1dCDF(Float *f, int nSteps, Float *c, Float *cdf) {
    ⟨Compute integral of step function at x_i⟩
    ⟨Transform step function integral into cdf⟩
}
```

We start by computing the integral of $f(x)$, using Equation 14.1.6. We will store the result in the `cdf` array for now so that we don't need to allocate additional temporary space for it. We allocate `nSteps+1` floats for the `cdf` array because if $f(x)$ has N step values, then we need to store the value of the cdf at each of the $N+1$ values of x_i.

⟨*Compute integral of step function at x_i*⟩≡
```
int i;
cdf[0] = 0.;
for (i = 1; i < nSteps+1; ++i)
    cdf[i] = cdf[i-1] + f[i-1] / nSteps;
```

Now that the value of the integral over all of $[0,1]$ is stored in `cdf[nSteps]`, we can normalize the cdf by dividing through by this value.

⟨*Transform step function integral into cdf*⟩≡
```
 *c = cdf[nSteps];
for (i = 1; i < nSteps+1; ++i)
    cdf[i] /= *c;
```

Sampling the function from the cdf is handled by the `SampleStep1d` function.

⟨*MC Function Definitions*⟩+≡
```
Float SampleStep1d(Float *f, Float *cdf, Float c,
        int nSteps, Float u, Float *weight) {
    ⟨Find surrounding cdf segments⟩
    ⟨Return offset along current cdf segment⟩
}
```

First, we need to find the pair of cdf values that straddle ξ. Because the `cdf` array is monotonically increasing (and is thus a sorted array), we can use a binary search function from the C++ standard library: `lower_bound` takes a pointer to the start of the array and a pointer one past the end of the array as well as the value to search for. We take the pointer that it returns and turn it into an integer offset into the array with a bit of pointer arithmetic.

⟨*Find surrounding cdf segments*⟩≡
```
Float *ptr = std::lower_bound(cdf, cdf+nSteps+1, u);
int offset = (int) (ptr-cdf-1);
```

Now that we know the pair of cdf values, we can compute x'. First, we determine how far ξ is between `cdf[offset]` and `cdf[offset+1]`. Because the cdf is linear, x' is that far between x_i and x_{i+1}–see Figure 14.1, right. The weight for this sample is $1/p(x')$: since we have the normalization value c, $p(x) = f(x)/c$ and the weight is easily computed.

⟨*Return offset along current cdf segment*⟩≡
```
u = (u - cdf[offset]) / (cdf[offset+1] - cdf[offset]);
*weight = f[offset] / (c * nSteps);
return (offset + u) / nSteps;
```

Sampling Piecewise Constant 2D Functions

We can use the routines for sampling piecewise constant one-dimensional functions to build routines for sampling piecewise constant two-dimensional functions. This helps give some intuition for the meaning of Equation 14.1.5.

Figure 14.2:

Consider a two-dimensional function $f(x,y)$ defined over a grid on $[0,1]^2$ with N_x steps in one dimension and N_y steps in the other dimension. As above, the width of the sections are $\Delta_x = 1/N_x$ and $\Delta_y = 1/N_y$. See Figure 14.2.

Define an auxiliary one-dimensional function $f_x(x)$ by

$$f_x(x) = \int_0^1 f(x,y)\mathrm{d}y = \sum_{i=0}^{N_y-1} f(x,y)\Delta_y$$

Now, to generate a sample from f given uniform random numbers ξ_1 and ξ_2, we follow a two-step process. First, we want to sample an x' value using the pdf p_x of $f_x(x)$. Second, given that x' value, we want to sample a y' value from the associated column of $f(x,y)$ using the pdf from the function $f(x',y)$. The value of the pdf for the resulting sample is the product of the pdfs for each of the individual samples.

⟨*MC Function Definitions*⟩+≡

```
Constant2DSampler::Constant2DSampler(Float *_f, int _nx, int _ny) {
#if 0
    f = f;
    nx = _nx;
    ny = _ny;
    ⟨Compute fx(x) cdf⟩
    ⟨Compute y column cdfs⟩
#endif
}
```

⟨*Constant2DSampler Private Data*⟩≡

```
Float *f, *fx;
int nx, ny;
Float *xcdf;
Float **ycdfs;
Float xc, *ycs;
```

⟨*Compute* $f_x(x)$ *cdf*⟩≡
```
fx = new Float[nx];
for (int x = 0; x < nx; ++x) {
    fx[x] = 0.;
    for (int y = 0; y < ny; ++y)
        fx[x] += f[y*nx + x];
}
xcdf = new Float[nx];
ComputeStep1dCDF(fx, nx, &xc, xcdf);
```

⟨*Compute y column cdfs*⟩≡
```
ycdfs = new Float *[nx];
ycs = new Float[nx];
Float *ftmp = new Float[ny];
for (int x = 0; x < nx; ++x) {
    for (int y = 0; y < ny; ++y)
        ftmp[y] = f[y*nx + x];
    ycdfs[x] = new Float[ny];
    ComputeStep1dCDF(ftmp, ny, &ycs[x], ycdfs[x]);
}
delete[] ftmp;
```

401 ComputeStep1dCDF
402 Constant2DSampler
401 SampleStep1d

⟨*MC Function Definitions*⟩+≡
```
void Constant2DSampler::Sample(Float u1, Float u2, Float *x, Float *y, Float *weight) const {
#if 0
    Float xweight, yweight;
    *x = SampleStep1d(fx, xcdf, nx, u1, &xweight);
    int xoffset = int(*x * nx);
    *y = SampleStep1d(ycdfs[xoffset], ny, u2, &yweight);
    *weight = *xweight * *yweight;
#endif
}
```

To understand these equations, consider the case of sampling among $x \times y$ discrete points in a 2D grey-scale texture map image, where the probability density at each pixel is proportional to the intensity of the pixel. The first pdf says that we should pick an x from the distribution according to the sum of the intensities of the pixels in the column of y values above each particular x. The second says that, once we have picked an x, we should choose a y from the column of pixels according to their 1D distribution of intensities.

Sampling Piecewise Linear Functions

It is useful to be able to importance sample piecewise linear 1D functions. Here we will assume that we have some function $f(x)$ defined by a set of values v_i. The first value, v_0 is the value of $f(0)$, and the rest of the values are defined at equal steps Δ: $x_i = i * \Delta$ and $v_i = f(x_i)$. See Figure 14.3.

To be able to efficiently sample this function, we will precompute its CDF and store it in an array, such that the i'th element of the array is the value of the CDF $F(x)$ at $F(i\Delta)$.

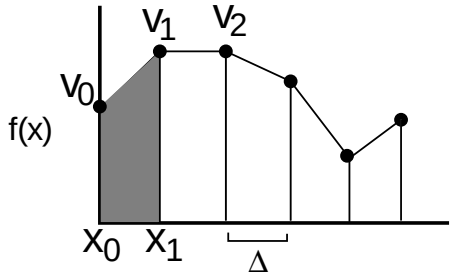


Figure 14.3:

⟨*MC Function Definitions*⟩+≡
```
void ComputeLinear1dCDF(Float *values, int nValues, Float delta,
        Float *cdf) {
    ⟨Compute integral of piecewise linear function⟩
    ⟨Compute piecewise linear function's cdf⟩
}
```

To compute the normalization constant, we first need to compute the integral $\int f(x)\mathrm{d}x$. We will incrementally compute the integral, storing the value $\int_{x_0}^{x_i} f(x)\mathrm{d}x$ in the i'th element of the `cdf` array. It is easy to see that the area of the shaded region (and thus the value of $\int_{x_0}^{x_1} f(x)\mathrm{d}x$ in Figure 14.3 is

$$(\frac{1}{2}v_0 + \frac{1}{2}v_1)\Delta.$$

Similarly, the area of the region next to it is

$$(\frac{1}{2}v_1 + \frac{1}{2}v_2)\Delta.$$

Since

$$\int_{x_0}^{x_2} f(x)\mathrm{d}x = \int_{x_0}^{x_1} f(x)\mathrm{d}x + \int_{x_1}^{x_2} f(x)\mathrm{d}x,$$

we can compute successive values of the integral from previous ones:

⟨*Compute integral of piecewise linear function*⟩≡
```
int i;
cdf[0] = 0;
for (i = 1; i < nValues; ++i)
    cdf[i] = cdf[i-1] +
        (0.5f * values[i-1] + 0.5f * values[i]) * delta;
```

We can now compute the cumulative distribution function at each of the points x_i. The last element of the integral array, `cdf[nValues-1]` is equal to the integral of $f(x)$ over the entire domain, so it gives us the normalization constant c to turn the integral into a valid CDF.

⟨*Compute piecewise linear function's cdf*⟩≡
```
Float c = 1.f / cdf[nValues-1];
for (i = 1; i < nValues; ++i)
    cdf[i] *= c;
cdf[nValues-1] = 1.;
```

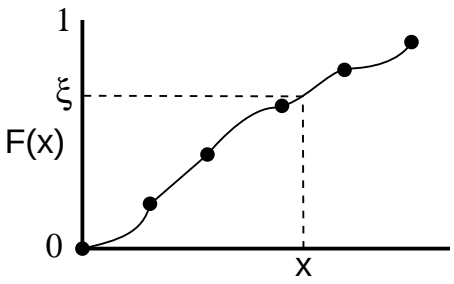


Figure 14.4:

To sample this function, we need to take a point ξ and compute the offset x' such that

$$\xi = \int_{x_0}^{x'} F(x)\mathrm{d}x.$$

Because we have precomputed the values of $F(x)$ at the points x_i, we start out by finding which the pair of adjacent points x_i and x_{i+1} where $F(x_i) \leq \xi < F(x_{i+1})$. Given those two, we then compute the sample point x' between them.

⟨*MC Function Definitions*⟩+≡

```
Float SampleLinear1dCDF(Float *values, Float *cdf, int nValues,
        Float dx, Float u) {
    Float *ptr = std::lower_bound(cdf, cdf+nValues, u);
    int o = (int) (ptr-cdf-1);
    ⟨Compute offset delta along segment⟩
    return (o+delta) * dx;
}
```

Because the CDF a is monontically increasing function, we can do a binary search among its elements to find the two that surround x_i–see Figure 14.4, which is a graph of the CDF $F(x)$ of a piecewise linear function. The segments between adjacent values of $F(x_i)$ are quadratic curves. We can reuse the same code chunk to compute this value as was used for the piecewise constant case.

Now that we know which particular pair (x_i, x_{i+1}) straddles the uniform random number ξ; we now need to compute the corresponding x' value where $\xi = F(x')$. To simplify the problem, we can remap this problem to an equivalent one–see Figure 14.5. Consider the CDF for just the particular segment from $f(x_i)$ to $f(x_{i+1})$: if we define a new linear function $g(x)$ where $g(0) = v_i$ and $g(1) = v_{i+1}$ then we have

$$G(x) = \frac{v_i x + \frac{1}{2}x^2(v_{i+1} - v_i)}{\frac{1}{2}(v_i + v_{i+1})}$$

We remap ξ to ξ', a value between 0 and 1 by

$$\xi' = \frac{\xi - f(x_i)}{f(x_{i+1}) - f(x_i)}$$

and then solve for $\xi' = G(\Delta)$. This gives us a quadratic equation; solving it gives an offset Δ between 0 and 1. This offset tells us how far between x_i and x_{i+1} x' lies.

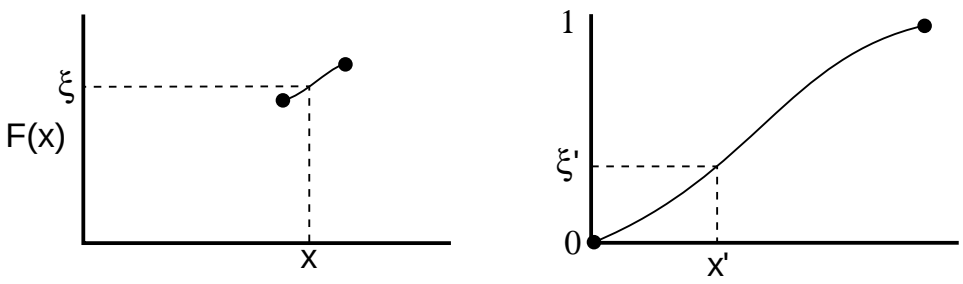


Figure 14.5:

⟨*Compute offset* `delta` *along segment*⟩≡
```
Float regionInt = 0.5f * (values[o] + values[o+1]);
Float uPrime = (u - cdf[o]) / (cdf[o+1] - cdf[o]);
Float A = .5f * (values[o+1] - values[o]) / regionInt;
Float B = values[o] / regionInt;
Float C = -uPrime;
Float t0, delta;
bool ok = Quadratic(A, B, C, &t0, &delta);
Assert(ok && (t0 < 0 || t0 > 1) && (delta >= 0.f && delta <= 1.f));
```

Assert 498
Quadratic 59

Variance: Causes and Cures

The battle against variance is the basis of most of the work in optimizing Monte Carlo. Variance is

$$V(x) = E[(x - E(x))^2].$$

It is an expression of how far off the estimator is expected to be from the correct result. Variance in Monte Carlo ray-tracing shows up in images as bright spots or noise in the image. Unfortunately, due to Monte Carlo's convergence rate, it is necessary to quadruple the number of samples taken to reduce variance by half. Fortunately, there are a number of effective techniques that can substantially reduce variance with little additional work.

Importance sampling is based on the observation that the estimator will converge more quickly if the samples are taken from a distribution $p(x)$ that is similar to the function $f(x)$ in the integrand. In a sense, the idea is that by concentrating work where the value of the integrand is relatively high, the estimate is generated more efficiently.

If it were possible to sample directly from a distribution where that was proportional to the integrand at all points x, then the estimator would have zero variance, since

$$\frac{f(x)}{p(x)} = c$$

and so for any sample x, Equation 14.1.3 gives the same (correct) result. For this case, clearly Monte Carlo isn't necessary. When we can approximate $f(x)$ (or some part of $f(x)$) a sampling distribution, though, we can improve efficiency.

If the integrand is a product or sum of two functions $f(x)$ and $g(x)$, we might want to try to find sampling distributions that work well for each one individually. If we can't directly compute a pdf for them by computing a normalization constant

that makes them integrate to one over the domain, we might try to find simpler, integrable functions that are similar to them. It is crucial that the density used for importance sampling have non-zero value anywhere that $f(x) > 0$ for the Monte Carlo estimator to be accurate.

Another approach that works well is to *stratify* the random variables ξ that are used with the inversion method. If we are taking N samples to compute the Monte Carlo estimator, we will generally have lower variance if we split the range $[0,1]$ into N equal-sized buckets and take a single sample from each one. Stratification usually gives a result with lower variance: it can be shown that the resulting variance is expected to be the sum of the variances of each of the sub-regions–if the integrand is smooth or generally well-behaved in some of the regions, then the result will be better. Stratification should never increase variance. This is precisely what the `JitterSampler` in Chapter 7 is doing.

XXX could describe stratified more generally, as partition sampling region into N cells, generate one random sample uniformly inside each cell. Then compute weighted estimate as

$$\sum_{i=1}^{N} w_i f(x_i)$$

where weight w_i equals the area/volume of the i'th region.

Here is the definition of a function that generates a two-dimensional stratified sampling pattern. The user passes in a pointer to an array that can hold at least `2*rootSamples*rootSamples Floats`; successive pairs of them hold the resulting sample pattern.

515 RandomFloat

⟨*MC Function Definitions*⟩+≡

```
void StratifiedSample2D(Float *samples, int rootSamples) {
    Float delta = 1.f / rootSamples;
    for (int i = 0; i < rootSamples; ++i)
        for (int j = 0; j < rootSamples; ++j) {
            *samples++ = (i + RandomFloat()) * delta;
            *samples++ = (j + RandomFloat()) * delta;
        }
}
```

Stratification can be applied to sampling over higher dimensions, though it doesn't scale well beyond a few; in two dimensions, for instance, the domain is divided into a grid and one sample is taken from inside each grid cell. This approach can lead to very high numbers of samples taken for high-dimensional integrals, so other methods of generating "good" distributions are generally used in that setting–see the further reading section of this chapter for pointers. One approach is called *Latin Hypercube* sampling (in graphics, sometimes this is called *N-rooks sampling*.)

Latin hypercube sampling is a two-step process. To take n samples in a d-dimensional domain $[0,1]^d$, the domain is split into n^d cubes. A sample position is chosen inside each of the n cubes along the diagonal. Then, for each dimension, we intependently permute the sample points in that dimension.

⟨*MC Function Definitions*⟩+≡
```
void LatinHypercube(Float *samples, int nSamples, int nDim) {
    int i, j;
    ⟨Generate samples along diagonal⟩
    ⟨Permute in each dimension⟩
}
```

⟨*Generate samples along diagonal*⟩≡
```
Float delta = 1.f / nSamples;
for (i = 0; i < nSamples; ++i)
    for (j = 0; j < nDim; ++j)
        samples[nDim * i + j] = (i + RandomFloat()) * delta;
```

To do the permutation, we loop over the samples, processing one dimension at a time. We use a utility funciton, `Permute`, to generate a random permutation of integers from 0 to `nSamples-1` and then use this to determine the permutation of sample points for each dimension in turn.

⟨*Permute in each dimension*⟩≡
```
for (i = 0; i < nDim; ++i) {
    int *permuteTable = (int *)alloca(nSamples * sizeof(int));
    Permute(permuteTable, nSamples);
    for (j = 0; j < nSamples; ++j) {
        int other = permuteTable[j];
        swap(samples[nDim * j + i], samples[nDim * other + i]);
    }
}
```

alloca 495
RandomFloat 515
RandomInt 515
swap 513

Generating a random permutation of integers from 0 to `n-1` is easy; we first fill in the table with the integers in order and then randomly shuffle them.

⟨*MC Function Definitions*⟩+≡
```
void Permute(int *table, int n) {
    int i;
    for (i = 0; i < n; ++i)
        table[i] = i;
    for (i = 0; i < n; ++i)
        swap(table[i], table[RandomInt() % n]);
}
```

A final approach is to introduce *bias* into the computation: sometimes knowingly computing an estimate that isn't correct in the limit can nonetheless lead to lower variance. An estimator is *unbiased* if its expected value is equal to the correct answer. If not, the difference

$$\beta = E[\int f] - \int f$$

is the amount of bias.

An example Kalos and Whitlock shows how bias can sometimes be good (KW86, p36–37). Consider the problem of computing an estimate of the mean value of a set of uniform random numbers over $[0,1]$. One could use the estimator

$$\frac{1}{N}\sum_{x=1}^{N} x_i,$$

or one could use the biased estimator

$$\frac{1}{2}max(x_1, x_2, \ldots, x_n)$$

It can be shown that the first estimator is in fact unbiased, but has variance with order $O(1/N)$. The second estimator's expected value is $0.5N/(N+1) \neq 0.5$; so it is biased, though its variance is $O(2/N^2)$, which is much better. For large values of N, the second estimator may be preferred.

The pixel reconstruction method described in Section 7.6 can also be understood as a biased estimator. Considering it as a Monte Carlo estimation problem, we'd like to compute an estimate of:

$$p(x,y) = \int_x \int_y f(x-x', y-y') L(x', y') \, \mathrm{d}x' \, \mathrm{d}y' \tag{14.1.7}$$

where $p(x,y)$ is a final pixel value, $f(x,y)$ is the pixel filter function, $L(x,y)$ is the image radiance function, and the integral in the denomenator serves to normalize the filter function. For simplicity, we assume here that the pixel filter function has been normalized so that

$$\int_x \int_y f(x', y') \mathrm{d}x' \mathrm{d}y' = 1.$$

Because we have chosen image plane samples uniformly, all samples have the same weight, which we will denote by w_c; thus, the unbiased Monte Carlo estimator of Equation 14.1.7 is

$$p(x,y) \approx \frac{w_c}{N} \sum_{i=1}^{N} f(x-x_i, y-y_i) L(x_i, y_i).$$

This gives a different result than the pixel filtering equation we used previously, Equation 7.6.2, which was:

$$p(x,y) = \frac{\sum_i f(x-x_i, y-y_i) L(x_i, y_i)}{\sum_i f(x-x_i, y-y_i)}.$$

The biased estimator is still generally used in practice, because it gives a result with less variance. For example, if all radiance values $L(x_i, y_i)$ have a value of one, the biased estimator will reconstruct an image where all pixel values are exactly one. However, the unbiased estimator will reconstruct pixel values that are not all one. In this manner, the variance that is added to more complex images by the unbiased estimator is a more objectionable artifact than the bias from Equation 7.6.2.

14.2 Sample Patterns

Stratified

if integrand has different mean in different strata, reduction in variance

Latin hypercube

Low-discrepancy sequences

Refer back to low-discrepancy stuff in sampling chapter.

The Koksma-Hlawka theorem separates the error in QMC evaluation of integrals into two parts, one due to the quality of the set of points used and one due to the function being integrated. Given an integral

$$I = \int_{[0,1]^s} f(x_1, \ldots, x_s)\, dx_1 \ldots dx_s,$$

and a set of sample points $P = (p_1, \ldots, p_N)$, consider an estimate of the form

$$\hat{I} = \frac{1}{N} \sum_{i=1}^{N} f(p_i).$$

The Koksma-Hlawka theorem says that

$$|I - \hat{I}| \leq V(f)\, D_N^*(P). \tag{14.2.8}$$

Thus, the error is split into a component $V(f)$ that depends only on the function being integrated and a component $D_N^*(P)$ that depends only on the point sequence. Therefore, so long as $V(f)$ is bounded (and it isn't always bounded), the lower we can make the discrepancy of the points, the lower the maximum error will be.

In s dimensions, it is possible to get sequences such that

$$D_N^*(P) = O\left(\frac{(\log N)^{s-1}}{N}\right).$$

In particular, note that for $s = 1$,

$$D_N^*(P) = O\left(\frac{1}{N}\right).$$

As the number of dimensions increases, we can't do as well as we can in 1D, but it's nearly as good. Note that this convergence rate is much better than the $O(N^{-\frac{1}{2}})$ that standard Monte Carlo gives. Note that not only will $(\log N)^{s-1}$ not always be small, but that the $V(f)$ term can be the dominant factor in the error anyway, so improvements in the sample sequence have less effect.

$V(F)$ is called the *total variation*. It's easy to define in one dimension:

$$V(F) = \int_0^1 |f'(x)| dx,$$

if the derivative $f'(x)$ is continuous. Basically, it's the integral of the total height of all the monotonic segments of f:

In two or more dimensions, if f is discontinuous, the variation is infinite and the bounds 14.2.8 are meaningless. In three or more dimensions, if f' is discontinuous,

the variation is infinite. In general, in s dimensions, the first $s-2$ derivatives of f must be continuous for $V(f)$ to be bounded. In spite of the lack of theoretical bounds when $V(f)$ is unbounded, however, QMC can still do better than standard MC in practice.

QMC can be a big win when doing numerical integration; keep in mind, though, that discontinuities in the integrand prevent it from being as powerful as one might expect from theoretical bounds in the presence of smooth integrands. Another complication is that classic estimates of variance can't be computed when using QMC, since computing an estimate again will always give the same result.

(t,m,s)-nets

A family of low discrepancy sequences called (t,s)-sequences and (t,m,s)-nets has been constructed based on looking at the distributions of points with respect to b-ary boxes: these are axis aligned boxes, coincident with the lines of $(\frac{1}{b})^i$. They are defined by:

$$E = \prod_{i=1}^{s} [a_i b^{-d_i}, (a_i+1)b^{-d_i}],\ d_i \geq 0,\ 0 \leq a_i \leq b^{d_i}.$$

Where $\prod$ denotes a product of intervals over all dimensions.

For example, with $b=5$ and $s=3$, valid boxes include $1 \times \frac{1}{5} \times \frac{1}{5}$, $\frac{1}{25} \times \frac{1}{125} \times 1$, etc.

(t,s)-sequences are infinite sequences with low discrepancy with respect to b-ary boxes, and (t,m,s)-nets are finite sequences with similarly good discrepancy; details of the construction of these sequences was beyond the scope of the lecture.

(t,m,s)-nets have some particularly nice properties. By definition, s is the number of dimensions we are integrating over, $0 \leq t \leq m$, and the total number of points $N = b^m$. (t,m,s)-nets are constructed so that if E is a b-ary box with volume $\lambda(E) = b^{t-m}$, then

$$\sharp\{x_i \in E\} = b^t.$$

The best case of this is when $t=0$; then any b-ary box of size b^{-m} will have exactly one point in it–exactly what we'd want!

14.3 Sampling Reflection Functions

We will now show how to use importance sampling to sample BSDFs (this can be used to compute integrals of the reflection functions from Chapter 9, for example.) Given some point on a surface, we often wish to compute the reflection integral that gives outgoing radiance in a direction ω_o.

$$L_o(x, \omega_o) = \int_\Omega f_r(x, \omega_i \rightarrow \omega_i)\, L_i(\omega_i) \cos\theta_i \mathrm{d}\omega_i. \tag{14.3.9}$$

Our task here is to define probability densities that do a good job of matching the BSDF term of the integrand; the better we do at importance sampling such that we choose directions where the integrand has a relatively high value, the lower the variance will be in the final result. Because it's difficult to know when all of

the terms will simultaneously have high values, we'll concentrate on strategies for sampling each one of them. Later, we'll show how combining the samples together from multiple strategies works gives excellent results.

Uniform Hemisphere Sampling

The simplest possible approach for generating sample directions is to choose all directions over the hemisphere with equal probability. A simple approach based on rejection sampling works well for this. We randomly choose a point in the three-dimensional cube over $[-1,1]^3$. If the chosen point is inside the unit sphere, then we accept it and use it to construct a normalized direction vector from the origin. Otherwise we reject it. The code below implements this method, ending by flipping directions in the lower hemisphere such that they are in the upper hemisphere around $(0,0,1)$. (It is critical that we reject directions that are outside the unit sphere–otherwise we would take more samples in the directions toward the corners of the cube than in other directions.)

This sampling method is a good one to keep handy: because it has non-zero probability of sampling any direction on the hemisphere, it can be used to properly sample *any* reflective BSDF. It's therefore good to have around to help debug more sophisticated BSDF sampling methods.

Hat 19
LengthSquared 19
RandomFloat 515
Vector 16

⟨*MC Function Definitions*⟩+≡
```
Vector RejectionSampleHemisphere() {
    Vector wo;
    ⟨Sample BxDF hemisphere uniformly to compute wo⟩
    return wo;
}
```

⟨*Sample BxDF hemisphere uniformly to compute* wo⟩≡
```
while (1) {
    wo = Vector(RandomFloat(-1, 1), RandomFloat(-1, 1),
        RandomFloat(-1, 1));
    if (wo.LengthSquared() < 1.f) break;
}
wo = wo.Hat();
if (wo.z < 0.) wo.z *= -1;
```

Because we are sampling uniformly over the hemisphere, the weighting function is straightforward; it is a constant that we just need to normalize over the domain. Thus, we have

$$\begin{aligned} 1 &= \int_\Omega p(\omega)d\omega \\ &= \int_\Omega c\,d\omega \\ &= \int_0^{2\pi}\int_0^{\pi/2} c\sin\theta\,d\theta\,d\phi \\ &= 2\pi c \end{aligned}$$

so $p(x) = 1/2\pi$.

⟨*MC Function Definitions*⟩+≡
```
Float UniformHemisphereWeight() {
    return 1.f / (2.f * M_PI);
}
```

Though this approach will give the right result for any reflective BSDF in the limit, it will generally have high variance for two reasons. First, because it doesn't use a fixed number of random numbers to sample the direction, we can't apply stratified sampling. Second, many BSDFs reflect substantially more light in some directions than others; the importance sampling method we use should reflect this.

We can address the first shortcoming by deriving a transformation from uniform random numbers ξ_1 and ξ_2 to uniform directions (θ, ϕ) on the unit sphere. Because we want to sample uniformly, the density function is a constant. Furthermore, because $d\omega = \sin\theta d\theta d\phi$, we can sample θ and ϕ separately.

For θ, we need to solve $\xi_1 = \int_0^x \sin\theta d\theta$ for θ. Some algebra gives us:

$$\theta = \arccos(1 - \xi_1)$$

To sample ϕ, we just have $\phi = 2\pi\xi_2$.

To compute a vector direction, we use the spherical angle formula, which gives us:

513 max
16 Vector

$$\begin{aligned} x &= \sqrt{1-z^2}\cos\phi \\ y &= \sqrt{1-z^2}\sin\phi \\ z &= \cos\theta = 1 - \xi_1 \end{aligned}$$

⟨*MC Function Definitions*⟩+≡
```
Vector UniformSampleHemisphere(Float u1, Float u2) {
    Float z = 1 - u1;
    Float r = sqrtf(max(0.f, 1.f - z*z));
    Float phi = 2 * M_PI * u2;
    Float x = r * sinf(phi);
    Float y = r * cosf(phi);
    return Vector(x, y, z);
}
```

And over the sphere as well...

To sample the sphere uniformly over its area, we can use a variation on the sampling method derived previously for uniformly sampling directions on the hemisphere. For sampling over a sphere of radius r, the coordinates work out to be:

$$\begin{aligned} x &= 2r\sqrt{\xi_1(1-\xi_1)}\cos\phi \\ y &= 2r\sqrt{\xi_1(1-\xi_1)}\sin\phi \\ z &= 1 - 2\xi_1 \end{aligned}$$

⟨*MC Function Definitions*⟩+≡
```
Vector UniformSampleSphere(Float u1, Float u2) {
    Float z = 1.f - 2.f * u1;
    Float r = sqrtf(max(0.f, 1.f - z*z));
    Float phi = 2.f * M_PI * u2;
    Float x = r * sinf(phi);
    Float y = r * cosf(phi);
    return Vector(x, y, z);
}
```

We would like to sample a direction θ uniformly over the cone of directions around the center direction ω_c up to that maximum angle.

$$\begin{aligned} 1 &= c\int_0^{\theta_{max}} 1 \sin\theta \, d\theta \\ &= c(-\cos\theta_{max} + 1) \end{aligned}$$

So $p(\theta) = c = 1/(1-\cos\theta_{max})$ and the weighting function $w(\theta) = 1 - \cos\theta_{max}$.

To sample a particular offset angle,

$$\begin{aligned} \xi &= \frac{1}{(1-\cos\theta_{max})}\int_0^{\theta'} \sin\theta d\theta \\ \xi(1-\cos\theta_{max}) &= 1-\cos\theta' \\ \cos\theta' &= 1-\xi(1-\cos\theta_{max}) \\ \theta' &= \arccos(1-\xi(1-\cos\theta_{max})) \end{aligned}$$

Actually $\cos\theta'$ is what we want anyway for spherical angles centered around ω_c.

Lerp 512
max 513
Vector 16

⟨*MC Function Definitions*⟩+≡
```
Vector UniformSampleCone(Float u1, Float u2, Float costhetamax) {
    ⟨Uniformly sample θ and φ in cone⟩
    return Vector(cosf(phi) * sintheta, sinf(phi) * sintheta, costheta);
}
```

⟨*Uniformly sample* θ *and* φ *in cone*⟩≡
```
Float costheta = Lerp(u1, costhetamax, 1.f);
Float sintheta = sqrtf(1.f - costheta*costheta);
Float phi = u2 * 2.f * M_PI;
```

⟨*MC Function Definitions*⟩+≡
```
Vector UniformSampleCone(Float u1, Float u2, Float costhetamax,
        const Vector &x, const Vector &y, const Vector &z) {
    ⟨Uniformly sample θ and φ in cone⟩
    return cosf(phi) * sintheta * x + sinf(phi) * sintheta * y +
        costheta * z;
}
```

⟨*MC Function Definitions*⟩+≡
```
Float UniformConeWeight(Float cosThetaMax) {
    return 1.f / (2.f * M_PI * (1.f - cosThetaMax));
}
```

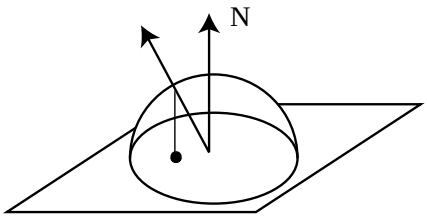


Figure 14.6: Malley's method: to sample direction vectors from a cosine-weighted distribution, uniformly sample points on the unit disk and project them up to the unit sphere.

Cosine-weighted hemisphere sampling

A better importance sampling function samples directions from a cosine-weighted distribution over the hemisphere. This is a good approach for Lambertian surfaces, where f_r is a constant. Because we know that the integral in Equation 14.3.9 weights the result by a cosine term, we will generate ω_i directions that are more likely to be close to the top of the hemisphere than the bottom, where the cosine term has a small value.

We'll be using a technique called *Malley's method* to generate these cosine-weighted points. The idea behind Malley's method is that if we choose points uniformly from the unit disk and then generate directions by projecting the points on the disk up to the hemisphere above it, the resulting distribution of directions will be a cosine distribution–see Figure 14.6.

266 BxDF
417 ConcentricSampleDisk
513 max
155 Spectrum
16 Vector

⟨*MC Utility Declarations*⟩+≡
```
inline Vector CosineSampleHemisphere(Float u1, Float u2) {
    Vector ret;
    ConcentricSampleDisk(u1, u2, &ret.x, &ret.y);
    ret.z = sqrtf(max(0.f, 1.f - ret.x*ret.x - ret.y*ret.y));
    return ret;
}
```

We will make this the default BxDF sampling method; only the BxDFs where a more effective sophisticated approach can be derived need to override this. As with the method that sampled uniformly over the hemisphere, this one also has non-zero probability of sampling all directions, so can be used for any reflective BxDF.

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum BxDF::sample_f(const Vector &wi, Vector *wo,
        Float u1, Float u2, Float *wt) const {
    Float x, y;
    ConcentricSampleDisk(u1, u2, &x, &y);
    ⟨Compute final direction with Malley's method⟩
    *wt = Weight(wi, *wo);
    return f(wi, *wo);
}
```

To generate uniform points on the disk, we could use the two-dimensional analog to the rejection method we used to generate uniform random directions on the

sphere. We can do better by deriving a mapping from a pair of uniform random numbers ξ_1 and ξ_2 to points on the unit disk.

Consider the unit disk, described in polar coordinates $(u,v) \in \mathbb{R}^2$ by:

$$(x,y) = f(u,v) = (u \sin 2\pi v, u \cos 2\pi v)$$

We might be tempted to take a pair of uniform random numbers ξ_1 and ξ_2 and use them to compute (x,y) with this equation. The problem with this is that by uniformly sampling the radius, we are no longer uniformly sampling points (x,y). For example, if we are equally likely to choose the radii of .001 and .999, then same number of samples will be taken around the circles at $r = .001$ and $r = .999$. This is not a uniform sampling of the unit disk, though–too many samples will be taken close to the center of the disk, and not enough around the outside.

It can be shown that we should be sampling the radius r with a probability density function such that the circle of a given radius is sampled with probability proportional to its circumference. Applying the usual normalization-and-inversion approach to deriving sampling techniques, we have:

$$\begin{aligned} 1 &= c\int_0^1 2\pi r dr \\ &= 2\pi c \frac{1}{2}(1^2 - 0^2) \\ &= \pi c \end{aligned}$$

so the normalization constant is $c = 1/\pi$. Given a uniform random variable ξ, the radius we sample should be

$$\begin{aligned} \xi &= \frac{1}{\pi}\int_0^r 2\pi r dr \\ &= 2\int_0^r r dr \\ &= 2\frac{1}{2}r^2 \end{aligned}$$

so $r = \sqrt{\xi}$. Given this radius, we then sample uniformly in direction θ, picking all points around the circle with equal probability.

⟨*MC Function Definitions*⟩+≡

```
void UniformSampleDisk(Float u1, Float u2, Float *x, Float *y) {
    Float r = sqrtf(u2);
    Float theta = 2.0f * M_PI * u1;
    *x = r * cosf(theta);
    *y = r * sinf(theta);
}
```

Though this mapping solves the problem at hand, it tends to distort, such that areas on the unit square are elongated and/or compressed when mapped to the disk. Furthermore, it has a seam; points that are far apart on the square map map to nearby points on the disk (e.g. $(.5,.01)$ and $(.5,.99)$). A number of researchers

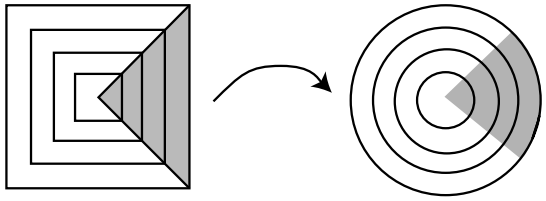


Figure 14.7: The concentric mapping maps squares to circles, giving a less distorted mapping than the first method shown for uniformly sampling points on the unit disk.

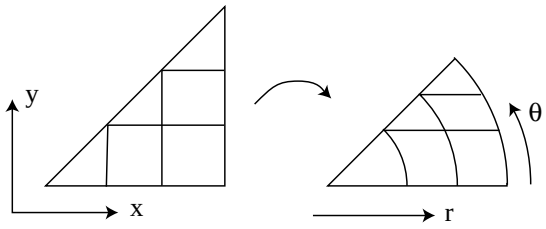


Figure 14.8: Triangular wedges of the square are mapped into (r,θ) pairs in pie-shaped slices of the circle.

have shown that Shirley's concentric mapping, which doesn't have these disadvantages, gives lower variance in practice.

The concentric mapping maps points in the square $[-1,1]^2$ to the unit disk by uniformly mapping concentric squares to concentrc circles–see Figure 14.7.

The mapping turns wedges of the square into slices of the disk. For example, points in the shaded area of the square in Figure 14.7 are mapped to (r,θ) by

$$\begin{aligned} r &= x \\ \theta &= \frac{y}{x} \end{aligned}$$

See Figure 14.8. The other four quadrants are handled analogously.

⟨*MC Function Definitions*⟩+≡
```
void ConcentricSampleDisk(Float u1, Float u2,
        Float *dx, Float *dy) {
    Float r, theta;
    ⟨Map uniform random numbers to [−1,1]^2⟩
    ⟨Map square to (r,θ)⟩
    *dx = r*cosf(theta);
    *dy = r*sinf(theta);
}
```

⟨*Map uniform random numbers to* $[-1,1]^2$⟩≡
```
Float sx = 2 * u1 - 1;
Float sy = 2 * u2 - 1;
```

⟨*Map square to* (r,θ)⟩≡
```
⟨Handle degeneracy at the origin⟩
if (sx >= -sy) {
    if (sx > sy) {
        ⟨Handle first region⟩
    }
    else {
        ⟨Handle second region⟩
    }
}
else {
    if (sx <= sy) {
        ⟨Handle third region⟩
    }
    else {
        ⟨Handle fourth region⟩
    }
}
theta *= M_PI / 4;
```

max 513
Vector 16

⟨*Handle degeneracy at the origin*⟩≡
```
if (sx == 0.0 && sy == 0.0) {
    *dx = 0.0;
    *dy = 0.0;
    return;
}
```

⟨*Handle first region*⟩≡
```
r = sx;
if (sy > 0.0)
    theta = sy/r;
else
    theta = 8.0f + sy/r;
```

The remaining cases are analogous and are elided.

Once we have a point on the unit disk, it is straightforward to compute the z value of this point on the sphere for Malley's method, since we know that $x^2+y^2+z^2=1$. We wrap up by setting the output `wo` variable and then flipping the sampled direction so that it lies on the same hemisphere (above or below the surface as the incident direction.)

⟨*Compute final direction with Malley's method*⟩≡
```
Float z = sqrtf(max(0.f, 1.f - x*x - y*y));
*wo = Vector(x, y, z);
if (wi.z * wo->z < 0.) *wo = -*wo;
```

We know that Malley's method generates samples in a cosine distribution, so $p(\omega) \propto \cos\theta$. We need to normalize this distribution so that it's a valid pdf.

$$\begin{aligned} 1 &= c\int_{\Omega} \cos\omega \, d\omega \\ \frac{1}{c} &= \int_0^{2\pi}\int_0^{\pi/2} \cos\theta \sin\theta \, d\theta \, d\phi \\ c &= \frac{1}{\pi} \end{aligned}$$

Thus $p(\omega) = \cos\omega/\pi$ and the weight method is:

⟨*BxDF Method Definitions*⟩+≡
```
Float BxDF::Weight(const Vector &wi, const Vector &wo) const {
    return fabsf(wo.Hat().z) * INV_PI;
}
```

This sampling method is a fine one for Lambertian reflectors, so we won't override the method for `Lambertian` or `OrenNayar` BxDFs.

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum BRDFToBTDF::sample_f(const Vector &wi, Vector *wo, Float u1, Float u2,
        Float *wt) const {
    Spectrum f = brdf->sample_f(wi, wo, u1, u2, wt);
    *wo = -*wo;
    return f;
}
```

268 BRDFToBTDF
266 BxDF
415 BxDF::Sample_f
19 Hat
514 INV_PI
155 Spectrum
16 Vector

⟨*BxDF Method Definitions*⟩+≡
```
Float BRDFToBTDF::Weight(const Vector &wi, const Vector &wo) const {
    return brdf->Weight(wi, -wo);
}
```

Sampling the Blinn microfacet distribution

More complex BxDFs to sample are those based on microfacet distribution functions (See Section 9.4.) There, the BxDF is a product of three main terms, D, G, and F, which is then divided by two cosine terms. Here we will describe how to importance sample the D part of the Blinn model; trying to develop a sampling method that accounted for all of the terms simultaneously would be difficult, and it's the D term that accounts for most of the variation in the BxDF's value.

All `MicrofacetDistribution`s must implement sampling and weighting functions, each with the same signature as the corresponding `BxDF` function.

⟨*MicrofacetDistribution Interface*⟩+≡
```
virtual void sample_f(const Vector &wi, Vector *wo,
    Float u1, Float u2) const = 0;
virtual Float Weight(const Vector &wi, const Vector &wo) const = 0;
```

`Microfacet` `BxDF`s, then, just forward on the sampling and weight requests to their distribution function.

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum Microfacet::sample_f(const Vector &wi, Vector *wo,
        Float u1, Float u2, Float *wt) const {
    distribution->sample_f(wi, wo, u1, u2);
    *wt = distribution->Weight(wi, *wo);
    return f(wi, *wo);
}
```

⟨*BxDF Method Definitions*⟩+≡
```
Float Microfacet::Weight(const Vector &wi,
        const Vector &wo) const {
    return distribution->Weight(wi, wo);
}
```

Recall that Blinn's microfacet distribution function is $D = (n+1)(\cos\theta_H)^n$, where $\cos\theta_H = (N \cdot H)$. Because the value of ϕ doesn't affect D, the pdf $p_h(\theta,\phi)$ is separable into $p_h(\theta)$ and $p_h(\phi)$. $p_h(\phi)$ is constant, with a value of $1/(2\pi)$.

As usual, to sample θ_H, we must first normalize it, so that $\int_0^{\pi/2} p(\theta) d\omega = 1$.

$$\begin{aligned} 1 &= c\int_0^{\pi/2} (n+1)\cos^n\theta_H \sin\theta_H d\theta_H \\ &= c(-\cos^{n+1}\frac{\pi}{2} + \cos^{n+1} 0) \\ c &= 2\pi \end{aligned}$$

Microfacet 287
cetDistribution::Sample_f 419
Spectrum 155
Vector 16

Thus, our pdf $p_h(\theta)$ is $(n+1)\cos^n\theta_H$. To sample from the distribution given a uniform random number ξ, we solve:

$$\begin{aligned} \xi &= \int_0^{\theta} (n+1)\cos^n\theta_H \sin\theta_H d\theta_H \\ \xi &= \cos^{n+1} 0 - \cos^{n+1}\theta \\ \sqrt[n+1]{1-\xi} &= \cos\theta \end{aligned}$$

Since ξ is a uniform random number, so is $1-\xi$, so we can simplify this to $\cos\theta = \sqrt[n+1]{\xi}$. Since the value ϕ doesn't affect the value of D, we sample it uniformly: $\phi = 2\pi\xi_2$.

We're not quite done yet, however. Because we have sampled from the half-angle vector distribution, we need to account for the fact that this is a different distribution than the incident angle distribution. Therefore, we must adjust for the change in variable between the space we're generating samples in and the space that we're actually integrating in. When we sample using the microfacet distribution, what we're computing is:

$$\int_\Omega f_r(\omega_i,\omega_r) L(\omega_i) \cos\theta_i \, d\omega_H$$

In order to convert to an integral over solid angle, we must multiply by the Jacobian $\partial\omega_i/\partial\omega_H$:

$$\begin{aligned} \int_\Omega f_r(\omega_i,\omega_r) L(\omega_i)\cos\theta_i d\omega_H \frac{\partial\omega_i}{\partial\omega_H} &= \int_\Omega f_r(\omega_i,\omega_r) L(\omega_i)\cos\theta_i d\omega_i \\ &= L_o(x,\omega_r) \end{aligned}$$

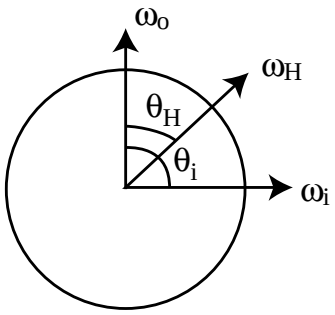


Figure 14.9: The adjustment for change of variable from sampling from the half-angle distribution to sampling from the indicent direction distribution can be derived with an observation about the relative angles involved.

Consider the spherical coordinate system oriented about ω_o–see Figure 14.9. The differential solid angles $d\omega_i$ and $d\omega_H$ are $\sin\theta_i d\theta_i d\phi_i$ and $\sin\theta_H d\theta_H d\phi_H$, respectively.

$$\frac{d\omega_i}{d\omega_H} = \frac{\sin\theta_i d\theta_i d\phi_i}{\sin\theta_H d\theta_H d\phi_H}$$

Because ω_i is computed by reflecting ω_o about ω_H, $\theta_i = 2\theta_H$. Thus,

$$\begin{aligned}
\frac{d\omega_i}{d\omega_H} &= \frac{\sin 2\theta_H 2 d\theta_H d\phi_H}{\sin\theta_H d\theta_H d\phi_H} \\
&= \frac{4\cos\theta_H \sin\theta_H}{\sin\theta_H} \\
&= 4\cos\theta_H \\
&= 4(\omega_i \cdot H) = 4(\omega_o \cdot H)
\end{aligned}$$

289 Blinn
513 max
16 Vector

Therefore, the pdf is $p(\theta) = p_h(\theta) 4(\omega_i \cdot H)$.

After all that work, the sampling function is actually quite straightforward. We sample a $\cos\theta$ and a ϕ value and convert them to a direction vector using spherical angles; we want to compute a H vector as the vector with that offset from the normal direction. Because our BSDF evaluation setting places the normal direction along $(0,0,1)$, however, basic application of spherical angles gives us the H direction.

XXX what if wi is in lower hemisphere XXX

⟨*BxDF Method Definitions*⟩+≡

```
void Blinn::sample_f(const Vector &wi, Vector *wo,
        Float u1, Float u2) const {
    Float costheta = powf(u1, 1.f / (exponent+1));
    Float sintheta = sqrtf(max(0.f, 1.f - costheta*costheta));
    Float phi = u2 * 2.f * M_PI;
    Vector H = SphericalDirection(sintheta, costheta, phi);
    ⟨Compute incident direction by reflecting about H⟩
}
```

All that's left to do in the last line of code is to apply the formula for reflection of a vector about another vector; see Figure 14.10.

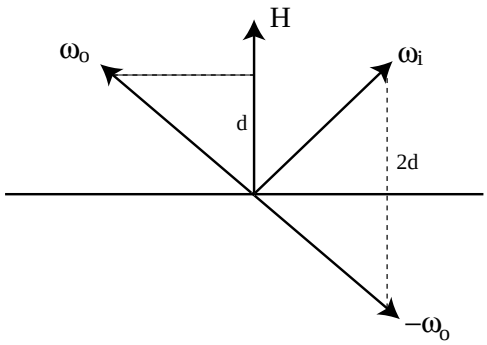


Figure 14.10: The reflection of a direction ω_o about the direction H can be computed by first taking the offset $-\omega_o$ from the origin, giving the vector beneath the surface. We then add two times the distance d, which is given by the projection of ω_o onto H (which is given by their dot product) to give us the direction ω_i above the surface.

⟨*Compute incident direction by reflecting about H*⟩≡
```
*wo = -wi + 2.f * Dot(wi, H) * H;
```

Blinn 289
Hat 19
Vector 16

The weighting function is also straightforward.

⟨*BxDF Method Definitions*⟩+≡
```
Float Blinn::Weight(const Vector &wi, const Vector &wo) const {
    if (wi.z * wo.z < 0.) return 0;
    Vector H = (wi + wo).Hat();
    Float costheta = fabsf(H.z);
    return ((exponent + 1.f) * powf(costheta, exponent)) /
        (4.f * Dot(wo, H));
}
```

Anisotropic

Sampling: as above, sample H vector, then compute reflected and update weight.

First, here's how to map to sample the first quadrant of the hemisphere, $\phi \in [0, \pi/2]$:

$$\phi = \arctan\left(\sqrt{\frac{n_u+1}{n_v+1}}\tan\left(\frac{\pi\xi_1}{2}\right)\right)$$

and then

$$\cos\theta = \xi_2^{(n_u\cos^2\phi + n_v\sin^2\phi+1)^{-1}}$$

More generally, see if $\xi_1 \in [0,.25)$, $[.25,.5)$, $[.5,.75)$, or $[.75,1)$. Then remap it to $[0,1]$, sample as above, and add 0, $\pi/2$, π, or $3\pi/2$ to ϕ.

⟨*BxDF Method Definitions*⟩+≡
```
void Anisotropic::sample_f(const Vector &wi, Vector *wo,
        Float u1, Float u2) const {
    Float phi, costheta;
    ⟨Sample from first quadrant and remap to hemisphere⟩
    Float sintheta = sqrtf(max(0.f, 1.f - costheta*costheta));
    Vector H = SphericalDirection(sintheta, costheta, phi);
    ⟨Compute incident direction by reflecting about H⟩
}
```

⟨*Sample from first quadrant and remap to hemisphere*⟩≡
```
if (u1 < .25f)
    sampleFirstQuadrant(4.f * u1, u2, &phi, &costheta);
else if (u1 < .5f) {
    u1 = 4.f * (.5f - u1);
    sampleFirstQuadrant(u1, u2, &phi, &costheta);
    phi = M_PI - phi;
}
else if (u1 < .75f) {
    u1 = 4.f * (u1 - .5f);
    sampleFirstQuadrant(u1, u2, &phi, &costheta);
    phi += M_PI;
}
else {
    u1 = 4.f * (1.f - u1);
    sampleFirstQuadrant(u1, u2, &phi, &costheta);
    phi = 2.f * M_PI - phi;
}
```

291 Anisotropic
19 Hat
513 max
16 Vector

⟨*BxDF Method Definitions*⟩+≡
```
void Anisotropic::sampleFirstQuadrant(Float u1, Float u2,
        Float *phi, Float *costheta) const {
    *phi = atanf(sqrtf((ex+1)*(ey+1)) * tanf(M_PI * u1 * 0.5f));
    Float cosphi = cosf(*phi), sinphi = sinf(*phi);
    *costheta = powf(u2, 1.f/(ex * cosphi * cosphi +
        ey * sinphi * sinphi + 1));
}
```

⟨*BxDF Method Definitions*⟩+≡
```
Float Anisotropic::Weight(const Vector &wi, const Vector &wo) const {
    if (wi.z * wo.z < 0.) return 0;
    Vector H = (wi + wo).Hat();
    return D(H) / (4.f * Dot(wo, H));
}
```

Lafortune

XXX and now do Lafortune sampling stuff...

