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
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 interdisciplinary 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 included implementations 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:
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 reducing 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 precompute 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 Stroustrup’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 encyclopedic 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 (SCW98). *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 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

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Processing Camera Samples</td>
<td>237</td>
</tr>
<tr>
<td>8.2 Spectral Image Storage</td>
<td>239</td>
</tr>
<tr>
<td>8.3 Image Display Pipeline</td>
<td>240</td>
</tr>
<tr>
<td>8.4 Tone Mapping</td>
<td>244</td>
</tr>
<tr>
<td>8.5 Device RGB Conversion and Output</td>
<td>258</td>
</tr>
</tbody>
</table>

# 9 Reflection Models

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Basic Interface</td>
<td>265</td>
</tr>
<tr>
<td>9.2 Specular Reflection and Transmission</td>
<td>269</td>
</tr>
<tr>
<td>9.3 Lambertian Reflection</td>
<td>280</td>
</tr>
<tr>
<td>9.4 Microfacet Models</td>
<td>281</td>
</tr>
<tr>
<td>9.5 Lafortune Model</td>
<td>291</td>
</tr>
<tr>
<td>9.6 Fresnel Incidence Effects</td>
<td>294</td>
</tr>
</tbody>
</table>

# 10 Materials

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 BSDFs</td>
<td>297</td>
</tr>
<tr>
<td>10.2 Material Interface and Bump Mapping</td>
<td>302</td>
</tr>
<tr>
<td>10.3 Matte</td>
<td>311</td>
</tr>
<tr>
<td>10.4 Plastic</td>
<td>312</td>
</tr>
<tr>
<td>10.5 Translucent</td>
<td>313</td>
</tr>
<tr>
<td>10.6 Glass</td>
<td>314</td>
</tr>
<tr>
<td>10.7 Shiny Metal</td>
<td>315</td>
</tr>
<tr>
<td>10.8 Diffuse Substrate</td>
<td>316</td>
</tr>
<tr>
<td>10.9 Measured Data</td>
<td>318</td>
</tr>
</tbody>
</table>

# 11 Texture

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 Sampling and Anti-Aliasing</td>
<td>323</td>
</tr>
<tr>
<td>11.2 Texture Interface</td>
<td>323</td>
</tr>
<tr>
<td>11.3 Basic Textures</td>
<td>324</td>
</tr>
<tr>
<td>11.4 2D Mappings</td>
<td>326</td>
</tr>
<tr>
<td>11.5 Interpolated Textures</td>
<td>330</td>
</tr>
<tr>
<td>11.6 Image Maps</td>
<td>333</td>
</tr>
<tr>
<td>11.7 Solid and Procedural Texturing</td>
<td>343</td>
</tr>
<tr>
<td>11.8 Noise</td>
<td>347</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>A.2</td>
<td>Error Reporting</td>
</tr>
<tr>
<td>A.3</td>
<td>Statistics</td>
</tr>
<tr>
<td>A.4</td>
<td>Memory Management</td>
</tr>
<tr>
<td>A.5</td>
<td>Matrix Routines</td>
</tr>
<tr>
<td>A.6</td>
<td>Mathematical Utility Functions</td>
</tr>
<tr>
<td>A.7</td>
<td>Random Numbers</td>
</tr>
<tr>
<td>A.8</td>
<td>Hash Tables</td>
</tr>
<tr>
<td>A.9</td>
<td>Octrees</td>
</tr>
<tr>
<td>A.10</td>
<td>Kd Trees</td>
</tr>
<tr>
<td>A.11</td>
<td>Main Include File</td>
</tr>
<tr>
<td>B</td>
<td>TIFF Input and Output</td>
</tr>
<tr>
<td>B.1</td>
<td>Input</td>
</tr>
<tr>
<td>B.2</td>
<td>Output</td>
</tr>
<tr>
<td>C</td>
<td>Dynamic Object Creation</td>
</tr>
<tr>
<td>C.1</td>
<td>Parameter Sets</td>
</tr>
<tr>
<td>C.2</td>
<td>Reading Dynamic Libraries</td>
</tr>
<tr>
<td>D</td>
<td>Rendering Interface</td>
</tr>
<tr>
<td>D.1</td>
<td>Tokens and Parameter Management</td>
</tr>
<tr>
<td>D.2</td>
<td>Global Settings and Graphics Options</td>
</tr>
<tr>
<td>D.3</td>
<td>Graphics State</td>
</tr>
<tr>
<td>D.4</td>
<td>Transformations</td>
</tr>
<tr>
<td>D.5</td>
<td>Geometric Primitives</td>
</tr>
<tr>
<td>D.6</td>
<td>Light Sources</td>
</tr>
<tr>
<td>D.7</td>
<td>Volumes</td>
</tr>
</tbody>
</table>
This chapter provides a high-level description of \textit{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 \textit{programs should be written more for people’s consumption than for computers’ consumption.} He named this methodology \textit{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 \textit{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:

```c
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:

```c
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:

```c
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:

```c
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:

```c
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 preceding 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.
int main(int argc, char *argv[]) {
    fprintf(stderr, "lrt version %1.3f of %s at %s\n", LRT_VERSION, __DATE__, __TIME__);

    RiBegin();

    if (argc == 1) {
        ParseFile("-");
    } else {
        for (int i = 1 ; i < argc ; i++) {
            if (!ParseFile(argv[i]))
                Error("Couldn't open input file "%s\n", argv[i]);
        }
    }

    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.

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.

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.

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 computed, 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.

\[ Scene \ Data \]

\begin{verbatim}
  vector<Light *> lights;
  Primitive *prims;
  Camera *camera;
  Sampler *sampler;
  SurfaceIntegrator *surfaceIntegrator;
  VolumeIntegrator *volumeIntegrator;
  vector<VolumeRegion *> volumeRegions;
  DisplayInfo *displayInfo;
\end{verbatim}

\[ Scene \ Data \]

\begin{verbatim}
  BBox bound;
  Point center;
  Float radius;
\end{verbatim}

\[ Scene \ Method \ Declarations \]

\begin{verbatim}
  BBox Scene::WorldBound() const {
    return bound;
  }
\end{verbatim}

\[ Scene \ Method \ Declarations \]

\begin{verbatim}
  void BoundingBoxSphere(Point *c, Float *rad) const {
    *c = center;
    *rad = radius;
  }
\end{verbatim}
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

```cpp
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);
}
```

Camera Interface Declarations

```cpp
virtual void GenerateRay(const Sample *sample, Ray &ray) const = 0;
```

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.
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 compute a color to store in the image.

\[ \text{Generate ray differentials for camera ray} \]

```cpp
++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.