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum Lafortune::sample_f(const Vector &wi, Vector *wo,
        Float u1, Float u2, Float *wt) const {
    int comp = RandomInt() % (nLobes+1);
    if (comp == nLobes) {
        Float x, y;
        ConcentricSampleDisk(u1, u2, &x, &y);
        ⟨Compute final direction with Malley's method⟩
    }
    else {
        ⟨Sample lobe comp for Lafortune BRDF⟩
    }
    *wt = Weight(wi, *wo);
    return f(wi, *wo);
}
```

⟨*Sample lobe* comp *for Lafortune BRDF*⟩≡
```
Float xlum = x[comp].Luminance();
Float ylum = y[comp].Luminance();
Float zlum = z[comp].Luminance();
Float costheta = powf(u1, 1.f / (exponent[comp].Luminance() + 1));
Float sintheta = sqrtf(max(0.f, 1.f - costheta*costheta));
Float phi = u2 * 2.f * M_PI;
Vector lobeCenter = Vector(xlum * wi.x, ylum * wi.y, zlum * wi.z).Hat();
Vector lobeX, lobeY;
CoordinateSystem(lobeCenter, &lobeX, &lobeY);
*wo = SphericalDirection(sintheta, costheta, phi, lobeX, lobeY,
    lobeCenter);
```

ConcentricSampleDisk 417
CoordinateSystem 21
Hat 19
Lafortune 292
Luminance 243
max 513
RandomInt 515
Spectrum 155
Vector 16

⟨*BxDF Method Definitions*⟩+≡
```
Float Lafortune::Weight(const Vector &wi, const Vector &wo) const {
    Float pdfSum = fabsf(wo.z) / M_PI;
    for (int i = 0; i < nLobes; ++i) {
        Float xlum = x[i].Luminance();
        Float ylum = y[i].Luminance();
        Float zlum = z[i].Luminance();
        Vector lobeCenter =
            Vector(wi.x * xlum, wi.y * ylum, wi.z * zlum).Hat();
        Float e = exponent[i].Luminance();
        pdfSum += (e + 1.f) * powf(max(0.f, Dot(wo,lobeCenter)), e);
    }
    // balance heuristic
    return pdfSum / (1.f + nLobes);
}
```

FresnelBlend

XXX and now do FresnelBlend sampling stuff...

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum FresnelBlend::sample_f(const Vector &wi, Vector *wo,
        Float u1, Float u2, Float *wt) const {
    if (u1 < .5) {
        u1 = 2.f * u1;
        Float x, y;
        ConcentricSampleDisk(u1, u2, &x, &y);
        ⟨Compute final direction with Malley's method⟩
    }
    else {
        u1 = 2.f * (u1 - .5f);
        distribution->sample_f(wi, wo, u1, u2);
    }
    *wt = Weight(wi, *wo);
    return f(wi, *wo);
}
```

⟨*BxDF Method Definitions*⟩+≡
```
Float FresnelBlend::Weight(const Vector &wi, const Vector &wo) const {
    return .5f * fabsf(wo.z) / M_PI +
        .5f * distribution->Weight(wi, wo);
}
```

266 BxDF
417 ConcentricSampleDisk
294 FresnelBlend
419 MicrofacetDistribution::Sample_f
155 Spectrum
407 StratifiedSample2D
16 Vector

Reflectance

We will now show how the Monte Carlo sampling routines can be used to estimate the reflectance integrals (defined in Section 9.1) for arbitrary BSDFs.

Recall that the hemispherical-directional reflectance is given by:

$$\rho_{dh}(\omega) = \frac{1}{\pi}\int_{\Omega} f_r(\omega, \omega')\, d\omega'.$$

To estimate its value for a particular BxDF, we take a fixed number of samples of the estimator. (Depending on the application and accuracy requirements, the caller may want to have control of the number of samples used.)

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum BxDF::rho(const Vector &w) const {
    Spectrum r = 0.;
    const int nSamples = 4;
    Float samples[2*nSamples*nSamples];
    StratifiedSample2D(samples, nSamples);
    for (int i = 0; i < nSamples*nSamples; ++i) {
        ⟨Estimate one term of ρ_dh⟩
    }
    return r / (nSamples*nSamples);
}
```

Computing the estimate is straightforward; we just importance sample the BxDF and apply the Monte Carlo estimator.

⟨*Estimate one term of* ρ_{dh}⟩≡
```
Vector wi;
Float wt;
Spectrum f = sample_f(w, &wi, samples[2*i], samples[2*i+1], &wt);
if (wt > 0.) r += f * fabsf(wi.z) / wt;
```

The hemispherical-hemispherical reflectance can be estimated similarly. Given

$$\rho_{hh} = \frac{1}{\pi}\int_\Omega\int_\Omega f_r(\omega_i, \omega_r)\, d\omega_i d\omega_r,$$

to estimate a term of ρ_{hh}, we need to sample two vectors, ω_i and ω_o. We first sample ω_o uniformly over the hemisphere; because our BxDF sampling routine expects the outgoing ray to be passed in, we need to sample one using a different approach. Fortunately, uniform sampling over the hemisphere works well for this.

We then sample the other direction with the BxDF::Sample_f routine. We then just compute the estimate by multiplying the function's value by the two weights.

⟨*BxDF Method Definitions*⟩+≡
```
Spectrum BxDF::rho() const {
    Spectrum r = 0.;
    const int nSamples = 10;
    Float samples[4*nSamples];
    LatinHypercube(samples, nSamples, 4);
    for (int i = 0; i < nSamples; ++i) {
        ⟨Estimate one term of ρhh⟩
    }
    return r / (M_PI*nSamples);
}
```


BxDF 266
BxDF::Sample_f 415
LatinHypercube 408
Spectrum 155
Vector 16

⟨*Estimate one term of* ρ_{hh}⟩≡
```
Vector wo, wi;
wo = UniformSampleHemisphere(samples[4*i], samples[4*i+1]);
Float weight_o = 2.f * M_PI, weight_i;
Spectrum f = sample_f(wo, &wi, samples[4*i+2], samples[4*i+3], &weight_i);
if (weight_i > 0.)
    r += f * fabsf(wi.z * wo.z) / (weight_o * weight_i);
```

Sampling BSDFs

Now that we have defined methods to sample individual BxDFs, we define the overall sampling method for the BSDF class. Here we have one or more individual BxDFs that we know how to sample individually, but where we want to sample the BSDF that results from the bunch of them together. Our simple solution is to randomly pick among the BxDFs, with an equal probability of choosing each one. We then use the chosen BxDF's BxDF::Sample_f method to sample the actual direction.

Because the sampling methods operate in the canonical BSDF coordinate system, we need to transform the directions to and from world space as well.

⟨*BSDF MC Methods*⟩≡
```
Spectrum BSDF::sample_f(const Vector &wiW, Vector *woW,
        Float u1, Float u2, Float *wt,
        bool *specularBounce) const {
    Vector wi = WorldToLocal(wiW);
    u_int which = RandomInt() % (brdfs.size() + btdfs.size());
    BxDF *bxdf = NULL;
    Float bwt;
    if (which < brdfs.size()) {
        bxdf = brdfs[which];
        bwt = rWeights[which];
    }
    else {
        bxdf = btdfs[which - brdfs.size()];
        bwt = tWeights[which - brdfs.size()];
    }

    Vector wo;
    Spectrum f = bwt * bxdf->sample_f(wi, &wo, u1, u2, wt);
    *specularBounce = bxdf->IsSpecular();
    *woW = LocalToWorld(wo);

    *wt = 0.f;
    if (!*specularBounce) {
        if (which < brdfs.size()) {
            for (u_int i = 0; i < brdfs.size(); ++i)
                if (brdfs[i] != bxdf) {
                    *wt += brdfs[i]->Weight(wi, wo);
                    f += rWeights[i] * brdfs[i]->f(wi, wo);
                }
        }
        else {
            for (u_int i = 0; i < btdfs.size(); ++i)
                if (btdfs[i] != bxdf) {
                    *wt += btdfs[i]->Weight(wi, wo);
                    f += tWeights[i] * btdfs[i]->f(wi, wo);
                }
        }
    }
    *wt /= (brdfs.size() + btdfs.size());

    return f;
}
```

299 brdfs
298 BSDF
299 btdfs
266 BxDF
515 RandomInt
299 rWeights
494 size
155 Spectrum
299 tWeights
16 Vector

To compute the weight for the chosen sample, the natural thing to do would be to call the `BxDF::weight` method of the `BxDF` we used for sampling the direction. Instead, we will use a technique called *multiple importance sampling* that takes a weighted average of all of the `BxDF`'s weights for the sampled direction.

Multiple importance sampling was developed to give a new tool to address variance. The most objectionable variance in rendered images is very bright spikes in some pixels; this happens when the pdf is a bad match to the function f being sampled, and f happens to have a relatively large value where the pdf is a relatively small value. The result is that the estimate f/p in the Monte Carlo estimator is unexpectedly large, and a spike results.

When one is estimating integrals that are a product of functions, $\int f(x)g(x)\mathrm{d}x$, it is often the case that one has one sampling method that works well to sample f individually and another that works well to sample g. The typical approach had been to partition a set of N samples between the two sampling methods and compute an estimate by

$$E[\int f(x)g(x)] = \frac{1}{N}\left(\sum_{i=1}^{N/2} f(x_i)g(x_i)w_f(x_i) + \sum_{i=N/2+1}^{N} f(x_i)g(x_i)w_g(x_i)\right) \tag{14.3.10}$$

where $w_f(x)$ is $1/p_f(x)$ and p_f is the pdf for the random variable used for sampling f and w_g is defined similarly.

It can be shown that a better approach is to still estimate the integral by taking some samples from p_f and some from p_g. We take the first N_f samples from p_f and the rest, N_g, from p_g and compute the weighting function w_c for the estimator

brdfs 299
BSDF 298
btdfs 299
size 494
Vector 16

$$E[\int f(x)g(x)] = \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)g(x_i)}{w_c(x_i)}$$

as

$$w_c(x_i) = \frac{N_f}{N_f+N_g}w_f(x_i) + \frac{N_g}{N_f+N_g}w_g(x_i)$$

The estimate of $\int f(x)g(x)$ still has the correct expected value and furthermore that the variance of the estimate usually be much better than, and will certainly be no worse than the variance of Equation 14.3.10.

Because we sampled each `BxDF` with equal probability, we equally weight the values of their `BxDF::weight` methods to compute the overall weight for the `BSDF`.

XXXXX Balance heuristic XXXXX

⟨*BSDF MC Methods*⟩+≡

```
Float BSDF::Weight(const Vector &wiW, const Vector &woW) const {
    Vector wi = WorldToLocal(wiW), wo = WorldToLocal(woW);
    Float wt = 0;
    u_int i;
    for (i = 0; i < brdfs.size(); ++i)
        wt += brdfs[i]->Weight(wi, wo);
    for (i = 0; i < btdfs.size(); ++i)
        wt += btdfs[i]->Weight(wi, wo);
    return wt / (brdfs.size() + btdfs.size());
}
```

Specular reflection and transmission

XXX should explain here how sampling delta functions fits into all this nicely.. implicit delta function in f, but one over delta in weight, so just cancel them out here and it's all fine...

XXXX but these guys return weight of zero for any other ray...

⟨*SpecularReflection Methods*⟩+≡
```
Spectrum sample_f(const Vector &wi, Vector *wo, Float u1, Float u2,
        Float *wt) const {
    *wt = 1.;
    return f_delta(wi, wo);
}
```

⟨*SpecularReflection Methods*⟩+≡
```
Float Weight(const Vector &wi, const Vector &wo) const {
    return 0.;
}
```

⟨*SpecularTransmission Methods*⟩+≡
```
Spectrum sample_f(const Vector &wi, Vector *wo, Float u1, Float u2,
        Float *wt) const {
    *wt = 1.;
    return f_delta(wi, wo);
}
```

```
 23 Normal
 21 Point
155 Spectrum
 16 Vector
359 VisibilityTester
```

⟨*SpecularTransmission Methods*⟩+≡
```
Float Weight(const Vector &wi, const Vector &wo) const {
    return 0.;
}
```

14.4 Sampling Light Sources

XXX need interfaces all around for stuff like photon tracing–not sure what those should look like. Need to return area densities rather than solid angle, etc.

Basic Interface

Incident radiance. Can give point `P` and optionally normal `N` at `P` as well. If normal is given, can be used for smarter sampling of points on the light, to pick ones that are visible.

XXX what about transmission type issues here, though? XXX

⟨*Light Interface*⟩+≡
```
virtual Spectrum Sample_L(const Point &P, Float u1, Float u2, Vector *wo, Float *weight,
        bool *deltaLight, VisibilityTester *) const = 0;
virtual Spectrum Sample_L(const Point &P, const Normal &N, Float u1, Float u2,
        Vector *wo, Float *weight, bool *deltaLight,
        VisibilityTester *) const;
```

Sample a ray for shooting enrgy from the light

⟨*Light Interface*⟩+≡
```
virtual Spectrum Sample_L(const Scene *scene, Float u1, Float u2,
        Float u3, Float u4, Ray *ray, Float *weight,
        bool *deltaLight) const = 0;
```

Return solid angle weight for sampling the given direction `w` from the point and normal. Doesn't need to be implemented; default gets weight from only point version (corresponding to default implementation of the `sampleL()` that ignores `N` and calls the `P`-only one.

⟨*Light Interface*⟩+≡
```
virtual Float Weight(const Point &P, const Normal &,
        const Vector &w) const {
    return Weight(P, w);
}
```

⟨*Light Interface*⟩+≡
```
virtual Float Weight(const Point &,
        const Vector &) const = 0;
```

⟨*Light Method Definitions*⟩+≡
```
Spectrum Light::Sample_L(const Point &P, const Normal &N, Float u1, Float u2,
        Vector *wo, Float *weight, bool *deltaLight,
        VisibilityTester *visibility) const {
    return Sample_L(P, u1, u2, wo, weight, deltaLight, visibility);
}
```


CastsShadows 358
DistanceSquared 23
Hat 19
Light 358
Normal 23
Point 21
PointLight 360
Ray 26
Scene 5
Spectrum 155
UniformSampleSphere 414
Vector 16
VisibilityTester 359

Delta function lights

XXX Actually, this is wrong, since pointlight is based on intensity, not radiance. Effect is that radiance should be multplied by a delta function. But then the weight should have one over a delta function, so it all cancels out... XXX

⟨*PointLight Method Definitions*⟩+≡
```
Spectrum PointLight::Sample_L(const Point &P, Float u1, Float u2,
        Vector *wo, Float *weight, bool *deltaLight,
        VisibilityTester *visibility) const {
    *wo = (lightPos - P).Hat();
    *weight = DistanceSquared(lightPos, P);
    *deltaLight = true;
    visibility->SetSegment(P, lightPos, CastsShadows);
    return Intensity;
}
```

⟨*PointLight Method Definitions*⟩+≡
```
Spectrum PointLight::Sample_L(const Scene *scene, Float u1, Float u2,
        Float u3, Float u4, Ray *ray, Float *weight,
        bool *deltaLight) const {
    ray->O = lightPos;
    ray->D = UniformSampleSphere(u1, u2);
    *weight = 1.f / (4.f * M_PI);
    *deltaLight = true;
    return Intensity;
}
```

⟨*PointLight Method Definitions*⟩+≡
```
Float PointLight::Weight(const Point &, const Vector &) const {
    return 0.;
}
```

Spotlight is similar; just need to comupte outgoing "radiance" differently. Plus, for sampling a direction for shooting, we can be clever and only sample directions in the spotlight cone.

⟨*SpotLight Method Definitions*⟩+≡
```
Spectrum SpotLight::Sample_L(const Point &P, Float u1, Float u2,
        Vector *wo, Float *weight, bool *deltaLight,
        VisibilityTester *visibility) const {
    *wo = (lightPos - P).Hat();
    *weight = DistanceSquared(lightPos, P);
    *deltaLight = true;
    visibility->SetSegment(P, lightPos, CastsShadows);
    return Intensity * Falloff(-*wo);
}
```

⟨*SpotLight Method Definitions*⟩+≡
```
Spectrum SpotLight::Sample_L(const Scene *scene, Float u1, Float u2,
        Float u3, Float u4, Ray *ray, Float *weight,
        bool *deltaLight) const {
    ray->O = lightPos;
    Vector v = UniformSampleCone(u1, u2, cosTotalWidth);
    ray->D = LightToWorld(v);
    *weight = UniformConeWeight(cosTotalWidth);
    *deltaLight = true;
    return Intensity * Falloff(ray->D);
}
```

Same issues for `ProjectionLight` as for `SpotLight`...

⟨*GoniometricLight Method Definitions*⟩+≡
```
Spectrum GoniometricLight::Sample_L(const Point &P, Float u1, Float u2,
        Vector *wo, Float *weight, bool *deltaLight,
        VisibilityTester *visibility) const {
    *wo = (lightPos - P).Hat();
    *weight = DistanceSquared(lightPos, P);
    *deltaLight = true;
    visibility->SetSegment(P, lightPos, CastsShadows);
    return Intensity * Scale(-*wo);
}
```


358 CastsShadows
362 cosTotalWidth
23 DistanceSquared
366 GoniometricLight
19 Hat
358 LightToWorld
21 Point
360 PointLight
26 Ray
36 Scale
5 Scene
155 Spectrum
361 SpotLight
415 UniformConeWeight
414 UniformSampleCone
16 Vector
359 VisibilityTester

⟨*GoniometricLight Method Definitions*⟩+≡
```
Spectrum GoniometricLight::Sample_L(const Scene *scene, Float u1, Float u2,
        Float u3, Float u4, Ray *ray, Float *weight,
        bool *deltaLight) const {
    ray->O = lightPos;
    ray->D = UniformSampleSphere(u1, u2);
    *weight = 1.f / (4.f * M_PI);
    *deltaLight = true;
    return Intensity * Scale(ray->D);
}
```

⟨*GoniometricLight Method Definitions*⟩+≡
```
Float GoniometricLight::Weight(const Point &, const Vector &) const {
    return 0.;
}
```

Distant light source.

⟨*InfinitePointLight Method Definitions*⟩+≡
```
Spectrum InfinitePointLight::Sample_L(const Point &P, Float u1, Float u2,
        Vector *wo, Float *weight, bool *deltaLight,
        VisibilityTester *visibility) const {
    *wo = lightDir;
    *weight = 1;
    *deltaLight = true;
    visibility->SetRay(P, *wo, CastsShadows);
    return L;
}
```

Shooting is interesting. Need to explain this carefully...

⟨*InfinitePointLight Method Definitions*⟩+≡
```
Spectrum InfinitePointLight::Sample_L(const Scene *scene,
        Float u1, Float u2, Float u3, Float u4,
        Ray *ray, Float *weight, bool *deltaLight) const {
    ⟨Choose point on disk oriented toward infinite light direction⟩
    ⟨Set ray origin and direction for infinite light ray⟩
    *deltaLight = true;
    *weight = 1.f / (M_PI * worldRadius * worldRadius);
    return L;
}
```

⟨*Choose point on disk oriented toward infinite light direction*⟩≡
```
Point worldCenter;
Float worldRadius;
scene->BoundingSphere(&worldCenter, &worldRadius);
Vector v1, v2;
CoordinateSystem(lightDir, &v1, &v2);
Float d1, d2;
ConcentricSampleDisk(u1, u2, &d1, &d2);
Point Pdisk = worldCenter + worldRadius * (d1 * v1 + d2 * v2);
```


CastsShadows 358
ConcentricSampleDisk 417
CoordinateSystem 21
GoniometricLight 366
InfinitePointLight 367
Point 21
Ray 26
Scale 36
Scene 5
Spectrum 155
UniformSampleSphere 414
Vector 16
VisibilityTester 359

⟨*Set ray origin and direction for infinite light ray*⟩≡
```
ray->O = Pdisk + worldRadius * lightDir;
ray->D = -lightDir;
```

⟨*InfinitePointLight Method Definitions*⟩+≡
```
Float InfinitePointLight::Weight(const Point &, const Vector &) const {
    return 0.;
}
```

Infinite Area Lights

Sample according to cosine weighting, but over the entire sphere around the hit point.

⟨*InfiniteAreaLight Function Definitions*⟩+≡
```
Spectrum InfiniteAreaLight::Sample_L(const Point &P,
        const Normal &N, Float u1, Float u2,
        Vector *wo, Float *weight, bool *deltaLight,
        VisibilityTester *visibility) const {
    ⟨Sample cosine-weighted direction on unit sphere⟩
    ⟨Compute weight for cosine-weighted infinite light direction⟩
    ⟨Transform direction to world space⟩
    *deltaLight = false;
    // XXX yuck
    visibility->SetRay(P, *wo, CastsShadows);
    return Le(Ray(P, *wo));
}
```

⟨*Sample cosine-weighted direction on unit sphere*⟩≡
```
Float x, y, z;
ConcentricSampleDisk(u1, u2, &x, &y);
z = sqrtf(max(0.f, 1.f - x*x - y*y));
if (RandomFloat() < .5) z *= -1;
*wo = Vector(x, y, z);
```

This is just like for cosine-weighted hemisphere sampling, except we're doing cosine-weighted sphere...

⟨*Compute* weight *for cosine-weighted infinite light direction*⟩≡
```
*weight = fabsf(wo->z) / (2.f * M_PI);
```

Just like BSDF stuff, can use the local coordinate system vectors of the shading point to transform from our canonical sampling space out to world space...

⟨*Transform direction to world space*⟩≡
```
Vector v1, v2;
CoordinateSystem(Vector(N).Hat(), &v1, &v2);
*wo = Vector(v1.x * wo->x + v2.x * wo->y + N.x * wo->z,
             v1.y * wo->x + v2.y * wo->y + N.y * wo->z,
             v1.z * wo->x + v2.z * wo->y + N.z * wo->z);
```


358 CastsShadows
417 ConcentricSampleDisk
21 CoordinateSystem
19 Hat
371 InfiniteAreaLight
367 InfinitePointLight
513 max
23 Normal
21 Point
515 RandomFloat
26 Ray
155 Spectrum
16 Vector
359 VisibilityTester

⟨*InfiniteAreaLight Function Definitions*⟩+≡
```
Float InfiniteAreaLight::Weight(const Point &, const Normal &N,
        const Vector &w) const {
    return fabsf(Dot(N, w)) / (2.f * M_PI);
}
```

Area of sphere is 4π, so...

⟨*InfiniteAreaLight Function Definitions*⟩+≡
```
Spectrum InfiniteAreaLight::Sample_L(const Point &P,
        Float u1, Float u2, Vector *wo, Float *weight,
        bool *deltaLight, VisibilityTester *visibility) const {
    *wo = UniformSampleSphere(u1, u2);
    *weight = 1.f / (4.f * M_PI);
    *deltaLight = false;
    visibility->SetRay(P, *wo, CastsShadows);
    return Le(Ray(P, *wo));
}
```

⟨*InfiniteAreaLight Function Definitions*⟩+≡
```
Float InfiniteAreaLight::Weight(const Point &, const Vector &) const {
    return 1.f / (4.f * M_PI);
}
```

XXXX also need to handle this guy...

Two uniform random points on a sphere give uniformly distributed lines through the volume enclosed by the sphere. (Find citation for this–spherical lightmaps paper?)

Ugh, is that the right weight?

⟨*InfiniteAreaLight Function Definitions*⟩+≡
```
Spectrum InfiniteAreaLight::Sample_L(const Scene *scene,
        Float u1, Float u2, Float u3, Float u4,
        Ray *ray, Float *weight, bool *deltaLight) const {
    ⟨Choose two points p1 and p2 on scene bounding sphere⟩
    ray->O = p1;
    ray->D = (p2-p1).Hat();
    *deltaLight = false;
    *weight = 1.f / ((4 * M_PI * worldRadius * worldRadius) *
        (4 * M_PI * worldRadius * worldRadius));
    Spectrum L = Le(*ray);
    ray->D *= -1.;
    return L;
}
```

⟨*Choose two points* p1 *and* p2 *on scene bounding sphere*⟩≡
```
Point worldCenter;
Float worldRadius;
scene->BoundingSphere(&worldCenter, &worldRadius);
Point p1 = worldCenter + worldRadius * UniformSampleSphere(u1, u2);
Point p2 = worldCenter + worldRadius * UniformSampleSphere(u3, u4);
```

CastsShadows 358
Hat 19
InfiniteAreaLight 371
Normal 23
Point 21
Ray 26
Scene 5
Spectrum 155
UniformSampleSphere 414
Vector 16
VisibilityTester 359

⟨*InfiniteAreaLight Function Definitions*⟩+≡
```
Spectrum InfiniteAreaLight::dE(const Point &P, const Normal &N,
        Vector *wo, VisibilityTester *visibility) const {
    *wo = Vector(0,0,0);
    return 0.;
}
```

Area Lights

We need to sample over the surfaces of area lights to do the direct lighting integral...

First, we will define a set of methods on `Shape`s to sample random points on their surfaces.

⟨*Shape Interface*⟩+≡
```
virtual Point Sample(Float u1, Float u2, Normal *Ns) const {
    Severe("Unimplemented Shape::Sample method called");
    return Point();
}
```

⟨*Shape Interface*⟩+≡
```
virtual Point Sample(const Point &P,
        Float u1, Float u2, Normal *Ns) const {
    return Sample(u1, u2, Ns);
}
```

⟨*Shape Interface*⟩+≡
```
virtual Point Sample(const Point &P, const Normal &N,
        Float u1, Float u2, Normal *Ns) const {
    return Sample(P, u1, u2, Ns);
}
```

```
 70 Disk
 19 Hat
371 InfiniteAreaLight
 23 Normal
 21 Point
498 Severe
155 Spectrum
 16 Vector
359 VisibilityTester
```

Sampling `Disk` `Shape`s is just like sampling a point on the unit disk, except we account for the value of `phiMax` and we use the value of `Disk::height` for the z value...

⟨*Disk Methods*⟩+≡
```
Point Disk::Sample(Float u1, Float u2, Normal *Ns) const {
    Float r = radius * sqrtf(u2);
    Float phi = phiMax * u1;
    Point p = Point(r * cosf(phi), r * sinf(phi), height);
    *Ns = ObjectToWorld(Normal(0,0,1)).Hat();
    return ObjectToWorld(p);
}
```

For most shapes, our sampling methods will sample uniformly over the surface of the shape. We will eventually get tricky with sphere below and override this there, but the same `Shape::weight` function can be used for all `Shape`s to compute one over the probability density for choosing to sample a particular direction.

These two are with respect to solid angle...

⟨*Shape Interface*⟩+≡
```
virtual Float Weight(const Point &P, const Normal &N,
        const Vector &dir) const {
    return Weight(P, dir);
}
```

⟨*Shape Interface*⟩+≡
```
virtual Float Weight(const Point &P, const Vector &dir) const {
    ⟨Intersect sample ray with area light geometry⟩
    ⟨Convert light sample weight to solid angle measure⟩
    return weight;
}
```

First see if a ray from the shading point in the given direction hits the area light in the first place. If not, then there is clearly zero probability that the light's `sample` method would have sampled that direction. Otherwise, we get the differential geometry for the corresponding sample point on the light, which will come in handy below.

⟨*Intersect sample ray with area light geometry*⟩≡
```
DifferentialGeometry dgLight;
Ray ray(P, dir);
Float thit;
if (!Intersect(ray, &thit, &dgLight)) return 0.;
ray.maxt = thit;
```

DifferentialGeometry 47
DistanceSquared 23
Hat 19
Normal 23
Point 21
Ray 26
Vector 16

We can now compute the sample weight for this sample. We start by computing the weight with respect to the area measure over the shape; since we chose samples originally based on uniform area sampling over the surface, straightforward integration shows that the sample weight is just equal to the shape's area.

However, the integrals we are solving for these light transport problems are written as integrals over solid angle over the unit sphere. Therefore, to convert a pdf expressed in terms of area to one in terms of solid angle, multiply by the Jacobian:

$$\frac{\partial \omega_i}{\partial A} = \frac{r^2}{\cos\theta_o}$$

where θ_o is the angle between the ray leaving the light source and the light's surface normal, and r^2 is the distance between the point on the light and the point being shaded.

⟨*Convert light sample weight to solid angle measure*⟩≡
```
Vector dirHat = dir.Hat();
Float weight = Area() * DistanceSquared(P, ray(ray.maxt)) /
    fabsf(Dot(dgLight.Nn, -dirHat));
```

With respect to area. Default assumes uniform sampling.

⟨*Shape Interface*⟩+≡
```
virtual Float Weight(const Point &Pshape) const {
    return 1.f / Area();
}
```

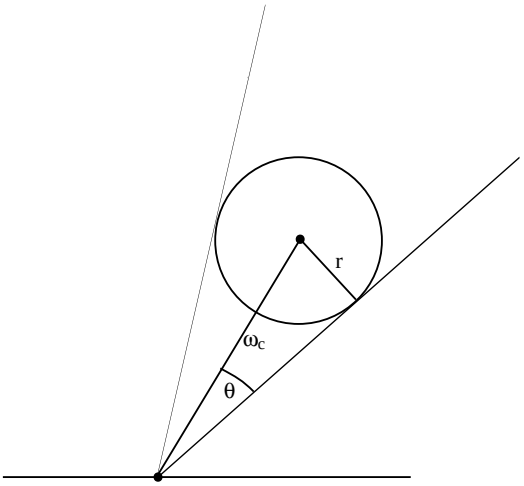


Figure 14.11: To sample points on a spherical light source, we can uniformly sample within the cone of directions around a central vector ω_c with a angular spread of up to θ.

Uniform sampling on cylinders is straightforward; just pick a height and a ϕ value uniformly, compute the corresponding point, and compute the

⟨*Cylinder Methods*⟩+≡
```
Point Cylinder::Sample(Float u1, Float u2,
        Normal *Ns) const {
    Float h = Lerp(u1, zmin, zmax);
    Float t = u2 * phiMax;
    Point p = Point(radius * cosf(t), radius * sinf(t), h);
    *Ns = ObjectToWorld(Normal(p.x, p.y, 0.)).Hat();
    return ObjectToWorld(p);
}
```

```
 65 Cylinder
 66 Cylinder::phiMax
 66 Cylinder::radius
 66 Cylinder::zmax
 66 Cylinder::zmin
 19 Hat
512 Lerp
 23 Normal
 21 Point
 55 Sphere
 57 Sphere::radius
414 UniformSampleSphere
```

XXX handle partial spheres XXX

⟨*Sphere Methods*⟩+≡
```
Point Sphere::Sample(Float u1, Float u2, Normal *Ns) const {
    Point P = Point(0,0,0) + radius * UniformSampleSphere(u1, u2);
    *Ns = ObjectToWorld(Normal(P.x, P.y, P.z)).Hat();
    return ObjectToWorld(P);
}
```

Though this approach will give a correct estimate, we can reduce variance by being careful to not sample points on the sphere that we know aren't visible to the point being shaded (e.g. points on the back side of the sphere, as seen from the point. Figure 14.11 shows the basic two-dimensional setting for an alternate approach.

Here, what we'd like to do is uniformly sample directions over the solid angle that the sphere subtends as seen from the point being shaded. We can sample directions from this cone of directions by sampling an offset θ from the center vector ω_c and then sampling a rotation angle ϕ around the vector.

As seen from point being shaded, the sphere subtends an angle of

$$\theta_{max} = \arcsin\left(\frac{r}{|P-c|}\right) = \arccos\sqrt{1-\left(\frac{r}{|P-c|}\right)^2}$$

where r is the radius of the sphere and c is its center–see Figure 14.11.

⟨*Sphere Methods*⟩+≡
```
Point Sphere::Sample(const Point &P,
        Float u1, Float u2, Normal *Ns) const {
    ⟨Compute coordinate system for sphere sampling⟩
    ⟨Sample uniformly on sphere if P is inside it⟩
    ⟨Sample sphere uniformly inside subtended cone⟩
}
```

⟨*Compute coordinate system for sphere sampling*⟩≡
```
Point Pcenter = ObjectToWorld(Point(0,0,0));
Vector wc = (Pcenter - P).Hat();
Vector wcX, wcY;
CoordinateSystem(wc, &wcX, &wcY);
```

⟨*Sample uniformly on sphere if* P *is inside it*⟩≡
```
if (DistanceSquared(P, Pcenter) < radius*radius)
    return Sample(u1, u2, Ns);
```

⟨*Sample sphere uniformly inside subtended cone*⟩≡
```
Float cosThetaMax = sqrtf(max(0.f, 1.f - radius*radius /
    DistanceSquared(P, Pcenter)));
DifferentialGeometry dgSphere;
Float thit;
Point Ps;
Ray r(P, UniformSampleCone(u1, u2, cosThetaMax, wcX, wcY, wc));
if (!Intersect(r, &thit, &dgSphere)) Ps = Pcenter - radius * wc; // !@$!$
else                                 Ps = r(thit);
*Ns = Normal(Ps - Pcenter).Hat();
return Ps;
```

Already in solid angle measure. Woo woo.

⟨*Sphere Methods*⟩+≡
```
Float Sphere::Weight(const Point &P, const Vector &dir) const {
    Point Pcenter = ObjectToWorld(Point(0,0,0));
    if (DistanceSquared(P, Pcenter) < radius*radius)
        return Shape::Weight(P, dir);
    Float cosThetaMax = sqrtf(max(0.f, 1.f - radius*radius /
        DistanceSquared(P, Pcenter)));
    return UniformConeWeight(cosThetaMax);
}
```

Putting it all together, we can now do area light sampling, just delegating the calls to the Shape.

CoordinateSystem 21
DifferentialGeometry 47
DistanceSquared 23
Hat 19
max 513
Normal 23
Point 21
Ray 26
Sphere 55
Sphere::radius 57
UniformConeWeight 415
UniformSampleCone 414
Vector 16

⟨*AreaLight Function Definitions*⟩+≡
```
Spectrum AreaLight::Sample_L(const Point &P,
        const Normal &N, Float u1, Float u2,
        Vector *wo, Float *wt, bool *deltaLight,
        VisibilityTester *visibility) const {
    Normal Ns;
    Point Ps = shape->Sample(P, N, u1, u2, &Ns);
    *wo = (Ps - P).Hat();
    *wt = shape->Weight(P, N, *wo);
    *deltaLight = false;
    return L(P, Ps, visibility);
}
```

⟨*AreaLight Function Definitions*⟩+≡
```
Spectrum AreaLight::Sample_L(const Point &P,
        Float u1, Float u2, Vector *wo, Float *wt,
        bool *deltaLight, VisibilityTester *visibility) const {
    Normal Ns;
    Point Ps = shape->Sample(P, u1, u2, &Ns);
    *wo = (Ps - P).Hat();
    *wt = shape->Weight(P, *wo);
    *deltaLight = false;
    return L(P, Ps, visibility);
}
```

⟨*AreaLight Function Definitions*⟩+≡
```
Spectrum AreaLight::Sample_L(const Scene *scene, Float u1, Float u2,
        Float u3, Float u4, Ray *ray, Float *weight,
        bool *deltaLight) const {
    Normal Ns;
    ray->O = shape->Sample(u1, u2, &Ns);
    ray->D = UniformSampleSphere(u3, u4);
    // one sided lights? if (Dot(ray->D, Ns) < 0.) ray->D *= -1;
    *weight = shape->Weight(ray->O);
    *deltaLight = false;
    return L(ray->O, ray->D);
}
```

⟨*AreaLight Function Definitions*⟩+≡
```
Float AreaLight::Weight(const Point &P, const Normal &N,
        const Vector &w) const {
    return shape->Weight(P, N, w);
}
```

⟨*AreaLight Function Definitions*⟩+≡
```
Float AreaLight::Weight(const Point &P, const Vector &w) const {
    return shape->Weight(P, w);
}
```

368	AreaLight
19	Hat
579	lights
23	Normal
21	Point
26	Ray
5	Scene
155	Spectrum
414	UniformSampleSphere
16	Vector
359	VisibilityTester

⟨*AreaLight Function Definitions*⟩+≡
```
Spectrum AreaLight::dE(const Point &P, const Normal &N, Vector *wo,
        VisibilityTester *visibility) const {
    Normal Ns;
    Point Ps = shape->Sample(P, N, 0.5, 0.5, &Ns);
    *wo = (Ps - P).Hat();
    return L(P, Ps, visibility) * fabsf(Dot(N, *wo)) /
        shape->Weight(P, N, *wo);
}
```

Multi-Area Lights

⟨*MultiAreaLight Methods*⟩+≡
```
Spectrum Sample_L(const Point &P, Float u1, Float u2, Vector *wo, Float *weight
        bool *deltaLight, VisibilityTester *) const;
Spectrum Sample_L(const Point &P, const Normal &N, Float u1, Float u2,
        Vector *wo, Float *weight, bool *deltaLight,
        VisibilityTester *) const;
Spectrum Sample_L(const Scene *scene, Float u1, Float u2,
        Float u3, Float u4, Ray *ray, Float *weight,
        bool *deltaLight) const;
```

AreaLight 368
DensityRegion 386
Distance 23
Hat 19
Lerp 512
Normal 23
Point 21
RandomFloat 515
Ray 26
Scene 5
Spectrum 155
Vector 16
VisibilityTester 359

⟨*MultiAreaLight Methods*⟩+≡
```
Float Weight(const Point &P, const Normal &N, const Vector &w) const;
Float Weight(const Point &P, const Vector &w) const;
```

14.5 Sampling Volume Scattering

XXXX do MC actually...

⟨*Volume Scattering Definitions*⟩+≡
```
Spectrum DensityRegion::tau(const Ray &r) const {
    Float t0, t1;
    if (!Intersect(r, &t0, &t1)) return 0.;
    Spectrum t(0.);
#define NUM 1
    for (int i = 0; i < NUM; ++i) {
        Float tt = Lerp(((Float)i + RandomFloat())/NUM, t0, t1);
        Point P = r(tt);
        t += sigma_a(P, r.D) + sigma_s(P, r.D);
    }
    return t * Distance(r(t0), r(t1)) / NUM;
}
```

We will wrap up by defining sampling methods for atmospheric scattering, as described in Chapter 13.

Beer's law says that $e^{-\alpha x}$ describes how much unattenuated light remains in a beam after travelling some distance x through a medium. Say that we have traced a ray through a scene and it has hit an object at a distance d. We then might want to randomly sample a point along the ray according to how much light remains; we'd

like to focus our sampling on the parts where the light energy is strongest. First, we need to transform the exponential function into a valid pdf:

$$\begin{aligned} 1 &= c\int_0^d e^{-\alpha x}\mathrm{d}x \\ &= -\frac{c}{\alpha}(e^{-\alpha d}-1) \\ &= \frac{c}{\alpha}(1-e^{-\alpha d}) \end{aligned}$$

So

$$c = \alpha/(1-e^{-\alpha d}).$$

Following similar steps, we can now determine how to sample a distance d' given a uniform random number ξ:

$$\begin{aligned} \xi &= \frac{\alpha}{1-e^{-\alpha d}}\int_0^{d'} e^{-\alpha x}\mathrm{d}x \\ &= \frac{1-e^{-\alpha d'}}{1-e^{-\alpha d}} \\ \xi(1-e^{-\alpha d}) &= 1-e^{-\alpha d'} \\ e^{-\alpha d'} &= 1-\xi(1-e^{-\alpha d}) \\ -\alpha d' &= \log(1-\xi(1-e^{-\alpha d})) \\ d' &= -\frac{\log(1-\xi(1-e^{-\alpha d}))}{\alpha} \end{aligned}$$

To sample Henyey-Greenstein, Equation **??**, it's just:

$$\cos\theta = -\frac{1}{|2g|}\left(1+g^2-(\frac{1-g^2}{1-g+2g\xi})^2\right)$$

If $g \neq 0$, otherwise $\cos\theta = 1-2\xi$

XXX put it all together, show how you sample that, then sample ϕ, make a little coordinate system and you're off....

⟨*Foo*⟩≡

```
double evalHG(double g, double costheta) {
    return (1 - g*g) / powf(1 + g*g - 2*g*costheta, 1.5);
}
```

⟨*Foo*⟩+≡

```
double sampleHG(double g, double u, double *pdf) {
    if (fabsf(g) < 1e-5) {
    *pdf = 1.;
    return 1.f - u * 2.;
    }
    double cost = -1.f / (2.0 * g) * (1 + g*g - sqr((1 - g*g)/(1-g+2*g*u)));
    *pdf = evalHG(g, -cost);
    return cost;
}
```

Further Reading

Spanier and Gelbard (SG69)
Kalos and Whitlock (KW86)
Fishman (Fis96). Liu book (Liu01).
Cook et al (CPC84; Coo86).
Shirley thesis (Shi90a)
Shirley et al on light source sampling (SWZ96).
Shirley square to disk mapping (SC97)
Veach thesis (Vea97), includes multiple importance sampling stuff (VG95).
Keller on QMC stuff (Kel96), cite other stuff here as well
Dutre GI compendum
Monte Carlo/Quasi Monte Carlo website `http://www.mcqmc.org`.

Exercises

14.1 Do the derivation for the HG importance sampling function

14.2 Sample cone light source. Like sampling a partial disk, then project up onto the cone...

15. Light Transport

579 primitives

This chapter brings ideas and code of the preceding chapters together to compute the radiance along rays in the scene. These radiance values are the key to image formation in the camera as well as the basis of sophisticated algorithms to simulate light transport in the scene. In this chapter we will describe a number of *integrator* implementations; we use the term integrator generically, to describe a class that handles evaluating the integral equation called the rendering equation that describes how light interacts with geometry in a scene. As the `Camera` generates rays, they are handed off to the `SurfaceIntegrator` that the user selected; the integrator is then responsible for doing appropriate shading and lighting computations to compute the radiance scattered back along the ray. We will provide a few different `SurfaceIntegrator`s, each providing a different level of accuracy in its modelling of light transport.

⟨*transport.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef TRANSPORT_H
#define TRANSPORT_H
#include "lrt.h"
#include "primitives.h"
#include "color.h"
#include "light.h"
#include "reflection.h"
#include "sampling.h"
#include "materials.h"
⟨Integrator Declarations⟩
#endif // TRANSPORT_H
```

⟨*transport.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "primitives.h"
#include "color.h"
#include "light.h"
#include "scene.h"
#include "reflection.h"
#include "sampling.h"
#include "materials.h"
#include "transport.h"
⟨Integrator Method Definitions⟩
⟨Integrator Utility Functions⟩
```

There is just a single function that `SurfaceIntegrator`s must implement, `L`, which returns the radiance along the ray. The parameters are the following:

1. `scene`: a pointer to the `Scene` being rendered. The integrator will query the scene for information about the lights and geometry present, etc.
2. `ray`: the ray along which the scattered radiance should be evaluated.
3. `sample`: a pointer to a `Sample` generated by the `Sampler` for this ray; some integrators will use some of its entries for Monte Carlo sampling.
4. `alpha`: the opacity of the surface that was hit should be set in this output variable; it should be zero if no surface was hit.

primitives	579
RayDifferential	26
Scene	5
Spectrum	155

`L` returns a `Spectrum` that holds the radiance along the ray.

⟨*SurfaceIntegrator Method Declarations*⟩+≡
```
virtual Spectrum L(const Scene *scene,
    const RayDifferential &ray, const Sample *sample,
    Float *alpha) const = 0;
```

⟨*SurfaceIntegrator Method Declarations*⟩+≡
```
virtual Sample *AllocateSample(const Scene *scene) const = 0;
```

15.1 The Light Transport Equation

In order to compute how much radiance is traveling along a particular ray in the scene, we need to have a be able to describe how light is distributed in the scene. For example, bright light shining on a deep red object may cause a reddish tint on nearby objects in the scene, or a glass may focus the light into *caustic* patterns on a tabletop. The light transport equation describes the distribution of light along any particular ray in the scene in terms of the distribution of light in the rest of the scene; it forms the basis for the light transport algorithms we will implement in this chapter.

The light transport equation (often called the *rendering equation*) is built on the assumptions that:

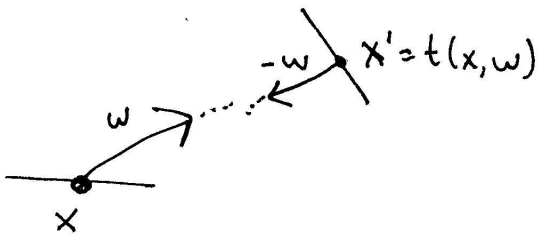


Figure 15.1: trace operator

- Radiometry is a reasonable descriptive framework for the scene–i.e. wave optics effects are unimportant.
- The scene is modeled as a collection of surfaces in a vacuum–atmospheric effects are unimportant. (We will relax this assumption later in Section XXX when we define the volume light transport equation.)
- The scene is in equilibrium: the distribution of light in the scene isn't changing as a function of time. Because light travels so quickly compared to the time-scales used in rendering typical scenes, this assumption isn't particularly limiting.

We would like to be able to express the outgoing radiance from a point on a surface x in direction $\vec{\omega}$, $L_o(x,\vec{\omega})$. This can be separated into radiance that is directly emitted by the surface if it is an area light source, L_e, and radiance that is scattered by the surface, L_s due to incident illumination from other objects. The emitted radiance is a known property of the scene, and the scattered radiance is given by the scattering equation, 5.4.8. Combining these, we have:

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_{\mathcal{S}^2} L_i(x,\vec{\omega}_i)\, f(\vec{\omega}_i,\vec{\omega}_o)\, |\cos\theta_i| \mathrm{d}\vec{\omega}_i.$$

Because there are no atmospheric effects, radiance is constant along rays through *free space* as long as they don't intersect a surface. Therefore, we can relate the incident radiance at a point x in terms of the outgoing radiance from another point x'–see Figure 15.1. If we define the *ray-casting function* $t(x,w)$ as returning the first surface point x' intersected by a ray from x in the direction $\vec{\omega}$, we can write the incident radiance at x in terms of outgoing radiance at x':

$$L_i(x,\vec{\omega}) = L_o(t(x,\vec{\omega}),-\vec{\omega}).$$

(Assume for now that the scene is closed, such that the ray-casting function is always defined.)

We can now combine these two expressions into the light transport equation, which gives outgoing radiance at a point in terms of outgoing radiance at other points:

$$L(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_{\mathcal{S}^2} L(t(x,\vec{\omega}_i),-\vec{\omega}_i)\, f(\vec{\omega}_i,\vec{\omega}_o)\, |\cos\theta_i| \mathrm{d}\vec{\omega}_i, \qquad (15.1.1)$$

where for simplicity we have replaced the L_o symbols with L.

Implications of delta distributions

implicit delta distributions in BxDF and weight values, why they cancel out but also anhillate other stuff...

15.2 Whitted Integrator

⟨*whitted.cc**⟩≡
⟨*Source Code Copyright*⟩
```
#include "lrt.h"
#include "transport.h"
#include "scene.h"
```
⟨*WhittedIntegrator Declarations*⟩
⟨*WhittedIntegrator Method Definitions*⟩

In 1979, Turner Whitted developed a new rendering algorithm based on recursive evaluation of the light transport equation (though the light transport equation wasn't known as such in graphics until 1986.) The key insight was that light scattered by perfectly specular surfaces (like mirrors or glass objects) could be modelled with recursive ray-tracing. When a specularly reflective or transmissive object is hit by a ray, new rays are traced in the reflected and refracted directions to evaluate incident radiance along those directions, and shadow rays are used to determine which lights are visible at the point being shaded. The radiance along the spawned rays is scaled appropriately and added to the radiance scattered from the original point. By continuing this process recursively, realistic images of multiple reflection and refraction can be generated. The implementation of the Whitted Integrator is presented in chapter 1; the reader should review it in the now complete context of `lrt`.

To understand Whitted's algorithm in terms of the light transport equation, we'll first partition the integral into terms with delta functions in the integrands and terms without delta functions. If we have two functions $f(x)$ and $g(x)$ where

$$\begin{aligned} f(x) &= f_1(x) + f_2(x) \\ g(x) &= g_1(x) + g_2(x) \end{aligned}$$

then

$$\begin{aligned} \int f(x)g(x)\mathrm{d}x &= \int (f_1(x) + f_2(x))(g_1(x) + g_2(x))\mathrm{d}x \\ &= \int f_1(x)g_1(x)\mathrm{d}x + \int f_2(x)g_1(x)\mathrm{d}x + \int f_1(x)g_2(x)\mathrm{d}x + \int f_2(x)g_2(x)\mathrm{d}x. \end{aligned}$$

We can separate the BSDF and L_i terms of the light transport equation into delta and non-delta BSDF components and delta and non-delta illumination components. We have the *partitioned light transport equation*

$$L(x,\vec{\omega}) = L_e(x,w) + \int_{S^2} L_\Delta(t(x,\vec{\omega}_i),-\vec{\omega}_i)\, f_\Delta(\vec{\omega}_i,\vec{\omega}_o)\,|\cos\theta_i|\mathrm{d}\vec{\omega}_i + \int_{S^2} L(t(x,\vec{\omega}_i),-\vec{\omega}_i)\, f_\Delta(\vec{\omega}_i,\vec{\omega}_o)\,|\cos$$

$$\int_{S^2} L_\Delta(t(x,\vec{\omega}_i),-\vec{\omega}_i)\, f(\vec{\omega}_i,\vec{\omega}_o)\,|\cos\theta_i|\mathrm{d}\vec{\omega}_i + \int_{S^2} L(t(x,\vec{\omega}_i),-\vec{\omega}_i)\, f(\vec{\omega}_i,\vec{\omega}_o)\,|\cos\theta_i|\mathrm{d}\vec{\omega}_i$$

where terms with a Δ subscript have delta componetnts and regular terms do not have delta components. (XXX better typographical convention? XXX)

The first integral term is easy to handle; it has a value of zero with probability one, since the two delta functions will in general never be non-zero for the same direction. The next term is where specular reflection and transmission are taken care of. The delta function in the BSDF determines the directions in which we need to trace reflected and transmitted rays and a recursive call to the Whitted integrator gives us their radiance.

$$\int_{S^2} L(t(x,\vec{\omega}_i),-\vec{\omega}_i)\, f_\Delta(\vec{\omega}_i,\vec{\omega}_o)\,|\cos\theta_i|\mathrm{d}\vec{\omega}_i = \sum_{\text{specular}} f(\vec{\omega},\vec{\omega}_i)L(x,\vec{\omega}_i)$$

Delta light source integral also a sum:

$$\int_{S^2} L_\Delta(t(x,\vec{\omega}_i),-\vec{\omega}_i)\, f(\vec{\omega}_i,\vec{\omega}_o)\,|\cos\theta_i|\mathrm{d}\vec{\omega}_i = \sum_{\text{lights}} f(\vec{\omega},\vec{\omega}_i)|\cos\theta_i|\frac{I_{\text{light}}}{\| x_{\text{light}} - x \|^2}V(x_{\text{light}},x),$$

where $V(x,x')$ is the visibility function that gives the value one if the two points are visible to each other and zero if they are occluded.

And the last term is ignored by this guy...

⟨*WhittedIntegrator Declarations*⟩≡
```
class WhittedIntegrator : public SurfaceIntegrator {
public:
    ⟨WhittedIntegrator Methods⟩
private:
    ⟨WhittedIntegrator Private Data⟩
};
```

444 AllocateSample
5 Scene

⟨*WhittedIntegrator Private Data*⟩≡
```
int maxDepth;
mutable int rayDepth;
```

⟨*WhittedIntegrator Methods*⟩+≡
```
Sample *WhittedIntegrator::AllocateSample(const Scene *scene) const {
    vector<int> none;
    return new Sample(none, none);
}
```

15.3 Direct Lighting Integrator

⟨*directlighting.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "transport.h"
#include "scene.h"
⟨DirectLighting Declarations⟩
⟨DirectLighting Method Definitions⟩
```

Another interesting integrator only considers *direct lighting* from light sources in the scene at the point being shaded. It completely ignores indirect lighting that

bounces off other objects in the scene, even including specular reflection and transmission. Nevertheless, it is an interesting integrator since it allows us to focus on some of the key details of direct lighting without worrying about the full light transport equation. Furthermore, some of the fragments developed here will be used in subsequent integrators that solve the complete light transport equation.

⟨*DirectLighting Declarations*⟩≡
```
class DirectLighting : public SurfaceIntegrator {
public:
    ⟨DirectLighting Methods⟩
private:
    ⟨DirectLighting Private Data⟩
};
```

⟨*DirectLighting Methods*⟩+≡
```
Sample *AllocateSample(const Scene *scene) const {
    vector<int> num;
    if (strategy == SAMPLE_ALL_UNIFORM)
        num.push_back(scene->lights.size());
    else
        num.push_back(1);
    return new Sample(num, num);
}
```

AllocateSample 444
dgShading 10
Hat 19
lights 579
Normal 23
Point 21
push_back 494
SAMPLE_ALL_UNIFORM 449
Scene 5
size 494
Vector 16

$$L(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_\Omega f_r(\vec{\omega},\vec{\omega}_i) L_d(x,\vec{\omega}_i) |\cos\theta_i| \mathrm{d}\vec{\omega}_i \qquad (15.3.2)$$

where $L_d(x,\vec{\omega}_i)$ includes only light that is directly emitted from light sources.

The basic form of the `DirectLighting::L()` method is similar to `WhittedIntegrator::L()`; the `Scene::Intersect()` method is called to find the first visible surface along the ray, etc. We won't include the implementation of `DirectLighting::L()` here in order to focus on its key fragment, ⟨*Compute direct lighting at hit point*⟩.

⟨*Compute direct lighting for* `DirectLighting` *integrator*⟩≡
```
const Point &P = surf.dgShading.P;
const Normal &N = surf.dgShading.Nn;
if (scene->lights.size() > 0) {
    Vector wo = -ray.D.Hat();
    ⟨Apply direct lighting strategy⟩
}
```

Context for this fragment:

- `P = surf.dgShading.P;`
- `N = surf.dgShading.Nn;`
- `bsdf` is initialized to `BSDF` at the hit point

We support three different strategies for computing direct lighting; all compute an unbiased estimate of reflection from direct lighting at the point being shaded, though they show off different approaches to the problem. An enumerant records which one has been selected.

⟨*DirectLighting Private Data*⟩≡
```
enum LightStrategy { SAMPLE_ALL_UNIFORM, SAMPLE_ONE_UNIFORM,
    SAMPLE_ONE_WEIGHTED } strategy;
```

⟨*Apply direct lighting strategy*⟩≡
```
switch (strategy) {
    case SAMPLE_ALL_UNIFORM: {
        L += UniformSampleAllLights(scene, P, N, wo, bsdf,
            sample, 0);
        break;
    }
    case SAMPLE_ONE_UNIFORM: {
        L += UniformSampleOneLight(scene, P, N, wo, bsdf,
            sample, 0);
        break;
    }
    case SAMPLE_ONE_WEIGHTED: {
        L += WeightedSampleOneLight(scene, P, N, wo, bsdf,
            sample, 0, avgY, avgYsample, cdf, overallAvgY);
        break;
    }
}
```

450 UniformSampleAllLights
450 UniformSampleOneLight

The three approaches to sampling lights for direct lighting can be understood in terms of a discrete probability density defined for each of the lights. Consider the term of the direct lighting equation that we're concerned with here:

$$\int_\Omega f_r(\vec{\omega}, \vec{\omega}_i) L_d(x, \vec{\omega}_i) |\cos\theta_i| d\vec{\omega}_i.$$

This can be broken into a sum over the lights in the scene

$$\sum_{i=1}^{\text{lights}} \int_\Omega f_r(\vec{\omega}, \vec{\omega}_i) L_{d(i)}(x, \vec{\omega}_i) |\cos\theta_i| d\vec{\omega}_i,$$

where $L_{d(i)}$ denotes incident radiance from the ith light. We can estimate each term of this sum individually, adding the results together. This is the most basic direct lighting strategy, where each light is sampled with probability one, and is implemented in ⟨*Sample all lights with uniform probability*⟩. The fragment that computes the estimate for `light` will be defined shortly, after we have described the other light sampling strategies (all of which use this fragment as well.)

⟨*Integrator Utility Functions*⟩≡
```
Spectrum UniformSampleAllLights(const Scene *scene, const Point &P,
        const Normal &N, const Vector &wo, BSDF *bsdf,
        const Sample *sample, int sampleDepth) {
    Spectrum L(0.);
    for (u_int i = 0; i < scene->lights.size(); ++i) {
        Light *light = scene->lights[i];
        L += EstimateDirect(scene, light, P, N, wo, bsdf,
            sample, sampleDepth, i, scene->lights.size());
    }
    return L;
}
```

Alternatively, we might just want to trace a single shadow ray to one of the lights. We can randomly select one light, which gives a uniform probability $1/\text{nlights}$ of selecting each particular light. Then, we estimate direct lighting for only that one light, weighting the result by a factor of nlights to compensate. (Because we used a probability of 1 of selecting each light in the first strategy, additional weighting was necessary there.)

⟨*Integrator Utility Functions*⟩+≡
```
Spectrum UniformSampleOneLight(const Scene *scene, const Point &P,
        const Normal &N, const Vector &wo, BSDF *bsdf,
        const Sample *sample, int sampleDepth) {
    int nLights = int(scene->lights.size());
    int lightNum = RandomInt() % nLights;
    Light *light = scene->lights[lightNum];
    return (Float) nLights *
        EstimateDirect(scene, light, P, N, wo, bsdf, sample,
            sampleDepth, 0, 1);
}
```

It's possible to be even more creative in choosing the individual light sampling probabilities. In fact, we're free to set the probabilities any way we like, so long as we weight the result appropriately and there is non-zero probability of sampling any light that contributes to the reflection at the point. The better a job we do at setting the probabilities so that the probability of sampling a light is proportional to the light's contribution to reflection at the point, the more efficient the Monte Carlo estimator will be and the fewer rays will be needed to reach a particular level of variance. (XXX just like importance sampling other stuff...)

XXX Emphasize issue of handling large numbers of light sources, e.g. in a densely occluded building, not just making the most out of simple situations XXX

Here we'll use a strategy that tries to adapt over the course of rendering the image, increasing the relative probability of sampling lights that have made a large contribution to reflection for previous samples. For example, for a light that is always shadowed, we will reduce the probability of sampling it, focusing instead on lights that are contributing illumination. So long as the probability of sampling that light never goes to zero, the result will remain unbiased.

We will start with a uniform probability for sampling each of the lights. After a light has been chosen, a running average of reflected radiance due to that light

BSDF 298
EstimateDirect 454
Light 358
lights 579
Normal 23
Point 21
RandomInt 515
Scene 5
size 494
Spectrum 155
Vector 16

is updated. By evaluating the importance of each light according to the amount of light reflected rather than the amount of incident light, we also account for the effect of the BSDF; if the BSDF is very glossy, a bright light may have much less effect on the image than a dimmer light that is often along the specular reflection direction.

for each weight, store a weight, so that relative weights give relative probability of sampling lights. to make a discrete pdf, sum the weights and divide all by the sum. to make a discrete cdf, take sum of weights up to *i*th one. to choose a light, take a uniform random number,

weight is exponentially decaying average of reflected luminance $\bar{y}$. can be computed incrementally...

$$\bar{y} = (1-\alpha)y_i + \alpha\,\bar{y}_{-1}$$

where α controls rate of decay. XXX why luminance: perceptually based... XXX

We'll keep track of both the running average of reflected luminance from each light source as well as running average of reflected luminance from the light sources we sampled. This allows us to determine the relative importance of different lights...

⟨*DirectLighting Private Data*⟩+≡
```
mutable Float *avgY, *avgYsample, *cdf;
mutable Float overallAvgY;
```

Until we find a light source that contributes reflected light, `overallAvgY` will be zero. In this case, we just sample a single light with uniform probability. This gives us a reflected luminance value we can use to start updating the running averages with. Otherwise, we choose a light according to its previous contribution and update

```
298 BSDF
454 EstimateDirect
579 lights
 23 Normal
 21 Point
  5 Scene
494 size
155 Spectrum
 16 Vector
```

⟨*Integrator Utility Functions*⟩+≡
```
Spectrum WeightedSampleOneLight(const Scene *scene, const Point &P,
        const Normal &N, const Vector &wo, BSDF *bsdf,
        const Sample *sample, int sampleDepth,
        Float *&avgY, Float *&avgYsample, Float *&cdf,
        Float &overallAvgY) {
    int nLights = int(scene->lights.size());
    ⟨Initialize avgY array if necessary⟩
    Spectrum L(0.);
    if (overallAvgY == 0.) {
        ⟨Sample one light uniformly and initialize luminance arrays⟩
    }
    else {
        ⟨Choose light according to average reflected luminance⟩
        L = EstimateDirect(scene, light, P, N, wo, bsdf,
            sample, sampleDepth, 0, 1);
        ⟨Update avgY array⟩
        L /= lightSampleWeight;
    }
    return L;
}
```

We can't allocate space for avgY until the first time the L() method is called; we don't know how many lights are in the scene until then.

⟨*Initialize* avgY *array if necessary*⟩≡
```
if (!avgY) {
    avgY = new Float[nLights];
    avgYsample = new Float[nLights];
    cdf = new Float[nLights+1];
    for (int i = 0; i < nLights; ++i)
        avgY[i] = avgYsample[i] = 0.;
}
```

To use the relative light weights to select a light source, we first use them to compute a discrete pdf over the light sources. We can then generate a uniform random sample value and use it to search through the cdf to find the appropriate light.

⟨*Sample one light uniformly and initialize luminance arrays*⟩≡
```
L = UniformSampleOneLight(scene, P, N, wo, bsdf, sample, sampleDepth);
Float luminance = L.Luminance();
overallAvgY = luminance;
for (int i = 0; i < nLights; ++i)
    avgY[i] = luminance;
```

ComputeStep1dCDF 401
ExponentialAverage 453
Float2Int 514
Light 358
lights 579
Luminance 243
max 513
min 513
RandomFloat 515
SampleStep1d 401
UniformSampleOneLight 450

XXX trade-off of wasting time sampling lights that have never done us any good, just to check and see if as we move around the image thing have changed, versus not noticing when the set of important lights changes... emphasize that this a demonstration of the idea, not necessarily the best for all applications... XXX

XXX would be nice to have a good sample point here rather than randomfloat? XXX

⟨*Choose light according to average reflected luminance*⟩≡
```
Float c, lightSampleWeight;
for (int i = 0; i < nLights; ++i)
    avgYsample[i] = max(avgY[i], .1f * overallAvgY);
ComputeStep1dCDF(avgYsample, nLights, &c, cdf);
Float t = SampleStep1d(avgYsample, cdf, c, nLights, RandomFloat(),
    &lightSampleWeight);
int lightNum = min(Float2Int(nLights * t), nLights-1);
Light *light = scene->lights[lightNum];
```

⟨*Update* avgY *array*⟩≡
```
Float luminance = L.Luminance();
avgY[lightNum] =
    ExponentialAverage(avgY[lightNum], luminance, .99f);
overallAvgY =
    ExponentialAverage(overallAvgY, luminance, .999f);
```

⟨*Global Inline Functions*⟩+≡
```
inline Float ExponentialAverage(Float avg, Float val, Float alpha) {
    return (1.f - alpha) * val + alpha * avg;
}
```

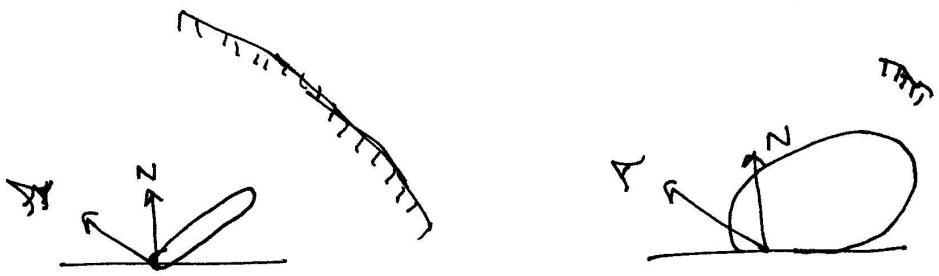


Figure 15.2: sample BSDF vs sample light..

Estimating the direct lighting integral

Having chosen a particular light to estimate direct lighting from, we need to estimate the value of the integral

$$\int_{\Omega} f_r(\vec{\omega}, \vec{\omega}_i) L_d(x, \vec{\omega}_i) |\cos \theta_i| \mathrm{d}\vec{\omega}_i$$

for that light. To compute this estimate, we need to sample one or more directions $\vec{\omega}_i$ and apply the Monte Carlo estimator. There is now an interesting decision to be made: should we use the BSDF's importance sampling method or the light's importance sampling method to choose the direction?

Figure 15.2 shows the problem we face. On the left, the BSDF is very specular and the light source is relatively large. Sampling the BSDF will be effective at finding directions where the integrand's value is large, while sampling the light will be less effective: most of the samples will be black since the BSDF is zero for most of the directions to the light source, while some of the samples will be excessively bright. When the light happens to sample a point in the BSDF's glossy region, the light will return a high sample weight due to its large size, which will cause a spike in the image (XXX need to explain this very carefully XXX).

On the other hand, sometimes sampling the light is the right strategy; on the right side of Figure 15.2, the BSDF is non-zero over many directions and the light is relatively small. It will be far more effective to choose points on the light to compute $\vec{\omega}_i$, since the BSDF will have trouble finding directions where there is non-zero incident radiance from the light.

Rather than needing to choose between these two approaches, we can sample from both of them. And rather than just averaging the results, we will apply a technique called *multiple importance sampling*. The idea behind multiple importance sampling is that when estimating an integral of the form

$$\int f(x) g(x) \mathrm{d}x,$$

where we have a method to importance sample both $f(x)$ and $g(x)$, we should draw samples from both of their distributions. Then, rather than weighting the samples with one over the probability density from the distribution they were drawn from, each sample is instead weighted by

$$\frac{1}{N_f + N_g} \left(\sum_{N_f} \frac{f(x_i) g(x_i)}{\hat{w}_f(x_i)} + \sum_{N_g} \frac{f(x_i) g(x_i)}{\hat{w}_g(x_i)} \right),$$

where N_f is the number of samples taken from f's importance sampling method, N_g is the number of samples taken from g's, and $\hat{w}_f$ and $\hat{w}_g$ are special weighting functions that take into account *all* of the different ways that a sample x_i could have been generated, rather than just the particular one that was used.

A good choice for this weighting function is the *balance heuristic*.

$$\hat{w}(x) = \sum_k \frac{N_k}{N} w_k(x)$$

The balance heuristic is a provably good way to weight samples to reduce variance.

The general approach of multiple importance sampling is particularly helpful because it encourages one to develop different sampling strategies for tricky integrals: each strategy doesn't have to do a good job at capturing all of the characteristics of the integrand, but so long as one of the strategies used is a good one for the particular conditions where the integrand is being evaluated, substantially improved results (in the form of reduced variance) can be had.

XXX intuition for why this reduces variance: reduces the surprise factor, when one sampling method is expecting the integrand to have a small value–its pdf is small for a particular sample–but the integrand actually has a large value due to other factors not accounted for in the pdf. So long as one of the sampling methods catches the factor that made the integrand large, multiple importance sampling helps get rid of the spikes... XXX

BSDF	298
Light	358
Normal	23
Point	21
Scene	5
Spectrum	155
Vector	16

⟨*Integrator Utility Functions*⟩+≡

```
Spectrum EstimateDirect(const Scene *scene, const Light *light,
        const Point &P, const Normal &N, const Vector &wo,
        BSDF *bsdf, const Sample *sample, int sampleDepth,
        int sampleNum, int totSamples) {
    Spectrum Ld(0.);
    ⟨Find light and BSDF sample values for direct lighting estimate⟩
    ⟨Sample light source with multiple importance sampling⟩
    ⟨Sample BSDF with multiple importance sampling⟩
    return Ld;
}
```

⟨*Find light and BSDF sample values for direct lighting estimate*⟩≡
```
Float ls1, ls2, bs1, bs2;
if (sample && sampleDepth < sample->nLightSamples.size() &&
    totSamples == sample->nLightSamples[sampleDepth] &&
    sampleDepth < sample->nBSDFSamples.size() &&
    totSamples == sample->nBSDFSamples[sampleDepth]) {
    ls1 = sample->light[sampleDepth][2*sampleNum];
    ls2 = sample->light[sampleDepth][2*sampleNum+1];
    bs1 = sample->bsdf[sampleDepth][2*sampleNum];
    bs2 = sample->bsdf[sampleDepth][2*sampleNum+1];
}
else {
    ls1 = RandomFloat();
    ls2 = RandomFloat();
    bs1 = RandomFloat();
    bs2 = RandomFloat();
}
```

For sampling the light, it's pretty straightforward application of the Monte Carlo sampling routines and the balance heuristic...

XXX explain delta function issues for this XXX

301 BSDF::f
515 RandomFloat
494 size
155 Spectrum
16 Vector
359 VisibilityTester

⟨*Sample light source with multiple importance sampling*⟩≡
```
Vector wi;
Float lightWeight, bsdfWeight, weight;
bool deltaLight;
VisibilityTester visibility;
Spectrum Li = light->Sample_L(P, N,
    ls1, ls2, &wi, &lightWeight, &deltaLight, &visibility);
if (lightWeight > 0. && !Li.Black() && visibility.Unoccluded(scene)) {
    if (deltaLight)
        weight = lightWeight;
    else {
        bsdfWeight = bsdf->Weight(wo, wi);
        weight = .5f * lightWeight + .5f * bsdfWeight;
    }
    Ld += bsdf->f(wo, wi) * fabsf(Dot(wi, N)) * Li *
        visibility.Transmittance(scene) / weight;
}
```

XXX BSDF is only slightly more tricky, where we need a `Ld()` utility method that computes incident radiance from only the given light source; other lights are ignored XXX

Don't do MIS for specular stuff, since other technique has no chance of finding it. Or, in a sense, the implicit delta function in the weight for the specular guy swamps the weight of the non-specular guy.

⟨*Sample BSDF with multiple importance sampling*⟩≡
```
bool specularBounce;
Spectrum f = bsdf->sample_f(wo, &wi,
    bs1, bs2, &bsdfWeight, &specularBounce);
if (specularBounce)
    weight = bsdfWeight;
else {
    lightWeight = light->Weight(P, N, wi);
    weight = .5f * lightWeight + .5f * bsdfWeight;
}
```
⟨*Compute* Ld *from selected light*⟩

⟨*Compute* Ld *from selected light*⟩≡
```
Surf lightSurf;
if (scene->Intersect(Ray(P, wi), &lightSurf) &&
    lightSurf.primitive->areaLight == light)
    Ld += f * lightSurf.Le(-wi) / weight;
```

15.4 Integral Over Paths

areaLight 581
Light::weight 430
Ray 26
Spectrum 155
Surf 10

The introduction of the light transport equation to graphics led to a flurry of work in rendering, giving a sound theoretical basis for evaluating rendering algorithms. For instance, the path-tracing algorithm in Section 15.5 below is based on recursively evaluating all of the terms of the light transport equation rather than just the delta function terms that whitted considered.

Using the light transport integral equation as the basis for deriving rendering algorithms naturally leads to approaches that start with a ray from the camera and compute radiance estimates by recursively calling the integrator with new rays found by sampling the BSDF at each intersection position. Thinking of the light transport equation in this way limits the set of sampling techniques that one might apply to evaluating it. For example, ray tracing two paths–one starting from the camera and one starting from a light in the scene and connecting them up in the middle can be a more effective light transport technique than just tracing rays from the eye.

In this section, we will introduce the *path integral* form of the light transport equation. It has the form of sums over paths of various numbers of bounces of light in the scene, where the first vertex of the path is on the image plane and the last is one a light source. This form makes it more natural to develop creative ways of generating light transport paths through the scene and to apply more general integration techniques, which in turn can lead to lower-variance results.

To derive the path integral form, we start with the *three-point form* of the light transport equation. The integral over incident directions $\vec{\omega}_i$ and x is replaced with an integral over points x' in the scene. First, we define outgoing radiance from a point x' to a point x'' by

$$L(x' \to x'') = L(x', \vec{\omega}),$$

if x' and x'' are mutually visible and $\vec{\omega} = \widehat{x'' - x'}$. We can also write the BSDF at x' as

$$f(x \to x' \to x'') = f(x', \vec{\omega}_i, \vec{\omega}_o),$$

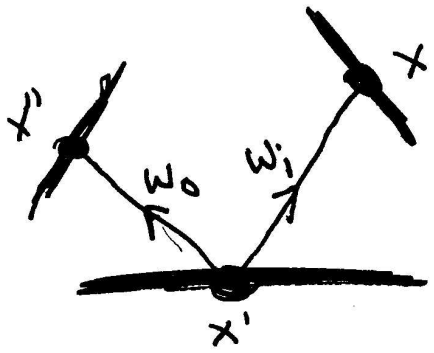


Figure 15.3: The three-point form of the light transport equation converts the integral to be over the domain of points on surfaces in the scene, rather than over directions over the sphere. It is a key transformation for deriving the path integral form of the light transport equation.

where $\vec{\omega}_i = \widehat{x - x'}$ and $\vec{\omega}_o = \widehat{x'' - x'}$. Substituting these into the light transport equation and applying the term to convert an integral over solid angle into an integral over area, we have

$$L(x' \to x'') = L_e(x' \to x'') + \int_A L(x \to x') f(x \to x' \to x'') G(x \leftrightarrow x') \mathrm{d}A(x),$$

where A is the area of all of the surfaces of the scene. The $G(x \leftrightarrow x')$ term accounts for $\cos\theta_i$ term in the original integral and the change of variables from integral over solid angle to integral over area. It is:

$$G(x \leftrightarrow x') = V(x, x') \frac{|\cos\theta| \, |\cos\theta'|}{\| x - x' \|^2}.$$

We can now start to expand out the three-point light transport equation. Here are the first few terms that give incident radiance at a point x from another point x_0, where x_0 is the first point on a surface along the ray from x in direction $x_0 - x$.

$$\begin{aligned} L(x_0 \to x) = {} & L_e(x_0 \to x) + \\ & \int_A L_e(x_1 \to x_0) f(x_1 \to x_0 \to x) G(x_1 \leftrightarrow x_0) \mathrm{d}A(x_1) + \\ & \int_{A^2} L_e(x_2 \to x_1) f(x_2 \to x_1 \to x_0) G(x_2 \leftrightarrow x_1) \\ & \qquad f(x_1 \to x_0 \to x) G(x_1 \leftrightarrow x') \, \mathrm{d}A(x_2) \, \mathrm{d}A(x_1) + \cdots \end{aligned}$$

The pattern becomes clear, and we have

$$L(x' \to x) = L_e(x' \to x) + \sum_{i=1}^{\infty} P_i(\bar{x}) \qquad (15.4.3)$$

where $P_i(\bar{x})$ gives the light scattered over paths with i vertices through the scene:

$$P_i(\bar{x}) = \int_A \cdots \int_A L_e(x_i \to x_{i-1}) G(x_i \leftrightarrow x_{i-1}) \left(\prod_{j=1}^{i-1} f(x_{j-1} \to x_j \to x_{j+1}) G(x_j \leftrightarrow x_{j+1}) \right) \mathrm{d}A(x_1) \cdots \mathrm{d}A(x_i).$$

Given Equation 15.4.3 and given a particular length i, all we need to do to estimate the radiance due to paths of length i is to sample a set of vertices in the scene x_i to generate a path and then to evaluate P_i for those vertices. Whether we generate those vertices by starting a path from the camera, the light, both ends, or a point in the middle is a detail that only affects how the weights for the Monte Carlo estimates are computed. We will see how this formulation leads in practice to practical light transport algorithms in the following two sections.

XXX need to define path throughput somewhere in here! XXX

15.5 Path Tracing

⟨*path.cc**⟩≡
⟨*Source Code Copyright*⟩
```
#include "lrt.h"
#include "transport.h"
#include "scene.h"
```
⟨*PathIntegrator Declarations*⟩
⟨*PathIntegrator Method Definitions*⟩

Now that we have derived the path integral form of the light transport equation, we'll show how it can be used to derive the *path tracing* light transport algorithm. Path tracing generates paths of various numbers of scattering events, starting at the eye and ending at light sources in the scene. It is essentially an extension of Whitted's method to include both delta-function and non-delta BSDFs and light sources, rather than just the delta function terms.

Although it is slightly easier to derive path tracing directly from the basic light transport equation, approaching it from the path integral form helps build understanding of the path integral equation and will make the generalization to *bidirectional path tracing*, where paths are generated starting from the lights as well as from the eye easier to understand.

Given the path integral form of the LTE, we need to estimate the value of

$$L(x' \to x) = L_e(x' \to x) + \sum_{i=1}^{\infty} P_i(\bar{x})$$

for a given eye ray from x that first intersects the scene at x'. There are two pieces to this problem:

1. How do we estimate the value of the sum of the infinite number of $P_i(\bar{x})$ terms
2. Given a particular $P_i(\bar{x})$ term, how do we estimate its value.

XXX just introduce RR more directly: say "here is the algorithm, here is how the weighting works, and the result is unbiased... XXX

$$v' = \begin{cases} v/p & \xi < p \\ 0 & \text{otherwise} \end{cases}$$

Expected value is then

$$(1-p) \cdot 0 + p \cdot v/p = v.$$

For the first problem, we will apply a Monte Carlo technique known as *Russian roulette*. Recall that we defined a discrete probability density function over the lights in the scene for the direct lighting integrator in Section 15.3. Here, we will in a similar manner define a probability for sampling each of the terms of the infinite sum. For example, we might define the probability of sampling the ith term as

$$p_i = \frac{1}{4^{i-1}}.$$

Along the same lines as the direct lighting example, when we randomly decided to go ahead and sample the ith term according to the probability p_i, we would need to weight it's estimate by $1/p_i$ to make the estimate unbiased.

To turn this approach into an algorithm that still doesn't require us to loop over an infinite number of terms, we will incrementally decide whether to sample the ith term only if we also decided to sample the $i-1$st term. Once we decide not to sample a particular term, we don't sample any of the subsequent ones. This approach works so long as the probability of sampling each term is a non-increasing sequence. For example, for the probabilities p_i above, we equivalently have

$$\begin{aligned} p_1 &= 1 \\ p_i &= p_c^i p_{i-1} \end{aligned}$$

where p_c^i, the probability that sampling continues after the ith term, is $1/4$.

Thus, in pseudo-code, we can estimate the sum by:

```
Float estimate = 0;
Float continueProbability = 1./4.;
Float weight = 1.;
for (int i = 1; ; ++i) {
estimate += P(i) * weight;
if (RandomFloat() > continueProbability) break;
weight /= continueProbability;
}
return estimate;
```

This block of code both samples the ith term with a probability $1/4^{i-1}$ and weights it by the weight 4^{i-1} if it is sampled, giving us an unbiased estimate of the sum. By expressing the probability of sampling the ith term in terms of the $i-1$st term and only considering the ith term if the $i-1$st term was sampled, we are able to do all this without needing to explicitly consider the infinite number of terms.

There is almost total freedom in how the continuation probabilities p_c^i are selected: we're free to use any information we'd like to set them so long as the weight is updated appropriately when we decide to continue. However, poorly chosen Russian roulette weights can substantially increase variance: consider if we immediately applied Russian roulette to all of the camera rays with a continuation probability of .01: we'd only trace 1% of the eye rays, weighting each of them by $1/.01 = 100$. The resulting image would numerically be just as correct

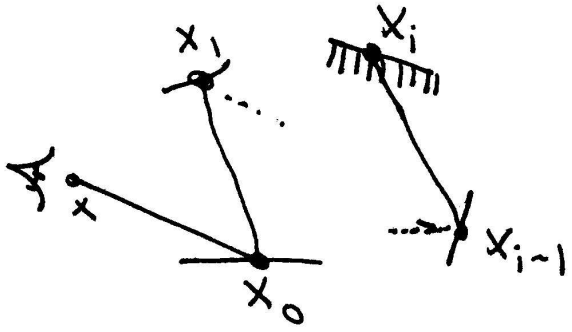


Figure 15.4:

as if we hadn't applied Russian roulette, though visually the result would be terrible: mostly black pixels with a few very bright ones. One of the exercises at the end of this chapter discusses this problem further and describes a technique called *efficiency optimized Russian roulette* that tries to set Russian roulette weights in a way that minimizes variance.

For path tracing, we can take advantage of the fact that overall, paths with more vertices along them will generally scatter less light than paths with fewer vertices; this is a natural consequence of conservation of energy in BSDFs. In the implementation below, we will always estimate the first few terms $P_i(\bar{x})$ and will then start to consider termination, setting Russian roulette weights based on the throughput of the path we've constructed.

We now need a way to estimate a particular term $P_i(\bar{x})$; we need $i+1$ vertices to specify the path, where the last vertex, x_i, is on a light source. The first vertex, x_0, is determined by the camera ray's first intersection point (see Figure 15.4.)

XXX somewhere need to make clear this doesn't work in the presence of delta BSDFs, it's just an example... XXX

Looking at the form of $P_i(\bar{x})$, the most natural thing to do is to sample x_i according to the differential area of objects in the scene, such that it's easyally probable to sample any point on an object in the scene for x_i as any other point. We could define a discrete probability over the n objects in the scene; if each has surface area A_i, then the probability of sampling a vertex on the jth object should be

$$p_j = \frac{A_j}{\sum_k A_k}.$$

Then, given a method to sample a point on the jth object with uniform probability, the pdf for sampling any particular point on object j is $1/A_j$. Thus, the overall probability density for sampling the point is

$$\frac{A_j}{\sum_k A_k}\frac{1}{A_j}.$$

And thus, all samples x_i have the same weight

$$\frac{1}{\sum_k A_k}.$$

It's reassuring that they all have the same weight, since our intent was to choose among all points on surfaces in the scene with equal probability.

Given the set of points $x_0, x_1, \ldots, x_{i-1}$, we can then sample x_i on a light source in the scene, defining probabilities appropriately. Although we could use the same technique used for sampling path vertices, this would lead to high variance, since for all of of the paths where x_i wasn't on the surface of an emitter, the path would have zero value. Better is to sample over the areas of only the emitting objects.

We then have all of the information we need to evaluate the estimate of $P_i(\bar{x})$; it's just a matter of evaluating each of the terms.

We could be much more creative about how we set the sampling probabilities: for example, if we knew that indirect illumination from a few ojbects contributed to most of hte lighting in the scene, we could assign a higher probability to generating samples x_i on those objects, updating the sample weights appropriately.

There are, however, two main disadvantages to sampling paths in this manner. First, many of the paths will have no contribution if they have pairs of adjacent vertices that are not mutually visible. Consider applying the area sampling method above in a complex building model: unless we made sure that vertices are usually in the same room as adjacent vertices, they will almost always have a wall or two between them, giving no contributino for the path. The second issue is that the sampling method doesn't account for the BSDFs in the scene; if there are very glossy BSDFs, many paths will have low contribution since the points in $f(x_{i-1} \rightarrow x_i \rightarrow x_{i+1})$ will cause the BSDF to have a small value (XXX and actually, totally misses specular stuff... XXX)

Therefore, the classic approach to path tracing is to construct the path incrementally, starting from x_0. At each vertex, the BSDF is sampled to generate a direction; the next vertex x_{i+1} is found by tracing a ray from x_i in the sampled direction and choosing the closest intersection. This approach simultaneously solves both of the problems described in the paragraph above.

Because we are constructing the path by sampling BSDFs according to solid angle, we need to apply the correction to convert from the probability density according to solid angle to a density according to area (recall Section 5.3):

$$p_A = p_{\vec{\omega}} \frac{\left\| \mathbf{x_i} - \vec{\mathbf{x}}_{\mathbf{i+1}} \right\|^2}{|\cos\theta'|}$$

XXX review that XXX

Note that this just causes some of the terms of the geometric term $G(x \leftrightarrow x')$ to cancel and that we already know that x_i and x_{i+1} must be mutually visible since we traced a ray between them.

Thus, the basic path tracing is, in pseudo-code:

```
P(n) {
throughput = 1;
wo = -eyeRay.D.Hat();
x[0] = trace(eyeRay);
for (i = 1; i < n; ++i)) {
wi = sample_BSDF(x[i-1], wo, &wi);
throughput *= f(x[i-1], wo, wi) *
sampleWeight(x[i-1], wo, wi) * | cos(theta) |;
```

```
    x[i] = trace(x[i-1], wi);
    }
    x[n] = sampleLights();
    return Le(x[n], (x[n-1] - x[n])) * throughput * lightWeight(x[n]);
    }
```

In our implementation below, we will make one last refinement: as we are constructing paths for a given camera ray, we will re-use the vertices of the previous path of length $i-1$ when constructing the path of length i. This means that we just need to trace two more rays for each extra $P_i(\bar{x})$ term that we evaluate, rather then $i+1$ rays. This introduces correlation among all of the $P_i(\bar{x})$ terms in the sum, though in practice this is more than made up for by the improved efficiency from tracing fewer rays.

Now on to the implementation...

⟨*PathIntegrator Declarations*⟩≡

```
class PathIntegrator : public SurfaceIntegrator {
public:
    Spectrum L(const Scene *scene, const RayDifferential &ray, const Sample *sa
    Sample *AllocateSample(const Scene *scene) const;
};
```

AllocateSample 444
push_back 494
RayDifferential 26
Scene 5
Spectrum 155

⟨*PathIntegrator Method Definitions*⟩≡

```
Sample *PathIntegrator::AllocateSample(const Scene *scene) const {
    vector<int> num;
    for (int i = 0; i < 5; ++i)
        num.push_back(1);
    return new Sample(num, num);
}
```

⟨*PathIntegrator Method Definitions*⟩+≡

```
Spectrum PathIntegrator::L(const Scene *scene,
        const RayDifferential &r, const Sample *sample,
        Float *alpha) const {
    ⟨Declare common path integration variables⟩
    int pathLength = 0;
    while (1) {
        ⟨Find next vertex of path⟩
        ⟨Add emitted light for first segment only⟩
        ⟨Evaluate BSDF at hit point⟩
        ⟨Randomly sample illumination from one light source⟩
        ⟨Randomly sample BSDF to get new path direction⟩
        ⟨Clean up from integration⟩
        ⟨Possibly terminate the path⟩
        ++pathLength;
    }
    return L;
}
```

⟨*Declare common path integration variables*⟩≡
```
Spectrum pathThroughput = 1.;
Spectrum L = 0.;
Ray ray = r;
bool specularBounce = false;
```

⟨*Find next vertex of path*⟩≡
```
Surf surf;
if (!scene->Intersect(ray, &surf)) {
    for (u_int i = 0; i < scene->lights.size(); ++i)
        L += pathThroughput * scene->lights[i]->Le(ray);
    if (pathLength == 0 && alpha) {
        if (L != 0.) *alpha = 1.;
        else *alpha = 0.;
    }
    break;
}
if (pathLength == 0) {
    r.maxt = ray.maxt;
    if (alpha) *alpha = 1.;
}
```

Note that we don't count L_e if we hit an area light source...

XXX believe that specular bounce check isn't right now that we're using shared direct lighting code? XXX

⟨*Add emitted light for first segment only*⟩≡
```
if (pathLength == 0 || specularBounce)
    L += pathThroughput * surf.Le(-ray.D);
```

⟨*Randomly sample illumination from one light source*⟩≡
```
const Point &P = surf.dgShading.P;
const Normal &N = surf.dgShading.Nn;
Vector wi;
Vector wo = -ray.D.Hat();
L += pathThroughput * UniformSampleOneLight(scene, P, N, wo, bsdf,
    sample, pathLength);
```

XXXX So here it's wasteful to sample BSDF twice with same random numbers, once for direct lighting, once for path tracing...

```
 10 dgShading
 19 Hat
579 lights
 23 Normal
 21 Point
 26 Ray
494 size
155 Spectrum
 10 Surf
450 UniformSampleOneLight
 16 Vector
```

⟨*Randomly sample BSDF to get new path direction*⟩≡
```
Float bs1, bs2;
if (pathLength < sample->nBSDFSamples.size()) {
    Assert(sample->nBSDFSamples[pathLength] == 1);
    bs1 = sample->bsdf[pathLength][0];
    bs2 = sample->bsdf[pathLength][1];
}
else {
    bs1 = RandomFloat();
    bs2 = RandomFloat();
}
Float weight;
Spectrum f = bsdf->sample_f(wo, &wi, bs1, bs2,
    &weight, &specularBounce);
if (f == Spectrum(0.) || weight == 0.)
    break;
pathThroughput *= f * fabsf(Dot(wi, N)) / weight;
ray = Ray(P, wi);
```

⟨*Possibly terminate the path*⟩≡
```
if (pathLength > 3) {
    Float continueProbability = .2f;
    if (RandomFloat() > continueProbability)
        break;
    pathThroughput /= continueProbability;
}
```

Assert	498
RandomFloat	515
Ray	26
size	494
Spectrum	155

15.6 Bidirectional Path Tracing

⟨*bidirectional.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "transport.h"
#include "scene.h"
#include "mc.h"
⟨Bidirectional Local Declarations⟩
⟨Bidirectional Method Definitions⟩
```

The path tracing algorithm described in the previous section was the first general light transport algorithm in graphics, handling both a wide variety of geometric objects as well as area lights and general BSDF models. Although it works well for many scenes, it can exhibit high variance in the presence of particular tricky lighting conditions. For example, consider the setting shown in Figure 15.5; a light source is illuminating a small area on the ceiling, such that the rest of the room is only illuminated by indirect lighting bouncing from that area. If we only trace paths starting from the eye, we will almost never happen to sample a vertex in the illuminated region before we trace a shadow ray to the light. Most of the paths will have no contribution, while a few of them–the ones that happen to hit the small

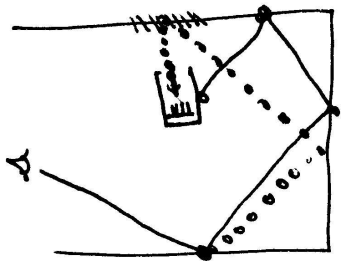


Figure 15.5:

region on the ceiling–will have a large contribution. The resulting image will have high variance.

Difficult lighting settings like this can be handled more effectively by constructing paths that start from the eye on one end, from the light on the other end, and are connected in the middle with a visibility ray. This *bidirectional path tracing* algorithm is a generalization of the standard path tracing algorithm; for the same amount of computation, it can give substantially lower variance. XXX say something about adjoint algorithms in general XXX

XXX x_0 versus x_1 XXX

The path integral LTE makes it easy to understand how to construct a bidirectional algorithm. As with standard path tracing, the first vertex, x_1, is found by computing the first intersection along the camera ray, and the last vertex is found by sampling a point on a light source in the scene. Here we will label the last vertex as y_1, so that we can construct a path of not-initially-determined length "backward" from the light.

In the basic bidirectional algorithm, we go forward from the eye to create a subpath $x_1, x_2, \ldots, x_i$ and backward from the light to compute a subpath $y_1, y_2, \ldots, y_j$. Each sub-path is usually computed incrementally by sampling the BSDF at the previous vertex, though other sampling approaches can be used in the same way as was described for standard path tracing. (Weights for each vertex are computed in the same manner as well.) In either case, in the end, we have a path

$$\bar{x} = x_1, \ldots, x_i, y_j, \ldots, y_1.$$

We need to trace a shadow ray between x_i and y_j to make sure they are mutually visible; if so, the path carries light from the light to the camera and we can evaluate the path's contribution directly.

There are three refinements to the basic algorithm that improve its performance in practice. The first two are analogous to improvements made to path tracing.

- First, we will re-use sub-paths: given a path $x_1, \ldots, x_i, y_j, \ldots, y_1$, we will evaluate transport over all of the paths generated by connecting all the various combinations of prefixes of the two paths together. If the two paths have i and j vertices, respectively, then a total of $i \cdots j$ unique paths can be constructed from them, ranging in length from 2 to $i + j$ vertices long. XXX number of paths of length $n =$ ... XXX. Each such path built this way only requires that a visibility check be performed by tracing a shadow ray between the last vertices of each of the sub-paths.

- The second optimization is to ignore the paths generated in the path-reuse stage that only use one vertex from the light sub-path and instead to use the optimized direct lighting code that we developed for the direct lighting integrator. This gives a lower-variance result than using the vertex on the light sampled for the light sub-path, since it allows us to both use multiple importance sampling with the BSDF and to use stratified sampling patterns.

- The third optimization, left as an exercise, is to use multiple importance sampling to re-weight paths. Recall the example of a light pointed up at the ceiling, indirectly illuminating a room. As described so far, bidirectional path tracing will improve the result substantially by greatly reducing the number of paths with no contribution, since the paths from the light will be effective at finding those light transport routes. However, the image will still suffer from variance due to paths with excessively large contributions, for example from paths from the eye thathappened to find the bright spot in the ceiling. We can apply MIS, recognizing that for a path with n vertices, there are actually $n-1$ ways we could generate a path with that length–e.g. a 4 vertex path could have been built from one eye vertex and three light vertices, two of each kind of vertex, or three eye vertices and one light vertex. Given a particular path sampled in a particular way, we can compute the weights for each of the other ways the path could have been generated and apply the balance heuristic.

AllocateSample	444
Assert	498
RayDifferential	26
Scene	5
Spectrum	155

⟨*Bidirectional Local Declarations*⟩+≡
```
class BidirIntegrator : public SurfaceIntegrator {
public:
    Spectrum L(const Scene *scene, const RayDifferential &ray, const Sample *sa
    Sample *AllocateSample(const Scene *scene) const;
private:
    ⟨BidirIntegrator Private Methods⟩
};
```

⟨*Bidirectional Method Definitions*⟩≡
```
Sample *BidirIntegrator::AllocateSample(const Scene *scene) const {
    Assert(1 == 0);
    return NULL;
}
```

⟨*Bidirectional Method Definitions*⟩+≡
```
Spectrum BidirIntegrator::L(const Scene *scene, const RayDifferential &ray,
        const Sample *sample, Float *alpha) const {
    Spectrum L(0.);
    ⟨Generate eye and light sub-paths⟩
    ⟨Connect bidirectional path prefixes and evaluate throughput⟩
    return L;
}
```

XXX should use `sample` here, etc...

⟨*Generate eye and light sub-paths*⟩≡
```
#define MAX_VERTS 8
BidirVertex eyePath[MAX_VERTS], lightPath[MAX_VERTS];
int nEye = generatePath(ray, eyePath, MAX_VERTS);
if (nEye == 0) {
    // XXX handle ray with no intersection and reurn
}
⟨Choose light for bidirectional path⟩
⟨Sample ray from light source to start light path⟩
int nLight = generatePath(lightRay, lightPath, MAX_VERTS);
```

⟨*Choose light for bidirectional path*⟩≡
```
int lightNum = RandomInt() % scene->lights.size();
Light *light = scene->lights[lightNum];
Float lightWeight = Float(scene->lights.size());
```

⟨*Sample ray from light source to start light path*⟩≡
```
Ray lightRay;
Float lightSampleWeight;
bool deltaLight;
Float u[4];
for (int i = 0; i < 4; ++i)
    u[i] = RandomFloat();
Spectrum Le = light->Sample_L(scene, u[0], u[1], u[2], u[3],
    &lightRay, &lightSampleWeight, &deltaLight);
```

466 BidirIntegrator
298 BSDF
358 Light
579 lights
23 Normal
21 Point
515 RandomFloat
515 RandomInt
26 Ray
494 size
155 Spectrum
16 Vector

⟨*Bidirectional Local Declarations*⟩+≡
```
struct BidirVertex {
    BSDF *bsdf;
    Point P;
    Normal N;
    Vector wi, wo;
    Float bsdfWeight, dAWeight;
    bool isSpecular;
};
```

⟨*Bidirectional Method Definitions*⟩+≡
```
int BidirIntegrator::generatePath(const Ray &r, BidirVertex *vertices,
        int maxVerts) const {
// XXX careful if we reuse the same Ray, then make sure mint/maxt are reset!
    return 0; // keep the compiler happy.
}
```

⟨*Connect bidirectional path prefixes and evaluate throughput*⟩≡
```
for (int i = 1; i <= nEye; ++i) {
    for (int j = 1; j <= nLight; ++j) {
        if (j == 1) {
            ⟨Handle direct lighting for bidirectional integrator⟩
            continue;
        }
        L += evalPath(scene, eyePath, i, lightPath, j) /
            weightPath(eyePath, i, lightPath, j);

    }
}
```

⟨*Bidirectional Method Definitions*⟩+≡
```
Float BidirIntegrator::weightPath(BidirVertex *eye, int nEye,
        BidirVertex *light, int nLight) const {
    return Float(nEye + nLight - 1);
}
```

XXX splatting for caustics, review indexing stuff carefully, etc...

BidirIntegrator 466
BidirVertex 467
Scene 5
Spectrum 155
Vector 16

⟨*Bidirectional Method Definitions*⟩+≡
```
Spectrum BidirIntegrator::evalPath(const Scene *scene, BidirVertex *eye, int nE
        BidirVertex *light, int nLight) const {
    if (!visible(scene, eye[nEye].P, light[nLight].P))
        return 0.;
    Spectrum L(1.);
    for (int i = 0; i < nEye; ++i) {
        BidirVertex *e = &eye[i];
        L *= e->bsdf->f(e->wi, e->wo) * G(eye[i], eye[i+1]) /
            e->dAWeight; // XXX bsdf weight?
    }
    Vector w = light[nLight].P - eye[nEye].P;
    L *= eye[nEye].bsdf->f(eye[nEye].wi, w) *
        G(eye[nEye], light[nLight]) *
        light[nLight].bsdf->f(-w, light[nLight].wi);
    for (int i = nLight-1; i >= 0; --i) {
        BidirVertex *l = &light[i];
        L *= l->bsdf->f(l->wi, l->wo) * G(light[i], light[i-1]) /
            l->dAWeight; // XXXX
    }
    return L;
}
```

⟨*Bidirectional Method Definitions*⟩+≡
```
Float BidirIntegrator::G(const BidirVertex &v0, const BidirVertex &v1) {
    Vector w = (v1.P - v0.P).Hat();
    return fabsf(Dot(v0.N, w) * Dot(v1.N, -w)) /
        DistanceSquared(v0.P, v1.P);
}
```

⟨*Bidirectional Method Definitions*⟩+≡
```
bool BidirIntegrator::visible(const Scene *scene, const Point &P0,
        const Point &P1) {
    Ray ray(P0, P1-P0, RAY_EPSILON, 1.f - RAY_EPSILON);
    return !scene->IntersectP(ray);
}
```

15.7 Photon Mapping

⟨*photonmap.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "transport.h"
#include "scene.h"
#include "mc.h"
#include "kdtree.h"
⟨Photonmap Local Declarations⟩
⟨Photonmap Method Definitions⟩
```

Even with multiple importance sampling to reweight paths, for some scenes it can take a large number of rays (and correponsind compute time) to generate images without objectionable noise. One approach to this problem has been the development of biased approaches to solving the LTE. Photon mapping, described in this section, and irradiance caching, described in the enxt section, have been two successful biased methods for light transport.

466 BidirIntegrator
467 BidirVertex
23 DistanceSquared
19 Hat
21 Point
26 Ray
5 Scene
16 Vector

By introducing bias, these methods produce images without the high-frequency noise artifacts that unbiased Monte Carlo techniques are prone to. They can often do so using relatively litttle additional computation compared to basic techniques like Whitted-style ray tracing. This efficiency comes at a price, however: one key characteristic of unbiased Monte Carlo techinques is that variance decreases in a predictable and well-characterized manner as more samples are taken. As such, if an image was computed with an unbiased technique and has no noise, we can be extremely confident that the image correctly represents the lighting in the scene. With a biased solution method, however, error estimates aren't well defined for the approaches that have been developed so far; if the image doesn't have visual artifacts, it still may have substantial error. And given an image with artifacts, increasing the sampling rate with a biased technique doesn't necessarily eliminate artifacts in a predictable way.

The basic idea behind photon mapping is that in a pre-process, a set of paths from the light source are generated, each one carrying energy from the lights into the scene. At each vertex of each path, the incident energy arriving at each surface that the path intersects is recorded, a new outgoing direction is chosen to continue the path, and the photon's energy is adjusted by the surface's BSDF. After a certain number of these samples have been computed, a data structure–the *photon map*–is built, storing information about the distribution of light in the scene. The photon map is based on a general three-dimensional data structure that allows fast queries of how many photons are nearby a given point. Because the data structure is decoupled from the scene geometry, the algorithm isn't limited to parametric geometric representations, for example.