\[ \text{Evaluate radiance along ray} \]

```cpp
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.

\[\text{Add sample to image}\]\
\[
\text{Float screenz = camera->ScreenDepth(ray(ray.maxt));}
\]
\[
\text{if (screenz > 1.) {}
L = 0.;
alpha = 0.;
}
\]
\[
\text{Point Praster(sample->imagex, sample->imagey, screenz);}
\]
\[
\text{sampler->AddSample(camera->film, Praster, L, alpha);}\
\]

1.6 Ray Intersections

\[\text{Primitive Declarations}\]\
\[
\text{class Primitive : public ReferenceCounted<Primitive> {}
\]
\[
\text{public:
}
\]
\[
\text{\textit{Primitive Interface}}
\]
\[
\text{virtual ~Primitive();}
\]
\[
\};
\]

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:

\[\text{Primitive Interface}\]\
\[
\text{virtual BBox WorldBound() const = 0;}
\]
\[
\text{virtual bool CanIntersect() const;}
\]
\[
\text{virtual bool Intersect(const Ray &r, Surf *s) const = 0;}
\]
\[
\text{virtual bool IntersectP(const Ray &r) const = 0;}
\]
\[
\text{virtual void Refine(vector<Reference<Primitive> > &refined) const;}
\]

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.

\[\text{GeometricPrimitive Methods}\]\
\[
\text{void GeometricPrimitive::Bump(const DifferentialGeometry &dg,}
\]
\[
\text{DifferentialGeometry *dgShading) const {
material->Bump(dg, Ns, Ss, dgShading);
}}
\]

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.

\[
\text{GeometricPrimitive Data} \equiv \\
\text{Reference<Shape> shape;}
\text{Reference<Material> material;}
\text{AreaLight *areaLight;}
\text{Texture<Normal> *Ns;}
\text{Texture<Vector> *Ss;}
\]

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.

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.

\[
\text{Primitive Declarations} \equiv \\
\text{struct Surf {}
\text{Surf Method Declarations}
\text{Surf Data Members}
\text{}}
\]

\[
\text{Surf Data Members} \equiv \\
\text{DifferentialGeometry dgGeom;}
\text{mutable DifferentialGeometry dgShading;}
\text{const GeometricPrimitive *primitive;}
\]

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.
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.

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.

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, $-\text{ray.D}$. 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 $\text{rayDepth}$. 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 ??.

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

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 in Chapter 12.

```plaintext
(Compute emitted light if an area light source) \equiv
   L += \text{surf.Le}(-\text{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.

```plaintext
(Evaluate BSDF at hit point) \equiv
   \text{BSDF} * \text{bsdf} = \text{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.)

\[
\text{Compute reflection by integrating over the lights}\equiv
\]

```cpp
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);
}
```

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.

\[
\text{Trace rays for specular reflection and refraction}\equiv
\]

```cpp
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.
Finally, rays that don’t hit any geometry in the scene may still carry radiance back to the image:

```c
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

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. Stroustrup XXX is the definitive reference and extensively describes the use of the standard library.


Appel ray tracing shadows, etc (App68)

Kay and Greenberg on transparency (KG79)

Whitted original paper (Whi80)
2. Geometry and Transformations

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 \( p_o \) (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 \( \vec{v} \) in the space can be expressed as a linear combination of the basis vectors. Given a vector \( \vec{v} \) and the basis vectors \( \vec{v}_i \), we can compute scalar values \( s_i \) such that

\[
\vec{v} = s_1 \vec{v}_1 + \cdots + s_n \vec{v}_n
\]

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

\[
p = p_0 + s_1 \vec{v}_1 + \cdots + s_n \vec{v}_n
\]

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

\(\{\text{Geometry Classes}\}\)\

\{
\begin{align*}
\text{class Vector} & \{
\text{public:}
\{\text{Vector Constructor}s\}
\{\text{Vector Methods}\}
\{\text{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 \( \vec{v} \) will be written \( v_x \), \( v_y \), and \( v_z \).

\(\{\text{Vector Public Data}\}\)\

\begin{align*}
\text{Float} & \ x, y, z;
\end{align*}

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

\(\{\text{Vector Constructors}\}\)\

\begin{align*}
\text{Vector()} & \ :
\begin{cases}
\text{x}(0.), \ y(0.), \ z(0.)
\end{cases}
\}
\end{align*}

\(\{\text{Vector Constructors}\} + \)\

\begin{align*}
\text{Vector(Float xx, Float yy, Float zz)} & 
\begin{cases}
\text{x(xx), y(yy), z(zz)}
\end{cases}
\}
\end{align*}

**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.
Sec. 2.1] Vectors

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

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

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.
\textit{Vector Methods} \(\equiv\)

\begin{verbatim}
Vector operator-() const {
  return Vector(-x, -y, -z);
}
\end{verbatim}

\textbf{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 \(\|\mathbf{v}\|\). The method to do this is called \textit{Hat}, which is a common mathematical notation for a normalized vector.

\textit{Vector Methods} \(\equiv\)

\begin{verbatim}
Float LengthSquared() const { return x*x + y*y + z*z; }
Float Length() const { return sqrtf( LengthSquared() ); }
Vector Hat() const { return (*this)/Length(); }
\end{verbatim}

\textbf{Dot and Cross Product}

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

\[v_x w_x + v_y w_y + v_z w_z\]

\textit{Geometry Inline Functions} \(\equiv\)

\begin{verbatim}
inline Float Dot(const Vector &v1, const Vector &v2) {
  return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}
\end{verbatim}

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

\[(\mathbf{v} \cdot \mathbf{w}) = \|\mathbf{v}\| \|\mathbf{w}\| \cos \theta\]

where \(\theta\) is the angle between \(\mathbf{v}\) and \(\mathbf{w}\). It follows from this that \((\mathbf{v} \cdot \mathbf{w})\) is zero if and only if \(\mathbf{v}\) and \(\mathbf{w}\) are perpendicular (provided that neither \(\mathbf{v}\) nor \(\mathbf{w}\) is \textit{degenerate}—equal to \((0,0,0)\)). Furthermore, if \(\mathbf{v}\) and \(\mathbf{w}\) 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 \(\mathbf{u}\), \(\mathbf{v}\), and \(\mathbf{w}\) are vectors and \(s\) is a scalar value, then

\[(\mathbf{u} \cdot \mathbf{v}) = (\mathbf{v} \cdot \mathbf{u})\]

\[(s\mathbf{u} \cdot \mathbf{v}) = s(\mathbf{v} \cdot \mathbf{u})\]

\[(\mathbf{u} \cdot (\mathbf{v} + \mathbf{w})) = (\mathbf{u} \cdot \mathbf{v}) + (\mathbf{u} \cdot \mathbf{w})\]

The \textit{cross product} is another useful vector operation. Given two vectors in 3D, the cross product \(\mathbf{v} \times \mathbf{w}\) 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 \( \mathbf{v}_1 \) and \( \mathbf{v}_2 \) is equal to \( \mathbf{v}_2 h \). The cross product can easily compute this value as \( \mathbf{v}_1 \times \mathbf{v}_2 \).

(\( \mathbf{v}_1 \times \mathbf{v}_2 \))_x = (v_y w_z) - (v_z w_y) 
(\( \mathbf{v}_1 \times \mathbf{v}_2 \))_y = (v_z w_x) - (v_x w_z) 
(\( \mathbf{v}_1 \times \mathbf{v}_2 \))_z = (v_x w_y) - (v_y w_x)

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

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

\[
\begin{vmatrix}
  i & v_x & w_x \\
  j & v_y & w_y \\
  k & v_z & w_z \\
\end{vmatrix}
\]

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

**Geometry Inline Functions**