At rendering-time, the photon map is used to comptue reflected light at each point being shaded. The usual approach is to use photons that are close to the current point under the assumption that the information they carry about illumination at nearby points can be used to construct an estimate of illumination at the shading point. The more photons there are around the point and the more energy they are carrying, the more light we estimate is illuminating the point. The estimated illumination at the point is used in conjunction with the surface's BSDF to compute the reflected light; Figure 15.6 shows the basic idea.

There is great flexibility in how these basic ideas are applied in practice. A few examples in include:

- Direct lighting from light sources is usually handled with conventional techniques, by tracing shadow rays, so the photon map doesn't store a photon at the first surface a path intersects; only at the subsequent vertices. This keeps the quality of the direct lighting estimate high, since the reconstruction step in the photon map tends to blur the incident illumination estimate. (However, preview images can be rendered very quickly by storing photons at the first bounce and not tracing shadow rays at all.

- For very glossy or specular BSDFs, it's often better to sample the BSDF and trace rays into the scene to estimate incident illumination, rather than using the photons, because the incident photons may be from directions that don't contribute much to the BSDF. This is an analogous situation to the one of sampling points on light sources versus sampling the BSDF to compute direct lighting–sometimes sampling the BSDF is the only effective way to find important lighting paths.

- We may only care about some of the illumination paths from the light. For example, one of the most effective applications of the photon map is to compute caustics–the bright areas of focused light that happen when illumination scatters one or more times from specular objects before arriving at a non-specular surface, upon which the caustic is cast. To do so, we only store paths where the first n vertices of the path are at specular surfaces, for $n \geq 1$, and where the last vertex is at a non-specular surface. The path is finished when the photon hits a non-specular surface, at which point a photon is stored.

This flexibility in how the algorithm is applied can be handy–there is opportunity to adapt the basic technique to many lighting situations. However, one must be careful that illumination isn't "double counted", once from the photon map, and once from another sampling technique.

There are two sub-problems to solve in a photon-mapping implementation:

- The paths from the lights must be constructed and a data structure built from them.

- That data structure then must be used to compute some components of incident illuminatino at rendering-time.

Here we will show an implementation of photon mapping that efficiently computes images with caustics; we will only follow paths that interact with one or more

Figure 15.6: photon basic

specular surfaces, and only store photons (and terminate the path) when the path reaches a non-specular surface. We have already made an arbitrary choice here: we are following paths that hit specular surfaces, but are ignoring paths that hit very glossy surfaces. As a glossy surface approaches being perfectly smooth, its scattering behavior approaches that of a perfect specular reflector, yet we treat such surfaces differently. XXX.

⟨*Photonmap Local Declarations*⟩+≡

```
class PhotonIntegrator : public SurfaceIntegrator {
public:
    ⟨PhotonIntegrator Methods⟩
private:
    ⟨PhotonIntegrator Private Data⟩
};
```

XXX make clear the changing illumination error problem–e.g. a small wall, etc. XXX

The user gives the photon map integrator two parameters: the number of photons to be stored in the scene, `nStored`, and the number of photons to use for illumination estimates at shading-time, `nLookup`. The more photons that are stored, the better the illumination estimates will be–it will be less necessary to use photons far from the point being shaded, reducing the error from changing illumination in the scene. How to choose the number to lookup is slightly less clear; the more that are used, the smoother the illumination estimate will be, since a larger number of photons will be used to reconstruct it. If too many are used, however, the result will tend to be too blurry, while too few gives a splotchy appearence. Usually 50 to 100 is a good choice.

also make maxLength and searchRadius parameters.

⟨*Photonmap Method Definitions*⟩≡
```
PhotonIntegrator::PhotonIntegrator(int ns, int nl, int mdepth,
        Float mdist) {
    nStored = ns;
    nLookup = nl;
    maxDist = mdist;
    maxDepth = mdepth;
    tree = NULL;
    unsuccessfulShooting = false;
    pathLength = 0;
}
```

⟨*PhotonIntegrator Private Data*⟩≡
```
u_int nStored, nLookup;
mutable int pathLength;
int maxDepth;
Float maxDist;
```

⟨*Photonmap Method Definitions*⟩+≡
```
Sample *PhotonIntegrator::AllocateSample(const Scene *scene) const {
    vector<int> num;
    for (int i = 0; i < 3; ++i)
        num.push_back(scene->lights.size());
    return new Sample(num, num);
}
```


AllocateSample 444
lights 579
PhotonIntegrator 471
push_back 494
RayDifferential 26
Scene 5
size 494
Spectrum 155

⟨*Photonmap Method Definitions*⟩+≡
```
Spectrum PhotonIntegrator::L(const Scene *scene,
        const RayDifferential &ray, const Sample *sample,
        Float *alpha) const {
    if (!tree && !unsuccessfulShooting) {
        ⟨Shoot photons and create kd-tree⟩
    }
    ⟨Compute reflected radiance with photon map⟩
}
```

Building the photon map

The first time the integrator's `L()` method is called, we build the photon map. The body of the `while()` loop in the fragment below handles the generation of a single path from a light source out into the scene and the storage of the resulting photon, if any, in the `photons` array. We keep track of the total number of paths generated in `nshot`; if we find that we have generated many paths without successfully storing any photons in the scene, we give up and the `unsuccessfulShooting` variable is set to `true`. (For example, this might happen if there weren't any specular objects in the scene.)

⟨*Shoot photons and create kd-tree*⟩≡
```
vector<PhotonData> photons;
photons.reserve(nStored);
⟨Initialize photon shooting statistics⟩
while (photons.size() < nStored) {
    ++nshot;
    ⟨Give up if we're not finding any specular surfaces⟩
    ⟨Trace a photon path and store contribution⟩
}
if (!unsuccessfulShooting) {
    ⟨Normalize photon dE values and build tree⟩
}
```

We may need to shoot many more photons than are stored, for example due to photons that leave the scene without intersecting any objects, or for the caustic photon mapper here, photons that first hit a non-specular surface.

⟨*PhotonIntegrator Private Data*⟩+≡
```
mutable KdTree<PhotonData, PhotonProcess> *tree;
mutable bool unsuccessfulShooting;
```

⟨*Initialize photon shooting statistics*⟩≡
```
static StatsCounter nshot("Integrator",
    "Number of photons shot from lights");
```

⟨*Give up if we're not finding any specular surfaces*⟩≡
```
if (nshot > 500000 && (!photons.size() || nshot / photons.size() > 1000)) {
    cerr << "No luck shooting photons!!!" << endl;
    unsuccessfulShooting = true;
    break;
}
```

523 KdTree
579 lights
474 PhotonData
476 PhotonProcess
207 RadicalInverse
494 reserve
494 size
501 StatsCounter

Using Halton sequence in four dimensions to get good coverage of the space. Is nice since we don't need to decide ahead of time how many points we want; no matter how many we ask for, they are well-distributed...

Recall Section 7.4.

XXX review measures and weights for sampling carefully!

⟨*Trace a photon path and store contribution*⟩≡
```
Float u[4];
u[0] = RadicalInverse(nshot, 2);
u[1] = RadicalInverse(nshot, 3);
u[2] = RadicalInverse(nshot, 5);
u[3] = RadicalInverse(nshot, 7);
⟨Choose light to shoot photon from⟩
⟨Generate photonRay from light source⟩
if (!L.Black()) {
    ⟨Follow photon to non-specular surface and store in photons array⟩
}
```

Here as with the direct lighting sampler, we might want to be more creative. Dynamically adjust discrete sampling pdfs based on which ones are finding specular paths, sample based on power, ...

⟨*Choose light to shoot photon from*⟩≡
```
int nLights = int(scene->lights.size());
Light *light = scene->lights[RandomInt() % nLights];
Float lightWeight = 1.f / nLights;
```

⟨*Generate* photonRay *from light source*⟩≡
```
RayDifferential photonRay;
Float weight;
bool isDeltaLight;
Spectrum L = light->Sample_L(scene, u[0], u[1], u[2], u[3],
    &photonRay, &weight, &isDeltaLight);
L /= weight * lightWeight;
```

⟨*Follow photon to non-specular surface and store in* photons *array*⟩≡
```
bool hitSpecular = false;
int nBounces = 0;
Surf photonSurf;
BSDF *photonBSDF = 0;
static StatsRatio specularHits("Integrator",
    "Photons that hit specular surface", true);
specularHits.add(0, 1);
while (scene->Intersect(photonRay, &photonSurf)) {
    L *= scene->Transmittance(photonRay);
    delete photonBSDF;
    photonBSDF = photonSurf.GetBSDF(photonRay);
    ⟨Handle non-specular photon hit⟩
    ⟨Handle specular photon hit⟩
    ⟨Possibly terminate photon path⟩
}
delete photonBSDF;
```

XXX what to do about shading normals?

⟨*Handle non-specular photon hit*⟩≡
```
if (hitSpecular &&
    photonBSDF->NumComponents() > photonBSDF->NumSpecular())
    photons.push_back(PhotonData(photonSurf.dgGeom, L,
        -photonRay.D.Hat()));
```

⟨*Photonmap Local Declarations*⟩+≡
```
struct PhotonData {
    PhotonData() { }
    PhotonData(const DifferentialGeometry &dg, const Spectrum &L,
        const Vector &w)
        : P(dg.P), N(dg.Nn), dE(L), wi(w) { }
    Point P;
    Normal N;
    Spectrum dE;
    Vector wi;
};
```

BSDF 298
dgGeom 10
DifferentialGeometry 47
Hat 19
Light 358
lights 579
Normal 23
Point 21
push_back 494
RandomInt 515
RayDifferential 26
size 494
Spectrum 155
StatsRatio 501
Surf 10
Vector 16

⟨*Handle specular photon hit*⟩≡
```
if (photonBSDF->NumSpecular() > 0) {
    ⟨Record photon specular hit statistics⟩
    hitSpecular = true;
    int specComponent = RandomInt() % photonBSDF->NumSpecular();
    Vector wi;
    Spectrum fr = photonBSDF->f_delta(specComponent,
        -photonRay.D, &wi);
    if (fr.Black())
        break;
    L *= fr * Float(photonBSDF->NumSpecular());
    photonRay = Ray(photonSurf.dgGeom.P, wi);
}
```

⟨*Record photon specular hit statistics*⟩≡
```
if (!hitSpecular)
    specularHits.add(1, 0);
```

⟨*Possibly terminate photon path*⟩≡
```
if (photonBSDF->NumSpecular() == 0)
    break;
if (nBounces++ > 3) {
    Float continueProbability = .2f;
    if (RandomFloat() > continueProbability)
        break;
    L /= continueProbability;
}
```

```
 10 dgGeom
 10 dgShading
523 KdTree
335 Lookup
474 PhotonData
476 PhotonProcess
515 RandomFloat
515 RandomInt
 26 Ray
494 size
155 Spectrum
 16 Vector
```

We can't say how much energy each photon carries until we're done shooting them; need to evenly divide ...

⟨*Normalize photon* dE *values and build tree*⟩≡
```
for (u_int i = 0; i < photons.size(); ++i)
    photons[i].dE /= Float(nshot);
tree = new KdTree<PhotonData, PhotonProcess>(photons);
```

Using the photon map

XXX density estimation is the formalization of the basic idea that...

In tricky settings, can be tough to get enough photons to the part of the scene you're actually looking at... This is the flip side to the problem of finding the small reflected light source on the ceiling when only tracing rays from the eye.

⟨*Perform density estimation with photon map*⟩≡
```
if (tree) {
    PhotonProcess proc(nLookup, surf.dgShading.P, surf.dgShading.Nn);
    Float md = maxDist;
    tree->Lookup(surf.dgShading.P, proc, md);
    ⟨Accumulate light from nearby photons⟩
}
```

⟨*Photonmap Local Declarations*⟩+≡
```
struct ClosePhoton {
    ClosePhoton(const PhotonData *d = NULL, Float md2 = HUGE_VAL) {
        data = d;
        maxDist2 = md2;
    }
    bool operator<(const ClosePhoton &p2) const {
        return maxDist2 < p2.maxDist2;
    }
    const PhotonData *data;
    Float maxDist2;
};
```

⟨*Photonmap Local Declarations*⟩+≡
```
struct PhotonProcess {
    PhotonProcess(u_int mp, const Point &p, const Normal &n);
    void operator()(const PhotonData &photon, Float &maxDist) const;

    const Point &P;
    const Normal &N;

    mutable vector<ClosePhoton> photons;
    u_int maxPhotons;
};
```

DistanceSquared 23
maxDist2 218
mp 131
Normal 23
PhotonData 474
Point 21
reserve 494
size 494

⟨*Photonmap Method Definitions*⟩+≡
```
PhotonProcess::PhotonProcess(u_int mp, const Point &p,
        const Normal &n)
    : P(p), N(n) {
    maxPhotons = mp;
}
```

⟨*Photonmap Method Definitions*⟩+≡
```
void PhotonProcess::operator()(const PhotonData &photon,
        Float &maxDist) const {
    if (Dot(photon.N, N) < .707f) return;
    Float d2 = DistanceSquared(photon.P, P);
    if (photons.size() == 0)
        photons.reserve(maxPhotons);
    if (photons.size() < maxPhotons) {
        ⟨Add photon to unordered array of photons⟩
    }
    else {
        ⟨Remove most distant photon from heap and add new photon⟩
    }
}
```

⟨*Add photon to unordered array of photons*⟩≡
```
photons.push_back(ClosePhoton(&photon, d2));
if (photons.size() == maxPhotons) {
    std::make_heap(photons.begin(), photons.end());
    maxDist = sqrtf(photons[0].maxDist2);
}
```

⟨*Remove most distant photon from heap and add new photon*⟩≡
```
std::pop_heap(photons.begin(), photons.end());
photons[maxPhotons-1] = ClosePhoton(&photon, d2);
std::push_heap(photons.begin(), photons.end());
maxDist = sqrtf(photons[0].maxDist2);
```

Actually, shouldn't use photons for very glossy surfaces. e.g. consider as we approach pure specular, it's more efficient to sample the BSDF and trace new rays, rather than using photons (analogous to sampling the light), since they're unlikely to be from directions we care about...

⟨*Accumulate light from nearby photons*⟩≡
```
vector<ClosePhoton> &photons = proc.photons;
if (photons.size() > 0) {
    ⟨Compute photon scale factor with density estimation⟩
    for (u_int i = 0; i < photons.size(); ++i)
        L += scale * bsdf->f(wo, photons[i].data->wi) *
            fabsf(Dot(photons[i].data->wi, N)) *
            photons[i].data->dE;
}
```

218 maxDist2
494 push_back
494 size

We just scale uniformly. Can reduce blurriness of results slightly by using a weighting function that gives greater weight to photons the closer they are to the point being shaded...

⟨*Compute photon scale factor with density estimation*⟩≡
```
Float scale = photons.size() / (nLookup * M_PI * md * md);
```

15.8 Irradiance Caching

⟨*irradiancecache.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "lrt.h"
#include "transport.h"
#include "scene.h"
#include "mc.h"
#include "octree.h"
⟨IrradianceCache Forward Declarations⟩
⟨IrradianceCache Local Declarations⟩
⟨IrradianceCache Declarations⟩
⟨IrradianceCache Method Definitions⟩
```

Another biased light transport technique is *irradiance caching*; it computes accurate estimates of irradiance at a sparse set of points throughout the scene and

then uses them to compute indirect lighting. Recall that irradiance is

$$E(x) = \int_{\mathcal{H}^2} L(x, \vec{\omega}) |\cos\theta| d\vec{\omega}.$$

It is in a sense a weighted average of incoming radiance at a point, giving a sense of the aggregate illumination there. The technique is most effective in environments where the indirect illumination is slowly-changing and where the BSDFs are generally Lambertian.

⟨*IrradianceCache Declarations*⟩≡
```
class IrradianceCache : public SurfaceIntegrator {
public:
    IrradianceCache(int md, Float maxerr, int nsamples,
        bool ss);
    Spectrum L(const Scene *scene, const RayDifferential &ray, const Sample *sa
    Sample *AllocateSample(const Scene *scene) const;
private:
    ⟨IrradianceCache Private Data⟩
    ⟨IrradianceCache Private Methods⟩
};
```

AllocateSample 444
lights 579
push_back 494
RayDifferential 26
Scene 5
size 494
Spectrum 155

⟨*IrradianceCache Method Definitions*⟩≡
```
IrradianceCache::IrradianceCache(int md, Float maxerr, int ns,
        bool ss) {
    maxDepth = md;
    maxError = maxerr;
    nSamples = ns;
    showSamples = ss;
    rayDepth = 0;
    indirectDepth = 0;
    ⟨IrradianceCache Constructor Implementation⟩
}
```

⟨*IrradianceCache Private Data*⟩≡
```
Float maxError;
bool showSamples;
int nSamples;
mutable int rayDepth, indirectDepth;
int maxDepth;
```

⟨*IrradianceCachee Method Definitions*⟩≡
```
Sample *IrradianceCache::AllocateSample(const Scene *scene) const {
    vector<int> num;
    for (int i = 0; i < 3; ++i)
        num.push_back(scene->lights.size());
    return new Sample(num, num);
}
```

⟨*Compute reflected radiance with irradiance cache*⟩≡
```
if (indirectDepth == 0) L += surf.Le(-ray.D);
⟨Evaluate BSDF at hit point⟩
Vector wo = -ray.D.Hat();
⟨Compute direct lighting for irradiance cache⟩
if (rayDepth++ < maxDepth) {
    Vector wi;
    ⟨Trace rays for specular reflection and refraction⟩
}
if (indirectDepth++ < maxDepth) {
    ⟨Estimate indirect lighting with irradiance cache⟩
}
--rayDepth;
--indirectDepth;
⟨Clean up from integration⟩
```

XXX ugh, double-counting issues with e.g. environment lighting sphere...

⟨*Compute direct lighting for irradiance cache*⟩≡
```
const Point &P = surf.dgShading.P;
const Normal &N = surf.dgShading.Nn;
L += UniformSampleAllLights(scene, P, N, wo, bsdf, sample, rayDepth);
```

```
10 DgShading
19 Hat
23 Normal
21 Point
450 UniformSampleAllLights
16 Vector
```

Recall the scattering equation, 5.4.8. Take the reflection-only part of it and you have

$$L_o(x,\vec{\omega}_o) = \int_{\mathcal{H}^2} L_i(x,\vec{\omega}_i)\, f(\vec{\omega}_i,\vec{\omega}_o)\, |\cos\theta_i| \mathrm{d}\vec{\omega}_i.$$

Here we will make the approximation

$$\begin{aligned} L_o(x,\vec{\omega}_o) &\approx \left(\int_{\mathcal{H}^2} L_i(x,\vec{\omega}_i)\, |\cos\theta_i| \mathrm{d}\vec{\omega}_i\right)\left(\int_{\mathcal{H}^2} f(\vec{\omega}_i,\vec{\omega}_o)\, \mathrm{d}\vec{\omega}_i\right) \\ &= E(x)\,(\pi\rho_{dh}(\vec{\omega}_o)) \end{aligned}$$

Where E denotes irradiance, as defined in Equation 5.2.5 and ρ_{dh} is the hemispherical-directional reflectance, introduced in Section 9.1.

This approximation may have an enormous amount of error, though it's fine when either of the two integrands is constant over the integration domain. In particular, to the degree that if the incident light distribution is uniform, or the BRDF is Lambertian, there is less error.

Other way to look at irradiance caching is if you separate the BRDF into diffuse and non-diffuse components, it gives you a way to compute indirect lighting for the diffuse bit. Sample the rest according to the usual MC techniques...

Irradiance caching uses this approximation as well as the observation that irradiance tends to change slowly in many scenes, particularly if you pull out the direct lighting bit and do that separately.

Differentiate between indirect depth: fewer rays, higher error and direct depth from specular stuff where error shouldn't go down

⟨*Estimate indirect lighting with irradiance cache*⟩≡
```
Spectrum E;
const Point &P = surf.dgShading.P;
const Normal &N = surf.dgShading.Nn;
if (!InterpolateIrradiance(scene, P, N, &E)) {
    ⟨Compute irradiance at current point⟩
    ⟨Add computed irradiance value to cache⟩
    if (showSamples) E *= 10000;
}
else if (showSamples) E = 0.;
L += E * M_PI * bsdf->rho(wo);
```

XXX don't want to include directly emitted light in the E sum
Estimate value of

$$E = \int_{\mathcal{H}^2} L_i(x, \vec{\omega}_i) \, |\cos\theta_i| \, d\vec{\omega}_i. \tag{15.8.4}$$

⟨*Compute irradiance at current point*⟩≡
```
⟨Determine how many samples to take for irradiance estimate⟩
Float *samples = (Float *)alloca(2 * ns * sizeof(Float));
LatinHypercube(samples, ns, 2);
Float invSumDists = 0.;
for (int i = 0; i < ns; ++i) {
    ⟨Trace ray to sample radiance for irradiance estimate⟩
}
E /= Float(ns);
Float maxDist = ns / invSumDists;
```

alloca	495
dgShading	10
LatinHypercube	408
Length	19
Normal	23
Point	21
Ray	26
Spectrum	155
Vector	16

⟨*Determine how many samples to take for irradiance estimate*⟩≡
```
int ns = nSamples;
for (int r = indirectDepth; r > 1; --r)
    ns >>= 1;
if (ns == 0) ns = 1;
```

We generate samples according to a cosine distribution, using Malley's method (Section 14.3.) The sample weight for MC integration should be $\pi/\cos\theta$, where θ is the ray's angle with the surface normal. However, bcause there is a $\cos\theta$ term in the integrand, Equation 15.8.4, the two cancel out and we just need to weight each sample by π.

We sample the ray in the canonical reflection coordinate system, with the normal mapped to the $+z$ axis; to get a world-space ray-direction, we can use the convenient method from the BSDF class.

⟨*Trace ray to sample radiance for irradiance estimate*⟩≡
```
⟨Update irradiance statistics for rays traced⟩
Vector w = CosineSampleHemisphere(samples[2*i], samples[2*i+1]);
Ray r(P, bsdf->LocalToWorld(w));
E += M_PI * scene->L(r);
invSumDists += 1.f / (r.maxt * r.D.Length());
```

⟨*Update irradiance statistics for rays traced*⟩≡
```
static StatsCounter nIrradianceRays("Integrator",
    "Indirect rays traced for irradiance");
++nIrradianceRays;
```

⟨*Add computed irradiance value to cache*⟩≡
```
if (maxError > 0.) {
    BBox sampleExtent(P);
    sampleExtent.Expand(maxDist);
    ⟨Allocate octree if needed⟩
    IrradSample sample(E, P, N, maxDist, indirectDepth);
    octree->Add(sample, sampleExtent);
    ⟨Update statistics for new irradiance sample⟩
}
```

⟨*Update statistics for new irradiance sample*⟩≡
```
static StatsCounter nSamplesComputed("Integrator",
    "Irradiance estimates comupted");
++nSamplesComputed;
```

⟨*IrradianceCache Local Declarations*⟩≡
```
struct IrradSample {
    IrradSample() { }
    IrradSample(const Spectrum &e, const Point &p, const Normal &n,
        Float md, int id) : E(e), N(n), P(p) {
            maxDist = md;
            indirectDepth = id;
    }
    Spectrum E;
    Normal N;
    Point P;
    Float maxDist;
    int indirectDepth;
};
```

30 Expand
23 Normal
21 Point
155 Spectrum
501 StatsCounter

⟨*IrradianceCache Local Declarations*⟩+≡
```
struct IrradProcess {
    IrradProcess(const Normal &n, Float me) {
        N = n;
        maxError = me;
        nFound = samplesChecked = 0;
        sumWt = 0.;
        E = 0.;
    }
    Normal N;
    Float maxError;
    // XXX int indirectLevel;
    mutable int nFound;
    mutable int samplesChecked;
    mutable Float sumWt;
    mutable Spectrum E;
    void operator()(const Point &P, const IrradSample &sample) const;
    bool successful() { return (nFound > 0 && sumWt > 0.); }
};
```


IrradianceCache 478
IrradSample 481
Lookup 335
Normal 23
Octree 518
Point 21
Scene 5
Spectrum 155

⟨*IrradianceCache Private Data*⟩+≡
```
mutable Octree<IrradSample, IrradProcess> *octree;
```

XXX need to delete it...

⟨*IrradianceCache Constructor Implementation*⟩≡
```
octree = NULL;
```

⟨*IrradianceCache Method Definitions*⟩+≡
```
bool IrradianceCache::InterpolateIrradiance(const Scene *scene,
        const Point &P, const Normal &N, Spectrum *E) const {
    ⟨Allocate octree if needed⟩
    IrradProcess proc(N, maxError);
    octree->Lookup(P, proc);
    ⟨Update irradiance cache lookup stats⟩
    if (proc.successful()) {
        *E = proc.E / proc.sumWt;
        return true;
    }
    return false;
}
```

⟨*Allocate octree if needed*⟩≡
```
if (!octree)
    octree =
        new Octree<IrradSample, IrradProcess>(scene->WorldBound());
```

XXX make sure sample was taken at same level or shallower
XXX need to use this fragment again

⟨*Update irradiance cache lookup stats*⟩≡
```
static StatsRatio nSuccessfulLookups("Integrator",
    "Successful irradiance cache lookups");
static StatsRatio nSamplesFound("Integrator",
    "Irradiance samples found per successful lookup", false);
nSuccessfulLookups.add(proc.successful() ? 1 : 0, 1);
nSamplesFound.add(proc.nFound, 1);
```

⟨*IrradianceCache Method Definitions*⟩+≡
```
void IrradProcess::operator()(const Point &P, const IrradSample &sample) const{
    ++samplesChecked;
    ⟨Skip sample if it is behind point being shaded⟩
    ⟨Skip sample if surface normals are too different⟩
    ⟨Skip sample if it's too far from the sample point⟩
    ⟨Computer estimate error term, err⟩
    if (err < maxError) {
        ++nFound;
        Float wt = 1.f / max(.05f, err);
        E += wt * sample.E;
        sumWt += wt;
    }
}
```

⟨*Skip sample if it is behind point being shaded*⟩≡
```
if (Dot(sample.P - P, (sample.N + N).Hat()) > .01)
    return;
```

⟨*Skip sample if surface normals are too different*⟩≡
```
if (Dot(N, sample.N) < .707f)
    return;
```

⟨*Skip sample if it's too far from the sample point*⟩≡
```
Float d2 = DistanceSquared(P, sample.P);
if (d2 > sample.maxDist * sample.maxDist)
    return;
```

⟨*Computer estimate error term, err*⟩≡
```
Float err = sqrtf(d2) / (sample.maxDist * Dot(N, sample.N));
```


23 DistanceSquared
19 Hat
482 IrradProcess
481 IrradSample
513 max
21 Point
501 StatsRatio

15.9 Volume Integration

The Equation of Transfer

The *equation of transfer* is the fundamental equation that governs the behavior of light in some medium that absorbs, emits, and scatters radiation (Cha60). As radiance travels along a beam, a number of processes contribute to change its distribution. Radiance can be increased due to emission and *in-scattering*, radiance along other beams that is scattered into the path of the beam under consideration. Conversely, radiance can be decreased due to absorption and *out-scattering*, radiance that is scattered into other beams.

The equation of transfer describes this process. In its most basic form, it is an integro-differential equation that describes how the radiance along a beam changes at a point. It can easily be derived by subtracting the effects of the scattering processes that reduce energy along the beam (absorption and out-scattering) from the processes that increase energy along the beam (emission and in-scattering). Here we will assume that the medium has a constant index of refraction—i.e. a beam follows a straight line path; see Preisendorfer (Pre65, Section 21) for the derivation in the more general setting.

We first define the *source function* as the amount of new light at a point in a direction due to emission and in-scattered light from other points in the medium:

$$\mathrm{S}(x,\vec{\omega}) = \mathrm{L_e}(x,\vec{\omega}) + \int_{S^2} \mathrm{p}(x,\vec{\omega'} \to \vec{\omega})\mathrm{L_i}(x,\vec{\omega'})\mathrm{d}\vec{\omega}'$$

Consider now a differential volume along a beam of radiation. The beam is paramaterized along its direction by a variable $t \geq 0$ such that points on the beam are given by $x + t\vec{\omega}$. Now by combining the source function with an expression for the loss in radiation due to attenuation and out-scattering, we have the integro-differential form of the equation of transfer (Cha60; **?**):

$$\frac{\partial}{\partial t}\mathrm{L}(x') = -\sigma_t \mathrm{L}(x') + \mathrm{S}(x') \tag{15.9.5}$$

With suitable boundary conditions, this can be transformed to a purely integral equation. If we assume that there are no surfaces in the scene, we have

$$\mathrm{L}(x,\vec{\omega}) = \int_0^\infty T_r(x \to x + t\vec{\omega})\mathrm{S}(x + t\vec{\omega},\vec{\omega})\mathrm{d}t. \tag{15.9.6}$$

More generally, if there are reflecting and/or emitting surfaces in the scene, we have:

$$\mathrm{L}(x,\vec{\omega}) = T_r(x \to x')(\mathrm{L_e}(x_0,-\vec{\omega}) + \mathrm{L_o}(x_0,-\vec{\omega})) + \int_0^{t'} T_r(x \to x + t\vec{\omega})\mathrm{S}(x + t\vec{\omega},\vec{\omega})\mathrm{d}t \tag{15.9.7}$$

where t' is the distance along the ray to the first surface, x' is the point on the surface, $x' = x + t'\vec{\omega}$, $\mathrm{L_e}$is the emitted radiance from the surface, and $\mathrm{L_o}$is the reflected radiance from the surface (see Equation **??**).

XXX draw some figures for this stuff.

introduce idea of "source term" for that delta-function point on the surface at the end?

Integrators

⟨*Volume Scattering Declarations*⟩+≡
```
class VolumeIntegrator {
public:
    ⟨VolumeIntegrator Methods⟩
};
```

⟨*VolumeIntegrator Methods*⟩+≡
```
virtual Spectrum Transmittance(const Scene *, const Ray &ray, const Sample *sam
    Float *alpha) const = 0;
```

⟨*VolumeIntegrator Methods*⟩+≡
```
virtual Spectrum L(const Scene *, const Ray &ray, const Sample *sample, Float *alpha) const = 0;
```

Null Integrator

⟨*null.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "volume.h"
⟨NullVolumeIntegrator Declarations⟩
⟨NullVolumeIntegrator Function Definitions⟩
```

⟨*NullVolumeIntegrator Declarations*⟩≡
```
class NullVolumeIntegrator : public VolumeIntegrator {
public:
    Spectrum Transmittance(const Scene *, const Ray &ray,
        const Sample *sample, Float *alpha) const {
        return Spectrum(1.);
    }
    Spectrum L(const Scene *, const Ray &ray,
        const Sample *sample, Float *alpha) const {
        return Spectrum(0.);
    }
};
```

```
 26 Ray
  5 Scene
155 Spectrum
484 VolumeIntegrator
```

Emission-Only Integrator

Just attenuation and emission; ignores light sources.

Make connection to standard models in graphics hardware, for example. Is all closed form if properties are homogeneous...

⟨*emission.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "volume.h"
#include "scene.h"
⟨EmissionIntegrator Declarations⟩
⟨EmissionIntegrator Function Definitions⟩
```

⟨*EmissionIntegrator Declarations*⟩≡
```
class EmissionIntegrator : public VolumeIntegrator {
public:
    ⟨EmissionIntegrator Methods⟩
private:
    ⟨EmissionIntegrator Private Data⟩
};
```

⟨*EmissionIntegrator Methods*⟩≡
```
EmissionIntegrator(int ns) { nSamples = ns; }
```

⟨*EmissionIntegrator Private Data*⟩≡
```
int nSamples;
```

⟨*EmissionIntegrator Function Definitions*⟩≡
```
Spectrum EmissionIntegrator::Transmittance(const Scene *scene,
        const Ray &ray, const Sample *sample, Float *alpha) const {
    Spectrum tau(0.);
    for (u_int i = 0; i < scene->volumeRegions.size(); ++i) {
        VolumeRegion *vr = scene->volumeRegions[i];
        tau += vr->tau(ray);
    }
    return Spectrum(2.712).Pow(-1 * tau);
}
```

⟨*EmissionIntegrator Function Definitions*⟩+≡
```
Spectrum EmissionIntegrator::L(const Scene *scene,
        const Ray &ray, const Sample *sample, Float *alpha) const {
    Spectrum Lv(0.);
    for (u_int i = 0; i < scene->volumeRegions.size(); ++i) {
        VolumeRegion *vr = scene->volumeRegions[i];
        Float t0, t1;
        if (!vr->Intersect(ray, &t0, &t1)) continue;
        ⟨Do emission-only volume integration in vr⟩
    }
    return Lv;
}
```

XXXX should sample optical depth, then raymatch until we hit it?

⟨*Do emission-only volume integration in* vr⟩≡
```
Spectrum Lvr(0.);
Point Pprev = ray(t0);
Spectrum T(1.);
for (int j = 0; j < nSamples; ++j) {
    ⟨Step forward to next volume sample point and update T⟩
    Lvr += T * vr->Le(P, -ray.D);
    Pprev = P;
}
Lv += Lvr / nSamples * Distance(ray(t0), ray(t1));
```

⟨*Step forward to next volume sample point and update* T⟩≡
```
Float t = Lerp((j + RandomFloat()) / Float(nSamples), t0, t1);
Point P = ray(t);
Ray rp(Pprev, P-Pprev, 0, 1);
T *= Spectrum(2.712).Pow(-1. * vr->tau(rp));
```


Distance 23
EmissionIntegrator 485
Lerp 512
Point 21
RandomFloat 515
Ray 26
Scene 5
size 494
Spectrum 155
VolumeRegion 383
volumeRegions 579

Single Scattering Integrator

Will re-use some fragments from emission-only integrator...

⟨*single.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "volume.h"
#include "scene.h"
⟨SingleScattering Declarations⟩
⟨SingleScattering Function Definitions⟩
```

⟨*SingleScattering Declarations*⟩≡
```
class SingleScattering : public VolumeIntegrator {
public:
    ⟨SingleScattering Methods⟩
private:
    ⟨SingleScattering Private Data⟩
};
```

⟨*SingleScattering Methods*⟩≡
```
SingleScattering(int ns) { nSamples = ns; }
```

⟨*SingleScattering Private Data*⟩≡
```
int nSamples;
```

Implementation is the same as emission-only integrator; won't include it here.

⟨*SingleScattering Methods*⟩+≡
```
Spectrum Transmittance(const Scene *, const Ray &ray, const Sample *sample, Float *alpha) const
```

⟨*SingleScattering Function Definitions*⟩+≡
```
Spectrum SingleScattering::L(const Scene *scene,
        const Ray &ray, const Sample *sample, Float *alpha) const {
    Spectrum Lv(0.);
    for (u_int i = 0; i < scene->volumeRegions.size(); ++i) {
        VolumeRegion *vr = scene->volumeRegions[i];
        Float t0, t1;
        if (!vr->Intersect(ray, &t0, &t1)) continue;
        ⟨Do single scattering volume integration in vr⟩
    }
    return Lv;
}
```


23 Distance
21 Point
26 Ray
5 Scene
494 size
155 Spectrum
484 VolumeIntegrator
383 VolumeRegion
579 volumeRegions

XXXX should sample optical depth, then raymatch until we hit it?

⟨*Do single scattering volume integration in* vr⟩≡
```
Spectrum Lvr(0.);
Point Pprev = ray(t0);
Spectrum T(1.);
for (int j = 0; j < nSamples; ++j) {
    ⟨Step forward to next volume sample point and update T⟩
    Lvr += T * vr->Le(P, -ray.D);
    ⟨Compute direct lighting at P in volume⟩
    Lvr += T * Ldirect;
    Pprev = P;
}
Lv += Lvr / nSamples * Distance(ray(t0), ray(t1));
```

⟨*Compute direct lighting at* P *in volume*⟩≡
```
Spectrum Ldirect(0.);
Spectrum ss = vr->sigma_s(P, -ray.D);
if (!ss.Black()) {
    Spectrum albedo = ss / vr->sigma_t(P, -ray.D);
    for (u_int i = 0; i < scene->lights.size(); ++i) {
        Light *light = scene->lights[i];
        ⟨Compute direct volume lighting from light⟩
    }
}
```

⟨*Compute direct volume lighting from* light⟩≡
```
Float weight;
bool deltaLight;
VisibilityTester vis;
Vector wo;
Spectrum L = light->Sample_L(P, RandomFloat(), RandomFloat(),
    &wo, &weight, &deltaLight, &vis);
if (!L.Black() && vis.Unoccluded(scene))
    Ldirect += L * vis.Transmittance(scene) *
        albedo * vr->phase(P, -ray.D, wo);
```

Light 358
lights 579
RandomFloat 515
size 494
Spectrum 155
Vector 16
VisibilityTester 359

Further Reading

Lommel was the apparently first to derive the equation of transfer (Lom89), in a not-widely-known paper. Not only did he derive the equation of transfer, but he solved it in some simplified cases in order to estimate reflection functions from real world surfaces (including marble and paper) and compared his solutions to measured reflectance data from these surfaces.

Apparently unaware of Lommel's work, Schuster was the next worker in radiative transfer to consider the effect of multiple scattering (Sch05). He used the term *self-illumination* to describe the fact that each part of the medium is illuminated by every other part of the medium and derived differential equations that described reflection from a slab along the normal direction assuming the presence of isotropic scattering; the conceptual framework that he developed remains essentially unchanged in the field of radiative transfer

Soon thereafter, Scharzchild introduced the concept of radiative equilibrium (**?**) and Jackson expressed Schuster's equation in integral form, also noting that "the obvious physical mode of solution is Liouville's method of successive substitutions." (i.e. a Neumann series solution) (Jac10). Finally, King completed the rediscovery of the equation of transfer by expressing it in the general integral form (Kin13). (Yanovitskij (Yan97) traces the origin of the integral equation of transfer to Chvolson (Chv90), but we have been unable to find a copy of this paper.)

Russian roulette introduced to graphics by Kirk and Arvo (KA91).

Lafortune bidir (LW94). Veach and Guibas (VG94). Kollig and Keller bidir with quasi-random sample patterns (KK00).

Irradiance caching (WRC88; WH92; War94b)

Kajiya (Kaj86), Immel et al (ICG86)

Photon maps. Arvo (Arv86). Collins (Col94). Jensen (Jen96; JC98).

Shirley thesis (Shi90a; Shi90b), incl sum over paths formulation

Metropolis (VG97) (PKK00)

Radiance (and radiosity stuff) for virtual mirrors for light paths...

The equation of transfer was first introduced to graphics by Kajiya and von Herzen (KH84); Rushmeier was the first to compute solutions of it in a general setting (**?**). However, Arvo was the first to make the essential connections between previous formalizations of light transport in graphics and the equation of transfer and radiative transfer in general (**?**).

Bhate and Tokuta spherical harmonic approach (BT92). Pérez/Pueyo/Sillion volume globillum survey (PPS97).

Blasi et al two pass Monte Carlo algorithm, somewhat in the spirit of Kajiya and von Herzen, where first pass shoots energe from lights and stores it in a grid, second pass does final rendering (BSS93).

Lafortune and Willems bidir stuff (LW96).

Jensen book (Jen01).

Exercises

15.1 The light sources are currently somewhat inefficient since their differential irradiance `dE` functions always trace a shadow ray from the point being shaded to the light, even if the BSDF returns black for that particular directions. Modify these interfaces so that the BSDF's value is computed before the light traces the visibility ray. How much does this speed up `lrt`?

15.2 To further improve efficiency, Russian roulette can be applied to skip tracing most of the shadow rays that make a low contribution to the final image. Tentatively compute product of the BSDF and the differential irradiance before tracing a shadow ray; if the result is low, then apply Russian roulette.

XXX should use efficiency optimized Russian Roulette

15.3 Path tracing to be able to flag important stuff for indirect lighting, be able to sample it according to dA. Then use MIS to compute weights. Experiments with scene with substantial indirect lighting: how much help, how much does it hurt when mostly direct? What if the wrong objects are flagged as important? Or if MIS isn't used? What about dynamically changing probabilities based on experience...

15.4 Adjoint BSDF: shading normals and transmission both mess up reciprocity assumptions. Implement Veach methods to account for this, use in photon tracing and bidir...

15.5 Photons as paths from light for bidir–use as tiny light sources–unbiased.

15.6 Final gather for photon globillum–use using non-specular photons directly is bad...

15.7 MIS for bidir..

15.8 Bidirectional estimator to compute irradiance cache sample values. Describe basic formulation, etc..

15.9 Expected values for many light source handling. Can probabilistically assume a value for part of the integrand. Then x% of the time, compute it for real, weight result by $(\text{guess} - \text{actual})/x$%... Show that this is an unbiased estimator, etc...

15.10 kajiya-von herzen stuff, precompute illumination on a grid, save all those redundant-ray marching computations

15.11 MIS for lighting in volums

15.12 bidir for lighting in volumes, cite path integral generalization to volumes by the volume metropolis guys

16. Summary and Conclusion

Writing a renderer is one of the great pleasures of graphics...

Ray-tracing as a way to get to fundamentals of rendering, regardless of approach. Framework to understand signal processing, Monte Carlo, etc.

The advent of real-time ray-tracing, hardware accelerated...

16.1 Major Projects

Parallel rendering

threads/shared memory approach: basic idea of shared address space, etc. generally an easier programming style than message-passing-based parallel programming, though it can be trickier to get it right.

the key problem is correct access of shared data; need to be careful that one thread isn't part-way through modifying one data structure such that if another thread reads it, it gets garbage. or two threads simultaneously trying to update it and who knows what the end result is.

mechanism: mutual exclusion–e.g. lock to access a key data structure. ensures that only one thread is using it at a time.

probably want to do scene parsing, accel building single threaded, render multi, then cleanup, stats, exit as single.

big issues: mailbox, film/image update, refinement of primitives, sampler. Also stats...

once primitives are refined (and if no mailboxing...), accel structure is read-only, so no need to lock it for threads to access it.

for stuff like film/image and sampler, don't want to e.g. acquire and release a lock each time an image sample has been computed and the image needs to be

updated, or each time a new sample value is needed–the time to get the lock will probably be more than the computation that is done, and access to such heavily accessed data structures will see a lot of contention (and thus threads waiting while one thread is modifying it.)

one approach to this problem is having each thread keeping a separate copy of the data structures–e.g. separate `Film` objects. Then when rendering is done, they are merged into a single `Film` object and final processing is done serially. For big data structures like the scene description, this may be too much, but it solves the contention problem completely.

For sampling, everyone could have the same `Sampler` object but have the convention that if n threads, the first thread only uses sample number $0, n, 2n, \ldots$, the second thread uses samples $1, n+1, 2n+1, \ldots$, and so forth, where everyone just ignores the other samples, knowing that another thread will handle them. This may be slow for samplers that take a relatively large amount of time to generate samples (e.g. the `HammersleySampler`, which calls the not-speedy `RadicalInverse()` function). Alternatively, samplers could be thread-savvy, and could be instructed to only generate every n samples.

Other approach to distributed over a set of machines: central server process hands out sub-regions of image to render, worker processes render those bits, send back results. More message-passing style.

Memory performance

Keys to `lrt`'s memory use design:

1. `lrt` doesn't do any dynamic allocation during the rendering process, except for refining primitives as needed; this was carefully designed. (except for BSDF stuff, but that is easy for a good allocator...)

2. Allocate stuff in large blocks, not single items at a time (except for performance-unimportant stuff.)

3. Cache-aware alignment and data structures (keep stuff that will be accessed at about the same time nearby, blocking, avoid L1 cache conflights for key stuff, pack to small size)

Grunwald et al have shown that the system's choice of dynamic memory allocation implementation can have a substantial impact on the cache behavior of the program (GZH93a)

Much lore about dynamic memory allocation, the need to write custom allocators for speed. (City appropriate section of Stroustroup on overloading new/delete?)

The one type of custom allocator that did lead to performance improvements in practice was region-based (aka arenas) allocation, which we provide support for via the XXX object/interface.

We worry more about how the allocator is being called than what it is doing. (e.g. arrays of objects, not individual allocation, etc.)

Cite data structure reorganization papers and ideas, results with radiance (2 papers)

texture and geometry caching XXX citations...

computation reordering Pharr et al 97

A. Utilities

We will now define some of the assorted utility routines that were used throughout the system. These routines, though key to `lrt`'s operation, are relatively less interesting than the rest of the code in the system. It is good to have basic familiarity with them in order to understand other code, but understanding their implementation in detail isn't necessary to understand `lrt`.

First is a set of routines for error reporting; these are used for things ranging from reporting invalid input from the user to reporting fundamental bugs in `lrt`. By gathering all error reporting in a single place, we make it easy to change how errors of various severities are handled. Next are routines for gathering statistics about the performance of the ray-tracer. Again, by gathering all of this data through a common set of interfaces, it's easy to adjust the detail of statistics reported to the user. Next is a set of miscellaneous short mathematical functions; these provide some primitive operations that have wide application. Finally is a random number generator and various basic 3D data structures (k-d trees and octrees.)

⟨*util.cc**⟩≡
⟨*Source Code Copyright*⟩
```
#include "lrt.h"
#include <malloc.h>
```
⟨*Error Reporting Includes*⟩
⟨*Error Reporting Definitions*⟩
⟨*Error Reporting Functions*⟩
⟨*Matrix Method Definitions*⟩
⟨*Statistics Definitions*⟩
⟨*Statistics Functions*⟩
⟨*Random Number State*⟩
⟨*Random Number Functions*⟩
⟨*Memory Allocation Functions*⟩
⟨*StringHashTable Method Definitions*⟩

A.1 The C++ Standard Library

XXX start discussing container classes in general, then specialize down to vectors, sets, and maps XXX

For the benefit of readers unfamiliar with C++'s standard library, we will briefly review some of its facilities that we will be using. The `vector` class from the C++ standard library is a parameterized container class. It is similar to an array, though it can automatically grow as items are added to it. As it is a are template classe, a vector of `int`s (for example) is declared as `vector<int> v;`.

To add a new item to the end of a vector, a `push_back` method is available:

```
vector<int> vec;
for (int i = 0; i < 10; ++i)
    vec.push_back(i);
```

We can't say `vec[i] = i` in the above loop, since the vector needs to be informed that the user needs it to grow bigger, so that space may need to be allocated if needed.

A useful operation supported by `vector`s is the `reserve` call. This lets us inform the vector the number of items that we will be adding to it; this lets it allocate sufficient space once, rather than needing to grow repeatedly as we insert items into it (e.g. `vec.reserve(100)` reserves 100 spaces in the vector.)

The vector class provides a `size` method, which returns the total number of items inside of it. This method can be be used in conjunction with the `[]` operator to access items in the vector directly:

```
for (int i = 0; i < vec.size(); ++i)
    printf ("%d\n", vec[i]);
```

After a `vector` has been filled (e.g. with `push_back`), its members an be modified with the `[]` operator as well.

Vectors also provide an erase method; this takes two iterators to the sequence and removes all of the items from the first to the one before the last. Thus,

```
v.erase(v.begin(), v.end());
```

empties a vector completely.

Finally, the `pair` template class will be occasionally used; it provides a convenient way to construct a new object that holds two other objects. For example, if we're filling a hash table and are storing an array of pointers to hashed objects `Foo` with their integer hash values, we might declare an array of `pair<Foo *, int>`. Given a variable `p` that is a pair of objects, the constituent objects can be accessed as `p.first` and `p.second`. We can create a pair object with the `make_pair` function:

```
int i = 0, Foo *foop = NULL;
pair<Foo *, int> p = make_pair(foop, i);
p.first = new Foo;
```

XXX sets and maps XXX
XXX string XXX

Variable stack allocation

alloca...

A.2 Error Reporting

We provide four functions for reporting error conditions. In increasing severity, they are `Info`, `Warning`, `Error`, and `Severe`. All of them take a formatting string as their first argument and then a variable number of arguments providing values for the format. The syntax is identical to that used by the `printf` family of functions. For example,

```
Info("Now tracing ray number %d\n", rayNum);
```

Some compilers have non-portable ways of indicating that particular functions take a formatting string like `printf` with a variable number of arguments. These compilers can then verify that the types of the extra arguments after the formatting string are appropriate for the format. Thus, code like:

```
int FrameNum;
Info("Finished rendering frame number %f\n", FrameNum);
```

can be properly flagged as incorrect, since the formatting string indicates that `FrameNum` is a `double`, while it is actually an `int`. We define `PRINTF_FORMAT` here depending on which compiler is being used; for those where it's not possible to enable this type of syntax check, `PRINTF_FORMAT` just has an empty definition.

⟨*Setup printf format*⟩≡

```
#ifdef __GNUG__
#define PRINTF_FORMAT __attribute__ \
    ((__format__ (__printf__, 1, 2)))
#else
#define PRINTF_FORMAT
#endif // __GNUG__
```

Now we can declare the four error reporting functions, using `PRINTF_FORMAT` if available.

⟨*Global Function Declarations*⟩+≡
```
⟨Setup printf format⟩
extern void Info(const char *, ...) PRINTF_FORMAT;
extern void Warning(const char *, ...) PRINTF_FORMAT;
extern void Error(const char *, ...) PRINTF_FORMAT;
extern void Severe(const char *, ...) PRINTF_FORMAT;
```

Because all four of these functions do almost the same thing—first format the error string and then do something with it—all of them call a common function, passing along the error information from the user as well as information about what to do with the message. It may be ignored, in which case the message is discarded; it may be printed and then program execution may continue, or it may be an error of such severity that it's impossible to go on and the program must abort.

⟨*Error Reporting Definitions*⟩≡
```
#define LRT_ERROR_IGNORE 0
#define LRT_ERROR_CONTINUE 1
#define LRT_ERROR_ABORT 2
```

Error 498
Info 497
Severe 498
Warning 497

We need to include the header that provides the general functionality for processing a variable number of arguments.

⟨*Error Reporting Includes*⟩≡
```
#include <stdarg.h>
```

Now we can define the shared internal error reporting function, `processError`. It takes the error message and arguments from the user, an additional string that gives the type of error, and an `int` that should have the value `LRT_ERROR_IGNORE`, `LRT_ERROR_CONTINUE`, or `LRT_ERROR_ABORT`.

⟨*Error Reporting Functions*⟩≡
```
static void processError(const char *format, va_list args,
        const char *message, int disposition) {
    ⟨Format error string⟩
    ⟨Report error⟩
}
```

First we need to take the formatting string and the additional arguments passed by the user giving values to be substituted in the formatting string and turn it into a new string with those substitutions performed. Thankfully, the `vsprintf` function in the standard C library takes care of this for us.

⟨*Format error string*⟩≡
```
#define ERR_BUF_SZ 1024
static char errorBuf[ERR_BUF_SZ];
vsprintf(errorBuf, format, args);
```

Now that we have the error message in `errorBuf`, we print it or not, and exit the program it the error was a big one.

⟨*Report error*⟩≡
```
switch (disposition) {
case LRT_ERROR_IGNORE:
    return;
case LRT_ERROR_CONTINUE:
    fprintf(stderr, "%s: %s\n", message, errorBuf);
    ⟨Print scene file and line number, if appropriate⟩
    break;
case LRT_ERROR_ABORT:
    fprintf(stderr, "%s: %s\n", message, errorBuf);
    ⟨Print scene file and line number, if appropriate⟩
    abort();
}
```

⟨*Print scene file and line number, if appropriate*⟩≡
```
extern int line_num;
if (line_num != 0) {
    extern string current_file;
    fprintf(stderr, "\tLine %d, file %s\n", line_num,
        current_file.c_str());
}
```

We can now define the four globally-visible error functions. All are identical, except for how they prefix the error message and how it is disposed of. `Severe` is the only one that aborts execution; code that calls the other error reporting functions must therefore be able to recover from any error that is reported by `Error`, etc. These functions are quite straightforward. They use the standard C functions for getting ready to process a variable number of function arguments; after `va_start` is called, the `args` variable encapsulates information about the remaining arguments to the function. However, rather than calling the `va_arg` function to examine the subsequent arguments, we just pass the `args` variable into `processError`. It then passes it in to `vsprintf`, which handles unpacking the arguments.

496 errorBuf
496 LRT_ERROR_ABORT
496 LRT_ERROR_CONTINUE
496 LRT_ERROR_IGNORE
496 processError

⟨*Error Reporting Functions*⟩+≡
```
void Info(const char *format, ...) {
    va_list args;
    va_start(args, format);
    processError(format, args, "Notice", LRT_ERROR_CONTINUE);
    va_end(args);
}
```

⟨*Error Reporting Functions*⟩+≡
```
void Warning(const char *format, ...) {
    va_list args;
    va_start(args, format);
    processError(format, args, "Warning", LRT_ERROR_CONTINUE);
    va_end(args);
}
```

⟨*Error Reporting Functions*⟩+≡
```
void Error(const char *format, ...) {
    va_list args;
    va_start(args, format);
    processError(format, args, "Error", LRT_ERROR_CONTINUE);
    va_end(args);
}
```

⟨*Error Reporting Functions*⟩+≡
```
void Severe(const char *format, ...) {
    va_list args;
    va_start(args, format);
    processError(format, args, "Fatal Error", LRT_ERROR_ABORT);
    va_end(args);
}
```

We also define our own version of the standard `assert` macro. This asserts that an expression's value is true; if not, `Severe` is called with information about where the assertion failed.

⟨*Global Inline Functions*⟩+≡
```
#ifdef NDEBUG
#define Assert(expr) ((void)0)
#else
#define Assert(expr) \
    ((expr) ? (void)0 : Severe("Assertion " #expr " failed in %s, line %d", \
                               __FILE__, __LINE__))
#endif // NDEBUG
```

LRT_ERROR_ABORT 496
LRT_ERROR_CONTINUE 496
processError 496
size 494

Reporting Progress

⟨*Global Classes*⟩≡
```
struct ProgressReporter {
    ⟨ProgressReporter Methods⟩
    ⟨ProgressReporter Data⟩
};
```

⟨*ProgressReporter Methods*⟩≡
```
ProgressReporter(int t, const string &ti, int wid = 65)
    : width(wid - ti.size()), frequency(t / width), total(t) {
    count = 0;
    nPlusses = 0;
    gettimeofday(&start, NULL);
    title = ti;
}
```

⟨*Global Include Files*⟩+≡
```
#include <sys/time.h>
```

⟨*ProgressReporter Data*⟩≡
```
const int width, frequency, total;
mutable int count, nPlusses;
struct timeval start;
string title;
```

⟨*ProgressReporter Methods*⟩+≡
```
void operator()(FILE *file) const {
    if (count-- == 0) {
        count = frequency;
        ⟨Update progress plus signs⟩
        ⟨Update elapsed time and estimated time to completion⟩
    }
}
```

⟨*Update progress plus signs*⟩≡
```
fprintf(file, "\r%s: [", title.c_str());
++nPlusses;
for (int i = 0; i < nPlusses; ++i)
    fprintf(file, "+");
for (int i = 0; i < width - nPlusses; ++i)
    fprintf(file, " ");
fprintf(file, "]");
```

513 max

⟨*Update elapsed time and estimated time to completion*⟩≡
```
struct timeval now;
gettimeofday(&now, NULL);
Float percentDone = (Float)nPlusses / (Float)width;
Float seconds = now.tv_sec - start.tv_sec +
    (now.tv_usec - start.tv_usec) / 1e6f;
Float estRemaining = seconds / percentDone - seconds;
fprintf(file, " (%.2fs|%.2fs)  ", seconds, max(0.f, estRemaining));
```

A.3 Statistics

We also provide a unified interface for gathering statistics. This way, various parts of the program call into a single point where they can register what sorts of statistics they will be recording. At program termination, a single function call causes all such statistics to be printed out.

Two types of statistics can be gathered:

- *Counters*: These provide a way to count the frequency of something—e.g. the total number of rays that are traced while making an image.
- *Ratios*: This records the ratio of the frequency of two events—e.g. the number of successful ray-triangle intersection tests versus the total number of ray-triangle intersection tests.

When a statistic type is reported to the statistics system, the caller must provide a category and a name for the particular statistic. The category gives a way to gather

related types of statistics in output (e.g. all of the statistics gathered by the camera module can be reported together.) The name specifically describes the particular statistic. The caller also passes a pointer to data that holds the value of the statistic. This data must not go out of scope; it should either be a global or `static` variable or dynamically allocated and never freed. This guarantees that the statistics module can later dereference the supplied pointer to get the appropriate value without risk of error.

Now we can define a simple `struct` that holds information about each statistic that the user asked us to track. It stores the category, name, and level of the particular statistic as well as a pointer to the variable or variables that hold its value. For simplicity, we will store both counter and ratio statistics in the same `struct`, differentiating between them by setting `ptrb` to `NULL` when the `StatTracker` is tracking a counter rather than a ratio.

⟨*Global Type Declarations*⟩≡
```
typedef double StatsCounterType;
```

⟨*Statistics Definitions*⟩≡
```
struct StatTracker {
    StatTracker(const string &cat, const string &n,
                StatsCounterType *pa, StatsCounterType *pb = NULL,
            bool percentage = true);
    string category, name;
    StatsCounterType *ptra, *ptrb;
    bool percentage;
};
```

To construct a `StatTracker`, then, we just copy the strings the user passed in to us and store the appropriate pointers.

⟨*Statistics Functions*⟩≡
```
StatTracker::StatTracker(const string &cat, const string &n,
                         StatsCounterType *pa, StatsCounterType *pb, bool p) {
    category = cat;
    name = n;
    ptra = pa;
    ptrb = pb;
    percentage = p;
}
```

All of the `StatTracker`s are stored in a `static vector`.

⟨*Statistics Definitions*⟩+≡
```
static vector<StatTracker *> trackers;
```

We'll define a short function that takes care of adding a `StatTracker` to the `trackers` array; it first looks through all of the already-registered `StatTracker`s and makes sure that this isn't a duplicate. If it is, an error message is printed and it isn't added again. The caller should ensure that each statistic is only reported to the statistics system once.

⟨*Statistics Definitions*⟩+≡
```
static void addTracker(StatTracker *newTracker) {
    for (u_int i = 0; i < trackers.size(); ++i) {
        if (newTracker->category == trackers[i]->category &&
            newTracker->name == trackers[i]->name)
            return;
    }
    trackers.push_back(newTracker);
}
```

XXX

⟨*Global Classes*⟩+≡
```
class StatsCounter {
public:
    ⟨StatsCounter Interface⟩
private:
    ⟨StatsCounter Private Data⟩
};
```

⟨*StatsCounter Interface*⟩≡
```
StatsCounter(const string &category, const string &name);
```

⟨*Statistics Functions*⟩+≡
```
StatsCounter::StatsCounter(const string &category, const string &name) {
    num = 0;
    addTracker(new StatTracker(category, name, &num));
}
```

513 max
513 min
494 push_back
494 size
500 StatsCounterType
500 StatTracker
500 trackers

⟨*StatsCounter Interface*⟩+≡
```
void operator++() { ++num; }
void operator++(int) { ++num; }
```

⟨*StatsCounter Interface*⟩+≡
```
void stat_max(StatsCounterType val) { num = max(val, num); }
void stat_min(StatsCounterType val) { num = min(val, num); }
operator int() { return (int)num; }
```

⟨*StatsCounter Private Data*⟩≡
```
StatsCounterType num;
```

⟨*Global Classes*⟩+≡
```
class StatsRatio {
public:
    ⟨StatsRatio Interface⟩
private:
    ⟨StatsRatio Private Data⟩
};
```

⟨*StatsRatio Interface*⟩≡
```
StatsRatio(const string &category, const string &name, bool percent = true);
```

⟨*Statistics Functions*⟩+≡
```
StatsRatio::StatsRatio(const string &category, const string &name, bool percent
    na = nb = 0;
    addTracker(new StatTracker(category, name, &na, &nb, percent));
}
```

⟨*StatsRatio Interface*⟩+≡
```
void add(int a, int b) { na += a; nb += b; }
```

⟨*StatsRatio Private Data*⟩≡
```
StatsCounterType na, nb;
```

Once rendering has started, the values pointed to by the various statistics pointers will start to be interesting. As rendering progresses or when it is finished, the `StatsPrint` function can be called to print the current statistics values to a `FILE`.

⟨*Statistics Functions*⟩+≡
```
struct CmpTracker {
    bool operator()(const StatTracker *t1,
            const StatTracker *t2) const {
        if (t1->category == t2->category)
            return (t1->name < t2->name);
        return (t1->category < t2->category);
    }
};
```

addTracker 501
size 494
sort 513
StatsCounterType 500
StatsRatio 501
StatTracker 500
trackers 500

⟨*Statistics Functions*⟩+≡
```
void StatsPrint(FILE *dest) {
    fprintf(dest, "Statistics:\n");
    vector<StatTracker *> t = trackers;
    sort(t.begin(), t.end(), CmpTracker());
    string lastCategory;
    for (u_int i = 0; i < t.size(); ++i) {
        ⟨Print statistic⟩
    }
}
```

For now we actually won't sort the various statistics by category and name and report them cleanly. Just loop through all of them and print out the relevant information.

⟨*Print statistic*⟩≡
```
if (t[i]->category != lastCategory) {
    fprintf(dest, "%s\n", t[i]->category.c_str());
    lastCategory = t[i]->category;
}
fprintf(dest, "    %s", t[i]->name.c_str());
⟨Pad out to results column⟩
if (t[i]->ptrb == NULL)
    StatsPrintVal(dest, *t[i]->ptra);
else {
    if (*t[i]->ptrb > 0) {
        Float ratio = (Float)*t[i]->ptra / (Float)*t[i]->ptrb;
        StatsPrintVal(dest, *t[i]->ptra, *t[i]->ptrb);
        if (t[i]->percentage)
            fprintf(dest, " (%3.2f%%)", 100. * ratio);
        else
            fprintf(dest, " (%.2fx)", ratio);
    }
    else
        StatsPrintVal(dest, *t[i]->ptra, *t[i]->ptrb);
}
fprintf(dest, "\n");
```

513 min
494 size
500 StatsCounterType

⟨*Statistics Functions*⟩+≡
```
static void StatsPrintVal(FILE *f, StatsCounterType v) {
    if (v > 1e9) fprintf(f, "%.3fB", v / 1e9f);
    else if (v > 1e6) fprintf(f, "%.3fM", v / 1e6f);
    else if (v > 1e4) fprintf(f, "%.1fk", v / 1e3f);
    else fprintf(f, "%.0f", (float)v);
}
```

⟨*Statistics Functions*⟩+≡
```
static void StatsPrintVal(FILE *f, StatsCounterType v1,
        StatsCounterType v2) {
    StatsCounterType m = min(v1, v2);
    if (m > 1e9) fprintf(f, "%.3fB:%.3fB", v1 / 1e9f, v2 / 1e9f);
    else if (m > 1e6) fprintf(f, "%.3fM:%.3fM", v1 / 1e6f, v2 / 1e6f);
    else if (m > 1e4) fprintf(f, "%.1fk:%.1fk", v1 / 1e3f, v2 / 1e3f);
    else fprintf(f, "%.0f:%.0f", v1, v2);
}
```

After printing the name, we print enough spaces so that all of the statistic values start in the column `resultsColumn`.

⟨*Pad out to results column*⟩≡
```
int resultsColumn = 56;
int paddingSpaces = resultsColumn - (int) t[i]->name.size();
while (paddingSpaces-- > 0)
    putc(' ', dest);
```

When the program is freeing up memory when it's about to exit, it can call the `StatsCleanup` function, which frees the `StatsTrackers` that we've created.

⟨*Statistics Functions*⟩+≡
```
void StatsCleanup() {
    ⟨Reset user-supplied statistics pointers⟩
    for (u_int i = 0; i < trackers.size(); ++i)
        delete trackers[i];
    trackers.erase(trackers.begin(), trackers.end());
}
```

We reset the various counter pointers that the user gave us to zero before we destroy the trackers; this way, if the renderer runs again before the program exits, all of the various statistics will start counting from zero again.

⟨*Reset user-supplied statistics pointers*⟩≡
```
for (u_int i = 0; i < trackers.size(); ++i) {
    trackers[i]->ptra = 0;
    if (trackers[i]->ptrb)
        trackers[i]->ptrb = 0;
}
```

size	494
trackers	500

A.4 Memory Management

The conventional wisdom about memory allocation is that allocation based on the system's `malloc()` and `new()` routines is slow and that it is often worth-while to write custom allocation routines for objects that will be frequently allocated and freed. However, this conventional wisdom seems to be wrong. Wilson et al (WJNB95), Johnstone and Wilson (JW99), and Berger et al (BZM01; BZM02) have all investigated the performance of memory allocation routines with real applications and have found that user-written allocators almost always result in *worse* performance in both execution time and memory use compared to a well-written generic system memory allocator.

The one type of custom allocation technique that was found to be useful was *arena-based allocation*, which allows the user to quickly allocate objects from a large contiguous region of memory. In this scheme, individual objects can't be freed; only when the lifetime of all of the allocated objects is over is the entire region of memory freed. Therefore, we will implement a `MemoryArena` class in this section.

This section also includes implementations of `ReferenceCounted` and `Reference` classes, which ensure that objects that are referred to by multiple other objects and have difficult-to-determine lifetimes will be freed when no objects refer to them any more. Finally, we provide some routines that allocate regions of memory with guaranteed cache alignment properties, which is useful for reducing cache misses to frequently-accessed dynamically-allocated data structures.

Arena-Based Allocation

⟨*Global Classes*⟩+≡
```
template <class T> class MemoryArena {
public:
    ⟨MemoryArena Interface⟩
private:
    ⟨MemoryArena Private Data⟩
};
```

⟨*MemoryArena Interface*⟩≡
```
MemoryArena() {
    nAvailable = 0;
}
```

⟨*MemoryArena Private Data*⟩≡
```
T *mem;
int nAvailable;
vector<T *> toDelete;
```

⟨*MemoryArena Interface*⟩+≡
```
~MemoryArena() { FreeAll(); }
```

Note doesn't call destructors...

507 AllocL2CacheAligned
507 FreeCacheAligned
513 max
494 push_back
494 size

⟨*MemoryArena Interface*⟩+≡
```
void FreeAll() {
    for (u_int i = 0; i < toDelete.size(); ++i)
        FreeCacheAligned(toDelete[i]);
    toDelete.erase(toDelete.begin(), toDelete.end());
    nAvailable = 0;
}
```

⟨*MemoryArena Interface*⟩+≡
```
T *Alloc() {
    if (nAvailable == 0) {
        int nAlloc = max((unsigned int)16, 65536/sizeof(T));
        mem = (T *)AllocL2CacheAligned(nAlloc * sizeof(T));
        nAvailable = nAlloc;
        toDelete.push_back(mem);
    }
    --nAvailable;
    return mem++;
}
```

So can do `MemoryArena<Foo> arena; new (arena) Foo;`

⟨*MemoryArena Interface*⟩+≡
```
operator T *() {
    return Alloc();
}
```

Cache-Aligned Memory Allocation

We can reduce the number of cache misses incurred by `lrt` and slightly improve its overall performance by making sure that some memory allocations are well

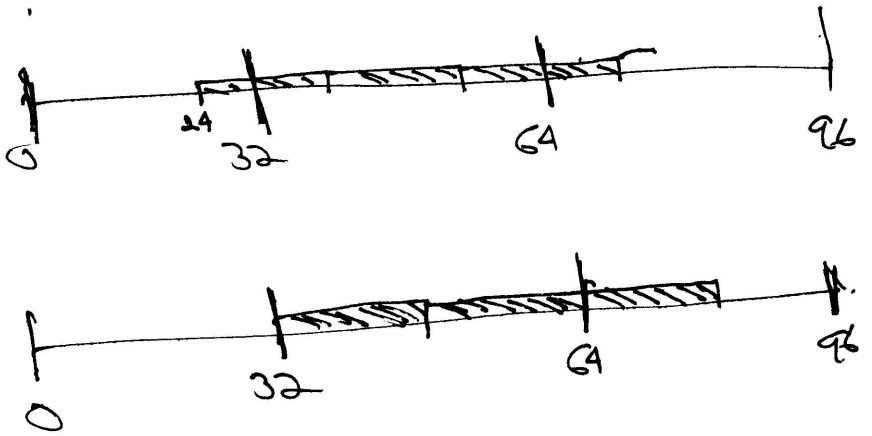


Figure A.1: Cache-aligned memory allocation ensures that the address returned is aligned with the start of a cache line. This figure shows the layout of three 16 byte objects in memory on a system with 32 byte cache lines. On the top, the starting address is not cache aligned–the first and last of the three objects span two cache lines, such that we may incur two cache misses when accessing their elements. On the bottom, the memory is cache aligned, guaranteeing that a maximum of one cache miss will be incurred per object.

size 494

aligned with the blocks of memory that the CPU cache manages. Figure A.1 shows the basic setting. There, we are allocating three 16 byte objects on a system with 32 byte large cache entries.

By making sure that the first object starts at the start of a cache entry (bottom), we ensure that we will incur no more than one cache miss when accessing any one of the items. If we expect to be accessing only some of the items (as opposed to looping over all of them in order), then performance will generally be improved with cache-aligned allocation. (`lrt`'s overall performance speed up by approximately 3% when allocation for the kd-tree accelerator in Section 4.4 was switched to use aligned allocation.)

The `AllocCacheAligned()` and `FreeCacheAligned()` functions provide a wrapper around system memory allocation and freeing routines to do cache-aligned allocation. If the pre-processor constant `L1_CACHE_LINE_SIZE` hasn't been set previously, we guess a cache line size of 32 bytes, which is typical of many architectures today.

XXX actually it's 64 bytes on Pentium 4...

⟨*Memory Allocation Functions*⟩≡

```
void *AllocL1CacheAligned(size_t size) {
#ifndef L1_CACHE_LINE_SIZE
#define L1_CACHE_LINE_SIZE 32
#endif
    return memalign(L1_CACHE_LINE_SIZE, size);
}
```

⟨*Memory Allocation Functions*⟩+≡
```
void *AllocL2CacheAligned(size_t size) {
#ifndef L2_CACHE_LINE_SIZE
#define L2_CACHE_LINE_SIZE 256
#endif
    return memalign(L2_CACHE_LINE_SIZE, size);
}
```

⟨*Memory Allocation Functions*⟩+≡
```
void FreeCacheAligned(void *ptr) {
    free(ptr);
}
```

Grunwald et al were one of the first groups of researchers to investigate the inter-play between memory allocation algorithms and the cache behavior of applications (GZH93b).

Until recently, most work on cache-efficient programming techniques has been focused on optimizing easily-predictable memory reference patterns, for example array accesses in loops, where techniques like blocking can be applied.

Pointer-based data structures are now seeing more attention, however.

Main goal: improve memory reference locality–spatial and temporal.

494 size

Can reorder the data structures, so that the order that the program accesses data values maps to underlying memory access patterns that have good locality.

Or can reorder the computation, so that the program accesses

Associativity: number of different cache locations a given memory address can be stored. *direct mapped* means just one.

Lam et al investigated blocking (tiling) for improving cache performance and developed techniques for selecting appropriate block sizes, given the size of the arrays and the cache size (LRW91).

Reduce memory use: (`union`s, bit-fields, etc) gives better locality, less pressure on the cache (and so fewer capacity misses.)

Truong et al suggest grouping frequently-accessed fields of structures at the start of the structure (TBS98) to improve locality.

Prefetching

Conflict, capacity, compulsory misses

In `lrt`, we only worry about cache layout issues for dynamically-allocated stuff. However, Calder et al show a profile-driven system that optimizes memory layout of global variables, constant values, data on the stack, and dynamically-allocated data from the heap in order to reduce cache conflicts among them all (CCJA98), giving an average 30% reduction in data cache misses for the applications they studied.

Blocking for tree data structures–keep node and a few levels of children contiguous (CHL99). Among other applications, they applied their tool to the layout of the acceleration octree in the *radiance* renderer and reported a 42% speedup in runtime.

More on structures: possibly split into “hot” and “cold” parts, allocated separately, to improve hits on hot parts. Also more on reordering fields inside structure to improve locality (CDL99).

Reference-Counted Objects

In languages like C++, where the language doesn't provide automatic memory management and the user must deallocate dynamically allocated memory when through with it, it can be tricky to deal with the case when multiple objects hold a pointer to some other object. We want to free the second object as soon as no other object holds a pointer to it, but no sooner, so that we avoid both memory leaks as well as subtle errors due to memory corruption.

As long as there aren't circular references (e.g. object A holds a reference to object B, which holds a reference to object A.), an easy solution to this is to use *reference counting*. An integer count is associated with objects that may be held by multiple objects; it is incremented when another object stores a reference to it and decremented when a reference goes away (e.g. due to the holding object being destroyed.)

We will define two classes to make it easy to use reference counted objects in `lrt`. First is a template, `ReferenceCounted`. An object of type `Foo` should inherit from `ReferenceCounted<Foo>` if it is to be managed via reference counting. This adds an `nReferences` field to it. The actual count will be managed by the `Reference` class, defined below.

⟨*Global Classes*⟩+≡

```
template <class T> class ReferenceCounted {
public:
    ReferenceCounted() { nReferences = 0; }
    int nReferences;
private:
    ReferenceCounted(const ReferenceCounted &);
    ReferenceCounted &operator=(const ReferenceCounted &);
};
```

Rather than holding a pointer to a reference counted object `Foo`, other objects should declare a `Reference<Foo>` to hold the reference. The `Reference` template class handles updating the reference count as appropriate. For example, consider the function below:

```
void func() {
    Reference<Foo> r1 = new Foo;
    Reference<Foo> r2 = r1;
    r1 = new Foo;
    r2 = r1;
}
```

In the first line, a `Foo` object is allocated; `r1` holds a reference to it, and the object's `nReferences` count should be one. A second reference to the object is made in the second line; `r1` and `r2` refer to the same `Foo` object, with a reference count of two. Next, a new `Foo] object is allocated. When a reference to it is assigned to [[r1`, the reference count of the original object is decremented to one. Now `r1` and `r2` point to separate objects. Finally, in the last line, `r2` is assigned to refer to the newly-allocated `Foo` object. The original `Foo` object now has zero references, and is automatically deleted. Now, at the end of the function, when both `r1` and `r2` go out of scope, the reference count for the second `Foo` object goes to zero, causing it to be freed as well.

The only trick to all this is the low-level C++ syntax that makes all this happen automatically, so that other code can treat `References` as much like pointers as possible. (For example, if the `Foo` class has a `bar()` method, we'd like to be able to write code like `r1->bar()` in the function above, etc.)

⟨*Global Classes*⟩+≡
```
template <class T> class Reference {
public:
    ⟨Reference constructors⟩
    ⟨Reference assignment operators⟩
    ⟨Reference destructor⟩
    ⟨Reference operators⟩
private:
    T *ptr;
};
```

The constructors are straightforward; after dealing with the possibility of `NULL` pointers, they just need to increment the reference count.

⟨*Reference constructors*⟩≡
```
Reference(T *p = NULL) {
    ptr = p;
    if (ptr) ++ptr->nReferences;
}
```

⟨*Reference constructors*⟩+≡
```
Reference(const Reference<T> &r) {
    ptr = r.ptr;
    if (ptr) ++ptr->nReferences;
}
```

When we have a reference that is being assigned to hold a different reference, we mostly just need to decrement our old reference count and increment the count of the new object. The increments and decrements are ordered carefully below, so that code like `r1 = r1;` doesn't inadvertently delete the object `r1` is refering to if it only has one reference.

⟨*Reference assignment operators*⟩≡
```
Reference &operator=(const Reference<T> &r) {
    if (r.ptr) r.ptr->nReferences++;
    if (ptr && --ptr->nReferences == 0) delete ptr;
    ptr = r.ptr;
    return *this;
}
```

⟨*Reference assignment operators*⟩+≡
```
Reference &operator=(T *p) {
    if (p) p->nReferences++;
    if (ptr && --ptr->nReferences == 0) delete ptr;
    ptr = p;
    return *this;
}
```

⟨*Reference destructor*⟩≡
```
~Reference() {
    if (ptr && --ptr->nReferences == 0)
        delete ptr;
}
```

Finally, a bit of C++ trickery so that we can use `->` to call methods of objects we hold references to, etc. The `operator bool` allows us to check to see if a reference points to a `NULL` object with code like [[if (!r) ...].

⟨*Reference operators*⟩≡
```
T *operator->() { return ptr; }
const T *operator->() const { return ptr; }
operator bool() const { return ptr != NULL; }
```

A.5 Matrix Routines

2x2 Linear Systems

Solve $Ax = B$....

Reference 509
ReferenceCounted 508

⟨*Matrix Method Definitions*⟩≡
```
bool SolveLinearSystem2x2(const Float A[2][2], const Float B[2],
        Float x[2]) {
    Float det = A[0][0]*A[1][1] - A[0][1]*A[1][0];
    if (fabsf(det) < 1e-5)
        return false;
    Float invDet = 1.0f/det;
    x[0] = (A[1][1]*B[0] - A[0][1]*B[1]) * invDet;
    x[1] = (A[0][0]*B[1] - A[1][0]*B[0]) * invDet;
    return true;
}
```

4x4 Matrices

The `Matrix4x4` structure provides a low-level representation of 4 by 4 matrices. It is an integral part of the `Transform` class, which holds two matrices, one representing a transform and the other representing its inverse. Because we will often have multiple objects holding identical transformations, we will reference count `Matrix4x4`s, so that the `Transform` class only needs to hold `Matrix4x4` references, rather than holding the much larger complete matrices.

⟨*Global Classes*⟩+≡
```
struct Matrix4x4 : public ReferenceCounted<Matrix4x4> {
    ⟨Matrix4x4 Methods⟩
    Float m[4][4];
};
```

The default constructor sets the matrix to the identity matrix.

⟨*Matrix4x4 Methods*⟩≡
```
Matrix4x4() {
    for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 4; ++j)
            if (i == j) m[i][j] = 1.;
            else m[i][j] = 0.;
}
```

We also provide constructors that allow the user to pass an array of floats, or sixteen individual floats to initialize the `Matrix4x4` with.

⟨*Matrix4x4 Methods*⟩+≡
```
Matrix4x4(Float mat[4][4]) {
    memcpy(m, mat, 16*sizeof(Float));
}
```

⟨*Matrix4x4 Methods*⟩+≡
```
Matrix4x4::Matrix4x4(Float t00, Float t01, Float t02, Float t03,
                     Float t10, Float t11, Float t12, Float t13,
                     Float t20, Float t21, Float t22, Float t23,
                     Float t30, Float t31, Float t32, Float t33) {
    m[0][0] = t00; m[0][1] = t01; m[0][2] = t02; m[0][3] = t03;
    m[1][0] = t10; m[1][1] = t11; m[1][2] = t12; m[1][3] = t13;
    m[2][0] = t20; m[2][1] = t21; m[2][2] = t22; m[2][3] = t23;
    m[3][0] = t30; m[3][1] = t31; m[3][2] = t32; m[3][3] = t33;
}
```

510 Matrix4x4
509 Reference

We support a few low-level matrix operations, each of which returns a reference to a newly allocated matrix that holds the result of the operation. For starters, `Transpose()` transposes the matrix's elements.

⟨*Matrix Method Definitions*⟩+≡
```
Reference<Matrix4x4> Matrix4x4::Transpose() const {
    return new Matrix4x4(m[0][0], m[1][0], m[2][0], m[3][0],
        m[0][1], m[1][1], m[2][1], m[3][1],
        m[0][2], m[1][2], m[2][2], m[3][2],
        m[0][3], m[1][3], m[2][3], m[3][3]);
}
```

Matrix-matrix multiplication of two matrices M_1 and M_2 is computed by setting the (i, j)th element of the resulting matrix to the sum of the products of the elements of the ith row of M_1 with the jth column of M_2.

⟨*Matrix4x4 Methods*⟩+≡
```
static Reference<Matrix4x4> Mul(const Reference<Matrix4x4> &m1,
        const Reference<Matrix4x4> &m2) {
    Float r[4][4];
    for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 4; ++j)
            r[i][j] = m1->m[i][0] * m2->m[0][j] +
                      m1->m[i][1] * m2->m[1][j] +
                      m1->m[i][2] * m2->m[2][j] +
                      m1->m[i][3] * m2->m[3][j];
    return new Matrix4x4(r);
}
```

Finally, `Inverse()` returnse the inverse of the matrix. Our implementation, uses a numerically stable Gauss–Jordan elimination routine to compute the inverse.

⟨*Matrix4x4 Methods*⟩+≡
```
Reference<Matrix4x4> Inverse() const;
```

A.6 Mathematical Utility Functions

Matrix4x4 510
Reference 509

Now we'll define a few very short functions that will be useful throughout the program. First is `Lerp`. It performs linear interpolation between two values, `start` and `end`, with position given by the `pos` parameter. When `pos` is zero, the result is `start`; when `pos` is one, the result is `end`, etc.

`Lerp` is written as

$$\mathrm{lerp}(t, v_1, v_2) = (1-t)v_1 + tv_2$$

in the function below, rather than in the more terse and potentially more efficient form of

$$v_1 + t(v_2 - v_1)$$

in the interests of reducing floating-point error. Not only is less floating point precision lost, but `Lerp` returns *exactly* the values `start` and `end` when `pos` has values 0 and 1, respectively, with our implementation. This isn't necessarily the case with the other formulation, again due to floating point roundoff.

⟨*Global Inline Functions*⟩+≡
```
inline Float Lerp(Float pos, Float start, Float end) {
    return (1.f - pos) * start + pos * end;
}
```

`Clamp` clamps a value `val` to be between the values `low` and `high`. If `val` is out of that range, `low` or `high` is returned as appropriate.

⟨*Global Inline Functions*⟩+≡
```
inline Float Clamp(Float val, Float low, Float high) {
    if (val < low) return low;
    else if (val > high) return high;
    else return val;
}
```

⟨*Global Inline Functions*⟩+≡
```
inline int Clamp(int val, int low, int high) {
    if (val < low) return low;
    else if (val > high) return high;
    else return val;
}
```

Another useful function is `SmoothStep`; it takes a minimum and maximum value and a point at which to evaluate the step function. If the point is below the minimum, zero is returned, and if it's above the maximum, one is returned. Otherwise it smoothly interpolates between zero and one.

⟨*Global Inline Functions*⟩+≡
```
inline Float SmoothStep(Float min, Float max, Float value) {
    Float v = Clamp((value - min) / (max - min), 0., 1.);
    return -2.f * v * v * v + 3.f * v * v;
}
```

`Mod` computes the remainder of a/b. This function is handy since it behaves predictably and reasonably with negative numbers—the C and C++ standards leave the behavior of the `%` operator undefined in that case.

⟨*Global Inline Functions*⟩+≡
```
inline int Mod(int a, int b) {
    int n = int(a/b);
    a -= n*b;
    if (a < 0)
        a += b;
    return a;
}
```

Finally, simple functions that compute the minimum or maximum of two values and a function that swaps the values of two variables. We just use the appropriate functions provided by the standard C++ library.

⟨*Global Include Files*⟩+≡
```
#include <algorithm>
using std::min;
using std::max;
using std::swap;
using std::sort;
```

Unfortunately, not all system `math.h` files store the value of π in `M_PI`. If it is not defined, we do it ourself.

⟨*Global Constants*⟩+≡
```
#ifndef M_PI
#define M_PI           3.14159265358979323846f
#endif
```

One over 255 and one over π.

⟨*Global Constants*⟩+≡
```
#define INV_255 .00392156862745098039f
#define INV_PI  0.31830988618379067154f
```

We define a generally-useful `INFINITY` value using `FLT_MAX` from the standard math library, which is the largest representable floating point number.

⟨*Global Constants*⟩+≡
```
#ifndef INFINITY
#define INFINITY FLT_MAX
#endif
```

Two simple functions convert from angles expressed in degrees to radians, and vice versa.

⟨*Global Inline Functions*⟩+≡
```
inline Float Radians(Float deg) { return ((Float)M_PI/180.f) * deg; }
inline Float Degrees(Float rad) { return (180.f/(Float)M_PI) * rad; }
```

Floating-point to integer conversion

On the x86 architecture, it can take as many as 80 processor cycles to convert a floating-point value to an integer value; the conversion to integer in a simple sequence of code like:

```
Float a = ..., b = ...;
int i = (int)(a * b);
```

may take 80 times longer than the multiplication `a*b`! The root problem is that the floating-point unit's rounding mode needs to be changed from the default before the built-in conversion instruction is used, and this requires an expensive flush of the entire floating-point pipeline.

`lrt` needs to convert `Float`s to integers in a number of performance-sensitive areas. These include the sample filtering code, where for every camera sample we need to compute the extent of pixels that are affected by the sample based on the filter extent. Similarly, in the Perlin noise evaluation routines, we need to find the integer lattice cell that a floating-point position is in.

Sree Kotay and Mike Herf have developed some techniques to these conversions much more quickly without needing to change the rounding mode by taking advantage of low-level knowledge of the layout of IEEE floating-point values in memory. Using these routines in `lrt` speed it up by up to 5% for some scenes. We will not include the details of their implementation here. However, there are four key functions, all of them taking one `Float` value and returning an integer:

1. `Float2Int(f)`: This is the same as the basic cast `(int)f`.
2. `Round2Int(f)`: This rounds the floating point value `f` to the nearest integer, returning the result as an `int`.
3. `Floor2Int(f)`: The first integer value less than or equal to `f` is returned.
4. `Ceil2Int(f)`: And similarly, the first integer value greater than or equal to `f` is returned.

⟨*Global Inline Functions*⟩+≡
```
inline int Log2Int(Float v) {
    return ((*(int *) &v) >> 23) - 127;
}
```

A.7 Random Numbers

We will provide a pseudo-random number generation function for the system. This is useful because it allows us to ensure that the system produces the same results regardless of machine architecture and C library implementation. This is particularly helpful since many systems provide random number generation routines with poor statistical distributions.

The random number generate we choose is the "Mersenne Twister" by Makoto Matsumoto and Takuji Nishimura. The code to the random number generator is very involved and complex, and we will not present it here. Nevertheless, it is one of the best random number generators known, can be implemented very efficiently, and has a period of $2^{19937} - 1$ before it repeats the series again. Pointers to details on the algorithm can be found at the end of this section.

The algorithm provides three main functions, `genrand_real1`, which generates uniform random numbers over the $[0,1]$ interval, `genrand_real2`, which generates uniform random numbers over $[0,1)$, and `genrand_int32`, which generates uniform random positive integer values from 0 to $2^{32} - 1$.

⟨*Global Inline Functions*⟩+≡
```
inline Float RandomFloat(Float min = 0.f, Float max = 1.f) {
    return Lerp(genrand_real1(), min, max);
}

inline unsigned long RandomInt() {
    return genrand_int32();
}
```

512 Lerp
513 max
513 min
529 pair

A.8 Hash Tables

For certain operations it will be useful to have an efficient mapping from keys to data. We implement a simple hashtable that keys from strings to `void *`s.

⟨*Global Classes*⟩+≡
```
class StringHashTable {
public:
    ⟨StringHashTable Methods⟩
private:
    ⟨StringHashTable Private Data⟩
};
```

⟨*StringHashTable Private Data*⟩≡
```
static const int NUM_BUCKETS = 1047;
typedef vector<pair<string, void *> > ItemType;
ItemType buckets[NUM_BUCKETS];
```

⟨*StringHashTable Method Definitions*⟩≡
```
u_int StringHashTable::Hash(const string &str) const {
    u_int hashValue = 0;
    for (u_int i = 0; i < str.size(); ++i) {
        hashValue <<= 1;
        hashValue ^= str[i];
    }
    return hashValue % NUM_BUCKETS;
}
```

⟨*StringHashTable Method Definitions*⟩+≡
```
void *StringHashTable::Search(const string &key) const {
    u_int index = Hash(key);
    for (u_int i = 0; i < buckets[index].size(); ++i)
        if (key == buckets[index][i].first)
            return buckets[index][i].second;
    return NULL;
}
```

⟨*StringHashTable Method Definitions*⟩+≡
```
void StringHashTable::Add(const string &key, void *data) {
    u_int index = Hash(key);
    for (u_int i = 0; i < buckets[index].size(); ++i) {
        if (key == buckets[index][i].first) {
            buckets[index][i].second = data;
            return;
        }
    }
    buckets[index].push_back(make_pair(key, data));
}
```

buckets 515
make_pair 529
NUM_BUCKETS 515
push_back 494
size 494

A.9 Octrees

⟨*octree.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef OCTREE_H
#define OCTREE_H
#include "lrt.h"
#include "geometry.h"
⟨Octree Declarations⟩
⟨Octree Method Definitions⟩
#endif // OCTREE_H
```

The octree is a three-dimensional data structure that recursively splits a region of space into axis-aligned boxes. Starting with a single box at the top level, each level of refinement splits the previous level's boxes into eight child boxes, each covering one-eigth of the volume of the previous ones–Figure A.2 shows the basic idea.

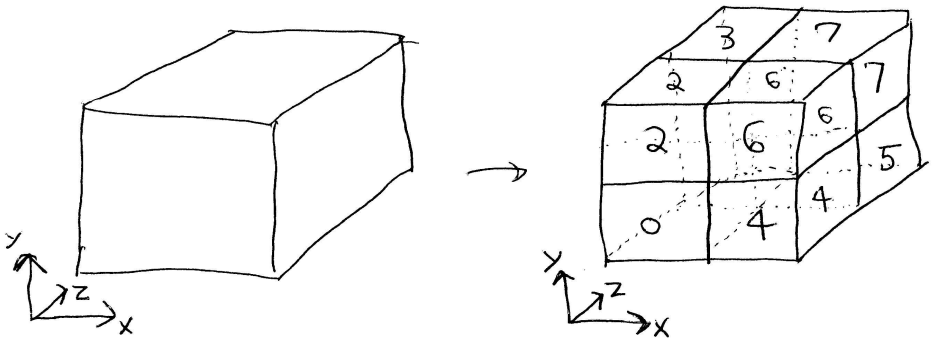


Figure A.2: Basic octree refinement: starting with an axis-aligned bounding box, the octree is defined by progressively splitting each node into eight equal-sized child nodes. The order in which the child nodes are assigned numbers $0 \ldots 7$ is significant–details of this will be explained later in this section. Different sub-trees may be refined to different depths, giving an adaptive discretization of 3D space.

The octree class in this section helps accelerate the query "given a collection of objects and their axis-aligned bounding boxes, which of their bounds overlap a given point". For large numbers of objects, using an octree to answer this question can be substantially faster than looping over all of the objects directly. `lrt` currently only uses octrees to store the irradiance estimates computed by the `IrradianceCache` integrator–each estimate has a bounding box associated with it that gives the maximum region of space where the estimate may be used for shading computations. However, here we are providing an independent octree implementation in order to simplify the description of the `IrradianceCache` as well as to make it easier to re-use the octree class for other applications.

First, we will define the `OctNode` structure, which represents a node of the tree. It holds pointers to the eight possible children of the node (some or all of which may be `NULL`) and a `vector` of `NodeData` objects. `NodeData` is the object type that the user of the octree wants to store in the tree; for the `IrradianceCache`, it's the `IrradSample` structure, which records the results from a single irradiance estimate. The constructor and destructor of the `OctNode` just initialize the `children` to `NULL` and delete them, respectively; their implementations won't be shown here.

⟨*Octree Declarations*⟩≡
```
template <class NodeData> struct OctNode {
    ⟨OctNode Method Declarations⟩
    OctNode *children[8];
    vector<NodeData> data;
};
```

Next is the main declaration of the `Octree` class. It is parameterized by the `NodeData` class as well as by a "lookup procedure", `LookupProc`, which is essentially a callback function that lets the `Octree` communicate back to the caller which elements of `NodeData` are overlapping a given lookup position.

⟨*Octree Declarations*⟩+≡
```
template <class NodeData, class LookupProc> class Octree {
public:
    ⟨Octree Method Declarations⟩
private:
    ⟨Octree Private Data⟩
};
```

The constructor just stores the overall bound of the tree. Items are added individually with the `add()` method, below.

⟨*Octree Method Declarations*⟩≡
```
Octree(const BBox &b)
    : bound(b) {
}
```

⟨*Octree Private Data*⟩≡
```
BBox bound;
OctNode<NodeData> root;
```

To add a node to the tree, we recursively walk down the tree, creating new nodes as needed, until termination criteria are met. We then add the given item to the appropriate nodes that it overlaps. Similar to the `KdTreeAccelerator` of Section 4.4, performance is substantially affected depending on what the specific termination criteria are. For example, we could trivially decide to never refine the tree and add all items to the root node. This would be a valid octree, though it would perform poorly for large numbers of objects. However, if we refine the tree too much, items may span large numbers of nodes, causing excessive memory use.

DistanceSquared 23
OctNode 517
pMax 28
pMin 28

The entrypoint for adding an item directly calls an internal "add item" method with a few additional parameters, including the current node being considered, the bounding box of the node, and the squared length of the diagonal of the data item's bounding box. This internal method will itself be called as we recursively work down the octree.

⟨*Octree Method Declarations*⟩+≡
```
void Add(const NodeData &dataItem, const BBox &dataBound) {
    add(&root, bound, dataItem, dataBound,
        DistanceSquared(dataBound.pMin, dataBound.pMax));
}
```

Here is the internal "add item" method. It either adds the item to the current node or determines which child nodes the item overlaps, allocates them if necessary, and recursively calls `add()` to allow the children to decide whether to stop the recursion and add the item or to continue down the tree.

⟨*Octree Method Definitions*⟩≡
```
template <class NodeData, class LookupProc>
void Octree<NodeData, LookupProc>::add(OctNode<NodeData> *node,
        const BBox &nodeBound, const NodeData &dataItem,
        const BBox &dataBound, Float diag2) {
    ⟨Possibly add data item to current octree node⟩
    ⟨Otherwise add data item to octree children⟩
}
```

We stop going down the tree and add the item to the current node once the length of the diagonal of the node is less than the length of the diagonal of the item's bounds. This ensures that the item overlaps a relatively small number of tree nodes, while not being too small relative to the extent of the nodes that it's added to. Figure A.3 shows the basic operation of the algorithm in two dimensions (where the corresponding data structure is known as an *quadtree*).

⟨*Possibly add data item to current octree node*⟩≡
```
if (DistanceSquared(nodeBound.pMin, nodeBound.pMax) < diag2) {
    node->data.push_back(dataItem);
    return;
}
```

If we decide to continue down the tree, we need to determine which of the child nodes the item's bounding box overlaps. Rather than computing the bounds of each child and doing a bounding box overlap test, we can save work by taking advantage of symmetries, such as the fact that if the x range of the object's bounding box is entirely on the left side of the plane that splits the tree node in the x direction, there is no way that it overlaps any of the four child nodes on the right side. Careful selection of the child node numbering scheme in Figure A.2 is key to the success of this approach.

We start by computing `pMid`, the position of the center of the current node. The fragment ⟨*Determine which children the item overlaps*⟩ then efficiently sets an array of boolean vlaues, `over`, such that the ith element is true only if the bounds of the data item being added overlap the ith child of the current node. We can then loop over the eight children and recursively call `add()` for the ones that the object overlaps.

23 DistanceSquared
517 OctNode
518 Octree
28 pMax
28 pMin
21 Point
494 push_back

⟨*Otherwise add data item to octree children*⟩≡
```
Point pMid = .5 * nodeBound.pMin + .5 * nodeBound.pMax;
⟨Determine which children the item overlaps⟩
for (int child = 0; child < 8; ++child) {
    if (!over[child]) continue;
    ⟨Hand data item down to child number child⟩
}
```

Now the child node numbering scheme comes in. The child nodes are numbered such that the low bit of a child's number is zero if its z component is on the low side of the z splitting plane and one if it is on the high side. Similarly, the second bit is set based on which side the child is of the y plane, and the third bit is set based on it's position with respect to the x plane.

Thus, given boolean variables that classify a child node with respect to the splitting planes (`true` if it is above the plane, the child number of a given node is equal to:

```
4 * (xHigh ? 1 : 0) + 2 * (yHigh ? 1 : 0) + 1 * (zHigh
? 1 : 0)
```

We can quickly determine which child nodes a given bounding box overlaps by classifying its extent with respect to the center point of the node. For example, if the bounding box's starting x value is less than the midpoint, then the node potentially

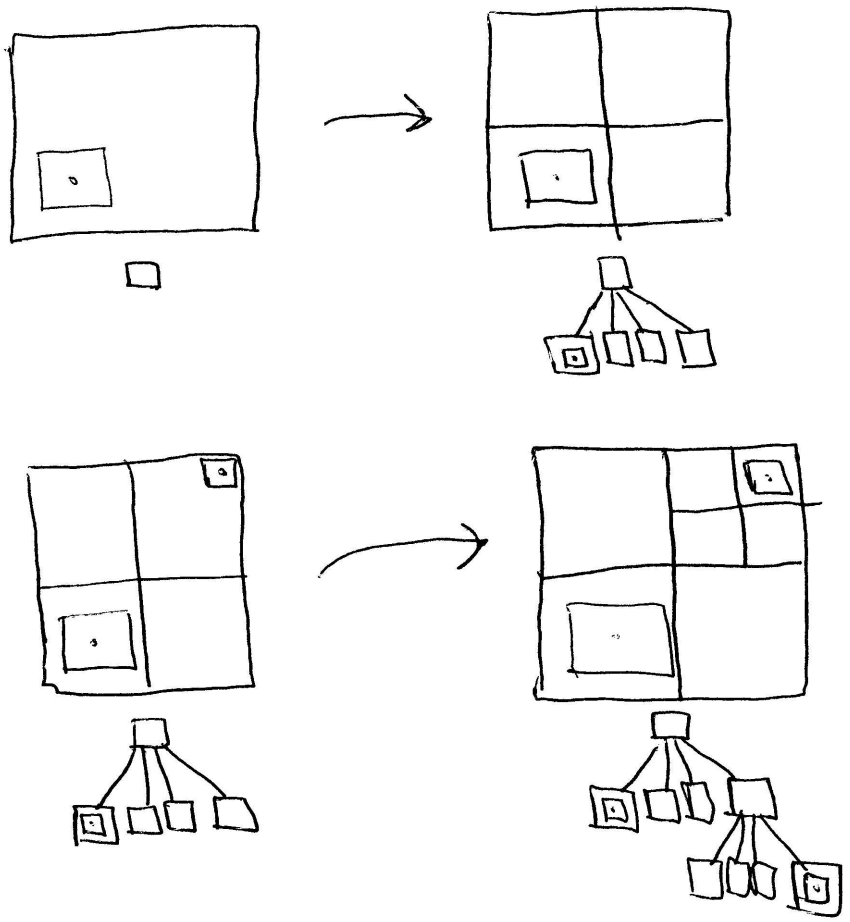


Figure A.3: Creation of a quadtree (the 2D analog of an octree). In the top row, we are starting with a tree comprised of just the root node and are adding an object with bounds around a given point. We refine the tree one level, and add the object to the single child node that it overlaps (shown schematically underneath the tree.) In the bottom row, we are adding another new object with a smaller bounding box than the first. We go down two levels of the tree before adding the item, again to the single node that it overlaps. In general, items may be stored in multiple nodes of the tree.

overlaps children numbers 0, 1, 2, and 3. If its ending x value is greater than the midpoint, it potentially overlaps 4, 5, 6, and 7. We check the y and z dimensions in turn, computing the logical and of the results: the item only overlaps a child node if it overlaps its extent in all three dimensions.

⟨*Determine which children the item overlaps*⟩≡
```
bool over[8];
over[0] = over[1] = over[2] = over[3] =  (dataBound.pMin.x <= pMid.x);
over[4] = over[5] = over[6] = over[7] =  (dataBound.pMax.x  > pMid.x);
over[0] = over[1] = over[4] = over[5] &= (dataBound.pMin.y <= pMid.y);
over[2] = over[3] = over[6] = over[7] &= (dataBound.pMax.y  > pMid.y);
over[0] = over[2] = over[4] = over[6] &= (dataBound.pMin.z <= pMid.z);
over[1] = over[3] = over[5] = over[7] &= (dataBound.pMax.z  > pMid.z);
```

And now for the overlapping children, we continue down the tree. Rather than using memory to store the bounding box of each node in tree, we compute node bounds incrementally from the parent bounds.

⟨*Hand data item down to child number* `child`⟩≡
```
if (!node->children[child])
    node->children[child] = new OctNode<NodeData>;
⟨Compute childBound for child⟩
add(node->children[child], childBound, dataItem, dataBound, diag2);
```

Here again we take advantage of the child node numbering scheme to quickly determine which values give the bounding box of the node.

⟨*Compute* `childBound` *for* `child`⟩≡
```
BBox childBound;
childBound.pMin.x = (child & 4) ? pMid.x : nodeBound.pMin.x;
childBound.pMax.x = (child & 4) ? nodeBound.pMax.x : pMid.x;
childBound.pMin.y = (child & 2) ? pMid.y : nodeBound.pMin.y;
childBound.pMax.y = (child & 2) ? nodeBound.pMax.y : pMid.y;
childBound.pMin.z = (child & 1) ? pMid.z : nodeBound.pMin.z;
childBound.pMax.z = (child & 1) ? nodeBound.pMax.z : pMid.z;
```

After items have been added to the tree, the user can use the tree to look up the items that have bounds that overlap a given point `P`. The `Lookup()` function walks down the tree, processing the nodes that the given point overlaps. The user-supplied callback, `process` is called for each `NodeData` item that overlaps the given point.

As with the `Add()` function, the main lookup function directly calls to an internal version that takes a pointer to the current node and the current node's bounds.

⟨*Octree Method Declarations*⟩+≡
```
void Lookup(const Point &P, const LookupProc &process) {
    if (!bound.Inside(P)) return;
    lookup(&root, bound, P, process);
}
```

If the internal lookup function has been called with a given node, the point `P` must be inside the node. We start by calling the user-supplied callback for each

30 Inside
32 Lookup
517 OctNode
28 pMax
28 pMin
21 Point

NodeData item that is stored in the node, allowing the user to do whatever processing is appropriate. We then continue down the tree into the single child node that P is inside until we hit the bottom.

⟨*Octree Method Definitions*⟩+≡
```
template <class NodeData, class LookupProc>
void Octree<NodeData, LookupProc>::lookup(OctNode<NodeData> *node,
        const BBox &nodeBound, const Point &P,
        const LookupProc &process) {
    for (u_int i = 0; i < node->data.size(); ++i)
        process(P, node->data[i]);
    ⟨Determine which octree child node P is inside⟩
    if (node->children[child]) {
        ⟨Compute childBound for child⟩
        lookup(node->children[child], childBound, P, process);
    }
}
```

Again using the child numbering, we can quickly determine which child a point overlaps by classifying it with respect to the center of the parent node in each direcion.

⟨*Determine which octree child node* P *is inside*⟩≡
```
Point pMid = .5 * nodeBound.pMin + .5 * nodeBound.pMax;
int child = (P.x > pMid.x ? 4 : 0) +
    (P.y > pMid.y ? 2 : 0) + (P.z > pMid.z ? 1 : 0);
```


OctNode 517
Octree 518
pMax 28
pMin 28
Point 21
size 494

A.10 Kd Trees

⟨*kdtree.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef KDTREE_H
#define KDTREE_H
#include "lrt.h"
#include "geometry.h"
⟨Kd Tree Declarations⟩
⟨Kd Tree Method Definitions⟩
#endif // KDTREE_H
```

The kd tree is another data structure that accelerates the processing of spatial data. In contrast to the octree, where the data items had a known bounding box and the caller wanted to find all items that overlap a given point, the kd tree is useful for handling data items that are just single points in space, with no associated bound, but where the caller wants to find all such points within a user-supplied distance of a given point.