```cpp
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 \( \mathbf{u} = \mathbf{v} \times \mathbf{w} \), then

\[
\| \mathbf{u} \| = \| \mathbf{v} \| \| \mathbf{w} \| \sin \theta,
\]

where \( \theta \) is the angle between \( \mathbf{v} \) and \( \mathbf{w} \). 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 \( \mathbf{v} \) and \( \mathbf{w} \) 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 \( \mathbf{v}_1 \) and \( \mathbf{v}_2 \), and has height \( h \), the area is given by \( \| \mathbf{v}_2 \| h \). Since \( h = \sin \theta \| \mathbf{v}_1 \| \), we can use Equation 2.1.1 to see that the area is \( \mathbf{v}_1 \times \mathbf{v}_2 \).

**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 coordinate 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, \( \mathbf{v1} \), 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 \( \mathbf{v2} \) will be normalized and that the dot product \( (\mathbf{v1} \cdot \mathbf{v2}) \) 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 perpendicular to the first two.

\[
\text{Geometry Inline Functions} \implies \\
\text{inline void CoordinateSystem(const Vector &}\mathbf{v1}, \text{ Vector *}\mathbf{v2}, \text{ Vector *}\mathbf{v3}) \{ \\
\quad \text{if (fabsf(v1.x) > fabsf(v1.y))} \{ \\
\quad\quad \text{Float invLen} = 1.f / \text{sqrtf(v1.x*v1.x + v1.z*v1.z)}; \\
\quad\quad *\mathbf{v2} = \text{Vector}(-v1.z * \text{invLen}, 0.f, v1.x * \text{invLen}); \\
\quad \} \\
\quad \text{else} \{ \\
\quad\quad \text{Float invLen} = 1.f / \text{sqrtf(v1.y*v1.y + v1.z*v1.z)}; \\
\quad\quad *\mathbf{v2} = \text{Vector}(0.f, v1.z * \text{invLen}, -v1.y * \text{invLen}); \\
\quad \} \\
\quad *\mathbf{v3} = \text{Cross(v1, *v2)}; \\
\}
\]

\[
\text{2.2 Points} \\
\text{Geometry Classes} \implies \\
\text{class Point} \{ \\
\text{public:} \\
\text{Point Constructors} \\
\text{Point Methods} \\
\text{Point Public Data} \\
\} \\
\]

A point is a zero-dimensional quantity that represents a location in 3D space. To represent a \text{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.

\[
\text{Point Public Data} \equiv \\
\text{Float } x, y, z;
\]

\[
\text{Point Constructors} \equiv \\
\text{Point}() \\
\quad : x(0.), y(0.), z(0.) \{ \\
\}
\]

\[
\text{Point Constructors} \equiv \\
\text{Point(Float xx, Float yy, Float zz) \\
\quad : x(xx), y(yy), z(zz) \{ \\
\}
\]
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 Methods)

```cpp
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.
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.

### Geometry Inline Functions

```cpp
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();
}
```

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 Normals and Vectors are very similar: like vectors, normals are represented by three Floats 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.

\[\text{Normal Constructors}\] +≡
explicit Normal(const Vector &v) :
\begin{align*}
x(v.x), y(v.y), z(v.z) & \{ \}
\end{align*}

\[\text{Vector Constructors}\] +≡
explicit Vector(const Normal &n);

\[\text{Geometry Inline Functions}\] +≡
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 \(n = \text{Normal}(v)\).

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

\[\text{Geometry Inline Functions}\] +≡
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

\[\text{Geometry Classes}\] +≡
class Ray {
public:
\begin{align*}
\text{Ray Constructor Declarations} & \\
\text{Ray Method Declarations} & \\
\text{Ray Public Data} & 
\end{align*}
};
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 \( \mathbf{o}(\mathbf{r}) \) and direction \( \mathbf{d}(\mathbf{r}) \), as shown in Figure 2.5.

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

\[
\mathbf{r}(t) = \mathbf{o}(\mathbf{r}) + t\mathbf{d}(\mathbf{r})
\]

Because we will be referring to these variables often throughout the code, the origin and direction members of a Ray are named simply \( \mathbf{O} \) and \( \mathbf{D} \).

\[
\langle \text{Ray Public Data} \rangle \equiv \\
\text{Point } \mathbf{O}; \\
\text{Vector } \mathbf{D};
\]

In addition, we include fields to restrict the ray to a particular segment. These fields, called \( \text{mint} \) and \( \text{maxt} \), 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 \( \text{const} \). Because we need to update these fields all the time, we elect to keep the code simpler rather than adding mutator methods.

\[
\langle \text{Ray Public Data} \rangle + \equiv \\
\text{mutable Float mint, maxt;}
\]

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.

\[
\langle \text{Ray Public Data} \rangle + \equiv \\
\text{Float time;}
\]

Constructing Rays is straightforward. A default constructor is provided, which lets the default constructors of Points and Vectors set the origin and direction to \((0,0,0)\). Alternatively, a particular point and direction can be provided. Also note that \( \text{mint} \) 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:

```cpp
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

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

\[\text{Geometry Classes}\]

```cpp
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 Floats when we are providing them with a 2D region on the image plane.

\{Extent2D Constructors\}

```cpp
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);
}
```

\{Extent2D Data\}

```cpp
Float x0, x1, y0, y1;
```

Three-dimensional bounding boxes

\[\text{Geometry Classes}\]

```cpp
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.
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.