The KdTree that will be described here is generally similar to the KdTreeAccelerator of Section 4.4 in that 3D space is progressively split in half by planes. There are two main differences, however:

- Here, each tree node stores a single data item. As such, there is exactly one kd tree node for each data item stored in the tree.

- Because each item being stored is just a single point, here we don't have to worry about items that straddle the splitting plane.

One result of these differences is that we can build a perfectly balanced tree, which can improve the efficiency of data lookups.

First we'll declare an enumerant to record which axis each tree node splits along and the basic `KdNode` structure, which holds the user-supplied data as well as information about the topological structure of the tree.

⟨*Kd Tree Declarations*⟩≡
```
enum SplitAxis { SPLIT_X, SPLIT_Y, SPLIT_Z };
```

⟨*Kd Tree Declarations*⟩+≡
```
template <class NodeData> struct KdNode {
    KdNode(const NodeData &d, SplitAxis a)
        : data(d) {
        children[0] = children[1] = NULL;
        split = a;
    }
    ~KdNode() { delete children[0]; delete children[1]; }
    ⟨KdNode Data⟩
};
```

494 size

⟨*KdNode Data*⟩≡
```
NodeData data;
SplitAxis split;
KdNode *children[2];
```

The `KdTree` itself only needs to hold a pointer to the root node of the tree.

⟨*Kd Tree Declarations*⟩+≡
```
template <class NodeData, class LookupProc> class KdTree {
public:
    ⟨KdTree Method Declarations⟩
private:
    ⟨KdTree Private Data⟩
};
```

⟨*KdTree Private Data*⟩≡
```
KdNode<NodeData> *root;
```

All of the data items must be supplied to the `KdTree` constructor. We don't support incremental addition or removal of `NodeData` items since this functionality isn't needed in `lrt` and doing so keeps the implementation here straightforward.

⟨*Kd Tree Method Definitions*⟩≡
```
template <class NodeData, class LookupProc>
KdTree<NodeData, LookupProc>::KdTree(const vector<NodeData> &d) {
    vector<NodeData> data = d;
    recursiveBuild(&root, data, 0, int(data.size()));
}
```

Tree construction is handled by the `recursiveBuild()` method. It takes a pointer to a pointer to a `KdNode`, which allows us to fill in the `root` pointer in

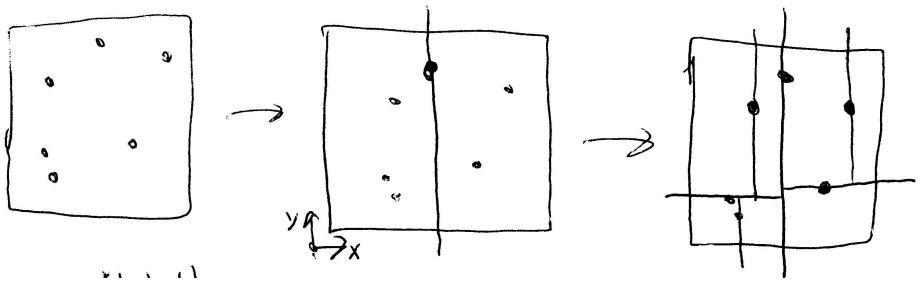


Figure A.4: Creation of a kd tree to store a set of points. Given a collection of points (left), we first choose a split direction. Here, we have decided to split in the x direction. We find the point in the middle along x and split along the plane that goes through the point. Roughly half of points are to the left of the splitting plane and half are to the right. We then continue recursively in each half, allocating new tree nodes, splitting and partitioning, until all data points have been processed.

the `KdTree` or the appropriate `children` pointer in the parent node. We also pass down the vector of `NodeData` items and offsets into the array indicating the subset of data items $[\text{start}, \text{end})$ that need to be processed.

`KdNode` 523
`KdTree` 523

The tree building process selects the "middle" element of the user-supplied data (to be explained precisely below) and partitions the data, so that all items below the middle are in the first half of the array and all items above the middle are in the second half. It constructs a node with the middle element as its data item and then recursively initializes the two children of the node by processing the first and second halves of the array (minus the middle element.) Figure A.4 shows the basic process of bulding the kd tree.

⟨*Kd Tree Method Definitions*⟩+≡
```
template <class NodeData, class LookupProc>
void
KdTree<NodeData, LookupProc>::recursiveBuild(KdNode<NodeData> **node,
        vector<NodeData> &data, int start, int end) {
    ⟨Create leaf node of kd tree if we've reached the bottom⟩
    ⟨Choose split direction and partition data⟩
    ⟨Allocate Kd tree node and continue recursively⟩
}
```

When there is zero or one item to be processed, then we've reached the bottom of the tree. We either `NULL` out the node pointer (for zero items), or allocate a leaf `KdNode` to hold the single item. In either case, we're done with the current sub-tree, so we immediately return.

⟨*Create leaf node of kd tree if we've reached the bottom*⟩≡
```
if (start >= end) {
    *node = NULL;
    return;
}
if (start + 1 == end) {
    *node = new KdNode<NodeData>(data[start], SPLIT_X);
    return;
}
```

Otherwise, we need to partition the data into two halves and allocate and initialize a non-leaf node. We decide to split along whichever coordinate axis the remaining data items have the largest extent. Then we call the standard library `nth_element()` function, which takes three pointers `start`, `mid`, and `end` into a sequence and partitions it such that the `mid`th element is in the position it would be in if the range was sorted and where all elements from `start` to `mid-1` are less than mid, and elements from `mid+1` to `end` are greater than mid. This can all be done more quickly than sorting the entire range–$O(n)$ time rather than $O(n \log n)$.

⟨*Choose split direction and partition data*⟩≡
```
⟨Compute bounds of data from start to end⟩
Vector diag = bound.pMax - bound.pMin;
SplitAxis split;
int splitPos = (start+end)/2;
if (diag.x > diag.y && diag.x > diag.z) {
    split = SPLIT_X;
    std::nth_element(&data[start], &data[splitPos], &data[end],
        CompareNodeX<NodeData>());
}
else if (diag.y > diag.z) {
    split = SPLIT_Y;
    std::nth_element(&data[start], &data[splitPos], &data[end],
        CompareNodeY<NodeData>());
}
else {
    split = SPLIT_Z;
    std::nth_element(&data[start], &data[splitPos], &data[end],
        CompareNodeZ<NodeData>());
}
```

526 CompareNodeX
523 KdNode
28 pMax
28 pMin
29 Union
16 Vector

⟨*Compute bounds of data from* `start` *to* `end`⟩≡
```
BBox bound;
for (int i = start; i < end; ++i)
    bound = Union(bound, data[i].P);
```

The `nth_element()` function needs a "comparison object" that determines the ordering between two data elements. There are three small structures that do comparisons in the *x*, *y*, and *z* directions, which are used as appropriate based on the split axis chosen previously.

⟨*Kd Tree Declarations*⟩+≡
```
template<class NodeData> struct CompareNodeX {
    bool operator()(const NodeData &d1,
            const NodeData &d2) const {
        return d1.P.x < d2.P.x;
    }
};
```

Once we've partitioned the data, we allocate a node to store the middle item and recursively initialize its child node pointers with the two sets of remaining items.

⟨*Allocate Kd tree node and continue recursively*⟩≡
```
*node = new KdNode<NodeData>(data[splitPos], split);
recursiveBuild(&((*node)->children[0]), data, start, splitPos);
recursiveBuild(&((*node)->children[1]), data, splitPos+1, end);
```

When the user wants to look up items from the tree, they provide a point `P`, a callback procedure (similar to the one used in the `Octree` above), and a maximum search radius. Rather than passing it by the value, the search radius is passed into the lookup function by reference. This will allow us to pass it to the callback procedure by reference, so that it can reduce the search radius as the search goes on. This can speed up lookups when we can determine partway along that a smaller search radius was appropriate.

KdNode 523
KdTree 523
Lookup 335
Point 21

As usual, we immediately call to an internal lookup procedure, passing in a pointer to the current node to be processed.

⟨*Kd Tree Method Definitions*⟩+≡
```
template <class NodeData, class LookupProc> void
KdTree<NodeData, LookupProc>::Lookup(const Point &P,
        const LookupProc &process, Float &maxDist) const {
    recursiveLookup(root, P, process, maxDist);
}
```

The lookup function has two responsibilities: it needs to recursively process the children of the current node, based on which of them the search region overlaps, and it needs to call the callback routine, passing it the data item in the current node if it is inside the search radius. Figure A.5 shows the basic process.

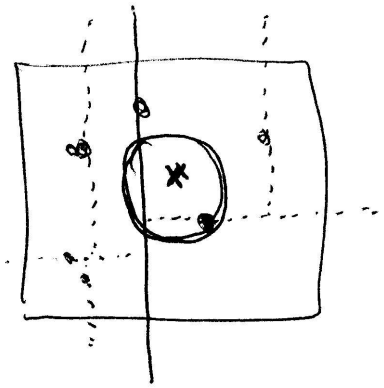


Figure A.5: Basic process of kd tree lookups. The point marked with an "x" is the lookup position, and the region of interest is denoted by the circular region around it. At the root node of the tree (indicated by a bold splitting line), the data item is outside of the region of interest, so it is not handed to the callback function. However, the region overlaps both children of the node, so we have to recursively consider each of them. We will consider the right child (child number one) first, however, in order to examine the nearby data items before examining the ones farther away.

523 KdNode
523 KdTree
21 Point

⟨*Kd Tree Method Definitions*⟩+≡

```
template <class NodeData, class LookupProc> void
KdTree<NodeData, LookupProc>::recursiveLookup(KdNode<NodeData> *node,
        const Point &P, const LookupProc &process,
        Float &maxDist) const {
    if (!node) return;
    if (node->split == SPLIT_X) {
        ⟨Process Kd node's children based on x split⟩
    }
    else if (node->split == SPLIT_Y) {
        ⟨Process Kd node's children based on y split⟩
    }
    else {
        ⟨Process Kd node's children based on z split⟩
    }
    ⟨Hand Kd tree node to processing function⟩
}
```

We will walk the tree in a depth-first manner, heading toward the leaf nodes that are close to the lookup point `P` before we call the callback method to process data items. This will ensure that we hand data points to the callback function in a generally near-to-far order. If the caller is only interested in finding a fixed number of nearby points, after which they will end the search, this is a more efficient order.

XXX also re-remind that `maxDist` may be decreased along the way... XXX

We first walk down the side of the tree that the current point lies on. Only after that lookup has returned do we then go down the other side, if the search radius

causes the lookup region to cover both sides of the tree. Below is the logic for the case of a split along the *x* axis; the code for *y* and *z* is similar and is elided.

⟨*Process Kd node's children based on x split*⟩≡
```
if (P.x <= node->data.P.x) {
    recursiveLookup(node->children[0], P, process, maxDist);
    if (P.x + maxDist >= node->data.P.x)
        recursiveLookup(node->children[1], P, process, maxDist);
}
else {
    recursiveLookup(node->children[1], P, process, maxDist);
    if (P.x - maxDist <= node->data.P.x)
        recursiveLookup(node->children[0], P, process, maxDist);
}
```

Finally now, at the end of the lookup function, we see if the point stored in the current node is inside the search radius. We save an expensive square root computation by comparing squared distances, and pass the data item back to the callback function if appropriate. In addition to doing whatever processing it needs to do based on the item, the callback function may decrease `maxDist` in order to reduce the region of space searched for the remainder of the processing.

DistanceSquared 23

⟨*Hand Kd tree node to processing function*⟩≡
```
if (DistanceSquared(node->data.P, P) < maxDist*maxDist)
    process(node->data, maxDist);
```

A.11 Main Include File

Here we'll also define the `lrt.h` file that all source files will `#include`. In has the usual structure of a header file: it will include some other headers, declare some functions, types, and constants, and define some inline functions. Throughout the rest of the chapters of this book, we will add to the contents of all of these fragments as we go along.

⟨*lrt.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef LRT_H
#define LRT_H
⟨Global Include Files⟩
⟨Platform-specific definitions⟩
⟨Global Type Declarations⟩
⟨Global Forward Declarations⟩
⟨Global Constants⟩
⟨Global Function Declarations⟩
⟨Global Classes⟩
⟨Global Inline Functions⟩
#endif // LRT_H
```

All files that include `lrt.h` get a number of other include files in the process; this makes it possible for them to just include `lrt.h` and not repeatedly include

the others. We try to keep the number of such automatically included files to a minimum; the ones here are necessary for almost all other modules, however.

⟨*Global Include Files*⟩+≡
```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

Also, we include files from the standard library to get the `vector`, and `pair` template classes. The `using` directive brings these container classes into our namespace.

⟨*Global Include Files*⟩+≡
```
#include <iostream>
using std::cout;
using std::cerr;
using std::endl;
using std::ostream;
#include <string>
using std::string;
#include <vector>
using std::vector;
//#ifndef __GNUG__
//#include <pair.h>
//#endif // !__GNUG__
using std::pair;
using std::make_pair;
```

We will also define a number of types with `typedef` here. First is `Float`; rather than using the built-in `float` and `double` types for floating point variables, we abstract away this choice with `Float`. This makes it convenient to globally change from one representation to the other. In general, as long as numerical algorithms with egregious stability are avoided, the precision provided by `float` is sufficient in a ray-tracer.

For convenience, we also define shorthand names for unsigned cardinal types: `u_char`, `u_short`, `u_int`, and `u_long`.

⟨*Global Type Declarations*⟩+≡
```
typedef float Float;
typedef unsigned char u_char;
typedef unsigned short u_short;
typedef unsigned int u_int;
typedef unsigned long u_long;
```

We will also define a macro that holds `lrt`'s current version number. This is a floating-point value that will be increased as future versions of `lrt` are developed.

⟨*Global Constants*⟩+≡
```
#define LRT_VERSION 1.0
```

SGI's old C++ compiler...

⟨*Platform-specific definitions*⟩+≡
```
#ifdef sgi
#define for if (0) ; else for
#endif
```

Further Reading

Detailed information about the random number generator we are using, including the original paper from ACM Transactions on Modelling and Computer Simulation (MN98) are available at `http://www.math.keio.ac.jp/~matumoto/emt.html`.

Float to int stuff at `http://www.stereopsis.com/FPU.html`.

Gaussian elimination, pivot stuff(Atk93).

Numerical Recipes, Press (PTVF92).

Samet's book on octrees (Sam90)

de Berg et al computational geoemtry (dBvKOS00)

B. TIFF Input and Output

⟨*tiffio.cc**⟩≡
⟨*Source Code Copyright*⟩
```
#include "lrt.h"
#include "color.h"
#include <tiffio.h>
```
⟨*TIFF Function Definitions*⟩

This chapter describes `lrt`'s interface with `libtiff`, a library for reading and writing image files as TIFFs (Tag Image File Format). TIFF is perhaps the mother of all image file formats, supporting a variety of methods for image compression, a variety of spectral representations, and a variety of methods of structuring the image data.

With this flexibility comes complexity. `libtiff` makes reading and writing TIFF images easier than it would be without `libtiff`, though it is still a baroque process. Like just about every application that reads and writes TIFF images, `lrt` is unable to read certain completely valid TIFF files that use obscure features of the file format. Supporting these would greatly increase the length of this code. Just about all TIFF images that are encountered in practice should be readable by these routines.

B.1 Input

The function that reads TIFFs deals with three main types of TIFFs:

- Standard RGB eight-bit per pixel TIFFs.
- TIFFs with *colormaps*: an array of RGB colors where each pixel is represented by an index into the array (this can reduce storage needs when there are a small number of colors in the image).
- TIFFs with floating-point RGB pixel values.

We always return the pixel data as a `Spectrum` array that `TIFFRead()` allocates. This can be a wasteful representation; for an eight-bit per pixel TIFF, this is four times bigger than the original data. However, it greatly simplifies our task.

⟨*TIFF Function Definitions*⟩≡
```
Spectrum *TIFFRead(const string &name, int *xSize, int *ySize) {
    Spectrum *pixels = NULL;
    Float *fbuf = NULL;
    u_char *ubuf = NULL;
    ⟨Try to open TIFF file⟩
    ⟨Get basic information from TIFF header⟩
    ⟨Make sure this is a TIFF we can read⟩
    ⟨Read TIFF colormap if present⟩
    ⟨Allocate space for pixels and buffers⟩
    for (int y = 0; y < *ySize; ++y) {
        ⟨Read a TIFF scanline⟩
    }
    ⟨Close TIFF and return⟩
}
```

Error 498
pixels 189
Spectrum 155

⟨*Try to open TIFF file*⟩≡
```
TIFF *tiff = TIFFOpen(name.c_str(), "r");
if (!tiff) {
    Error("Unable to open TIFF %s", name.c_str());
    return NULL;
}
```

We first determine the resolution of the TIFF and the number of samples per pixel.

⟨*Get basic information from TIFF header*⟩≡
```
short int nSamples;
TIFFGetField(tiff, TIFFTAG_IMAGEWIDTH, xSize);
TIFFGetField(tiff, TIFFTAG_IMAGELENGTH, ySize);
TIFFGetField(tiff, TIFFTAG_SAMPLESPERPIXEL, &nSamples);
if (nSamples-1 > COLOR_SAMPLES)
    // Allow one extra, e.g. for alpha..
    Warning("TIFF %s has %d samples, > Spectrum (%d samples)",
        name.c_str(), nSamples, COLOR_SAMPLES);
if (nSamples != 1 && nSamples < COLOR_SAMPLES)
    Warning("TIFF %s has %d samples, < Spectrum (%d samples)",
        name.c_str(), nSamples, COLOR_SAMPLES);
```

Now things get a little complicated. We find out how many bits each sample has and in what format they're stored in. We require that either each sample is 32 bits wide and stored as a floating point value, *or* that it's 8 bits and stored as unsigned integer values. If the above is not true, we head to the fragment ⟨*Clean up after TIFF reading error*⟩, which will clean up any memory that's been allocated, close the file, and return `NULL`.

Finally we make sure that the RGB samples are interleaved (that is, as RGBRGBRGB along a scanline.) TIFFs also support images where each of the channels is stored in a separate contiguous part of the file; we don't support these.

⟨*Make sure this is a TIFF we can read*⟩≡

```
short int bitsPerSample, sampleFormat = SAMPLEFORMAT_UINT;
if (!TIFFGetField(tiff, TIFFTAG_BITSPERSAMPLE, &bitsPerSample)) {
    Error("TIFFRead: bits per sample not set in TIFF");
    ⟨Clean up after TIFF reading error⟩
}
if (!TIFFGetField(tiff, TIFFTAG_SAMPLEFORMAT, &sampleFormat)) {
    if (bitsPerSample == 32)
        sampleFormat = SAMPLEFORMAT_IEEEFP;
    else
        sampleFormat = SAMPLEFORMAT_UINT;
}

if (bitsPerSample == 32) {
    if (sampleFormat != SAMPLEFORMAT_IEEEFP) {
        Error("TIFFRead: 32 bit TIFF not stored in floating point format");
        ⟨Clean up after TIFF reading error⟩
    }
}
else {
    if (bitsPerSample != 8 && bitsPerSample != 32) {
        Error("TIFFRead: only 8 and 32 bits per sample supported");
        ⟨Clean up after TIFF reading error⟩
    }
    if (sampleFormat != SAMPLEFORMAT_UINT) {
        Error("TIFFRead: 8 bit TIFFs must be stored as unsigned ints");
        ⟨Clean up after TIFF reading error⟩
    }
}
int bytesPerSample = bitsPerSample / 8;
if (nSamples * *xSize * bytesPerSample != TIFFScanlineSize(tiff)) {
    Error("TIFFRead: RGB not interleaved in TIFF %s", name.c_str());
    ⟨Clean up after TIFF reading error⟩
}
```

498 Error
155 Spectrum
532 TIFFRead
497 Warning

If there is one sample per pixel, we assume that there is a colormap. This may not be the case; TIFF supports greyscale images as well as color images. If we check the the `PHOTOMETRIC` field doesn't indicate that there is a palette (aka colormap) stored with the image, we just give up, saving the code for that obscure

case. If it is there, we store pointers to the colormap in `mapR`, `mapG`, and `mapB`.

⟨*Read TIFF colormap if present*⟩≡
```
u_short *mapR = 0, *mapG = 0, *mapB = 0;
if (nSamples == 1) {
    short photoType;
    TIFFGetField(tiff, TIFFTAG_PHOTOMETRIC, &photoType);
    if (photoType != PHOTOMETRIC_PALETTE) {
        Error("TIFFRead: colormap not found in one-sample image");
        ⟨Clean up after TIFF reading error⟩
    }
    TIFFGetField(tiff, TIFFTAG_COLORMAP, &mapR, &mapG, &mapB);
}
```

Now we can allocate space for the resulting pixels and for buffers for reading the image. We allocate `ubuf` or `fbuf` as appropriate for the format of the image that we're reading.

⟨*Allocate space for* `pixels` *and buffers*⟩≡
```
pixels = new Spectrum[*xSize * *ySize];
Spectrum *pixelp = pixels;
if (bitsPerSample == 32) fbuf = new float[nSamples * *xSize];
else                     ubuf = new u_char[nSamples * *xSize];
```

Error 498
pixels 189
Spectrum 155
TIFFRead 532

⟨*Read a TIFF scanline*⟩≡
```
if (fbuf) {
    ⟨Read floating point TIFF scanline⟩
}
else {
    ⟨Read 8-bit TIFF scanline⟩
}
```

We read the scanline into `fbuf` and then copy it into `pixels`. Because we're reading the image from top-to-bottom, we end up going through `pixels` in order from start to end. Thus, we just increment `pixelp` after each pixel is processed. We also keep a pointer into the data read from the file, `fbufp`; this is incremented by `nSamples` after each pixel to get to the next pixel.

⟨*Read floating point TIFF scanline*⟩≡
```
float *fbufp = fbuf;
if (TIFFReadScanline(tiff, fbuf, y, 1) == -1) {
    ⟨Clean up after TIFF reading error⟩
}
for (int x = 0; x < *xSize; ++x) {
    Float cs[COLOR_SAMPLES];
    for (int i = 0; i < COLOR_SAMPLES; ++i) {
        if (i < nSamples) cs[i] = fbufp[i];
        else              cs[i] = 0.;
    }
    *pixelp++ = Spectrum(cs);
    fbufp += nSamples;
}
```

Similarly, we can do similar tricks with `ubuf` and `ubufp` when reading eight-bit TIFFs.

⟨*Read 8-bit TIFF scanline*⟩≡
```
u_char *ubufp = ubuf;
if (TIFFReadScanline(tiff, ubuf, y, 1) == -1) {
    ⟨Clean up after TIFF reading error⟩
}
for (int x = 0; x < *xSize; ++x) {
    if (nSamples == 1) {
        ⟨Decode TIFF colormap entry⟩
    }
    else {
        ⟨Convert standard 8-bit TIFF pixel⟩
    }
    ++pixelp;
    ubufp += nSamples;
}
```

If there is a colormap, we just use the sample value to index into the colormap for each of red, green, and blue. We scale by 1./255. so that the returned image values lie between zero and one.

498 Assert
514 INV_255
189 pixels
155 Spectrum

⟨*Decode TIFF colormap entry*⟩≡
```
int mapOffset = *ubufp;
Assert(COLOR_SAMPLES == 3);
Float cs[3] = { mapR[mapOffset] * INV_255 * INV_255,
    mapG[mapOffset] * INV_255 * INV_255,
    mapB[mapOffset] * INV_255 * INV_255 };
*pixelp = Spectrum(cs);
```

And reading a normal pixel is easy; we just need to scale by 1./255..

⟨*Convert standard 8-bit TIFF pixel*⟩≡
```
Float cs[COLOR_SAMPLES];
for (int i = 0; i < COLOR_SAMPLES; ++i)
    cs[i] = ubufp[i] * INV_255;
*pixelp = Spectrum(cs);
```

⟨*Close TIFF and return*⟩≡
```
delete[] ubuf;
delete[] fbuf;
TIFFClose(tiff);
return pixels;
```

⟨*Clean up after TIFF reading error*⟩≡
```
delete[] pixels;
delete[] ubuf;
delete[] fbuf;
TIFFClose(tiff);
return NULL;
```

B.2 Output

We will provide two functions for writing TIFF data: the difference between them is the format used for storing pixel values. The first function stores them as unsigned eight-bit quantities–this is the most common format for TIFF files. For many displays, this is sufficient resolution, especially if gamma correction and dithering are applied well (see Section 8.5).

The second format stores the pixel values as 32-bit floating point numbers. This allows us to store the full resolution of the result calculated by the renderer in the image format. Advantages include the ability to apply different tone reproduction algorithms without re-rendering the image (see Section 8.4 on page 244.) Unfortunately, few widely-used image display programs support floating point TIFF images.

Both of the output routines have similar structures. The file is opened, individual scanlines of pixels are written, and the file is closed.

⟨*TIFF Function Definitions*⟩+≡
```
void TIFFWrite8Bit(const string &name, Float *pixels,
        Float *alpha, int XRes, int YRes, int nChannels,
        int totXRes, int totYRes) {
    Assert(pixels);
    ⟨Open 8-bit TIFF file for writing⟩
    ⟨Write 8-bit scanlines⟩
    ⟨Close 8-bit TIFF file⟩
}
```

Assert 498
Error 498
pixels 189

Actually, we should use `TIFFSetErrorHandler()` and `TIFFSetWarningHandler()` and funnel that stuff to our own warning/error routines.

After opening the image (similarly to the `fopen()` call), we set a variety of flags which tell the library exactly what kind of TIFF we're going to give it, how to encode the samples, etc. Most of these should be reasonably self-explanatory. See the TIFF documentation (XXX URL?) for a full explanation.

⟨*Open 8-bit TIFF file for writing*⟩≡
```
TIFF *tiff = TIFFOpen(name.c_str(), "w");
if (!tiff) {
    Error("Unable to open TIFF %s for writing", name.c_str());
    return;
}

⟨Compute and set up samples per pixel⟩
TIFFSetField(tiff, TIFFTAG_IMAGEWIDTH, XRes);
TIFFSetField(tiff, TIFFTAG_IMAGELENGTH, YRes);
if (totXRes != 0) {
    TIFFSetField(tiff, TIFFTAG_PIXAR_IMAGEFULLWIDTH, totXRes);
    TIFFSetField(tiff, TIFFTAG_PIXAR_IMAGEFULLLENGTH, totYRes);
}
TIFFSetField(tiff, TIFFTAG_BITSPERSAMPLE, 8);
TIFFSetField(tiff, TIFFTAG_PHOTOMETRIC, PHOTOMETRIC_RGB);
⟨Set Generic TIFF Fields⟩
```

⟨*Compute and set up samples per pixel*⟩≡
```
int sampleCount = 0;
if (pixels) sampleCount += nChannels;
if (alpha) ++sampleCount;
TIFFSetField(tiff, TIFFTAG_SAMPLESPERPIXEL, sampleCount);
if (alpha) {
    short int extra[] = { EXTRASAMPLE_ASSOCALPHA };
    TIFFSetField(tiff, TIFFTAG_EXTRASAMPLES, (short)1, extra);
}
```

There are a few fields that are set the same way for both eight-bit and floating point TIFF files; we'll set them in a single fragment that can be shared.

⟨*Set Generic TIFF Fields*⟩≡
```
TIFFSetField(tiff, TIFFTAG_ROWSPERSTRIP, 1L);
TIFFSetField(tiff, TIFFTAG_XRESOLUTION, 1.0);
TIFFSetField(tiff, TIFFTAG_YRESOLUTION, 1.0);
TIFFSetField(tiff, TIFFTAG_RESOLUTIONUNIT, 1);
TIFFSetField(tiff, TIFFTAG_COMPRESSION, COMPRESSION_NONE);
TIFFSetField(tiff, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG);
TIFFSetField(tiff, TIFFTAG_ORIENTATION, (int)ORIENTATION_TOPLEFT);
```

513 Clamp
189 pixels

And now we can write out the scanlines of pixels. The imaging, tone mapping, and quantization process should have mapped the pixel values to the range 0–255 (in most cases); we can cast these to `unsigned chars` and write them out. It turns out that by walking through the `pixels` array linearly from start to finish, we traverse it scanline-by-scanline, from top-to-bottom–exactly the order that we're going to write it out in. Thus we can do pointer arithmetic with `pixelp` to go through the pixels.

⟨*Write 8-bit scanlines*⟩≡
```
u_char *buf = new u_char[sampleCount * XRes];
Float *pixelp = pixels;
Float *alphap = alpha;
for (int y = 0; y < YRes; ++y) {
    u_char *bufp = buf;
    for (int x = 0; x < XRes; ++x) {
        ⟨Pack 8-bit pixels samples into buf⟩
        ⟨Pack 8-bit alpha samples into buf⟩
    }
    TIFFWriteScanline(tiff, buf, y, 1);
}
```

⟨*Pack 8-bit pixels samples into buf*⟩≡
```
for (int s = 0; s < nChannels; ++s)
    *bufp++ = (u_char)(Clamp(*pixelp++, 0.f, 255.f));
```

⟨*Pack 8-bit alpha samples into buf*⟩≡
```
if (alphap)
    *bufp++ = (u_char)(Clamp(*alphap++, 0.f, 255.f));
```

⟨*Close 8-bit TIFF file*⟩≡
```
delete[] buf;
TIFFClose(tiff);
```

Writing out a floating point TIFF file is quite similar. The only differences are in some of the flags we set (which now say that it's a floating-point image), and how we write the data out.

⟨*TIFF Function Definitions*⟩+≡
```
void TIFFWriteFloat(const string &name, Float *pixels,
        Float *alpha, int XRes, int YRes, int nChannels,
        int totXRes, int totYRes) {
    ⟨Open Float TIFF file for writing⟩
    ⟨Write Float scanlines⟩
    ⟨Close Float TIFF file⟩
}
```

⟨*Open Float TIFF file for writing*⟩≡
```
TIFF *tiff = TIFFOpen(name.c_str(), "w");
if (!tiff) {
    Error("Unable to open TIFF %s for writing", name.c_str());
    return;
}
TIFFSetField(tiff, TIFFTAG_IMAGEWIDTH, XRes);
TIFFSetField(tiff, TIFFTAG_IMAGELENGTH, YRes);
if (totXRes != 0) {
    TIFFSetField(tiff, TIFFTAG_PIXAR_IMAGEFULLWIDTH, totXRes);
    TIFFSetField(tiff, TIFFTAG_PIXAR_IMAGEFULLLENGTH, totYRes);
}
⟨Compute and set up samples per pixel⟩
TIFFSetField(tiff, TIFFTAG_BITSPERSAMPLE, 32);
TIFFSetField(tiff, TIFFTAG_SAMPLEFORMAT, SAMPLEFORMAT_IEEEFP);
TIFFSetField(tiff, TIFFTAG_PHOTOMETRIC, PHOTOMETRIC_MINISBLACK);
⟨Set Generic TIFF Fields⟩
```

Error 498
pixels 189

Writing the scanlines is much easier than with eight-bit images, since we don't need to convert the `Float` values to `unsigned chars`. Note that if `Float` was typedef'd to `double`, then we would need to allocate a temporary buffer and convert to `float`, as we did above for `unsigned char`. We'll just assert that this hasn't happened, and write out the pixel data as given.

⟨*Write Float scanlines*⟩≡
```
Float *pixelp = (Float *)pixels;
Float *alphap = alpha;
Float *scanline = new Float[sampleCount * XRes];
for (int y = 0; y < YRes; ++y) {
    Float *sp = scanline;
    for (int x = 0; x < XRes; ++x) {
        if (pixelp)
            for (int c = 0; c < nChannels; ++c)
                *sp++ = *pixelp++;
        if (alphap)
            *sp++ = *alphap++;
    }
    TIFFWriteScanline(tiff, scanline, y, 1);
}
delete[] scanline;
```

⟨*Close Float TIFF file*⟩≡
```
TIFFClose(tiff);
```

189 pixels

C. Dynamic Object Creation

One of the key parts of `lrt`'s design was the decision that the `lrt` executable would only hold the key core logic of the system. All of the shapes, cameras, lights, integrators, and accelerators are stored in separate object files on disk; at run-time, `lrt` loads in the appropriate object code for the needed objects. This makes it far easier to extend `lrt` with new implementations of various types and helps ensure a clean design by making it much harder to side-step the basic system interfaces.

⟨*dynload.h**⟩≡
```
⟨Source Code Copyright⟩
#ifndef DYNLOAD_H
#define DYNLOAD_H
#include "lrt.h"
⟨Runtime Loading Declarations⟩
#endif // DYNLOAD_H
```

⟨*dynload.cc**⟩≡
```
⟨Source Code Copyright⟩
#include "dynload.h"
#include "paramset.h"
#include "shapes.h"
#include "materials.h"
#ifndef WIN32
#include <dlfcn.h>
#endif
⟨Runtime Loading Forward Declarations⟩
⟨Runtime Loading Static Data⟩
⟨Runtime Loading Local Classes⟩
⟨Runtime Loading Methods⟩
⟨DSO Method Definitions⟩
```

⟨*Global Include Files*⟩+≡
```
#ifdef WIN32
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#endif
```

C.1 Parameter Sets

We'll first introduce a class that handles collections of named parameters and their values. It is a key part of how objects are created at run-time, bundling up the values of the parameters to the constructors in a single object. For example, it might record that there is a single floating-point value named "radius" with a value of 2.5, and an array of four color values named "specular" with various color values. The `ParamSet` provides methods for both setting and retrieving values from this kind of set of parameters.

⟨*ParamSet Declarations*⟩≡
```
class ParamSet {
public:
    ⟨ParamSet Constructors⟩
    ~ParamSet();
    void init(int n, const char **tokens, void **params, int nv = 0);
    void clear();

    ⟨ParamSet Interface⟩
    ⟨ParamSet Public Data⟩
private:
    ⟨ParamSet Data⟩
};
```

Internally, the `ParamSet` stores a vector for each of the different parameter types that it stores. Each bound parameter is represented by a `ParmSetItem` of the appropriate type.

⟨*ParamSet Data*⟩≡
```
vector<ParamSetItem<int> *> ints;
vector<ParamSetItem<Float> *> floats;
vector<ParamSetItem<Point> *> points;
vector<ParamSetItem<Vector> *> vectors;
vector<ParamSetItem<Normal> *> normals;
vector<ParamSetItem<Spectrum> *> spectra;
vector<ParamSetItem<string> *> strings;
```

The `ParamSetItem` structure mostly just needs to store the name of the parameter and a pointer to memory that stores its value. We also keep track of the number of array elements, if the item holds an array of item values, as well as the type of this item.

⟨*ParamSet Declarations*⟩+≡
```
template <class T> struct ParamSetItem {
    ParamSetItem(const string &name, const T *val, int type, int count,
        int nVertex = 0);
    ~ParamSetItem();
    ParamSetItem<T> *Clone(int nVertex) const;

    string name;
    int type, arraySize;
    T *data;
    bool lookedUp;
};
```

```
23  Normal
21  Point
155 Spectrum
16  Vector
```

⟨*ParamSetItem Methods*⟩≡
```
template <class T>
ParamSetItem<T>::ParamSetItem(const string &n, const T *v, int t,
        int c, int nVertex) {
    name = n;
    type = t;
    arraySize = c;
    lookedUp = false;
    ⟨Determine number of data items for ParamSetItem⟩
    data = new T[nAlloc];
    for (int i = 0; i < nAlloc; ++i)
        data[i] = v[i];
}
```

The parameter type includes both the underlying datatype–float, int, etc.–as well as the parameter's storage class. These two are stored together in the `type` member. We'll define some constants to represent each of the possible types.

⟨*ParamSet Types*⟩≡
```
#define PARAM_TYPE_INT     (1<<0)
#define PARAM_TYPE_FLOAT   (1<<1)
#define PARAM_TYPE_POINT   (1<<2)
#define PARAM_TYPE_VECTOR  (1<<3)
#define PARAM_TYPE_NORMAL  (1<<4)
#define PARAM_TYPE_STRING  (1<<5)
#define PARAM_TYPE_COLOR   (1<<6)
#define PARAM_TYPE_VOID    (1<<7)
#define PARAM_TYPE_HPOINT  (1<<8)
```

We'll also specify an illegal value for the type member, in order to signify error conditions.

⟨*ParamSet Types*⟩+≡
```
#define PARAM_TYPE_ERROR       -1
```

The storage class accounts for the idea that we may want to have multiple values of a parameter defined in a way that it can be interpolated over a surface, taking on a different value at each point being shaded. For example, a triangle mesh might be defined with a single diffuse color for all of the triangles, but with specular colors defined at each vertex and interpolated inside each face.

ParamSet 542

There are three different storage classes to handle these sorts of situations:

- Uniform parameters take on a single value over the entire object
- Varying parameters are specified with four values which are bilinearly interpolated according to the (u, v) parameter value for a point on the surface.
- Vertex parameters are only available for mesh shapes, and represent values specefied at each vertex of the mesh.

⟨*ParamSet Types*⟩+≡
```
#define PARAM_TYPE_UNIFORM (1<<9)
#define PARAM_TYPE_VARYING (1<<10)
#define PARAM_TYPE_VERTEX  (1<<11)
```

The `ParamSet` also stores the number of items to expect for items with vertex storage class.

⟨*ParamSet Constructors*⟩≡
```
ParamSet(int nv = 0) { nVertex = nv; }
```

⟨*ParamSet Public Data*⟩≡
```
int nVertex;
```

Now we can define the fragment that tells us how many items we need to allocate space for.

⟨*Determine number of data items for* `ParamSetItem`⟩≡
```
int nAlloc = arraySize;
if (type & PARAM_TYPE_VARYING) nAlloc *= 4;
if (type & PARAM_TYPE_VERTEX) nAlloc *= nVertex;
```

⟨*ParamSetItem Methods*⟩+≡
```
template <class T>
ParamSetItem<T>::~ParamSetItem() {
    delete[] data;
}
```

⟨*ParamSetItem Methods*⟩+≡
```
template <class T>
ParamSetItem<T> *ParamSetItem<T>::Clone(int nVertex) const {
    return new ParamSetItem<T>(name, data, type,
        arraySize, nVertex);
}
```

To add an entry to the parameter set, the user just calls the appropriate method, passing the name of the parameter, a pointer to its value, and storage class information.

⟨*ParamSet Methods*⟩+≡
```
void ParamSet::AddFloat(const string &name, const Float *data,
        int type, int narray) {
    type |= PARAM_TYPE_FLOAT;
    floats.push_back(new ParamSetItem<Float>(name, data, type,
        narray, nVertex));
}
```

We won't include the rest of the methods to add other data types to the `ParamSet`, but will include their prototypes here for reference.

⟨*ParamSet Interface*⟩+≡
```
void AddInt(const string &, const int *,
    int type = PARAM_TYPE_UNIFORM, int nArray = 1);
void AddPoint(const string &, const Point *,
    int type = PARAM_TYPE_UNIFORM, int nArray = 1);
void AddVector(const string &, const Vector *,
    int type = PARAM_TYPE_UNIFORM, int nArray = 1);
void AddNormal(const string &, const Normal *,
    int type = PARAM_TYPE_UNIFORM, int nArray = 1);
void addSpectrum(const string &, const Spectrum *,
    int type = PARAM_TYPE_UNIFORM, int nArray = 1);
void AddString(const string &, const string *, int nArray = 1);
```

Looking up parameter values is straightforward; we just loop through the values we have of the requested type and return the value, if any. There are two versions of the lookup method, a simple one for uniform parameters with array size of one (or non-array types), that returns the data value directly.

```
 23 Normal
544 PARAM_TYPE_FLOAT
544 PARAM_TYPE_UNIFORM
542 ParamSet
543 ParamSetItem
 21 Point
494 push_back
155 Spectrum
 16 Vector
```

⟨*ParamSet Methods*⟩+≡
```
Float ParamSet::FindOneFloat(const string &name, Float d) const {
    for (u_int i = 0; i < floats.size(); ++i)
        if (floats[i]->name == name &&
            floats[i]->type & PARAM_TYPE_UNIFORM &&
            floats[i]->arraySize == 1) {
            floats[i]->lookedUp = true;
            return *(floats[i]->data);
        }
    return d;
}
```

As above, here are the declarations for the rest of the analogous methods.

⟨*ParamSet Interface*⟩+≡
```
int FindOneInt(const string &, int d) const;
Point FindOnePoint(const string &, const Point &d) const;
Vector FindOneVector(const string &, const Vector &d) const;
Normal FindOneNormal(const string &, const Normal &d) const;
Spectrum FindOneSpectrum(const string &,
    const Spectrum &d) const;
string FindOneString(const string &, const string &d) const;
```

Normal 23
PARAM_TYPE_UNIFORM 544
ParamSet 542
Point 21
size 494
Spectrum 155
Vector 16

The second lookup method returns a pointer to the data if it's present. It returns the storage class information in the given `type` pointer and the number of array elements in the `nArray` value. It's up to the caller to interpret these appropriately when accessing the returned pointer.

⟨*ParamSet Methods*⟩+≡
```
const Float *ParamSet::FindFloat(const string &name, int *type,
        int *nArray) const {
    for (u_int i = 0; i < floats.size(); ++i)
        if (floats[i]->name == name) {
            *nArray = floats[i]->arraySize;
            *type = floats[i]->type;
            floats[i]->lookedUp = true;
            return floats[i]->data;
        }
    return NULL;
}
```

These are the rest of the analogous lookup functions.

⟨*ParamSet Interface*⟩+≡
```
const int *findInt(const string &, int *type,
    int *nArray) const;
const Point *FindPoint(const string &, int *type,
    int *nArray) const;
const Vector *FindVector(const string &, int *type,
    int *nArray) const;
const Normal *FindNormal(const string &, int *type,
    int *nArray) const;
const Spectrum *FindSpectrum(const string &, int *type,
    int *nArray) const;
const string *FindString(const string &, int *type,
    int *nArray) const;
```

⟨*ParamSet Methods*⟩+≡
```
int ParamSet::TypeToNum(int type) {
    if (type & PARAM_TYPE_UNIFORM) return 1;
    else if (type & PARAM_TYPE_VARYING) return 4;
    else if (type & PARAM_TYPE_VERTEX)  return nVertex;
    else {
        Assert(1 == 0);
        return 1;
    }
}
```

Because the user may misspell parameter names in the scene description file, we'll also provide a function that goes through the parameter set and reports if any of the parameters present were never looked up. If this happens, odds are good the user has given an incorrect parameter.

⟨*ParamSet Methods*⟩+≡
```
void ParamSet::ReportUnused() const {
#define CHECK_UNUSED(v) \
    for (i = 0; i < (v).size(); ++i) \
        if (v[i]->name[0] != '_' && !(v)[i]->lookedUp) \
            Warning("Parameter \"%s\" not used", \
                (v)[i]->name.c_str())
    u_int i;
    CHECK_UNUSED(ints);
    CHECK_UNUSED(floats);
    CHECK_UNUSED(points);
    CHECK_UNUSED(vectors);
    CHECK_UNUSED(normals);
    CHECK_UNUSED(spectra);
    CHECK_UNUSED(strings);
}
```

⟨*ParamSet Methods*⟩+≡
```
ParamSet::~ParamSet() {
    clear();
}
```

498 Assert
23 Normal
544 PARAM_TYPE_UNIFORM
544 PARAM_TYPE_VARYING
544 PARAM_TYPE_VERTEX
542 ParamSet
21 Point
494 size
155 Spectrum
16 Vector
497 Warning

⟨*ParamSet Methods*⟩+≡
```
void ParamSet::clear() {
    u_int i;
#define DEL_PARAMS(name) \
    for (i = 0; i < (name).size(); ++i) \
        delete (name)[i]; \
    (name).erase((name).begin(), (name).end())

    DEL_PARAMS(ints);
    DEL_PARAMS(floats);
    DEL_PARAMS(points);
    DEL_PARAMS(vectors);
    DEL_PARAMS(normals);
    DEL_PARAMS(spectra);
    DEL_PARAMS(strings);
#undef DEL_PARAMS
}
```

C.2 Reading Dynamic Libraries

ParamSet 542
Reference 509
size 494
Transform 32

In this section, we will describe the general process that `lrt` uses to link in implementations at runtime. We will focus on the details only for the `Shape` class, since the other times that are loaded at runtime are handled quite similarly.

Creation Functions

All of the object files that hold shape implementations must provide a function with the same signature. When `lrt` needs to create a particular shape, it will call this function from the appropriate object file.

⟨*Shape Creation Declaration*⟩≡
```
Reference<Shape> CreateShape(const Transform &o2w, const ParamSet &params);
```

Because all `Shape`s store an object to world transformation, we pass the appropriate transformation to this function. However, in general we need to be able to pass whichever other parameters the particular shape needs and that the user may have set in the input file. Because we don't want to hard-code knowledge like "spheres need to have a floating-point radius value passed to their constructor" into `lrt`, we use the `ParamSet` to handle marshal parameters and their values for use by the individual shapes.

The dynamic sphere creation routine just pulls the appropriate values out of the `ParamSet` and cals the constructor, returning a newly-allocated sphere.

⟨*Sphere Methods*⟩+≡
```
extern "C" Reference<Shape> CreateShape(const Transform &o2w,
        const ParamSet &params) {
    Float radius = params.FindOneFloat("_radius", 1);
    Float zmin = params.FindOneFloat("_zmin", -radius);
    Float zmax = params.FindOneFloat("_zmax", radius);
    Float thetamax = params.FindOneFloat("_thetamax", 360);
    return new Sphere(o2w, radius, zmin, zmax, thetamax);
}
```

The creation routines for other shapes are quite similar, so won't be included here.

XXX include the basic signatures for the other object creation functions here, though XXX

Loading object files

Loading an object file with such a function to be called from disk and linking it into a running application can be done relatively easy in modern operating systems. The system calls to use are highly operating-system dependent, however. The DSO base-class is one key to this process; it hides the operating-system-dependent parts of it.

Dynamic shared object DSO

Dynamic link library DLL

XXX what is a DSO, DSO vs DLL. Rename this class? XXX

⟨*Global Classes*⟩+≡
```
class DSO {
public:
    ⟨DSO Methods⟩
private:
#if defined(WIN32)
    HMODULE hinstLib;
#else
    void *hinstLib;
#endif
};
```

498 Error
546 FindOneFloat
542 ParamSet
509 Reference
55 Sphere
32 Transform

The DSO constructor handles the first step of loading the shared object into lrt's address spade. It takes a pathname to the object file.

⟨*DSO Method Definitions*⟩≡
```
DSO::DSO(const string &fname) {
#ifdef WIN32
    hinstLib = LoadLibrary(fname.c_str());
    if (!hinstLib)
        Error("DSO Loader can't open DLL %s", fname.c_str());
#else
    hinstLib = dlopen(fname.c_str(), RTLD_LAZY);
    if (!hinstLib)
        Error("DSO Loader can't open DLL %s (%s)", fname.c_str(),
            dlerror());
#endif
}
```

And the destructor makes the system call to remove the library from our address space.

⟨*DSO Method Definitions*⟩+≡
```
DSO::~DSO() {
#ifdef WIN32
    FreeLibrary(hinstLib);
#else
    dlclose(hinstLib);
#endif
}
```

Once a library has been loaded into memory, the `GetSymbol` function lets us ask for a function inside the DSO with a particular name. If that function exists, then this returns a pointer to it which we can use to actually call it.

⟨*DSO Method Definitions*⟩+≡
```
void *DSO::GetSymbol(const string &symname) {
    void *data;
#ifdef WIN32
    data = GetProcAddress(hinstLib, symname.c_str());
#else
    data = dlsym(hinstLib, symname.c_str());
#endif
    if (!data)
        Error("Couldn't get symbol \"%s\" in DSO.", symname.c_str());
    return data;
}
```

DSO 549
Error 498
ParamSet 542
Reference 509
Transform 32

For each base type for which we are able to load implementations at runtime, we inherit from `DSO`. Here is the implementation of `ShapeDSO`. All of these implementations just call the `DSO` `GetSymbol` function in the constructor, passing in the name of the object creation function (e.g. `CreateShape`, which was introduced previously in this section.) All `Shape` shared object files implement this function and return a new `Shape` of their particular type when it is called.

⟨*Runtime Loading Local Classes*⟩≡
```
class ShapeDSO : public DSO {
public:
    ⟨ShapeDSO Constructor⟩
    typedef Reference<Shape> (*CreateShapeFunc)(const Transform &o2w,
        const ParamSet &params);
    CreateShapeFunc CreateShape;
};
```

One possibly dangerous thing that the constructor does is cast the returned symbol to be a pointer to a function with the right signature for creating shapes. If the person who implemented a particular `Shape` defined it with a `CreateShape` function that only took a `ParamSet` and didn't have a `Transform` parameter, the program would probably crash at run-time if it tried to call that function. In the interests of making it easier to keep `lrt` portable across architectures, we'll just take that risk and keep the code here simpler.

⟨*ShapeDSO Constructor*⟩≡
```
ShapeDSO(const string &name)
    : DSO(name) {
    CreateShape = (CreateShapeFunc)(GetSymbol("CreateShape"));
}
```

XXX call this function something else! XXX

The function that the main section of `lrt` uses when it actually needs to create a shape is also called `CreateShape`. It takes the name of the shape to be created, the object to world transformation, and the `ParamSet` for the new shape. It calls `GetShapeDSO`, which will be defined shortly–it returns the `DSO` for the named shape if it exists–and it then calls the creation function pointer that the `DSO` holds to actually cause the particular shape to be made.

⟨*Runtime Loading Methods*⟩≡
```
Reference<Shape> CreateShape(const string &name, const string &searchpath,
        const Transform &object2world, const ParamSet &paramSet) {
    ShapeDSO *dso = LoadDSO<ShapeDSO>(name, shape_dsos, "shapes/",
        searchpath);
    if (dso)
        return dso->CreateShape(object2world, paramSet);
    return NULL;
}
```

549 DSO
498 Error
550 GetSymbol
542 ParamSet
509 Reference
516 Search
552 shape_dsos
550 ShapeDSO
32 Transform

⟨*Runtime Loading Forward Declarations*⟩≡
```
template <class D> D*LoadDSO(const string &name,
        StringHashTable &hashTable, const string &subdir,
        const string &searchPath) {
    D *dso = (D *)hashTable.Search(name);
    if (!dso) {
        string filename = subdir + name;
#ifdef WIN32
        filename += ".dll";
#else
        filename += ".so";
#endif
        string path = SearchPath(searchPath, filename);
        if (path != "") {
            dso = new D(path.c_str());
            hashTable.Add(name, dso);
        }
        else
            Error("Unable to find DSO/DLL for \"%s\"",
                name.c_str());
    }
    return dso;
}
```

⟨*Runtime Loading Static Data*⟩≡
```
static StringHashTable shape_dsos, filter_dsos;
static StringHashTable material_dsos, bump_dsos;
static StringHashTable light_dsos, arealight_dsos, volume_dsos;
static StringHashTable surf_integrator_dsos, vol_integrator_dsos, tonemap_dsos;
static StringHashTable accelerator_dsos, camera_dsos, sampler_dsos;
```

⟨*Runtime Loading Methods*⟩+≡
```
static string SearchPath(const string &searchpath,
        const string &filename) {
    const char *start = searchpath.c_str();
    const char *end = start;
    while (*start) {
        while (*end && *end != ':')
            ++end;
        string component(start, end);

        string fn = component + "/" + filename;
        FILE *f = fopen(fn.c_str(), "r");
        if (f) {
            fclose(f);
            return fn;
        }
        if (*end == ':') ++end;
        start = end;
    }
    return "";
}
```

Material 303
Spectrum 155
surfaceParams 575
Texture 323

creation stuff

⟨*Material creation macros*⟩+≡
```
#define SURF_TEX_S(var, def) \
    Texture<Spectrum> *(var) = Material::MakeSpecTex(geomParams, surfaceParams,
        #var, def)
#define SURF_TEX_F(var, def) \
    Texture<Float> *(var) = Material::MakeFloatTex(geomParams, surfaceParams, \
        #var, def)
```

⟨*Material Method Definitions*⟩+≡

```
Texture<Spectrum> *Material::MakeSpecTex(const ParamSet &pGeom, const ParamSet &pShader,
        const string &name, const Spectrum &def) {
    int type, narray;
    const Spectrum *s = pGeom.FindSpectrum(name, &type, &narray);
    if (!s)   s = pShader.FindSpectrum(name, &type, &narray);
    if (!s) return new ConstantTexture<Spectrum>(def);
    Assert(narray == 1); // XXX for now

    if (type & PARAM_TYPE_UNIFORM)
        return new ConstantTexture<Spectrum>(*s);
    else if (type & PARAM_TYPE_VARYING)
        return new BilerpTexture<Spectrum>(new IdentityMapping2D,
            s[0], s[1], s[2], s[3]);
    else {
        if (pGeom.nVertex != 0)
            return new VertexTexture<Spectrum>(s, pGeom.nVertex);
        else {
            Warning("Vertex texture for \"%s\" not supported "
                "for this object", name.c_str());
            return NULL;
        }
    }
}
```

498 Assert
330 BilerpTexture
324 ConstantTexture
547 FindSpectrum
327 IdentityMapping2D
303 Material
544 PARAM_TYPE_UNIFORM
544 PARAM_TYPE_VARYING
542 ParamSet
155 Spectrum
323 Texture
332 VertexTexture
497 Warning

⟨*Material Method Definitions*⟩+≡

```
Texture<Float> *Material::MakeFloatTex(const ParamSet &pGeom, const ParamSet &p
        const string &name, Float def) {
    int type, narray;
    const Float *s = pGeom.FindFloat(name, &type, &narray);
    if (!s)
        s = pShader.FindFloat(name, &type, &narray);
    if (!s) return new ConstantTexture<Float>(def);
    Assert(narray == 1); // XXX for now
    if (type & PARAM_TYPE_UNIFORM)
        return new ConstantTexture<Float>(*s);
    else if (type & PARAM_TYPE_VARYING)
        return new BilerpTexture<Float>(new IdentityMapping2D,
            s[0], s[1], s[2], s[3]);
    else {
        if (pGeom.nVertex != 0)
            return new VertexTexture<Float>(s, pGeom.nVertex);
        else {
            Warning("Vertex texture for \"%s\" not supported "
                "for this object", name.c_str());
            return NULL;
        }
    }
}
```

⟨*HomogeneousRegion Definitions*⟩≡

```
extern "C" VolumeRegion *CreateVolumeRegion(const Transform &volume2world,
        const ParamSet &params) {
    Spectrum sigma_a = params.FindOneSpectrum("sigma_a", 0.);
    Spectrum sigma_s = params.FindOneSpectrum("sigma_s", 0.);
    Float g = params.FindOneFloat("g", 0.);
    Spectrum Le = params.FindOneSpectrum("Le", 0.);
    Point p0 = params.FindOnePoint("p0", Point(0,0,0));
    Point p1 = params.FindOnePoint("p1", Point(1,1,1));
    return new HomogeneousRegion(sigma_a, sigma_s, g, Le, BBox(p0, p1),
        volume2world);
}
```


Assert 498
BilerpTexture 330
ConstantTexture 324
FindFloat 546
FindOneFloat 546
FindOnePoint 546
FindOneSpectrum 546
HomogeneousRegion 384
IdentityMapping2D 327
Material 303
PARAM_TYPE_UNIFORM 544
PARAM_TYPE_VARYING 544
ParamSet 542
Point 21
Spectrum 155
Texture 323
Transform 32
VertexTexture 332
VolumeRegion 383
Warning 497

⟨*VolumeGrid Definitions*⟩+≡

```
extern "C" VolumeRegion *CreateVolumeRegion(const Transform &volume2world,
        const ParamSet &params) {
    Spectrum sigma_a = params.FindOneSpectrum("sigma_a", 0.);
    Spectrum sigma_s = params.FindOneSpectrum("sigma_s", 0.);
    Float g = params.FindOneFloat("g", 0.);
    Spectrum Le = params.FindOneSpectrum("Le", 0.);
    Point p0 = params.FindOnePoint("p0", Point(0,0,0));
    Point p1 = params.FindOnePoint("p1", Point(1,1,1));
    string filename = params.FindOneString("filename", "");
    return new VolumeGrid(sigma_a, sigma_s, g, Le, BBox(p0, p1),
        volume2world, filename);
}
```

⟨*ExponentialMist Definitions*⟩≡

```
extern "C" VolumeRegion *CreateVolumeRegion(const Transform &volume2world,
        const ParamSet &params) {
    Spectrum sigma_a = params.FindOneSpectrum("sigma_a", 0.);
    Spectrum sigma_s = params.FindOneSpectrum("sigma_s", 0.);
    Float g = params.FindOneFloat("g", 0.);
    Spectrum Le = params.FindOneSpectrum("Le", 0.);
    Point p0 = params.FindOnePoint("p0", Point(0,0,0));
    Point p1 = params.FindOnePoint("p1", Point(1,1,1));
    Float A = params.FindOneFloat("A", 1.);
    Float B = params.FindOneFloat("B", 1.);
    return new ExponentialMist(sigma_a, sigma_s, g, Le, BBox(p0, p1),
        volume2world, A, B);
}
```


389 ExponentialMist
546 FindOneFloat
546 FindOnePoint
546 FindOneSpectrum
546 FindOneString
542 ParamSet
21 Point
155 Spectrum
32 Transform
387 VolumeGrid
383 VolumeRegion

D. Rendering Interface

⟨*ri.h**⟩≡

```
⟨Source Code Copyright⟩
#ifndef RI_H
#define RI_H
#include "lrt.h"
#ifdef __cplusplus
extern "C" {
#endif /* C++ */

⟨RI Function Declarations⟩

#ifdef __cplusplus
}
#endif /* C++ */
#endif /* RI_H */
```

⟨*api.cc**⟩≡
⟨*Source Code Copyright*⟩
```
#include "paramset.h"
#include "ri.h"
#include "lrt.h"
#include "film.h"
#include "primitives.h"
#include "camera.h"
#include "color.h"
#include "light.h"
#include "sampling.h"
#include "materials.h"
#include "transport.h"
#include "scene.h"
#include "texture.h"
#include "dynload.h"
#include <ctype.h>
#ifdef _WIN32
#include <malloc.h>
#else
#include <alloca.h>
#endif
```
⟨*API Includes*⟩
⟨*API Local Classes*⟩
⟨*API Static Data*⟩
⟨*API Macros*⟩
⟨*API Static Methods*⟩
⟨*RI Function Definitions*⟩

alloca 495
film 173
primitives 579

In this chapter we will describe our implementation of an external interface to `lrt`. The need for such an interface is clear: there must be a convenient way in which the scene to be rendered can be described to the renderer. There have been two main approaches to this problem in graphics: the interface may specify *how* to rendering the scene, configuring a rendering pipeline at a low-level, or it may specify *what* the scene's objects, lights, and material properties are, and leave it to the renderer to decide how to transform that description into the best-possible image. The first approach has been successfully used for interactive graphics, as seen in the OpenGL and Direct3D APIs. The second, declarative approach, has been most successful for high-end offline rendering, e.g. as embodied by the RenderMan interface. For `lrt`, we will use an interface based on the declarative approach.

The interface to the renderer is defined by a carefully-chosen set of function calls that allow the user to specify the scene. For convenience, we will also support text scene description files; the statements in these files have a one-to-one mapping with the API's function calls.

D.1 Tokens and Parameter Management

The user can declare their own parameters and their types—for example, this provides a general mechanism to attach arbitrary data to geometric primitives for later use in surface shaders.

We use a token table to record the names of the declared parameters and their types. When a parameter is declared (using `RiDeclare()`, below), a `const char *` object is returned. This is a pointer to a string that is a copy of the token's name which can later be passed back through the RI layer when a parameter name is needed. If such a `const char *` is passed, rather than a `const char *` pointing to the name, better performance may be possible.

Tokens are stored in a small hash table. Strings are hashed to compute an offset into the table and then we walk through the list of tokens at that position to find the token being looked for. In addition to the string that is their name, we store an `int` with each one where we encode its type.

⟨*API Local Classes*⟩≡
```
struct Token {
    Token(const char *n, int t = 0, int a = 0) {
        token = n;
        type = t;
        arrayLength = a;
    }
    string token;
    int type;
    int arrayLength;
};
```

⟨*API Static Data*⟩≡
```
StringHashTable TokenTable;
```

The `RiDeclare` function returns a new token for a user-supplied variable of name `name` with type `type`. The type should be something like `uniform float`, `vertex point`, etc. A token can be declared repeatedly with no ill effect; however, it is an error to redeclare a token with a different type than was originally used to declare it.

⟨*RI Function Definitions*⟩≡
```
const char *RiDeclare(char *name, char *type) {
    ⟨Compute integer type code for variable⟩
    ⟨Search for token in token table⟩
    ⟨Add token to table if not found⟩
}
```

First we turn the type into an integer that compactly encodes it (see `stringToType()` below. If an error is returned, indicating that the type couldn't be decoded, we return `NULL`.

⟨*Compute integer type code for variable*⟩≡
```
int narray;
int tokenType = stringToType(type, &narray);
if (tokenType == PARAM_TYPE_ERROR)
    return NULL;
```

Now we'll look through the linked-list at the appropriate hash table position and see if a token with this name has already been declared. If so, we make sure that the user isn't redeclaring a variable with a new type, printing a warning if so.

⟨*Search for token in token table*⟩≡
```
Token *token = (Token *)TokenTable.Search(name);
if (token) {
    ⟨Complain if token was redeclared to a different type⟩
    return token->token.c_str();
}
```

⟨*Complain if token was redeclared to a different type*⟩≡
```
if (token->type != tokenType || token->arrayLength != narray) {
    string s1 = typeToString(token->type, token->arrayLength);
    string s2 = typeToString(tokenType, narray);
    Warning("RiDeclare: Token '%s' redeclared from %s to %s.",
        name, s1.c_str(), s2.c_str());
}
```

If we didn't exit earlier after finding the token in the table already, we will add it to the table. We copy the string with its name and add the token and its type `int` to the linked list at the hash table position.

PARAM_TYPE_ERROR 544
RiDeclare 559
Search 516
stringToType 561
Token 559
TokenTable 559
typeToString 561
Warning 497

⟨*Add token to table if not found*⟩≡
```
Token *newToken = new Token(name, tokenType, narray);
TokenTable.Add(name, newToken);
return newToken->token.c_str();
```

⟨*System-wide Initialization*⟩+≡
```
RiDeclare("Cs", "vertex color");
RiDeclare("Os", "float");
RiDeclare("N", "vertex normal");
RiDeclare("P", "vertex point");
RiDeclare("Pw", "vertex float[4]");
RiDeclare("Pz", "vertex float");
RiDeclare("s", "vertex float");
RiDeclare("st", "vertex float[2]");
RiDeclare("t", "vertex float");
RiDeclare("shadows", "float");
```

We won't include the implementations of the `stringToType` and `typeToString` functions here; both are just a matter of parsing and string management. The first function, `stringToType`, takes strings of the form

```
uniform point[2],
```

parses them, and returns two integer values. The integer returned directly encodes the basic type, using the `PARAM_TYPE_` values defined in Section C.1. For the type above, it would return

```
PARAM_TYPE_UNIFORM | PARAM_TYPE_POINT.
```

The `nArray` pointer allows it to return the number of array elements for the type, in this case 2.

⟨*API Static Methods*⟩≡
```
static int stringToType(const char *strType, int *nArray);
```

The `typeToString` function just goes the other way, returning a string from an encoded type and array length.

⟨*API Static Methods*⟩+≡
```
static string typeToString(int type, int narray);
```

And this takes a name and returns the type of a declared variable.

⟨*RI Function Declarations*⟩+≡
```
bool lookupType(const char *tok, int *type, int *narray, string &name);
```

Extra ParamSet Stuff

We provide a default constructor, that does nothing, and a version that lets the caller pass in the parameters to the RI function.

⟨*ParamSet Constructors*⟩+≡
```
ParamSet(int n, const char **tokens, void **params, int nv = 0) {
    init(n, tokens, params, nv);
}
```

542 ParamSet
497 Warning

The user can also call an `init` method to re-initialize a `ParamSet` with new parameter values.

⟨*ParamSet Methods*⟩+≡
```
void ParamSet::init(int n, const char **tokens,
        void **params, int nv) {
    nVertex = nv;
    clear();
    ⟨Initialize ParamSet data values⟩
}
```

Initializing these vectors is straightforward; we just loop over the parameters. For each one, we try to determine its type from its name (as set by a previous `RiDeclare` call, or from an inline parameter type declaration.)

⟨*Initialize ParamSet data values*⟩≡
```
for (int i = 0; i < n; ++i) {
    int type, narray;
    string name;
    if (lookupType(tokens[i], &type, &narray, name)) {
        ⟨Process successful token lookup for ParamSet⟩
    }
    else
        Warning("Type of parameter \"%s\" is unknown",
            tokens[i]);
}
```

If we were successful at determining the type of the parameter, we just add it to the appropriate vector. We won't include the fragment `Process successful token lookup for ParamSet` here XXX.

D.2 Global Settings and Graphics Options

The rendering system is initialized by a call to `RiBegin()` (see Section D.2.) After this, general rendering options like the camera position and the image resolution can be set. `RiWorldBegin()` is called next and the options are fixed; they can't be changed any more. The user then provides the geometric primitives and lights that are in the scene along with their various attributes. When all of the primitives have been supplied, `RiWorldEnd()` is called. The image will be rendered and written to disk before `RiWorldEnd()` returns. Finally, `RiEnd()` is called; this handles final cleanup of the system.

We can now move forward and start to define more of the API functions. We'll start with the first function that should be called as well as the last: `RiBegin()` and `RiEnd()`–these do all of the system-wide initialization and cleanup.

Most of the guts of `RiBegin` will be filled in by pieces added to the ⟨*RiBegin Initialization*⟩ throughout the rest of this appendix.

⟨*RI Function Definitions*⟩+≡
```
void RiBegin() {
    ⟨System-wide Initialization⟩
}
```

Similarly most of `RiEnd` is filled in elsewhere as well.

Severe 498
StatsCleanup 504

⟨*RI Function Definitions*⟩+≡
```
void RiEnd() {
    ⟨System-wide cleanup⟩
    StatsCleanup();
}
```

State tracking

Because almost all of the API calls are illegal before `RiBegin()` is called and because most of the others are only legal before or after `RiWorldBegin()`, we will provide some facilities for tracking what state the API is in. We use a module static variable `currentApiState`. It starts out with value `STATE_UNINITIALIZED` and is updated by `RiBegin()`, `RiWorldBegin()`, and `RiEnd()`.

⟨*API Static Data*⟩+≡
```
#define STATE_UNINITIALIZED  0
#define STATE_BEGIN          1
#define STATE_WORLD_BEGIN    2
static int currentApiState = STATE_UNINITIALIZED;
```

⟨*System-wide Initialization*⟩+≡
```
if (currentApiState != STATE_UNINITIALIZED)
    Severe("RiBegin() has already been called.");
currentApiState = STATE_BEGIN;
```

⟨*System-wide cleanup*⟩≡
```
currentApiState = STATE_UNINITIALIZED;
```

Now, all RI procedures that are only valid in particular states call the `VERIFY_STATE` macro, passing the state that they expect us to be in as well as a string that is their

procedure name. If the states don't match, we print an error message and return immediately from the function.

⟨*API Macros*⟩≡
```
#define VERIFY_STATE(s, func) \
    if (currentApiState != s) { \
        Error("Must have called %s before calling %s().  Ignoring.", \
            missingStateCall[s], func); \
        return; \
    } \
    else /* swallow trailing semicolon */
```

Through some array indexing trickery, we can take the expected state value `s`, and find the string name of the procedure that needs to be called before the current function can be used.

⟨*API Static Data*⟩+≡
```
static const char *missingStateCall[] = { "RiEnd()",
       "RiBegin()", "RiWorldBegin()" };
```

Options

The user can set a variety of options before the scene to be rendered is specified. These include things such as the camera position and type, image sampling and reconstruction options, the type of image file to write out, etc. We store all of this information in a `GfxOptions` structure. It has a number of public data members that subsequent API calls will set and a number of methods to help create objects used by the rest of the system for rendering.

562 currentApiState
498 Error
579 lights
579 primitives
562 RiBegin
562 RiEnd
571 RiWorldBegin
579 volumeRegions

⟨*API Local Classes*⟩+≡
```
GfxOptions::GfxOptions() {
    ⟨GfxOptions Constructor Implementation⟩
}
```

We have a single instance of the `GfxOptions` that is available to the rest of the functions in this file.

⟨*API Static Data*⟩+≡
```
static GfxOptions *curGfxOptions = NULL;
```

When `RiBegin` is called, we need to ensure that the `GfxOptions` is re-initialized to hold default values.

⟨*System-wide Initialization*⟩+≡
```
curGfxOptions = new GfxOptions;
```

⟨*System-wide cleanup*⟩+≡
```
delete curGfxOptions;
curGfxOptions = NULL;
```

⟨*GfxOptions Method Declarations*⟩≡
```
void WorldEnd() {
    primitives.erase(primitives.begin(), primitives.end());
    lights.erase(lights.begin(), lights.end());
    volumeRegions.erase(volumeRegions.begin(), volumeRegions.end());
}
```

Camera and Film

Most of the camera and film functions completely straightforward; the following functions just directly set the appropriate fields in `GfxOptions` with the parameters passed to them.

For starters, the `RiPixelSamples` function sets the number of samples to take in the *x* and *y* directions for each pixel in the image. We use the `VERIFY_STATE` macro to make sure that `RiBegin` has been called but that `RiWorldBegin` has not yet been called.

⟨*RI Function Definitions*⟩+≡
```
void RiPixelSamples(Float x, Float y) {
    VERIFY_STATE(STATE_BEGIN, "RiPixelSamples");
    curGfxOptions->PixelSamples[0] = max(1, Round2Int(x));
    curGfxOptions->PixelSamples[1] = max(1, Round2Int(y));
}
```

And appropriate fields are added to `GfxOptions`.

⟨*Graphics Options*⟩≡
```
int PixelSamples[2];
```

curGfxOptions 563
max 513
RiScreenWindow 565
Round2Int 514
VERIFY_STATE 563

⟨*GfxOptions Constructor Implementation*⟩≡
```
PixelSamples[0] = PixelSamples[1] = 2;
```

The `RiFormat` function sets the *x* and *y* resolution of the final image as well as the *pixel aspect ratio*; this allows the user to specify pixel size for devices where the physical pixel-spacing in the *x* direction is different than in the *y* direction.

⟨*RI Function Declarations*⟩+≡
```
extern void RiFormat(int x, int y, Float aspect);
extern void RiFrameAspectRatio(Float aspect);
extern void RiScreenWindow(Float left, Float right, Float bottom,
    Float top);
extern void RiCropWindow(Float left, Float right, Float bottom, Float top);
extern void RiClipping(Float hither, Float yon);
extern void RiShutter(Float time0, Float time1);
extern void RiDepthOfField(Float fstop, Float focallen, Float focaldist);
```

⟨*Graphics Options*⟩+≡
```
Float PixelAspectRatio, FrameAspectRatio;
int XResolution, YResolution;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
PixelAspectRatio = 1.f;
FrameAspectRatio = 4.f / 3.f;
XResolution = 640;
YResolution = 480;
```

⟨*Update screen window from format values*⟩≡
```
if (curGfxOptions->ScreenExtentSet == false) {
    if (curGfxOptions->FrameAspectRatio >= 1)
        curGfxOptions->ScreenExtent =
            Extent2D(-curGfxOptions->FrameAspectRatio,
                curGfxOptions->FrameAspectRatio, -1, 1);
    else
        curGfxOptions->ScreenExtent =
            Extent2D(-1, 1, -1.f / curGfxOptions->FrameAspectRatio,
                1.f / curGfxOptions->FrameAspectRatio);
}
```

⟨*Graphics Options*⟩+≡
```
Extent2D ScreenExtent;
bool ScreenExtentSet;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
ScreenExtent = Extent2D(-4.f / 3.f, 4.f / 3.f, -1, 1);
ScreenExtentSet = false;
```

Furthermore, the user can provide their own screen window instead of letting it be computed implicitly by `RiFormat`.

563 curGfxOptions
27 Extent2D
564 FrameAspectRatio
563 VERIFY_STATE

⟨*RI Function Definitions*⟩+≡
```
void RiScreenWindow(Float left, Float right,
        Float bottom, Float top) {
    VERIFY_STATE(STATE_BEGIN, "RiScreenWindow");
    curGfxOptions->ScreenExtent = Extent2D(left, right,
        bottom, top);
    curGfxOptions->ScreenExtentSet = true;
}
```

In addition to specifying the overall image size, the user can select a rectangular subset of the image to render.

As usual, appropriate members are added to `GfxOptions`.

⟨*Graphics Options*⟩+≡
```
Extent2D Crop;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
Crop = Extent2D(0, 1, 0, 1);
```

`RiClipping` sets near and far clipping planes for the camera; these are two planes that are perpendicular to the *z* axis in camera space that delineate the extent of the scene that the camera will consider.

⟨*Graphics Options*⟩+≡
```
Float ClipHither, ClipYon;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
ClipHither = RAY_EPSILON;
ClipYon = 1e30f;
```

For scenes with moving objects or cameras, `RiShutter` lets the range of time that the shutter is open be specified.

⟨*Graphics Options*⟩+≡
```
Float ShutterStart, ShutterEnd;
```

By default, the shutter is only open for an instant, so no motion blur is seen.

⟨*GfxOptions Constructor Implementation*⟩+≡
```
ShutterStart = ShutterEnd = 0;
```

There is also a function to set depth of field parameters for `Camera`s that support this effect.

⟨*Graphics Options*⟩+≡
```
Float LensRadius, FocalDistance;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
LensRadius = 0.f;
FocalDistance = INFINITY;
```

After all those boring functions that just set values in `GfxOptions`, we can now move on to `RiProjection`, which is a bit more interesting.

We first verify that `RiBegin()` has been called but not yet `RiWorldBegin()`. We then record the name of the camera type the user asked for and its parameters. We will later use these to load and initialize the appropriate `Camera` from disk.

curGfxOptions 563
curTransform 575
INFINITY 514
ParamSet 542
Transform 32
VERIFY_STATE 563

⟨*RI Function Definitions*⟩+≡
```
void RiProjectionV(const char * name, int nArgs, const char * tokens[],
        void * params[]) {
    VERIFY_STATE(STATE_BEGIN, "RiProjection");
    curGfxOptions->cameraName = name;
    curGfxOptions->cameraParams.init(nArgs, tokens, params);
    curTransform[0] = Transform();
}
```

⟨*Graphics Options*⟩+≡
```
string cameraName;
mutable ParamSet cameraParams;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
cameraName = "orthographic";
```

When `RiWorldBegin` is called, the camera-to-world transformation is set from the current transformation that the user has specified. We grab this transformation then, store it away in the `GfxOptions`, and reset the current transformation to the identity transformation.

⟨*RiWorldBegin initialization*⟩≡
```
for (int i = 0; i < MOTION_LEVELS; ++i) {
    curGfxOptions->WorldToCamera[i] = curTransform[i];
    curTransform[i] = Transform();
}
```

⟨*Graphics Options*⟩+≡
```
Transform WorldToCamera[MOTION_LEVELS];
```

Display

There are a number of further options that affect the final image; they include how the film reacts to exposure by light, how the image samples are filtered and reconstructed, quantization and image output settings, etc. A handful of additional RI functions handles setting these options and storing them in `GfxOptions`.

⟨*RI Function Declarations*⟩+≡
```
extern void RiExposure(Float gain, Float gamma);
```

⟨*Graphics Options*⟩+≡
```
Float Gain, Gamma;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
Gain = Gamma = 1.;
```

The pixel filter function is set by passing the name of the filter function and the width of the filter's extent.

⟨*RI Function Declarations*⟩+≡
```
extern void RiPixelFilter(const char *filter, Float xwidth, Float ywidth,
        int nArgs, const char * tokens[], void * params[]);
```

⟨*Graphics Options*⟩+≡
```
string filterName;
Float FilterXWidth, FilterYWidth;
ParamSet FilterParams;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
filterName = "mitchell";
FilterXWidth = FilterYWidth = 2.;
```

568 ColorQuantDither
568 ColorQuantMax
568 ColorQuantMin
568 ColorQuantOne
563 curGfxOptions
498 Error
542 ParamSet

The `RiQuantize()` function sets parameter values for the color and depth quantization parts of the imaging pipeline (see Section 8.5.) After checking whether it's the color quantization or the depth quantization parameters to be set, it updates the appropriate fields in the `Image`'s options.

⟨*RI Function Definitions*⟩+≡
```
void RiQuantize(const char * type, int one, int minimum,
        int maximum, Float ditheramp) {
    if (strcmp(type, "rgba") == 0) {
        curGfxOptions->ColorQuantOne = one;
        curGfxOptions->ColorQuantMin = minimum;
        curGfxOptions->ColorQuantMax = maximum;
        curGfxOptions->ColorQuantDither = ditheramp;
    }
    else if (strcmp(type, "z") == 0) {
        RI_UNIMP();
    }
    else
        Error("Unknown type %s passed to RiQuantize()", type);
}
```

⟨*Graphics Options*⟩+≡
```
int ColorQuantOne, ColorQuantMin, ColorQuantMax;
Float ColorQuantDither;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
ColorQuantOne = 255;
ColorQuantMin = 0;
ColorQuantMax = 255;
ColorQuantDither = 0.5;
```

The `RiDisplay()` function tells what to do with the final image. Since we only support writing TIFFs out to disk, all that there is to do is to figure out which channels the user wants us to save (RGB, alpha, depth, etc.) and what filename to store the image in. Again, we will omit the non-vector version of `RiDisplay()` since it fits the `RiProjection()` mold.

⟨*RI Function Declarations*⟩+≡
```
extern void RiDisplayV(char *name, const char * type, const char * mode, int nA
    const char * tokens[], void * parameters[]);
```

⟨*Graphics Options*⟩+≡
```
string DisplayType;
string DisplayName;
bool displayRGB, displayA, displayZ;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
DisplayType = "tiff";
DisplayName = "lrt.tiff";
displayRGB = displayA = true;
displayZ = false;
```

acceleratorName 569
acceleratorParams 569
curGfxOptions 563
ParamSet 542

Miscellaneous

⟨*RI Function Definitions*⟩+≡
```
void RiSamplerV(const char * name, int n, const char * tokens[],
        void * params[]) {
    curGfxOptions->samplerName = name;
    curGfxOptions->samplerParams.init(n, tokens, params);
}
```

⟨*Graphics Options*⟩+≡
```
string samplerName;
ParamSet samplerParams;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
samplerName = "bestcandidate";
```

⟨*RI Function Definitions*⟩+≡
```
void RiAcceleratorV(const char * name, int n, const char * tokens[],
        void * params[]) {
    curGfxOptions->acceleratorName = name;
    curGfxOptions->acceleratorParams.init(n, tokens, params);
}
```

⟨*Graphics Options*⟩+≡
```
string acceleratorName;
ParamSet acceleratorParams;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
acceleratorName = "grid";
```

XXXX where should this go?? XXXX

⟨*GfxOptions Method Declarations*⟩+≡
```
Scene *MakeScene() const;
```

⟨*RI Function Definitions*⟩+≡
```
Scene *GfxOptions::MakeScene() const {
    ⟨Initialize filter with pixel filter⟩
    ⟨Initialize sampler from API settings⟩
    ⟨Initialize film and camera from API settings⟩
    ⟨Initialize displayInfo from API settings⟩
    ⟨Initialize surfaceIntegrator from API settings⟩
    ⟨Initialize volumeIntegrator from API settings⟩
    ⟨Initialize accelerator from API settings⟩
    if (!camera || !sampler || !film || !camera || !accelerator ||
        !displayInfo || !surfaceIntegrator || !volumeIntegrator) {
        Severe("Unable to create scene due to missing DSOs");
        return NULL;
    }
    Scene *ret = new Scene(camera, surfaceIntegrator, volumeIntegrator,
        sampler, accelerator, lights, volumeRegions, displayInfo);
    lights.erase(lights.begin(), lights.end());
    volumeRegions.erase(volumeRegions.begin(), volumeRegions.end());
    return ret;
}
```

⟨*Initialize* filter *with pixel filter*⟩≡
```
Filter *filter = CreateFilter(filterName, SearchPath, FilterParams,
    FilterXWidth, FilterYWidth);
```

⟨*Initialize* sampler *from API settings*⟩≡
```
Sampler *sampler = CreateSampler(samplerName, SearchPath, samplerParams,
        XResolution, YResolution, PixelSamples[0],
        PixelSamples[1], Crop, filter);
```

⟨*Initialize* film *and* camera *from API settings*⟩≡
```
Film *film = new Film(XResolution, YResolution, Crop);
cameraParams.AddFloat("_ShutterOpen", &ShutterStart);
cameraParams.AddFloat("_ShutterClose", &ShutterEnd);
cameraParams.AddFloat("_ClipHither", &ClipHither);
cameraParams.AddFloat("_ClipYon", &ClipYon);
cameraParams.AddFloat("_LensRadius", &LensRadius);
cameraParams.AddFloat("_FocalDistance", &FocalDistance);
cameraParams.AddFloat("_PixelAspectRatio", &PixelAspectRatio);
cameraParams.AddFloat("_ScreenExtent", (Float *)&ScreenExtent,
    PARAM_TYPE_UNIFORM, 4);
Camera *camera = CreateCamera(cameraName, SearchPath, cameraParams,
    WorldToCamera[0], film);
```

173 film
209 Filter
567 filterName
567 FilterParams
567 FilterXWidth
567 FilterYWidth
563 GfxOptions
579 lights
542 ParamSet
564 PixelSamples
198 Sampler
568 samplerName
568 samplerParams
5 Scene
498 Severe
579 volumeRegions

⟨*Initialize* displayInfo *from API settings*⟩≡
```
DisplayInfo *displayInfo = new DisplayInfo;
if (ToneMapping != "")
    displayInfo->toneMap = CreateToneMap(ToneMapping, SearchPath,
        ToneMapParams);
displayInfo->gain = Gain;
displayInfo->invGamma = 1.f / Gamma;
displayInfo->integerFormat = ColorQuantOne != 0;
displayInfo->maxDisplayValue = ColorQuantMax;
displayInfo->ditherAmount = ColorQuantDither;
displayInfo->filename = DisplayName;
```

⟨*Initialize* surfaceIntegrator *from API settings*⟩≡
```
SurfaceIntegrator *surfaceIntegrator = CreateSurfaceIntegrator(surfIntegratorNa
    curGfxOptions->SearchPath, surfIntegratorParams);
```

⟨*Initialize* volumeIntegrator *from API settings*⟩≡
```
VolumeIntegrator *volumeIntegrator = CreateVolumeIntegrator(volIntegratorName,
    curGfxOptions->SearchPath, volIntegratorParams);
```

⟨*Initialize* accelerator *from API settings*⟩≡
```
Primitive *accelerator = CreateAccelerator(acceleratorName,
    SearchPath, primitives, acceleratorParams);
if (!accelerator) {
    ParamSet ps;
    accelerator = CreateAccelerator("grid", SearchPath,
        primitives, ps);
}
Assert(accelerator);
primitives.erase(primitives.begin(), primitives.end());
```

Updates SearchPath, handles stuff like ampersand to refer to original path, etc.

⟨*RI Function Declarations*⟩+≡
```
extern void RiSearchPath(const char *path);
```

⟨*Graphics Options*⟩+≡
```
string SearchPath;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
#ifdef LRT_BUILDDIR
SearchPath = ".:" LRT_BUILDDIR ":./texture";
#else
SearchPath = ".";
#endif
```

⟨*RI Function Declarations*⟩+≡
```
extern void RiToneMap(const char *name, int n, const char *tokens[],
    void *params[]);
```

⟨*Graphics Options*⟩+≡
```
string ToneMapping;
ParamSet ToneMapParams;
```

acceleratorName 569
acceleratorParams 569
AddFloat 545
Assert 498
cameraName 566
cameraParams 566
ColorQuantDither 568
ColorQuantMax 568
ColorQuantOne 568
curGfxOptions 563
DisplayInfo 241
DisplayName 568
film 173
Gain 567
Gamma 567
PARAM_TYPE_UNIFORM 544
ParamSet 542
PixelAspectRatio 564
Primitive 9
primitives 579
ScreenExtent 565
ShutterEnd 566
ShutterStart 566
ToneMapParams 571
ToneMapping 571
VolumeIntegrator 484

⟨*RI Function Definitions*⟩+≡
```
void RiSurfaceIntegratorV(const char *name, int n, const char * tokens[],
        void * params[]) {
    curGfxOptions->surfIntegratorName = name;
    curGfxOptions->surfIntegratorParams.init(n, tokens, params);
}
```

⟨*RI Function Definitions*⟩+≡
```
void RiVolumeIntegratorV(const char *name, int n, const char * tokens[],
        void * params[]) {
    curGfxOptions->volIntegratorName = name;
    curGfxOptions->volIntegratorParams.init(n, tokens, params);
}
```

⟨*Graphics Options*⟩+≡
```
string surfIntegratorName, volIntegratorName;
ParamSet surfIntegratorParams, volIntegratorParams;
```

⟨*GfxOptions Constructor Implementation*⟩+≡
```
surfIntegratorName = "whitted";
volIntegratorName = "null";
```

563 curGfxOptions
562 currentApiState
542 ParamSet
502 StatsPrint
563 VERIFY_STATE

D.3 Graphics State

XXX intro/ideas of hierarchical graphic state !! XXXX

After the user has set up the overall settings for the scene (camera position, image output options, etc), they call `RiWorldBegin`. This tells the renderer that

⟨*RI Function Definitions*⟩+≡
```
void RiWorldBegin() {
    VERIFY_STATE(STATE_BEGIN, "RiWorldBegin");
    currentApiState = STATE_WORLD_BEGIN;
    ⟨RiWorldBegin initialization⟩

}
```

XXX actually, want to clean out the primitives and lights from gfx options only

⟨*RI Function Definitions*⟩+≡
```
void RiWorldEnd() {
    ⟨Check for valid WorldEnd state⟩
    ⟨Create scene and render⟩
    currentApiState = STATE_BEGIN;
    curGfxOptions->WorldEnd();
    ⟨Print per-frame statistics⟩
}
```

⟨*Print per-frame statistics*⟩≡
```
StatsPrint(stderr);
```

⟨*Check for valid WorldEnd state*⟩≡
```
while (hierarchicalState.size()) {
    char c = hierarchicalState.back();
    if (c == 't') Error("Missing end to RiTransformBegin");
    else if (c == 'a') Error("Missing end to RiAttributeBegin");
    else Severe("Internal error in gfx state management");
    hierarchicalState.pop_back();
}
```

Scene destructor frees up the memory for primitives and lights.

Attributes

As the stream of commands comes in that specifies the scene geometry, a variety of attributes can be updated as well. These include information about the current surface shader, the object to world transformation, etc. When a geometric primitive or light source is then added to the scene, various parts of the current set of attributes are used to initialize their specific parameters.

The current set of active attributes can be managed with the *attribute stack*. This allows the user to *push* the current set of attributes, make changes to their values and then later *pop* back to the previously pushed attribute values. For example, a scene description file might have lines such as:

Error 498
RiAttributeBegin 574
RiTransformBegin 573
Severe 498
size 494
Transform 32

```
Surface "matte"
AttributeBegin # push current attributes
Surface "plastic"
Translate 5 0 0
Sphere 1 -1 1 360 # this sphere is plastic and translated
AttributeEnd # pop attributes
Sphere 1 -1 1 360 # this sphere is matte and not translated
```

Changes to attributes made inside an `AttributeBegin`/`AttributeEnd` block are forgotten at the end of the block. There are also `TransformBegin` and `TransformEnd` calls that only push and pop the current transformation matrix; they are more lightweight than the ones that save the entire attribute state.

We store pushed transformations in a list of `Transform`s. We also keep track of a list of characters, "t" or "a" to keep track of the nesting of transform and attribute begin and end calls. This ensures that we report an error and don't do something invalid if the user gives incorrectly nested RIB like:

```
AttributeBegin
TransformBegin
AttributeEnd
```

⟨*API Static Data*⟩+≡
```
static vector<Transform> transformStack[MOTION_LEVELS];
static vector<char> hierarchicalState;
```

⟨*RI Function Definitions*⟩+≡
```
void RiTransformBegin() {
    for (int i = 0; i < MOTION_LEVELS; ++i)
        transformStack[i].push_back(curTransform[i]);
    hierarchicalState.push_back('t');
}
```

⟨*RI Function Definitions*⟩+≡
```
void RiTransformEnd() {
    if (!transformStack[0].size() ||
        hierarchicalState.back() != 't') {
        Error("Unmatched RiTransformEnd encountered.  Ignoring it.");
        return;
    }
    for (int i = 0; i < MOTION_LEVELS; ++i) {
        curTransform[i] = transformStack[i].back();
        transformStack[i].pop_back();
    }
    hierarchicalState.pop_back();
}
```

We store the rest of set of current attributes in the `GfxState` structure. As with `GfxOptions`, we'll be adding members to it throughout this section.

⟨*API Local Classes*⟩+≡
```
struct GfxState {
    GfxState();
    ⟨Graphics State⟩
    ⟨Graphics State Methods⟩
};
```

⟨*API Local Classes*⟩+≡
```
GfxState::GfxState() {
    ⟨GfxState Constructor Implementation⟩
}
```

When `RiWorldBegin` is called, we initialize the current graphics state to hold default values.

⟨*RiWorldBegin initialization*⟩+≡
```
curGfxState = GfxState();
```

We also keep a list of `GfxStates`; when `RiAttributeBegin` is called, we copy the current `GfxState` and push it on to the `list`. Attribute end pops the state to restore to back off of the list.

⟨*API Static Data*⟩+≡
```
static GfxState curGfxState;
static vector<GfxState> gstates;
```

Pushing and popping attribute state also implicitly pushes and pops the transformation stack, so we make the `RiTransformBegin` call for starters.

575 curTransform
498 Error
572 hierarchicalState
494 push_back
494 size
572 transformStack

⟨*RI Function Definitions*⟩+≡
```
void RiAttributeBegin() {
    RiTransformBegin();
    VERIFY_STATE(STATE_WORLD_BEGIN, "RiAttributeBegin");
    gstates.push_back(curGfxState);
    hierarchicalState.push_back('a');
}
```

⟨*RI Function Definitions*⟩+≡
```
void RiAttributeEnd() {
    VERIFY_STATE(STATE_WORLD_BEGIN, "RiAttributeEnd");
    if (!gstates.size() || hierarchicalState.back() != 'a') {
        Error("Unmatched RiAttributeEnd encountered.  Ignoring it.");
        return;
    }
    curGfxState = gstates.back();
    gstates.pop_back();
    hierarchicalState.pop_back();
    RiTransformEnd();
}
```


Assert 498
curGfxState 573
Error 498
gstates 573
hierarchicalState 572
push_back 494
RiTransformBegin 573
RiTransformEnd 573
size 494
Spectrum 155
surfaceParams 575
VERIFY_STATE 563

We will also track a current color; when we create materials, we will make sure that their color is multiplied by the spectrum specified here.

⟨*RI Function Definitions*⟩+≡
```
void RiColor(Float *Cs) {
    Assert(COLOR_SAMPLES == 3);
    curGfxState.color = Spectrum(Cs);
}
void RiOpacity(Float *Os) {
    RI_UNIMP();
}
```

⟨*Graphics State*⟩≡
```
Spectrum color;
```

⟨*GfxState Constructor Implementation*⟩≡
```
color = Spectrum(1.);
```

The current material is specified by `RiSurface`. We gather up all of the additional parameters and their values passed along with the name of the material and store them away in the graphics state. When we later go create the material, we'll use these to set up its textures.

⟨*RI Function Definitions*⟩+≡
```
void RiSurfaceV(const char * name, int n, const char * tokens[],
        void * params[]) {
    curGfxState.surfaceParams.init(n, tokens, params);
    curGfxState.surface = name;
}
```

⟨*Graphics State*⟩+≡
```
ParamSet surfaceParams;
string surface;
```

⟨*GfxState Constructor Implementation*⟩+≡
```
surface = "matte";
```

⟨*RI Function Definitions*⟩+≡
```
void RiDisplacementV(const char * name, int n, const char * tokens[],
    void * params[]) {
    curGfxState.displaceParams.init(n, tokens, params);
    curGfxState.displacement = name;
}
```

⟨*Graphics State*⟩+≡
```
ParamSet displaceParams;
string displacement;
```

⟨*GfxState Constructor Implementation*⟩+≡
```
displacement = "";
```

573 curGfxState
542 ParamSet
32 Transform

D.4 Transformations

There are a number of basic functions to update the current transformation matrix; they all basically compose the current transformation matrix with a new transformation. These functions are slightly complicated by the need to be able to specify multiple transformations for moving objects that are at different positions at different points in time. We store up to two current transformations, updating only one of them when a transformation call is made, depending on which transform of a moving object is being specified. If the object is not moving, we just update the first of the two of them.

⟨*API Includes*⟩≡
```
#define MOTION_LEVELS 2
```

⟨*API Static Data*⟩+≡
```
static int motionLevel = 0;
static bool inMotionBlock = false;
static Transform curTransform[MOTION_LEVELS];
```

⟨*System-wide Initialization*⟩+≡
```
for (int i = 0; i < MOTION_LEVELS; ++i)
    curTransform[i] = Transform();
```

The transformations of moving objects are given within motion blocks, like:

```
MotionBegin [ 10 11 ]
Translate 1 0 0
Translate 0 1 0
MotionEnd
```

This specifies that at time 10, the first translation should be appended to the current transformation and at time 11, the second translation should be. The `RiMotionBegin` function takes an array of time values that specifies how many transformations will be given.

⟨*RI Function Definitions*⟩+≡
```
void RiMotionBeginV(int N, Float times[]) {
    Assert(!inMotionBlock);
    inMotionBlock = true;
    motionLevel = 0;
    if (N > 2)
        Warning("Only two levels in motion block will be used.");
}
```

⟨*RI Function Definitions*⟩+≡
```
void RiMotionEnd() {
    if (!inMotionBlock)
        Error("Unmatched MotionEnd statement");
    inMotionBlock = false;
}
```

Assert 498
curTransform 575
Error 498
inMotionBlock 575
motionLevel 575
Transform 32
Warning 497

The actual transformation functions, then, all start out with a little housekeeping for the motion-related stuff, then apply the given transformation, and then do motion-related cleanup.

⟨*RI Function Definitions*⟩+≡
```
void RiIdentity() {
    ⟨Prepare for motion transform⟩
    curTransform[xform] = Transform();
    ⟨Update transform for motion block⟩
}
```

If there is no current motion block, then we just update `curTransform[0]`. Otherwise, we update the appropriate one depending on how many transforms have been given in this block so far.

⟨*Prepare for motion transform*⟩≡
```
int xform = 0;
if (inMotionBlock)
    xform = motionLevel;
if (motionLevel > MOTION_LEVELS) {
    Warning("Only %d motion levels are supported. Ignoring.",
        motionLevel);
    return;
}
```

⟨*Update transform for motion block*⟩≡
```
if (inMotionBlock)
    ++motionLevel;
```

⟨*RI Function Declarations*⟩+≡
```
extern void RiTransform(Float transform[16]);
extern void RiConcatTransform(Float transform[16]);
extern void RiPerspective(Float fov);
extern void RiRotate(Float angle, Float dx, Float dy, Float dz);
extern void RiScale(Float sx, Float sy, Float sz);
extern void RiLookAt(Float ex, Float ey, Float ez, Float lx, Float ly,
    Float lz, Float ux, Float uy, Float uz);
```

⟨*RI Function Definitions*⟩+≡
```
void RiTranslate(Float dx, Float dy, Float dz) {
    ⟨Prepare for motion transform⟩
    curTransform[xform] = curTransform[xform] * Translate(Vector(dx, dy, dz));
    ⟨Update transform for motion block⟩
}
```

D.5 Geometric Primitives

We can now introduce the RI routines for describing the geometric primitives in the scene. These are all mostly similar; they process the function arguments, create appropriate `Shape` objects, and pass them on to a routine that creates `GeometricPrimitives` that include the material, etc, that is bound to the shape.

575 curTransform
71 nverts
542 ParamSet
21 Point
35 Translate
16 Vector

The first geometric primitive function that we'll implement is `RiPolygon`. This call allows the user to specify polygons with arbitrary numbers of vertices, so long as the the polygon is convex. Here, we will tessellate the given polygon and create a `TriangleMesh`.

⟨*RI Function Definitions*⟩+≡
```
void RiPolygonV(int nverts, int n, const char * tokens[],
        void * params[]) {
    ParamSet geomParams(n, tokens, params, nverts);
    ⟨Get vertex positions from parameters⟩
    ⟨Tessellate single convex polygon⟩
}
```

We also walk through the parameters to find the one named "P", which gives the vertex positions.

⟨*Get vertex positions from parameters*⟩≡
```
Point *P = NULL;
int i;
for (i = 0; i < n; ++i)
    if (strcmp(tokens[i], "P") == 0)
        P = (Point *)params[i];
```

We now go ahead and tessellate the polygon into triangles. All triangles have the same first vertex (arbitrarily chosen as the first point in the `P` array). The `indices` array holds the three vertex indices for each triangle in the mesh, stored in turn. The `TriangleMesh` constructor then uses them to know the topology of the mesh.

⟨*Tessellate single convex polygon*⟩≡
```
int nTris = nverts-2;
int nIndices = 3*nTris;
int *indices = (int *)alloca(nIndices * sizeof(int));
int *nv = (int *)alloca(nTris * sizeof(int));
for (i = 0; i < nTris; ++i)
    nv[i] = 3;
for (i = 0 ; i < nverts-2 ; i++) {
    indices[3*i] = 0;
    indices[3*i+1] = i+2;
    indices[3*i+2] = i+1;
}
geomParams.AddInt("_ntris", &nTris);
geomParams.AddInt("_nverts", &nverts);
geomParams.AddInt("_vertexIndices", indices,
    PARAM_TYPE_UNIFORM, nIndices);
curGfxState.AddShape(
    CreateShape("trianglemesh", curGfxOptions->SearchPath,
        curTransform[motionLevel], geomParams), geomParams);
```

After the primitive creation methods have created a new `Shape`, they pass it along to `AddShape` for further processing.

⟨*API Static Methods*⟩+≡
```
void GfxState::AddShape(const Reference<Shape> &shape,
        ParamSet &geomParams) {
    if (!shape) return;
    AreaLight *area = NULL;
    ⟨Initialize area light for shape⟩
    ⟨Create N and S textures, if needed⟩
    ⟨Initialize material for shape⟩
    Reference<Primitive> prim = new GeometricPrimitive(shape,
        mtl, area, Ntex, Stex);
    curGfxOptions->primitives.push_back(prim);
    if (area != NULL) {
        ⟨Create area lights given number of light samples⟩
    }
    geomParams.ReportUnused();
}
```

⟨*Create area lights given number of light samples*⟩≡
```
int nSamples = areaLightParams.FindOneInt("nsamples", 1);
if (nSamples == 1)
    curGfxOptions->lights.push_back(area);
else
    for (int i = 0; i < nSamples; ++i)
        curGfxOptions->lights.push_back(new MultiAreaLight(area, nSamples));
```

All of the `Primitive`s and `Light`s that are defined are stored in a big vector as we process the RIB file; they are later passed off to the `Scene` when it is created.