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.

\[\langle \text{BBox Constructors} \rangle \equiv \]
\[
\begin{align*}
\text{BBox}() & \{ \\
pMin = \text{Point}(\text{INFINITY}, \text{INFINITY}, \text{INFINITY}); \\
pMax = \text{Point}(-\text{INFINITY}, -\text{INFINITY}, -\text{INFINITY}); 
\}
\end{align*}
\]

\[\langle \text{BBox Public Data} \rangle \equiv \]
Point pMin, pMax;

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

\[\langle \text{BBox Constructors} \rangle + \equiv \]
\[
\begin{align*}
\text{BBox(const Point &p) : pMin(p), pMax(p) \{ \}}
\end{align*}
\]

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.
Sec. 2.5] Axis-Aligned Bounding Boxes

\[ \text{BBox Constructors} \]

\begin{align*}
\text{BBox}(\text{const Point} & \mathbf{p}_1, \text{const Point} & \mathbf{p}_2) \{ \\
pMin &= \text{Point}(\min(\mathbf{p}_1.x, \mathbf{p}_2.x), \\
& \quad \min(\mathbf{p}_1.y, \mathbf{p}_2.y), \\
& \quad \min(\mathbf{p}_1.z, \mathbf{p}_2.z)); \\
pMax &= \text{Point}(\max(\mathbf{p}_1.x, \mathbf{p}_2.x), \\
& \quad \max(\mathbf{p}_1.y, \mathbf{p}_2.y), \\
& \quad \max(\mathbf{p}_1.z, \mathbf{p}_2.z)); \\
\}
\end{align*}

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.

\[ \text{BBox Method Definitions} \]

\begin{align*}
\text{BBox Union(const BBox} & \mathbf{b}, \text{const Point} & \mathbf{p}) \{ \\
\text{BBox ret = } \mathbf{b}; \\
\text{ret.pMin.x = } \min(\mathbf{b}.pMin.x, \mathbf{p}.x); \\
\text{ret.pMin.y = } \min(\mathbf{b}.pMin.y, \mathbf{p}.y); \\
\text{ret.pMin.z = } \min(\mathbf{b}.pMin.z, \mathbf{p}.z); \\
\text{ret.pMax.x = } \max(\mathbf{b}.pMax.x, \mathbf{p}.x); \\
\text{ret.pMax.y = } \max(\mathbf{b}.pMax.y, \mathbf{p}.y); \\
\text{ret.pMax.z = } \max(\mathbf{b}.pMax.z, \mathbf{p}.z); \\
\text{return ret; }
\}
\end{align*}

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.

\[ \text{BBox Method Declarations} \]

\begin{align*}
\text{friend BBox Union(const BBox} & \mathbf{b}, \text{const BBox} & \mathbf{b2});
\end{align*}

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

\[ \text{BBox Method Definitions} \]

\begin{align*}
\text{BBox Intersection(const BBox} & \mathbf{b}_1, \text{const BBox} & \mathbf{b}_2) \{ \\
\text{BBox ret; } \\
\text{ret.pMin.x = } \max(b1.pMin.x, b2.pMin.x); \\
\text{ret.pMin.y = } \max(b1.pMin.y, b2.pMin.y); \\
\text{ret.pMin.z = } \max(b1.pMin.z, b2.pMin.z); \\
\text{ret.pMax.x = } \min(b1.pMax.x, b2.pMax.x); \\
\text{ret.pMax.y = } \min(b1.pMax.y, b2.pMax.y); \\
\text{ret.pMax.z = } \min(b1.pMax.z, b2.pMax.z); \\
\text{return ret; }
\}
\end{align*}

We can also easily determine if two BBoxes 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);
}

BBox Method Declarations

Float Volume() const {
    Vector d = pMax - pMin;
    return d.x * d.y * d.z;
}

2.6 Transformations

#include "lrt.h"
#include "geometry.h"
#include "1rt.h"
#include "geometry.h"

Transform Declarations

Transform Inline Functions

Transform Methods
In general, a *transformation* $T$ can be described as a mapping from points to points and from vectors to vectors:

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

The transformation $T$ 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 $T$ is an arbitrary linear transformation and $s$ is an arbitrary scalar, then $T(s\vec{v}) = sT(\vec{v})$ and $T(\vec{v_1} + \vec{v_2}) = T(\vec{v_1}) + T(\vec{v_2})$. These two properties can greatly simplify reasoning about transformations.

- **Continuous**: roughly speaking, $T$ leaves the neighborhoods around $p$ and $\vec{v}$ around $p$ and $\vec{v}$.

- **One-to-one and invertible**: for each $p$, $T$ maps $p$ to a single $p'$. Furthermore, for each $p'$, we can find an inverse transform such that $T^{-1}(p') = p$.

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 4x4 matrix can express the linear transformation of a point or vector from one frame to another. Furthermore, such a 4x4 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.

```cpp
\langle Transform Declarations \rangle

class Transform {
public:
    \langle Transform Constructor Declarations \rangle
    \langle Transform Method Declarations \rangle
private:
    \langle Transform Private Data \rangle
};
```
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 $m_{ij}$, 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:

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

**Transform Constructor Declarations**

Transform() {
    m = m_inv = new Matrix4x4;
}

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(const Reference<Matrix4x4> &mat) {
    m = mat;
    m_inv = m->Inverse();
}

Transform(const Reference<Matrix4x4> &mat, const Reference<Matrix4x4> &minv) {
    m = mat;
    m_inv = minv;
}

**Homogeneous coordinates**
Given a frame defined by \((\mathbf{p}, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)\), there is ambiguity between the representation of a point \((p_x, p_y, p_z)\) and a vector \((v_x, v_y, v_z)\) with equivalent coordinates. However, taking the definition of the representations of points and vectors, we can write the point as \(s_1 \mathbf{v}_1 + s_2 \mathbf{v}_2 + s_3 \mathbf{v}_3 + \mathbf{p}\) and the vector as \(s_1 \mathbf{v}_1 + s_2 \mathbf{v}_2 + s_3 \mathbf{v}_3 + \mathbf{p}\). 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

\[
\mathbf{M} = \begin{pmatrix}
m_{00} & m_{01} & m_{02} & m_{03} \\
m_{10} & m_{11} & m_{12} & m_{13} \\
m_{20} & m_{21} & m_{22} & m_{23} \\
m_{30} & m_{31} & m_{32} & m_{33}
\end{pmatrix}
\]

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

\[
\mathbf{x} = [1, 0, 0, 0]^T \\
\mathbf{y} = [0, 1, 0, 0]^T \\
\mathbf{z} = [0, 0, 1, 0]^T \\
\mathbf{p} = [0, 0, 0, 1]^T
\]

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 \(\Delta x\), \(\Delta y\), and \(\Delta z\), as shown in Figure 2.7.

The translation has some simple properties:
Translation should only affect points, leaving vectors unchanged.

In matrix form, the translation transformation is:

\[
\begin{bmatrix}
1 & 0 & 0 & \Delta x \\
0 & 1 & 0 & \Delta y \\
0 & 0 & 1 & \Delta z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

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 \( T(\Delta x, \Delta y, \Delta z) \) with a point \( p \) in homogeneous coordinates \([xyz 1]\):

\[
\begin{bmatrix}
1 & 0 & 0 & \Delta x \\
0 & 1 & 0 & \Delta y \\
0 & 0 & 1 & \Delta z \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix} =
\begin{bmatrix}
x + \Delta x \\
y + \Delta y \\
z + \Delta z \\
1
\end{bmatrix}
\]

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

\[
\begin{bmatrix}
1 & 0 & 0 & \Delta x \\
0 & 1 & 0 & \Delta y \\
0 & 0 & 1 & \Delta z \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
0
\end{bmatrix} =
\begin{bmatrix}
x \\
y \\
z \\
0
\end{bmatrix}
\]

The result is the same vector \( \mathbf{v} \). 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 Transforms could have a negative performance impact if called frequently, this doesn’t have a significant impact on lrt since new Transforms aren’t computed during the main rendering loop after the scene has been specified.

**Transform Methods**

```cpp
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

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 \(z\):

\[
S(2,2,1)(x,y,z) = (2x,2y,z) .
\]

It has the following basic properties:

\[
\begin{align*}
S(1,1,1) &= I \\
S(x_1,y_1,z_1) \times S(x_2,y_2,z_2) &= S(x_1x_2,y_1y_2,z_1z_2) \\
S^{-1}(x,y,z) &= S \left( \frac{1}{x}, \frac{1}{y}, \frac{1}{z} \right) 
\end{align*}
\]

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

\[
S(x,y,z) = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\]
Transform 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 $R_x(\theta)$, etc.. The rotation
around an arbitrary axis $(x,y,z)$ is denoted by $R_{(x,y,z)}(\theta)$.

Rotations also have some basic properties:

\[
R_\alpha(0) = I
\]
\[
R_\alpha(\theta_1) \times R_\alpha(\theta_2) = R_\alpha(\theta_1 + \theta_2)
\]
\[
R_\alpha(\theta_1) \times R_\alpha(\theta_2) = R_\alpha(\theta_2) \times R_\alpha(\theta_1)
\]
\[
R_\alpha^{-1}(\theta) = R_\alpha(-\theta) = R_\alpha^T(\theta)
\]

where $R^T$ is the matrix transpose of $R$. This property, that the inverse of $R$
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 $R$ is an orthonormal matrix; its
upper 3x3 components are all normalized and orthogonal to each other.

The matrix for rotation around the x axis is

\[
R_x(\theta) = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

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

\[
R_x(\theta) \cdot [1,0,0,0]^T = [1,0,0,0];
\]
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 $R_x(\theta)$, 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 $\theta$ 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.

\begin{verbatim}
(\textit{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());
}

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

$$R_y(\theta) = \begin{pmatrix} 
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1 
\end{pmatrix} \quad R_z(\theta) = \begin{pmatrix} 
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 
\end{pmatrix}$$

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

\textbf{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 $\theta$ 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 $v$.

$$p = \bar{a} \cos \alpha = \bar{a}(\vec{v} \cdot \bar{a}).$$
We now compute a pair of basis vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ in this plane. Trivially, one of them is

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

and the other can be computed with a cross product

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

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

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

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

**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());
}

**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 specifies 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.
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);
}

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;

And the other three columns aren’t much worse. First, we normalize the di-
rection 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);

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

```cpp
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.

\[\text{Transform Inline Functions}\] +≡

```c
inline void Transform::operator()(const Point &pt,
    Point *ptrans) const {
    Float x = pt.x, y = pt.y, z = pt.z;

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

### 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.

\[\text{Transform Inline Functions}\] +≡

```c
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);
}
```

\[\text{Transform Inline Functions}\] +≡

```c
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 \(\mathbf{N}\) and any tangent vector \(\mathbf{T}\) are orthogonal by construction, we know that

\[\mathbf{N} \cdot \mathbf{T} = \mathbf{N}^T \mathbf{T} = 0.\]
When we transform a point on the surface by some matrix $M$, the new tangent vector $\mathbf{T}'$ at the transformed point is simply $M\mathbf{T}$. The transformed normal $\mathbf{N}'$ should be equal to $S\mathbf{N}$ for some $4 \times 4$ matrix $S$. To maintain the orthogonality requirement, we must have:

$$0 = \mathbf{N}'^T \mathbf{T}' = S\mathbf{N'}^T M \mathbf{T} = \mathbf{N}^T S^T M \mathbf{T}$$

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

**Transform Method Declarations**

```cpp
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 Vectors).

**Transform Inline Functions**