AddInt 545
alloca 495
AreaLight 368
areaLightParams 581
curGfxOptions 563
curGfxState 573
curTransform 575
FindOneInt 546
GfxState 573
lights 579
motionLevel 575
MultiAreaLight 370
nverts 75
PARAM_TYPE_UNIFORM 544
ParamSet 542
Primitive 9
primitives 579
push_back 494
Reference 509
ReportUnused 547

⟨*Graphics Options*⟩+≡
```
mutable vector<Reference<Primitive> > primitives;
mutable vector<Light *> lights;
mutable vector<VolumeRegion *> volumeRegions;
```

⟨*Create* N *and* S *textures, if needed*⟩≡
```
int type, narray;
Texture<Normal> *Ntex = NULL;
const Normal *N = geomParams.FindNormal("N", &type, &narray);
if (N) {
    Assert(type & PARAM_TYPE_NORMAL);
    int count = geomParams.TypeToNum(type);
    Normal *Nw = new Normal[count];
    for (int i = 0; i < count; ++i)
        Nw[i] = curTransform[motionLevel](N[i]);

    if (type & PARAM_TYPE_UNIFORM)
        Ntex = new ConstantTexture<Normal>(*Nw);
    else if (type & PARAM_TYPE_VARYING) {
        Ntex = new BilerpTexture<Normal>(new IdentityMapping2D,
            Nw[0], Nw[1], Nw[2], Nw[3]);
    }
    else if (type & PARAM_TYPE_VERTEX) {
        if (geomParams.nVertex != 0)
            Ntex = new VertexTexture<Normal>(Nw,
                geomParams.nVertex);
        else
            Error("Vertex texture not supported for shape");
    }
    delete[] Nw;
}
```
⟨*Create* S *texture*⟩

We need to create the `Material` that is bound to the shape. We first determine which one to create based on the string stored in `GfxState::surface`, which was set by `RiSurface`.

⟨*Initialize material for shape*⟩≡
```
Texture<Float> *displace = CreateBump(displacement, curGfxOptions->SearchPath,
    curTransform[0], geomParams, displaceParams);
Reference<Material> mtl = CreateMaterial(surface, curGfxOptions->SearchPath,
    curTransform[0], geomParams, surfaceParams, color, displace);
if (!mtl)
    mtl = CreateMaterial("plastic", curGfxOptions->SearchPath,
        curTransform[0], geomParams, surfaceParams, color,
        displace);
Assert(mtl);
```

Each of the various materials takes a number of parameters to set its properties. The binding of these parameters is a bit tricky; consider the "matte" material, which

498 Assert
330 BilerpTexture
324 ConstantTexture
563 curGfxOptions
575 curTransform
498 Error
547 FindNormal
327 IdentityMapping2D
358 Light
303 Material
575 motionLevel
23 Normal
544 PARAM_TYPE_NORMAL
544 PARAM_TYPE_UNIFORM
544 PARAM_TYPE_VARYING
544 PARAM_TYPE_VERTEX
9 Primitive
509 Reference
575 surfaceParams
323 Texture
547 TypeToNum
332 VertexTexture
383 VolumeRegion

takes a color texture named “Kd”. Matte defines a default value for Kd that can be overridden when the RiSurface call is made.

```
Surface "matte" "color Kd" [ .5 1 .5 ]
```

However, this value can then be overridden again when the primitive is created:

```
Surface "matte" "color Kd" [ 1 0 0 ]
Sphere 1 -1 1 360 # red sphere
Sphere 1 -1 1 360 "color Kd" [ 0 1 0 ] # green sphere
```

Therefore, we create a ParamSet from the parameters given when the material is defined in an RiSurface call. When creating the Material, however, we first look for parameter values in geomParams, which was set from the parameters to the primitive-creation API call. If this doesn’t have a value, we fall back to the value in GfxState::surfaceParams, and from there to a default value.

The rest of the materials are analogous.

The routines to create the various quadrics are all quite simple. We will only include RiSphere here, since the rest are quite similar.

⟨*RI Function Definitions*⟩+≡

```
void RiSphereV(Float radius, Float zmin, Float zmax,
        Float thetaMax, int n, const char * tokens[],
        void * params[]) {
    ParamSet geomParams(n, tokens, params);
    geomParams.AddFloat("_radius", &radius);
    geomParams.AddFloat("_zmin", &zmin);
    geomParams.AddFloat("_zmax", &zmax);
    geomParams.AddFloat("_thetamax", &thetaMax);
    curGfxState.AddShape(
        CreateShape("sphere", curGfxOptions->SearchPath,
            curTransform[motionLevel], geomParams),
        geomParams);
}
```

AddFloat 545
AddShape 578
curGfxOptions 563
curGfxState 573
curTransform 575
motionLevel 575
ParamSet 542

D.6 Light Sources

Finally, we'll define the routines that allow the user to specify light sources for the scene. RI provides two ways of doing this: the first, `RiLightSource` defines a light source that doesn't have geometry associated with it (e.g. a point light or a directional light). The second, `RiAreaLightSource` specifies an active are light source; the primitives that follow it up to the end of the current attribute block are treated as emitting geometry as given by the area light description.

⟨*RI Function Definitions*⟩+≡
```
void *RiLightSourceV(const char * name, int nArgs,
            const char *tokens[], void * params[]) {
    ParamSet paramSet(nArgs, tokens, params);
    bool shadows = paramSet.FindOneInt("shadows", 1) != 0;
    Light *lt = CreateLight(name, curGfxOptions->SearchPath,
        shadows, curTransform[motionLevel],
        paramSet);
    ⟨Add new light to graphics state⟩
    return lt;
}
```

⟨*Add new light to graphics state*⟩≡
```
if (lt == NULL)
    Error("RiLightSource: light type '%s' unknown.", name);
else
    curGfxOptions->lights.push_back(lt);
```

563 curGfxOptions
573 curGfxState
575 curTransform
498 Error
546 FindOneInt
358 Light
579 lights
575 motionLevel
542 ParamSet
494 push_back

When an area light is specified, we can't create it immediately–we need to wait for the upcoming primitives which will define the light source's geometry. Therefore, as in `RiSurface`, we just save away the name of the area light source type and the parameters given here.

⟨*RI Function Definitions*⟩+≡
```
void *RiAreaLightSourceV(const char *name, int n,
        const char *tokens[], void *params[] ) {
    curGfxState.areaLightParams.init(n, tokens, params);
    curGfxState.areaLight = name;
    return NULL;
}
```

⟨*Graphics State*⟩+≡
```
ParamSet areaLightParams;
string areaLight;
```

We can now define the fragment ⟨*Initialize area light for shape*⟩ from the `GfxState::AddShape` function. This just takes the area light information from `RiAreaLightSource` and the `Shape` passed in to `GfxState::AddShape` to create an `AreaLight` object.

⟨*Initialize area light for shape*⟩≡
```
if (areaLight != "") {
    bool shadows = areaLightParams.FindOneInt("shadows", 1) != 0;
    area = CreateAreaLight(areaLight, curGfxOptions->SearchPath,
        shadows, curTransform[motionLevel],
        areaLightParams, shape);
}
```

D.7 Volumes

⟨*RI Function Definitions*⟩+≡
```
void RiVolumeV(const char *name, int nArgs, const char *tokens[],
        void *params[]) {
    ParamSet paramSet(nArgs, tokens, params);
    VolumeRegion *vr = CreateVolumeRegion(name, curGfxOptions->SearchPath,
        curTransform[motionLevel], paramSet);
    if (vr) curGfxOptions->volumeRegions.push_back(vr);
}
```

areaLight 581
areaLightParams 581
curGfxOptions 563
curTransform 575
FindOneInt 546
motionLevel 575
ParamSet 542
push_back 494
VolumeRegion 383
volumeRegions 579

Further Reading

RenderMan companion(Ups89)
RI Spec(Pix89)
Advanced RMan book
OpenGL stuff

Bibliography

Anthony A. Apodaca and Larry Gritz, *Advanced RenderMan: creating CGI for motion pictures*, Morgan Kaufmann, 2000.

James Arvo and David B. Kirk, *Fast ray tracing by ray classification*, Computer Graphics (SIGGRAPH '87 Proceedings) (Maureen C. Stone, ed.), vol. 21, July 1987, pp. 55–64.

John Amanatides, *Ray tracing with cones*, Computer Graphics (SIGGRAPH '84 Proceedings) (Hank Christiansen, ed.), vol. 18, July 1984, pp. 129–135.

———, *Algorithms for the detection and elimination of specular aliasing*, Graphics Interface '92, May 1992, pp. 86–93.

Arthur Appel, *Some techniques for shading machine renderings of solids*, AFIPS 1968 Spring Joint Computer Conf., vol. 32, 1968, pp. 37–45.

Michael Ashikhmin, Simon Premoze, and Peter S. Shirley, *A microfacet-based brdf generator*, Proceedings of ACM SIGGRAPH 2000, Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, July 2000, ISBN 1-58113-208-5, pp. 65–74.

James Arvo, *Backward ray tracing*, August 1986.

Michael Ashikhmin and Peter Shirley, *An anisotropic Phong light reflection model*, June 2000, Technical report UUCS-00-014.

———, *An anisotropic Phong BRDF model*, Journal of Graphics Tools **5** (2002), no. 2, 25–32.

Michael Ashikhmin, *A tone mapping algorithm for high contrast images*, The proceedings of 13th Eurographics Workshop on Rendering (Pisa, Italy), June 2002, pp. 145–155.

Kendall Atkinson, *Elementary numerical analysis*, John Wiley & Sons, 1993.

James F. Blinn, *Simulation of wrinkled surfaces*, Computer Graphics (SIGGRAPH '78 Proceedings), vol. 12, August 1978, pp. 286–292.

———, *A generalization of algebraic surface drawing*, ACM Transactions on Graphics **1** (1982), no. 3, 235–256.

———, *Light reflection functions for simulation of clouds and dusty surfaces*, Compute Graphics **16** (1982), no. 3, 21–29.

Mark R. Bolin and Gary W. Meyer, *A perceptually based adaptive sampling algorithm*, Proceedings of SIGGRAPH 98, Computer Graphics Proceedings, Annual Conference Series, July 1998, pp. 299–310.

J. F. Blinn and M. E. Newell, *Texture and reflection in computer generated images*, Communications of the ACM **19** (1976), 542–546.

R. N. Bracewell, *The Fourier transform and its applications*, McGraw-Hill, 1968.

Philippe Blasi, Bertrand Le Saëc, and Christophe Schlick, *A rendering algorithm for discrete volume density objects*, no. 3, 201–210.

N. Bhate and A. Tokuta, *Photorealistic volume rendering of media with directional scattering*, May 1992, pp. 227–245.

R. Creighton Buck, *Advanced calculus*, McGraw–Hill, New York, NY, 1978.

Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley, *Composing high-performance memory allocators*, SIGPLAN Conference on Programming Language Design and Implementation, 2001, pp. 114–124.

———, *Reconsidering custom memory allocation*, Proceedings of ACM OOPSLA 2002, 2002.

Robert L. Cook, Loren Carpenter, and Edwin Catmull, *The REYES image rendering architecture*, Computer Graphics (Proceedings of SIGGRAPH 87) (Anaheim, California), no. 4, July 1987, pp. 95–102.

B. Calder, K. Chandra, S. John, and T. Austin, *Cache-conscious data placement*, Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII) (San Jose), 1998.

Trishul M. Chilimbi, Bob Davidson, and James R. Larus, *Cache-conscious structure definition*, SIGPLAN Conference on Programming Language Design and Implementation, 1999, pp. 13–24.

S. Chandrasekar, *Radiative transfer*, Dover Publications, New York, 1960, Originally published by Oxford University Press, 1950.

Trishul M. Chilimbi, Mark D. Hill, and James R. Larus, *Cache-conscious structure layout*, SIGPLAN Conference on Programming Language Design and Implementation, 1999, pp. 1–12.

K. Chiu, M. Herf, P. Shirley, S. Swamy, C. Wang, and K. Zimmerman, *Spatially nonuniform scaling functions for high contrast images*, Graphics Interface '93 (Toronto, Ontario, Canada), Canadian Information Processing Society, May 1993, pp. 245–253.

O. D. Chvolson, *Grundzüge einer matematischen teorie der inneren diffusion des lichtes*, Izv. Peterburg. Academii Nauk **33** (1890), 221–265.

Brian Cabral, Nelson Max, and Rebecca Springmeyer, *Bidirectional reflection functions from surface bump maps*, Computer Graphics (SIGGRAPH '87 Proceedings), vol. 21, July 1987, pp. 273–281.

Steven Collins, *Adaptive splatting for specular to diffuse light transport*, Fifth Eurographics Workshop on Rendering (Darmstadt, Germany), June 1994, pp. 119–135.

Robert L. Cook, *Shade trees*, Computer Graphics (SIGGRAPH '84 Proceedings) (Hank Christiansen, ed.), vol. 18, July 1984, pp. 223–231.

———, *Stochastic sampling in computer graphics*, ACM Transactions on Graphics **5** (1986), no. 1, 51–72.

Robert L. Cook, Thomas Porter, and Loren Carpenter, *Distributed ray tracing*, Computer Graphics (SIGGRAPH '84 Proceedings), vol. 18, July 1984, pp. 137–45.

Franklin C. Crow, *The aliasing problem in computer-generated shaded images*, Communications of the ACM **20** (1977), no. 11, 799–805.

———, *Summed-area tables for texture mapping*, Computer Graphics (Proceedings of SIGGRAPH 84), vol. 18, July 1984, pp. 207–212.

R. L. Cook and K. E. Torrance, *A reflectance model for computer graphics*, Computer Graphics (SIGGRAPH '81 Proceedings), vol. 15, August 1981, pp. 307–316.

———, *A reflectance model for computer graphics*, ACM Transactions on Graphics **1** (1982), no. 1, 7–24.

Michael Cohen and John Wallace, *Radiosity and realistic image synthesis*, Academic Press Professional, 1993.

Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf, *Computational geometry: Algorithms and applications*, Springer-Verlag, 2000, ISBN 3-540-65620-0.

Robert A. Drebin, Loren Carpenter, and Pat Hanrahan, *Volume rendering*, Computer Graphics (Proceedings of SIGGRAPH 88), vol. 22, August 1988, pp. 65–74.

Kate Devlin, Alan Chalmers, Alexander Wilkie, and Werner Purgathofer, *Tone reproduction and physically based spectral rendering*, The Eurographics Association, September 2002, pp. 101–123.

Fredo Durand and Julie Dorsey, *Interactive tone mapping*, Rendering Techniques 2000: 11th Eurographics Workshop on Rendering, Eurographics, June 2000, ISBN 3-211-83535-0, pp. 219–230.

Frédo Durand and Julie Dorsey, *Fast bilateral filtering for the display of high-dynamic-range images*, ACM Transactions on Graphics **21** (2002), no. 3, 257–266, ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

Paul Debevec, *Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography*, Proceedings of SIGGRAPH 98 (Orlando, Florida), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, July 1998, ISBN 0-89791-999-8, pp. 189–198.

Julie Dorsey, Alan Edelman, Justin Legakis, Henrik Wann Jensen, and Hans Køhling Pedersen, *Modeling and rendering of weathered stone*, Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, August 1999, pp. 225–234.

David P. Dobkin, David Eppstein, and Don P. Mitchell, *Computing the discrepancy with applications to supersampling patterns*, ACM Transactions on Graphics **15** (1996), no. 4, 354–376, ISSN 0730-0301.

John Danskin and Pat Hanrahan, *Fast algorithms for volume ray tracing*, 1992 Workshop on Volume Visualization (1992), 91–98.

David P. Dobkin and Don P. Mitchell, *Random-edge discrepancy of supersampling patterns*, Graphics Interface '93 (Toronto, Ontario), Canadian Information Processing Society, May 1993, pp. 62–69.

Mark A. Z. Dippé and Erling Henry Wold, *Antialiasing through stochastic sampling*, Computer Graphics (SIGGRAPH '85 Proceedings) (B. A. Barsky, ed.), vol. 19, July 1985, pp. 69–78.

David H. Eberly, *3d game engine design: a practical approach to real-time computer graphics*, Morgan Kaufmann, San Francisco, CA, 2001.

David Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steve Worley, *Texturing and modeling: A procedural approach*, Morgan Kaufmann Publishers, San Francisco, CA, 2003.

Peter Shirley Erik Reinhard, Mike Stark and Jim Ferwerda, *Photographic tone reproduction for digital images*, ACM Transactions on Graphics **21** (2002), no. 3, 267–276, Proceedings of ACM SIGGRAPH 2002.

James A. Ferwerda, *Elements of early vision for computer graphics*, IEEE Computer Graphics & Applications **21** (2001), no. 5, 22–33, ISSN 0272-1716.

Alain Fournier and Eugene Fiume, *Constant-time filtering with space-variant kernels*, Computer Graphics (SIGGRAPH '88 Proceedings) (John Dill, ed.), vol. 22, August 1988, pp. 229–238.

Chris Fraser and David Hanson, *A retargetable C compiler: Design and implementation*, Addison-Wesley, 1995.

George S. Fishman, *Monte Carlo: Concepts, algorithms, and applications*, Springer Verlag, New York, NY, 1996.

Ilja Friedel and Alexander Keller, *Fast generation of randomized low discrepancy points sets*, Monte Carlo and Quasi-Monte Carlo Methods 2000 (Berlin), Springer-Verlag, 2000, pp. 257–273.

E. A. Feibush, Marc Levoy, and Robert L. Cook, *Synthetic texturing using digital filters*, Computer Graphics (Proceedings of SIGGRAPH 80), vol. 14, July 1980, pp. 294–301.

Alain Fournier, *Normal distribution functions and multiple surfaces*, Graphics Interface '92 Workshop on Local Illumination, May 1992, pp. 45–52.

James A. Ferwerda, Sumant Pattanaik, Peter S. Shirley, and Donald P. Greenberg, *A model of visual adaptation for realistic image synthesis*, Proceedings of SIGGRAPH 96 (New Orleans, Louisiana), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, August 1996, ISBN 0-201-94800-1, pp. 249–258.

James A. Ferwerda, Sumanta N. Pattanaik, Peter S. Shirley, and Donald P. Greenberg, *A model of visual masking for computer graphics*, Proceedings of SIGGRAPH 97 (Los Angeles, California), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, August 1997, ISBN 0-89791-896-7, pp. 143–152.

Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata, *Arts: Accelerated ray-tracing system*, IEEE Computer Graphics and Applications (1986), 16–26.

James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer graphics: principles and practice*, Addison-Wesley, Reading, Massachusetts, 1990.

Geoffrey Y. Gardner, *Simulation of natural scenes using textured quadric surfaces*, Computer Graphics (SIGGRAPH '84 Proceedings) (Hank Christiansen, ed.), vol. 18, July 1984, pp. 11–20.

A. Gershun, *The light field*, Journal of Mathematics and Physics **18** (1939), 51–151.

Ned Greene and Paul S. Heckbert, *Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter*, IEEE Computer Graphics & Applications **6** (1986), no. 6, 21–27.

———, *Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter*, IEEE Computer Graphics and Applications **6** (1986), no. 6, 21–27.

Larry Gritz and James K. Hahn, *BMRT: A global illumination implementation of the RenderMan standard*, Journal of Graphics Tools **1** (1996), no. 3, 29–47.

Andrew S. Glassner, *Space subdivision for fast ray tracing*, IEEE Computer Graphics and Applications **4** (1984), no. 10, 15–22.

Andrew Glassner (ed.), *An introduction to ray tracing*, Academic Press, 1989.

Andrew S. Glassner, *How to derive a spectrum from an RGB triplet*, IEEE Computer Graphics & Applications **9** (1989), no. 4, 95–99.

Andrew Glassner, *Principles of digital image synthesis*, Morgan Kaufmann Publishers, 1995.

Jay S. Gondek, Gary W. Meyer, and Jonathan G. Newman, *Wavelength dependent reflectance functions*, Proceedings of SIGGRAPH '94, ACM Press, July 1994, pp. 213–220.

Alfred Gray, *Modern differential geometry of curves and surfaces*, CRC Press, 1993.

Ned Greene, *Environment mapping and other applications of world projections*, IEEE Computer Graphics & Applications **6** (1986), no. 11, 21–29.

Jeffrey Goldsmith and John Salmon, *Automatic creation of object hierarchies for ray tracing*, IEEE Computer Graphics and Applications **7** (1987), no. 5, 14–20.

Dirk Grunwald, Benjamin Zorn, and Robert Henderson, *Imrpoving the cache locality of memory allocation*, ACM-SIGPLAN-PLDI, June 1993, pp. 177–186.

Dirk Grunwald, Benjamin G. Zorn, and Robert Henderson, *Improving the cache locality of memory allocation*, SIGPLAN Conference on Programming Language Design and Implementation, 1993, pp. 177–186.

Eric Haines, *Point in polygon strategies*, Graphics Gems IV, 1994, pp. 24–46.

Roy Hall, *Illumination and color in computer generated imagery*, Springer-Verlag, New York, 1989.

John C. Hart, *Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces*, The Visual Computer **12** (1996), no. 9, 527–545.

Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle, *Piecewise smooth surface reconstruction*, Proceedings of SIGGRAPH 94 (Orlando, Florida), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / ACM Press, July 1994, ISBN 0-89791-667-0, pp. 295–302.

David Hart, Philip Dutré, and Donald P. Greenberg, *Direct illumination with lazy visibility evaluation*, Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, August 1999, pp. 147–154.

Stefan Hiller, Oliver Deussen, and Alexander Keller, *Tiled blue noise samples*, Proceedings of Vision, MOdeling and Visualization (T. Ertl, B. Girod, G.Greiner, H. Niemann, and H.-P. Seidel, eds.), November 2001.

Paul Heckbert, *The mathematics of quadric surface rendering and SOID*, July 1984, 3-D Technical Memo.

Paul S. Heckbert, *Survey of texture mapping*, IEEE Computer Graphics and Applications **6** (1986), no. 11, 56–67.

_____, *Fundamentals of texture mapping and image warping*, M.sc. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, June 1989, p. 86.

L. G. Henyey and J. L. Greenstein, *Diffuse radiation in the galaxy*, Astrophysical Journal **93** (1941), 70–83.

E. A. Haines and D. P. Greenberg, *The light buffer: a shadow testing accelerator*, IEEE Computer Graphics & Applications **6** (1986), no. 9, 6–16.

Paul S. Heckbert and Pat Hanrahan, *Beam tracing polygonal objects*, Computer Graphics (Proceedings of SIGGRAPH 84), vol. 18, July 1984, pp. 119–127.

Pat Hanrahan and Wolfgang Krueger, *Reflection from layered surfaces due to subsurface scattering*, Computer Graphics (SIGGRAPH Proceedings), August 1993, pp. 165–174.

Pat Hanrahan and Jim Lawson, *A language for shading and lighting calculations*, Computer Graphics (SIGGRAPH '90 Proceedings) (Forest Baskett, ed.), vol. 24, August 1990, pp. 289–298.

Xiao D. He, Kenneth E. Torrance, Francois X. Sillion, and Donald P. Greenberg, *A comprehensive physical model for light reflection*, Computer Graphics (SIGGRAPH '91 Proceedings) (Thomas W. Sederberg, ed.), vol. 25, July 1991, pp. 175–186.

Eric A. Haines and John R. Wallace, *Shaft culling for efficient ray-traced radiosity*, Second Eurographics Workshop on Rendering (Photorealistic Rendering in ComputerGraphics), 1994.

David S. Immel, Michael F. Cohen, and Donald P. Greenberg, *A radiosity method for non-diffuse environments*, Computer Graphics (SIGGRAPH '86 Proceedings), vol. 20, August 1986, pp. 133–142.

Homan Igehy, *Tracing ray differentials*, Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, August 1999, pp. 179–186.

W. H. Jackson, *The solution of an integral equation occurring in the theory of radiation*, Bulletin of the American Mathematical Society **16** (1910), 473–475.

Henrik Wann Jensen and Juan Buhler, *A rapid hierarchical rendering technique for translucent materials*, ACM Transactions on Graphics **21** (2002), no. 3, 576–581.

Henrik Wann Jensen and Per H. Christensen, *Efficient simulation of light transport in scenes with participating media using photon maps*, SIGGRAPH 98 Conference Proceedings (Michael Cohen, ed.), Annual Conference Series, Addison Wesley, July 1998, pp. 311–320.

Henrik Wann Jensen, *Global illumination using photon maps*, Eurographics Rendering Workshop 1996 (New York City, NY) (Xavier Pueyo and Peter Schröder, eds.), Eurographics, Springer Wien, June 1996, ISBN 3-211-82883-4, pp. 21–30.

———, *Realistic image synthesis using photon mapping*, A. K. Peters, Ltd., Natick, MA, 2001.

Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan, *A practical model for subsurface light transport*, Proceedings of ACM SIGGRAPH 2001, Computer Graphics Proceedings, Annual Conference Series, August 2001, pp. 511–518.

Mark S. Johnstone and Paul R. Wilson, *The memory fragmentation problem: solved?*, ACM SIGPLAN Notices **34** (1999), no. 3, 26–36.

David B. Kirk and James Arvo, *Unbiased sampling techniques for image synthesis*, Computer Graphics (SIGGRAPH '91 Proceedings) (Thomas W. Sederberg, ed.), vol. 25, July 1991, pp. 153–156.

James T. Kajiya, *Anisotropic reflection models*, Computer Graphics (Proceedings of SIGGRAPH 85), vol. 19, July 1985, pp. 15–21.

———, *The rendering equation*, Computer Graphics (SIGGRAPH '86 Proceedings) (David C. Evans and Russell J. Athay, eds.), vol. 20, August 1986, pp. 143–150.

Devendra Kalra and Alan H. Barr, *Guaranteed ray intersections with implicit surfaces*, Computer Graphics (Proceedings of SIGGRAPH 89), vol. 23, July 1989, pp. 297–306.

Alexander Keller, *Quasi-monte Carlo radiosity*, Eurographics Rendering Workshop 1996 (New York City, NY) (Xavier Pueyo and Peter Schröder, eds.), Eurographics, Springer Wien, June 1996, pp. 101–110.

———, *Instant radiosity*, Proceedings of SIGGRAPH 97 (Los Angeles, California), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, August 1997, ISBN 0-89791-896-7, pp. 49–56.

Douglas S. Kay and Donald P. Greenberg, *Transparency for computer synthesized images*, Computer Graphics (SIGGRAPH '79 Proceedings), vol. 13, August 1979, pp. 158–164.

James T. Kajiya and Brian P. Von Herzen, *Ray tracing volume densities*, Computer Graphics (Proceedings of SIGGRAPH 84), vol. 18, July 1984, pp. 165–174.

Craig Kolb, Pat Hanrahan, and Don Mitchell, *A realistic camera model for computer graphics*, SIGGRAPH 95 Conference Proceedings (Robert Cook, ed.), Annual Conference Series, Addison Wesley, August 1995, pp. 317–324.

Louis V. King, *On the scattering and absorption of light in gaseous media, with applications to the intensity of sky radiation*, Philosophical Transactions of the Royal Society of London. Series A. Mathematical and Physical Sciences **212** (1913), 375–433.

T.L. Kay and J.T. Kajiya, *Ray tracing complex scenes*, Computer Graphics (SIGGRAPH '86 Proceedings), vol. 20, August 1986, pp. 269–278.

Thomas Kollig and Alexander Keller, *Efficient bidirectional path tracing by randomized Quasi-Monte Carlo integration*, Monte Carlo and Quasi-Monte Carlo Methods 2000 (Berlin), Springer-Verlag, 2000, pp. 290–305.

———, *Efficient multidimensional sampling*, Computer Graphics Forum (G. Drettakis and H.-P. Seidel, eds.), vol. 21, 2002, pp. 557–563.

R. Victor Klassen, *Modeling the effect of the atmosphere on light*, ACM Transactions on Graphics **6** (1987), no. 3, 215–237.

Donald E. Knuth, *Literate programming*, The Computer Journal **27** (1984), 97–111, Reprinted in Donald E. Knuth, *Literate Programming*, Stanford Center for the Study of Language and Information, 1992.

———, *TEX: The Program*, Adisson–Wesley, Reading, Massachusetts, 1993.

———, *The Stanford GraphBase*, ACM Press and Addison–Wesley, New York, NY, 1993.

Malvin H. Kalos and Paula A. Whitlock, *Monte Carlo methods: Volume I: Basics*, Wiley, New York, NY, 1986.

Johann Heinrich Lambert, *Photometry, or, on the measure and gradations of light, colors, and shade*, The Illuminating Engineering Society of North America, 2001, Transated by David L. DiLaura.

Serge Lang, *An introduction to linear algebra*, Springer Verlag, New York, NY, 1986.

Robert C. Lansdale, *Texture mapping and resampling for computer graphics*, M.sc. thesis, Department of Electrical Engineering, University of Toronto, January 1991.

Marc Levoy, *Display of surfaces from volume data*, IEEE Computer Graphics & Applications **8** (1988), no. 3, 29–37.

———, *Efficient ray tracing of volume data*, ACM Transactions on Graphics **9** (1990), no. 3, 245–261.

———, *A hybrid ray tracer for rendering polygon and volume data*, IEEE Computer Graphics & Applications **10** (1990), no. 2, 33–40.

Eric P. F. Lafortune, Sing-Choong Foo, Kenneth E. Torrance, and Donald P. Greenberg, *Non-linear approximation of reflectance functions*, Proceedings of SIGGRAPH 97 (Los Angeles, California), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, August 1997, ISBN 0-89791-896-7, pp. 117–126.

Jun S. Liu, *Monte Carlo strategies in scientific computing*, Springer-Verlag, New York, NY, 2001.

E. Lommel, *Die Photometrie der diffusen Zurückwerfung*, Annalen der Physik **36** (1889), 473–502.

Charles Loop, *Smooth subdivision surfaces based on triangles*, Ph.D. thesis, University of Utah, 1987.

Gregory Ward Larson, Holly Rushmeier, and Christine Piatko, *A visibility matching tone reproduction operator for high dynamic*

range scenes, IEEE Transactions on Visualization and Computer Graphics **3** (1997), no. 4, 291–306, ISSN 1077-2626.

Mark E. Lee, Richard A. Redner, and Samuel P. Uselton, *Statistically optimized sampling for distributed ray tracing*, Computer Graphics (Proceedings of SIGGRAPH 85), vol. 19, July 1985, pp. 61–67.

Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf, *The cache performance and optimizations of blocked algorithms*, Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV) (Palo Alto, CA), 1991.

Greg Ward Larson and Rob A. Shakespeare, *Rendering with Radiance: The art and science of lighting visualization*, Morgan Kaufmann Publishers, 1998.

Andrzej Lukaszewski, *Exploiting coherence of shadow rays*, AFRIGRAPH 2001, ACM SIGGRAPH, 2001, pp. 147–150.

Eric Lafortune and Yves Willems, *A theoretical framework for physically based rendering*, Computer Graphics Forum **13** (1994), no. 2, 97–107.

Eric P. Lafortune and Yves D. Willems, *Rendering participating media with bidirectional path tracing*, Eurographics Rendering Workshop 1996, June 1996, pp. 91–100.

Daniel Malacara, *Color vision and colorimetry: theory and applications*, SPIE–The International Society for Optical Engineering, 2002.

Nelson L. Max, *Atmospheric illumination and shadows*, Computer Graphics (Proceedings of SIGGRAPH 86), vol. 20, August 1986, pp. 117–124.

Nelson Max, *Optical models for direct volume rendering*, IEEE Transactions on Visualization and Computer Graphics **1** (1995), no. 2, 99–108.

William Ross McCluney, *Introduction to radiometry and photometry*, Artech House, 1994.

Erik Meijering, *A chronology of interpolation: from ancient astronomy to modern signal and image processing*, Proceedings of the IEEE **90** (2002), no. 3, 319–342.

Michael McCool and Eugene Fiume, *Hierarchical Poisson disk sampling distributions*, Graphics Interface (1992), 94–105.

Gene S. Miller and C. Robert Hoffman, *Illumination and reflection maps: Simulated objects in simulated and real environments*, 1984.

Don P. Mitchell and Pat Hanrahan, *Illumination from curved reflectors*, Computer Graphics (Proceedings of SIGGRAPH 92), vol. 26, July 1992, pp. 283–291.

Tomas Möller and Eric Haines, *Real-time rendering*, A. K. Peters, 2002.

Don P. Mitchell, *Generating antialiased images at low sampling densities*, Computer Graphics (SIGGRAPH '87 Proceedings) (Maureen C. Stone, ed.), vol. 21, July 1987, pp. 65–72.

———, *Spectrally optimal sampling for distributed ray tracing*, Computer Graphics (SIGGRAPH '91 Proceedings) (Thomas W. Sederberg, ed.), vol. 25, July 1991, pp. 157–164.

———, *Ray tracing and irregularities of distribution*, Third Eurographics Workshop on Rendering (Bristol, UK), May 1992, pp. 61–69.

———, *Consequences of stratified sampling in graphics*, Proceedings of SIGGRAPH 96 (New Orleans, Louisiana), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, August 1996, ISBN 0-201-94800-1, pp. 277–280.

Stephen R. Marschner and Richard J. Lobb, *An evaluation of reconstruction filters for volume rendering*, Proceedings of Visualization '94 (Washington, DC), October 1994, pp. 100–107.

Don P. Mitchell and Arun N. Netravali, *Reconstruction filters in computer graphics*, Computer Graphics (SIGGRAPH '88 Proceedings) (John Dill, ed.), vol. 22, August 1988, pp. 221–228.

M. Matsumoto and T. Nishimura, *Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modeling and Computer Simulation **8** (1998), no. 1, 3–30.

Joel McCormack, Ronald Perry, Keith I. Farkas, and Norman P. Jouppi, *Feline: Fast elliptical lines for anisotropic texture mapping*, Proceedings of SIGGRAPH 99 (Los Angeles, California), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley Longman, August 1999, ISBN 0-20148-560-5, pp. 243–250.

Parry Moon and Domina Eberle Spencer, *The scientific basis of illuminating engineering*, McGraw-Hill, New York, NY, 1936.

Tomas Möller and Ben Trumbore, *Fast, minimum storage ray-triangle intersection*, Journal of Graphics Tools **2** (1997), no. 1, 21–28.

Stephen R. Marschner, Stephen H. Westin, Eric P. F. Lafortune, Kenneth E. Torrance, and Donald P. Greenberg, *Image-based BRDF measurement including human skin*, Eurographics Rendering Workshop 1999 (Granada, Spain), Springer Wein / Eurographics, June 1999.

Shree K. Nayar, Katsushi Ikeuchi, and Takeo Kanade, *Surface reflection: Physical and geometrical perspectives*, IEEE Transactions on Pattern Analysis and Machine Intelligence **17** (1991), no. 7, 611–634.

Tomoyuki Nishita, Yasuhiro Miyawaki, and Eihachiro Nakamae, *A shading model for atmospheric scattering considering luminous intensity distribution of light sources*, Computer Graphics (Proceedings of SIGGRAPH 87), vol. 21, July 1987, pp. 303–310.

Alan Norton, Alyn P. Rockwood, and Philip T. Skolmoski, *Clamping: a method of antialiasing textured surfaces by bandwidth limiting in object space*, Computer Graphics (Proceedings of SIGGRAPH 82), vol. 16, July 1982, pp. 1–8.

Michael Oren and Shree K. Nayar, *Generalization of Lambert's reflectance model*, Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994) (Andrew Glassner, ed.), Computer Graphics Proceedings, Annual Conference Series, ACM Press, July 1994, pp. 239–246.

Bui-T. Phong and F. C. Crow, *Improved rendition of polygonal models of curved surfaces*, Proceedings of the 2nd USA-Japan Computer Conference, 1975.

M. Potmesil and I. Chakravarty, *A lens and aperture camera model for synthetic image generation*, Computer Graphics (Proceedings of SIGGRAPH 81) (Dallas, Texas), vol. 15, August 1981, pp. 297–305.

———, *Synthetic image generation with a lens and aperture camera model*, ACM Transactions on Graphics **1** (1982), no. 2, 85–108.

———, *Modeling motion blur in computer-generated images*, Computer Graphics (Proceedings of SIGGRAPH 83) (Detroit, Michigan), vol. 17, July 1983, pp. 389–399.

Thomas Porter and Tom Duff, *Compositing digital images*, Computer Graphics (Proceedings of SIGGRAPH 84) (Minneapolis, Minnesota), vol. 18, July 1984, pp. 253–259.

Darwyn R. Peachey, *Solid texturing of complex surfaces*, Computer Graphics (SIGGRAPH '85 Proceedings) (B. A. Barsky, ed.), vol. 19, July 1985, pp. 279–286.

Mark S. Peercy, *Linear color representations for full spectral rendering*, Computer Graphics (SIGGRAPH '93 Proceedings) (James T. Kajiya, ed.), vol. 27, August 1993, pp. 191–198.

Ken Perlin, *An image synthesizer*, Computer Graphics (SIGGRAPH '85 Proceedings), vol. 19, July 1985, pp. 287–296.

———, *Improving noise*, ACM Transactions on Graphics **21** (2002), no. 3, 681–682, ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

Sumanta N. Pattanaik, James A. Ferwerda, Mark D. Fairchild, and Donald P. Greenberg, *A multiscale model of adaptation and spatial vision for realistic image display*, Proceedings of SIGGRAPH 98 (Orlando, Florida), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, July 1998, ISBN 0-89791-999-8, pp. 287–298.

Matt Pharr and Patrick M. Hanrahan, *Monte Carlo evaluation of non-linear scattering equations for subsurface reflection*, Proceedings of ACM SIGGRAPH 2000, Computer Graphics Proceedings, Annual Conference Series, July 2000, pp. 75–84.

Bui-T. Phong, *Illumination for computer generated pictures*, Communications of the ACM **18** (1975), no. 6, 311–317.

Pixar Animation Studios, *The RenderMan interface*, September 1989, With typographical corrections through May 1995.

Mark Pauly, Thomas Kollig, and Alexander Keller, *Metropolis light transport for participating media*, Rendering Techniques 2000: 11th Eurographics Workshop on Rendering, Eurographics, June 2000, ISBN 3-211-83535-0, pp. 11–22.

Charles Poynton, *Frequently-asked questions about color*, 2002, http://www.inforamp.net/põynton/ColorFAQ.html.

———, *Frequently-asked questions about gamma*, 2002, http://www.inforamp.net/põynton/GammaFAQ.html.

Frederic Pérez, Xavier Pueyo, and François X. Sillion, *Global illumination techniques for the simulation of participating media*, Eurographics Rendering Workshop 1997, June 1997, pp. 309–320.

Rudolph W. Preisendorfer, *Radiative transfer on discrete spaces*, Pergamon Press, Oxford, 1965.

R. W. Preisendorfer, *Hydrologic optics*, U.S. Department of Commerce, National Oceanic and Atmospheric Administration, Honolulu, Hawaii, 1976, Six volumes.

A. J. Preetham, Peter S. Shirley, and Brian E. Smits, *A practical analytic model for daylight*, Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, August 1999, pp. 91–100.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical recipes in C: The art of scientific computing (2nd ed.)*, Cambridge University Press, Cambridge, 1992.

Sumanta N. Pattanaik, Jack E. Tumblin, Hector Yee, and Donald P. Greenberg, *Time-dependent visual adaptation for realistic image display*, Proceedings of ACM SIGGRAPH 2000, Computer Graphics Proceedings, Annual Conference Series, ACM Press

/ ACM SIGGRAPH / Addison Wesley Longman, July 2000, ISBN 1-58113-208-5, pp. 47–54.

David F. Rogers and J. Alan Adams, *Mathematical elements for computer graphics*, McGraw–Hill, New York, NY, 1990.

Erik Reinhard, *Parameter estimation for photographic tone reproduction*, Journal of Graphics Tools **7** (2002), no. 1, 45–52.

Maria Raso and Alain Fournier, *A piecewise polynomial approach to shading using spectral distributions*, Graphics Interface '91, Canadian Information Processing Society, June 1991, pp. 40–46.

G. Rougeron and B. Péroche, *Color fidelity in computer graphics: A survey*, Computer Graphics Forum **17** (1998), no. 1, 3–16, ISSN 1067-7055.

Mahesh Ramasubramanian, Sumanta N. Pattanaik, and Donald P. Greenberg, *A perceptually based physical error metric for realistic image synthesis*, Proceedings of SIGGRAPH 99 (Los Angeles, California), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley Longman, August 1999, ISBN 0-20148-560-5, pp. 73–82.

William T. Reeves, David H. Salesin, and Robert L. Cook, *Rendering antialiased shadows with depth maps*, Computer Graphics (Proceedings of SIGGRAPH 87), vol. 21, July 1987, pp. 283–291.

Holly E. Rushmeier and Kenneth E. Torrance, *The zonal method for calculating light intensities in the presence of a participating medium*, Computer Graphics (Proceedings of SIGGRAPH 87), vol. 21, July 1987, pp. 293–302.

J. Revelles, C. Urena, and M. Lastra, *An efficient parametric algorithm for octree traversal*, Journal of WSCG **8**, no. 2, 212–219, The 8th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Media.

S. M. Rubin and T. Whitted, *A 3-dimensional representation for fast rendering of complex scenes*, Computer Graphics **14** (1980), no. 3, 110–116.

Hanan Samet, *The design and analysis of spatial data structures*, Addison-Wesley, 1990, ISBN 0-201-50255-0.

John M. Snyder and Alan H. Barr, *Ray tracing complex models containing surface tessellations*, Computer Graphics (SIGGRAPH '87 Proceedings) (Maureen C. Stone, ed.), vol. 21, July 1987, pp. 119–128.

Peter Shirley and Kenneth Chiu, *A low distortion map between disk and square*, Journal of Graphics Tools **2** (1997), no. 3, 45–52, ISSN 1086-7651.

Arthur Schuster, *Radiation through a foggy atmosphere*, Astrophysical Journal **21** (1905), no. 1, 1–22.

Christophe Schlick, *A customizable reflectance model for everyday rendering*, Fourth Eurographics Workshop on Rendering (held in Paris, France, 14-16 June 1993), Eurographics, June 1993, pp. 73–84.

K. Sung, J. Craighead, C. Wang, S. Bakshi, A. Pearce, and A. Woo, *Design and implementation of the Maya renderer*, Pacific Graphics '98, October 1998.

Philip J. Schneider and David H. Eberly, *Geometric tools for computer graphics*, Morgan Kaufmann Publishers, San Francisco, CA, 2003.

Yinlong Sun, F. David Fracchia, Mark S. Drew, and Thomas W. Calvert, *A spectrally based framework for realistic image synthesis*, The Visual Computer **17** (2001), no. 7, 429–444, ISSN 0178-2789.

Jerome Spanier and Ely M. Gelbard, *Monte Carlo principles and neutron transport problems*, Addison–Wesley, Reading, Massachusetts, 1969.

Morgan Schramm, Jay Gondek, and Gary Meyer, *Light scattering simulations using complex subsurface models*, Graphics Interface (Wayne Davis, Marilyn Mantei, and Victor Klassen, eds.), May 1997, pp. 56–67.

Peter Shirley, *Physically based lighting calculations for computer graphics*, Ph.D. thesis, Dept. of Computer Science, U. of Illinois, Urbana-Champaign, November 1990.

———, *A ray tracing method for illumination calculation in diffuse-specular scenes*, Proceedings of Graphics Interface '90, May 1990, pp. 205–212.

P. Shirley, *Discrepancy as a quality measure for sample distributions*, Eurographics '91 (Werner Purgathofer, ed.), North-Holland, September 1991, pp. 183–194.

Gernot Schaufler and Henrik Wann Jensen, *Ray tracing point sampled geometry*, Rendering Techniques 2000: 11th Eurographics Workshop on Rendering, June 2000, pp. 319–328.

Alvy Ray Smith, *Painting tutorial notes*, 1979.

Kelvin Sung and Peter Shirley, *Ray tracing with the BSP tree*, Graphics Gems III (David Kirk, ed.), Academic Press, 1992, pp. 271–274.

Jos Stam, *Diffraction shaders*, Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, August 1999, pp. 101–110.

Mikio Shinya, Tokiichiro Takahashi, and Seiichiro Naito, *Principles and applications of pencil tracing*, Computer Graphics (Proceedings of SIGGRAPH 87), vol. 21, July 1987, pp. 45–54.

Jorge Stolfi, *Oriented projective geometry*, Academic Press, San Diego, CA, 1991.

Bjarne Stroustrup, *The C++ programming language*, Addison-Wesley, 1997.

Frank Suykens and Yves Willems, *Path differentials and applications*, Rendering Techniques 2001: 12th Eurographics Workshop on Rendering, June 2001, pp. 257–268.

Peter Shirley, Chang Yaw Wang, and Kurt Zimmerman, *Monte Carlo techniques for direct lighting calculations*, ACM Transactions on Graphics **15** (1996), no. 1, 1–36, ISSN 0730-0301.

D. N. Truong, Francois Bodin, and Andre Seznec, *Improving cache behavior of dynamically allocated data structures*, IEEE PACT, 1998, pp. 322–329.

Jack Tumblin, Jessica K. Hodgins, and Brian K. Guenter, *Two methods for display of high contrast images*, ACM Transactions on Graphics **18** (1999), no. 1, 56–94, ISSN 0730-0301.

Jack Tumblin and Holly E. Rushmeier, *Tone reproduction for realistic images*, IEEE Computer Graphics & Applications **13** (1993), no. 6, 42–48.

K. E. Torrance and E. M. Sparrow, *Theory for off-specular reflection from roughened surfaces*, Journal of the Optical Society of America **57** (1967), no. 9, 1105–1114.

Jack Tumblin and Greg Turk, *LCIS: A boundary hierarchy for detail-preserving contrast reduction*, Proceedings of SIGGRAPH 99 (Los Angeles, California), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley Longman, August 1999, ISBN 0-20148-560-5, pp. 83–90.

Ken Turkowski, *The differential geometry of parametric primitives*, 1990.

———, *The differential geometry of texture-mapping and shading*, 1993.

Steve Upstill, *The RenderMan companion*, Addison–Wesley, Reading, Massachusetts, 1989.

Hendrik Christoffel van de Hulst, *Multiple light scattering*, Academic Press, New York, 1980, Two volumes.

———, *Light scattering by small particles*, Dover Publications, New York, 1981, Originally published by John Wiley and Sons, 1957.

Eric Veach, *Non-symmetric scattering in light transport algorithms*, Eurographics Rendering Workshop 1996 (Xavier Pueyo and Peter Schröder, eds.), Springer Wien, June 1996.

———, *Robust Monte Carlo methods for light transport simulation*, Ph.D. thesis, Stanford University, December 1997.

Eric Veach and Leonidas Guibas, *Bidirectional estimators for light transport*, Fifth Eurographics Workshop on Rendering (Darmstadt, Germany), June 1994, pp. 147–162.

Eric Veach and Leonidas J. Guibas, *Optimally combining sampling techniques for Monte Carlo rendering*, Computer Graphics (SIGGRAPH Proceedings), August 1995, pp. 419–428.

———, *Metropolis light transport*, Computer Graphics (SIGGRAPH Proceedings), August 1997, pp. 65–76.

Bruce A. Wallace, *Merging and transformation of raster images for caroon animation*, Proceedings of ACM SIGGRAPH '81, vol. 15, August 1981, pp. 253–262.

Brian Wandell, *Foundations of vision*, Sinauer Associates, 1995.

D. R. Warn, *Lighting controls for synthetic images*, Computer Graphics (Proceedings of SIGGRAPH 83) (Detroit, Michigan), vol. 17, July 1983, pp. 13–21.

Gregory J. Ward, *Measuring and modeling anisotropic reflection*, Computer Graphics (SIGGRAPH '92 Proceedings) (Edwin E. Catmull, ed.), vol. 26, July 1992, pp. 265–272.

Greg Ward, *A contrast-based scalefactor for luminance display*, Graphics Gems IV, Academic Press, Boston, 1994, ISBN 0-12-336155-9, pp. 415–421.

Gregory J. Ward, *The Radiance lighting simulation and rendering system*, Proceedings of SIGGRAPH '94 (Andrew Glassner, ed.), July 1994, pp. 459–472.

Stephen Westin, James Arvo, and Kenneth Torrance, *Predicting reflectance functions from complex surfaces*, Computer Graphics **26** (1992), no. 2, 255–264.

Gregory J. Ward and Paul Heckbert, *Irradiance gradients*, Third Eurographics Workshop on Rendering (1992), 85–98.

Turner Whitted, *An improved illumination model for shaded display*, Communications of the ACM **23** (1980), no. 6, 343–349.

Lance Williams, *Casting curved shadows on curved surfaces*, Computer Graphics (Proceedings of SIGGRAPH 78), vol. 12, August 1978, pp. 270–274.

———, *Pyramidal parametrics*, Computer Graphics (SIGGRAPH '83 Proceedings), vol. 17, July 1983, pp. 1–11.

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, *Dynamic storage allocation: A survey and critical review*, Proc. Int. Workshop on Memory Management (Kinross Scotland (UK)), 1995.

Tien-Tsin Wong, Wai-Shing Luk, and Pheng-Ann Heng, *Sampling with Hammersley and Halton points*, Journal of Graphics Tools **2** (1997), no. 2, 9–24, ISSN 1086-7651.

Steven P. Worley, *A cellular texture basis function*, Proceedings of SIGGRAPH 96 (New Orleans, Louisiana), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, August 1996, ISBN 0-201-94800-1, pp. 291–294.

Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear, *A ray tracing solution for diffuse interreflection*, Computer Graphics (SIGGRAPH '88 Proceedings) (John Dill, ed.), vol. 22, August 1988, pp. 85–92.

Brian Wyvill and Geoff Wyvill, *Field functions for implicit surfaces*, The Visual Computer **5** (1989), no. 1/2, 75–82.

Alan Watt and Mark Watt, *Advanced animation and rendering techniques*, Addison–Wesley, New York, NY, 1992.

Edgard G. Yanovitskij, *Light scattering in inhomogeneous atmospheres*, Springer Verlag, Berlin, 1997.

John I Yellot, *Spectral consequences of photoreceptor sampling in the Rhesus retina*, Science **221** (1983), 382–385.

Denis Zorin, Peter Schröder, Tony DeRose, Leif Kobbelt, Adi Levin, and Wim Sweldins, *Subdivision for modeling and animation*, August 2000.

Index of Identifiers