```cpp
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);
}
```
\[\textbf{Transform Inline Functions}\]

\begin{verbatim}
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;
}
\end{verbatim}

\begin{verbatim}
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;
}
\end{verbatim}

\begin{verbatim}
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);
}
\end{verbatim}

\section*{Rays}

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

\begin{verbatim}
inline Ray 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);
}
\end{verbatim}

\section*{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.
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 $4\times4$ 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. $A(B(C(p))) = T(p)$. Such a transformation $T$ can be computed by multiplying the matrices of the transformations $A$, $B$, and $C$ together. In code, we can write:

$$ \text{Transform } T = A \cdot B \cdot C; $$

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

We use the C++ $\ast$ operator to compute the new transformation that results from post-multiplying the current transformation with a new transformation $t_2$. From the definition of matrix multiplication, the $(i,j)$th element of the resulting matrix $\text{ret}$ 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 $t_2.m_{\text{inv}} \ast m_{\text{inv}}$; this is a result of the matrix identity

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

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 \(\alpha\), and an angle \(\theta\) 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

\[
\mathbf{R}(x, y, z, \theta) = \mathbf{T}(x, y, z) \ast \mathbf{R}(\theta, \alpha) \ast \mathbf{T}(-x, -y, -z)
\]

We will not include code for this operation as it won’t be necessary for implementing \(\mathbf{lrt}\). 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 \(\mathbf{P}\)
- A local coordinate system at the hit point, given by the surface normal \(\mathbf{N}\) and two tangent vectors \(\mathbf{S}\) and \(\mathbf{T}\)
- The parametric partial derivatives \(\partial \mathbf{P} / \partial u\) and \(\partial \mathbf{P} / \partial v\).
- The partial derivatives of the change in surface normal \(\partial \mathbf{N} / \partial u\) and \(\partial \mathbf{N} / \partial v\).
- \(u, v\) coordinates from the parameterization of the surface.
- A pointer to the \texttt{Shape} that the differential geometry lies on; the shape class will be introduced in the next chapter. See Figure 2.12 for a 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 \(\mathbf{P} = f(u, v)\). Though this isn’t true for all of the shape representations that are used in graphics, all of the shapes that \(\mathbf{lrt}\) 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.

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

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

\begin{align*}
\text{DifferentialGeometry Method Declarations} & \equiv \\
\text{DifferentialGeometry}(\text{const Point } & p, \text{ const Vector } & dpdu, \\
\text{ const Vector } & dpdv, \text{ const Vector } & dndu, \\
\text{ const Vector } & dndv, \text{ Float } uu, \text{ Float } vv, \\
\text{ const Shape } & \ast sh) \\
& : P(p), dPdu(dpdu), dPdv(dpdv), dNdu(dndu), dNdv(dndv) \\
& \{ \\
& \text{Initialize DifferentialGeometry from parameters} \\
& \}
\end{align*}

\begin{align*}
\text{Initialize DifferentialGeometry from parameters} & \equiv \\
\text{Nb} = \text{Normal}(\text{Cross}(dPdu, dPdv)); \\
\text{Nn} = \text{Nb.Hat}(); \\
\text{S} = dPdu.Hat(); \\
\text{T} = \text{Cross}(S, Nn); \\
\text{u} = uu; \\
\text{v} = vv; \\
\text{shape} = sh;
\end{align*}

It is useful to be able to transform direction vectors from world space to the coordinate frame defined by the three basis directions $\vec{S}$, $\vec{T}$, and $\vec{N}$. 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 $\vec{S}$, $\vec{T}$, and $\vec{N}$ in world-space, the matrix $M$ that transforms vectors in world space to the local differential geometry space is:

\[
M = \begin{pmatrix}
S_x & S_y & S_z \\
T_x & T_y & T_z \\
N_x & N_y & N_z
\end{pmatrix} = \begin{pmatrix}
\vec{S} \\
\vec{T} \\
\vec{N}
\end{pmatrix}
\]

To confirm this yourself, consider the value of $M\vec{N} = (\vec{S} \cdot \vec{N}, \vec{T} \cdot \vec{N}, \vec{N} \cdot \vec{N})$. Since $\vec{S}$, $\vec{T}$, and $\vec{N}$ are all orthonormal, the $x$ and $y$ two components of $M\vec{N}$ are zero. Since $\vec{N}$ is normalized, $\vec{N} \cdot \vec{N} = 1$. Thus, $M\vec{N} = (0, 0, 1)$. (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.)

\begin{align*}
\text{DifferentialGeometry Method Declarations} & + \equiv \\
\text{Vector WorldToLocal(const Vector } & \ast v) \text{ const} \\
& \{ \\
& \text{return Vector(Dot}(v, S), \text{ Dot}(v, T), \text{ Dot}(v, \text{ Nn})).\text{Hat}(); \\
& \}
\end{align*}

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:

\begin{align*}
\text{DifferentialGeometry Method Declarations} & + \equiv \\
\text{Vector LocalToWorld(const Vector } & \ast v) \text{ const} \\
& \{ \\
& \text{return Vector}(S.x \ast v.x + T.x \ast v.y + Nn.x \ast v.z, \\
& S.y \ast v.x + T.y \ast v.y + Nn.y \ast v.z, \\
& S.z \ast v.x + T.z \ast v.y + Nn.z \ast v.z);
& \}
\end{align*}
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 coordinate-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.
3. Shapes

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

\(<Shape Declarations>\)≡

```cpp
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.

\(<Shape Method Definitions>\)≡

```cpp
Shape::Shape(const Transform &o2w)
    : ObjectToWorld(o2w) {
    WorldToObject = ObjectToWorld.GetInverse();
}
```

\(<Shape Protected Data>\)≡

```cpp
Transform ObjectToWorld, WorldToObject;
```

\(<Update shape creation statistics>\)≡

```cpp
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.

\(<Shape Interface>\)⁺≡

```cpp
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.

\begin{verbatim}
virtual BBox WorldBound() const {
    return ObjectToWorld(ObjectBound());
}
\end{verbatim}

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.

\begin{verbatim}
virtual bool CanIntersect() const { return true; }
\end{verbatim}

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.

\begin{verbatim}
virtual void Refine(vector<RefinedShape> &refined) const {
    Severe("Unimplemented Shape::Refine() method called");
}
\end{verbatim}

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.

\[ \text{(Shape Declarations)} \]

\[
\text{struct RefinedShape} \\
\text{(const Reference<Shape> &s, Texture<Normal> *ns = NULL,} \\
\text{Texture<Vector> *ss = NULL) {} \\
\text{shape = s;} \\
\text{Ns = ns;} \\
\text{Ss = ss;}
\]

\[
\text{Reference<Shape> shape;} \\
\text{Texture<Normal> *Ns;} \\
\text{Texture<Vector> *Ss;}
\]

\section*{Intersection}

We will provide two separate intersection routines. The first, \texttt{Intersect}, returns information about a single ray-shape intersection corresponding to the intersection in the \([\text{mint}, \text{maxt}]\) parametric range along the ray. The other, \texttt{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 \texttt{IntersectP}.

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

- Recall that the \texttt{Ray} structure contains \texttt{mint} and \texttt{maxt} variables which define a ray segment from the point \texttt{ray(mint)} to the point \texttt{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 \texttt{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 \texttt{DifferentialGeometry} structure, which completely captures the local geometric properties of a surface. This type will be used heavily throughout \texttt{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 \texttt{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.

\texttt{Shape Interface} \begin{verbatim}
    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;
    }
\end{verbatim}

\textbf{Surface Area}

In order to properly use \texttt{Shape}s as area lights, we 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.

\texttt{Shape Interface} \begin{verbatim}
    virtual Float Area() const {
        Severe("Unimplemented Shape::Area() method called");
        return 0.;
    }
\end{verbatim}

\textbf{3.2 Spheres}

\texttt{sphere.cc} \begin{verbatim}
#include "lrt.h"
#include "shapes.h"
#include "mc.h"
\end{verbatim}

\texttt{Sphere Declarations} \begin{verbatim}
class Sphere: public Shape {
    public:
        \texttt{Sphere Interface}
    private:
        \texttt{Sphere Data}
};
\end{verbatim}

Spheres are a special case of a general type of surface called \textit{quadrics}. Quadrics
are surfaces described by quadratic polynomials in \(x, y, \text{ and } z\); 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 function 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 \( (\theta, \phi) \) where \( \theta \) ranges from 0 to \( \pi \) and \( \phi \) ranges from 0 to \( 2\pi \) for a complete sphere.

\[
\begin{align*}
  x &= r \cos \phi \cos \theta \\
  y &= r \sin \phi \cos \theta \\
  z &= r \sin \theta
\end{align*}
\]

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

\[
\begin{align*}
  \phi &= u \cdot \phi_{\text{max}} \\
  \theta &= \theta_{\text{min}} + v \cdot (\theta_{\text{max}} - \theta_{\text{min}})
\end{align*}
\]

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 \( \theta \) value. The sphere sweeps out \( \theta \) values from 0 to the given \( \theta_{\text{max}} \) such that the section of the sphere with spherical \( \theta \) values above this \( \theta \) is also removed.
Sec. 3.2] Spheres

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 $\phi$ value.

\[ \text{Sphere Methods} \]
\begin{verbatim}
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));
  }
\end{verbatim}

\[ \text{Sphere Data} \]
\begin{verbatim}
Float radius;
Float phiMax;
Float zmin, zmax;
Float thetaMin, thetaMax;
\end{verbatim}

Bounding

Computing a bounding box for a sphere is straightforward. We will use the values of $z_{\text{min}}$ and $z_{\text{max}}$ 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 $\theta_{\text{max}}$ and see if we can compute a tighter bounding box when that is less than $2\pi$.

\[ \text{Sphere Methods} + \]
\begin{verbatim}
BBox Sphere::ObjectBound() const {
  return BBox(Point(-radius, -radius, zmin),
               Point( radius, radius, zmax));
}
\end{verbatim}

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.

```
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 \( \phi \)
        // Test sphere intersection against clipping parameters
        // Fill in DifferentialGeometry from sphere hit
        // Update thitp for quadric intersection
        return true;
    }
}
```

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.

\(\text{Transform Ray to object space}\)

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:

\[ (o(r)_x + t\hat{d}(r)_x)^2 + (o(r)_y + t\hat{d}(r)_y)^2 + (o(r)_z + t\hat{d}(r)_z)^2 = r^2.\]

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 efficiently 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

\[ A = \tilde{d}(r)_x^2 + \tilde{d}(r)_y^2 + \tilde{d}(r)_z^2 \]
\[ B = 2(\tilde{d}(r)_x o(r)_x + \tilde{d}(r)_y o(r)_y + \tilde{d}(r)_z o(r)_z) \]
\[ C = o(r)_x^2 + o(r)_y^2 + o(r)_z^2 - r^2 \]

This directly translates to this fragment of source code.

\textit{Compute quadratic sphere coefficients}:

\begin{verbatim}
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;
\end{verbatim}

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:

\[ t_0 = \frac{-B - \sqrt{B^2 - 4AC}}{2A} \]
\[ t_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \]

We will provide a utility \texttt{Quadratic} function that solves a quadratic equation, returning \texttt{false} if there are no real solutions and returning \texttt{true} and setting \( t_0 \) and \( t_1 \) appropriately if there are solutions.

\textit{Solve quadratic equation for \( t \) values}:

\begin{verbatim}
Float t0, t1;
if (!Quadratic(A, B, C, &t0, &t1))
    return false;
\end{verbatim}

\textit{Compute intersection distance along ray}:

\begin{verbatim}
inline bool Quadratic(Float A, Float B, Float C, Float *t0, Float *t1) {
    \textit{Find quadratic discriminant}
    Float discrim = B * B - 4.f * A * C;
    if (discrim < 0.) return false;
    Float rootDiscrim = sqrtf(discrim);
    \textit{Compute quadratic \( t \) values}
    \begin{align*}
    t_0 &= \frac{-B - \sqrt{B^2 - 4AC}}{2A} \\
    t_1 &= \frac{-B + \sqrt{B^2 - 4AC}}{2A}
    \end{align*}
}
\end{verbatim}

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

\textit{Find quadratic discriminant}:

\begin{verbatim}
    Float discrim = B * B - 4.f * A * C;
    if (discrim < 0.) return false;
    Float rootDiscrim = sqrtf(discrim);
\end{verbatim}

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

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

\(\langle\text{Compute quadratic t values}\rangle\equiv\)

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 \(\text{mint}\) to \(\text{maxt}\). Since \(t_0\) is guaranteed to be less than \(t_1\) (and \(\text{mint}\) less than \(\text{maxt}\)), if \(t_0\) is greater than \(\text{maxt}\) or \(t_1\) is less than \(\text{mint}\), 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 \(\text{mint}\), however, in which case we ignore it and try \(t_1\). If that is also out of range, we have no valid intersection. Otherwise \(t_{\text{hit}}\) holds the distance to the hit.

\(\langle\text{Compute intersection distance along ray}\rangle\equiv\)

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 \(\phi\) ranges. Intersections that are in clipped areas need to be ignored.

We start by computing the object space position of the intersection, \(P_{\text{hit}}\) and the \(\phi\) value for the hit point. Taking the parametric equations for the sphere,

\[\begin{align*}
\frac{y}{x} &= \frac{r \sin \phi \cos \theta}{r \cos \phi \cos \theta} = \tan \phi \\
\end{align*}\]

so \(\phi = \arctan y/x\).

\(\langle\text{Compute sphere hit position and } \phi\rangle\equiv\)

\(P_{\text{hit}} = \text{ray(\text{thit})};\)
\(\phi = \tan2f(P_{\text{hit}}.y, P_{\text{hit}}.x);\)
if (phi < 0.) phi += 2.f*M_PI;
We remap the result from the standard library’s `atan2` function to be between 0 and $2\pi$, to match the sphere’s original definition.

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

\[\text{Test sphere intersection against clipping parameters}\]
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 $\phi$\}
  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 $\phi$ value for the hit to lie between 0 and 1 and by computing a $\theta$ value for the hit point which is also mapped to $[0, 1]$, based on the range of $\theta$ 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.

\[\text{Fill in DifferentialGeometry from sphere hit}\]
Float u = phi / phiMax;
Float theta = asinf(Phit.z / radius);
Float v = (theta - thetaMin) / (thetaMax - thetaMin);
\(\{\text{Compute sphere } \partial P/\partial u \text{ and } \partial P/\partial v\}\)
\(\{\text{Compute sphere } \partial N/\partial u \text{ and } \partial N/\partial v\}\)
*dg = DifferentialGeometry(ObjectToWorld(Phit), ObjectToWorld(dPdu),
ObjectToWorld(dPdv), ObjectToWorld(dNdudv), ObjectToWorld(dNdvdv),
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
\[
x = r \cos \phi \cos \theta = r \cos (\phi_{\text{max}} u) \cos (\theta_{\text{min}} + v(\theta_{\text{max}} - \theta_{\text{min}}))
\]

Consider the first component of $\partial P/\partial u$, $\partial x/\partial u$:
\[
\frac{\partial x}{\partial u} = \frac{\partial}{\partial u} (r \cos \phi \cos \theta) = r \cos \theta \frac{\partial}{\partial u} (\cos \phi) = r \cos \theta (-\phi_{\text{max}} \sin \phi)
\]

Using a substitution based on the parametric definition of the sphere’s $y$ coordinate, this simplifies to
\[
\frac{\partial x}{\partial u} = -\phi_{\text{max}} y.
\]
Similarly
\[ \frac{\partial y}{\partial u} = \phi_{\text{max}}, \]
and
\[ \frac{\partial z}{\partial u} = 0. \]

A similar process gives us \( \partial P / \partial v \).

\[
\frac{\partial P}{\partial u} = (-\phi_{\text{max}}, \phi_{\text{max}}, 0)
\]
\[
\frac{\partial P}{\partial v} = (\theta_{\text{max}} - \theta_{\text{min}})(-z \cos \phi, -z \sin \phi, r \cos \theta)
\]

\textit{(Compute sphere} \( \partial P / \partial u \) \textit{and} \( \partial P / \partial v \)\textit{)}

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 \textit{Weingarten equations} from differential geometry. They
are:

\[
\frac{\partial N}{\partial u} = \frac{f F - e G \frac{\partial P}{\partial u} + e F - f E \frac{\partial P}{\partial v}}{E G - F^2 \frac{\partial u}{\partial u}}
\]
\[
\frac{\partial N}{\partial v} = \frac{g F - f G \frac{\partial P}{\partial u}}{E G - F^2 \frac{\partial u}{\partial v}} + \frac{f F - g E \frac{\partial P}{\partial v}}{E G - F^2 \frac{\partial v}{\partial v}}
\]

where \( E, F, \) and \( G \) are coefficients of the \textit{first fundamental form} and are given by

\[
E = \left| \frac{\partial P}{\partial u} \right|^2
\]
\[
F = \left( \frac{\partial P}{\partial u} \cdot \frac{\partial P}{\partial v} \right)
\]
\[
G = \left| \frac{\partial P}{\partial v} \right|^2.
\]

These are easily computed with the \( \partial P / \partial u \) and \( \partial P / \partial v \) values that are already avail-
abe. \( e, f, \) and \( g \) are coefficients of the \textit{second fundamental form},

\[
e = (N \cdot \frac{\partial^2 P}{\partial u^2})
\]
\[
f = (N \cdot \frac{\partial^2 P}{\partial u \partial v})
\]
\[
g = (N \cdot \frac{\partial^2 P}{\partial v^2}).
\]
For these, we need to compute the second order partial derivatives $\frac{\partial^2 P}{\partial u^2}$ 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:

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

\[\text{Compute sphere } \frac{\partial N}{\partial u} \text{ and } \frac{\partial N}{\partial v}\]

\begin{align*}
\text{Vector } d2Pduu &= -\phi_{\text{Max}} \times \phi_{\text{Max}} \times \text{Vector}(\text{Phit}.x, \text{Phit}.y, 0); \\
\text{Vector } d2Pduv &= (z_{\text{max}} - z_{\text{min}}) \times \text{Phit}.z \times \phi_{\text{Max}} \times \\
&\text{Vector}(\sin \phi, -\cos \phi, 0.); \\
\text{Vector } d2Pdvv &= -(\theta_{\text{Max}} - \theta_{\text{Min}}) \times (\theta_{\text{Max}} - \theta_{\text{Min}}) \times \\
&\text{Vector}(\text{Phit}.x, \text{Phit}.y, \text{Phit}.z);
\end{align*}

\[\text{Compute coefficients for fundamental forms}\]

\[\text{Compute } \frac{\partial N}{\partial u} \text{ and } \frac{\partial N}{\partial v} \text{ from fundamental form coefficients}\]

\begin{align*}
\text{Float } E &= \text{Dot}(dPdu, dPdu); \\
\text{Float } F &= \text{Dot}(dPdu, dPdv); \\
\text{Float } G &= \text{Dot}(dPdv, dPdv); \\
\text{Vector } N &= \text{Cross}(dPdu, dPdv); \\
\text{Float } e &= \text{Dot}(N, d2Pduu); \\
\text{Float } f &= \text{Dot}(N, d2Pduv); \\
\text{Float } g &= \text{Dot}(N, d2Pdvv);
\end{align*}

\[\text{Compute } \frac{\partial N}{\partial u} \text{ and } \frac{\partial N}{\partial v} \text{ from fundamental form coefficients}\]

\begin{align*}
\text{Float } \text{invEGF2} &= 1.f / (E*G - F*F); \\
\text{Vector } dNdu &= (f*F - e*G) \times \text{invEGF2} \times dPdu + (e*F - f*E) \times \text{invEGF2} \times dPdv; \\
\text{Vector } dNdv &= (g*F - f*G) \times \text{invEGF2} \times dPdu + (f*F - g*E) \times \text{invEGF2} \times dPdv;
\end{align*}

Since there is an intersection at parametric distance $\text{thit}$ along the ray, we update the $\text{thit}$ 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.

\[\text{Update } \text{thit} \text{ for quadric intersection}\]

* $\text{thit} = \text{thit};$

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**

```cpp
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 \( \phi \)
    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

\[
2\pi \int_a^b f(x) \sqrt{1 + (f'(x))^2} \, dx,
\]

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

\[
\phi_{\text{max}} \int_a^b f(x) \sqrt{1 + (f'(x))^2} \, dx.
\]

Our sphere is a surface of revolution of a circular arc. Recall that the sphere is clipped at \( z_{\text{min}} \) and \( z_{\text{max}} \). 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

\[
A = \phi_{\text{max}} \int_{z_0}^{z_1} \sqrt{r^2 - x^2} \sqrt{1 + \frac{x^2}{r^2 - x^2}} \, dx
\]

\[
= \phi_{\text{max}} \int_{z_0}^{z_1} \sqrt{r^2 - x^2 + x^2} \, dx
\]

\[
= \phi_{\text{max}} \int_{z_0}^{z_1} \sqrt{r^2} \, dx
\]

\[
= \phi_{\text{max}} r(z_1 - z_0)
\]

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

```cpp
float Sphere::Area() const {
    return phiMax * radius * (zmax-zmin);
}
```
3.3 Cylinders

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 \( \phi \) sweep value. In parametric form, a cylinder is described by the equations:

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

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 \( \phi \) value.

```cpp
// cylinder.cc

#include "lrt.h"
#include "shapes.h"

Cylinder Declarations

class Cylinder: public Shape {
public:
    // Cylinder Interface

protected:
    // Cylinder Data
};
```

Construction

...
\textbf{Cylinder Methods}:
\begin{verbatim}
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));
  }
\end{verbatim}

\textbf{Cylinder Data}:
\begin{verbatim}
Float radius;
Float zmin, zmax;
Float phiMax;
\end{verbatim}

\section*{Bounding}
Like the sphere, we compute a conservative bounding box for the cylinder using the \( z \) range but without taking into account the maximum \( \phi \).

\textbf{Cylinder Methods}:
\begin{verbatim}
BBox Cylinder::ObjectBound() const {
    Point p1 = Point(-radius, -radius, zmin);
    Point p2 = Point( radius, radius, zmax);
    return BBox(p1, p2);
}
\end{verbatim}

\section*{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:
\[ (o(r)_x + \tilde{a}(r)_x)^2 + (o(r)_y + \tilde{a}(r)_y)^2 = r^2. \]
When we expand this and find the coefficients of the quadratic equation \( At^2 + Bt + C \), we get:
\[ A = \tilde{d}(r)_x^2 + \tilde{a}(r)_y^2 \]
\[ B = 2(\tilde{d}(r)_x o(r)_x + \tilde{a}(r)_y o(r)_y) \]
\[ C = o(r)_x^2 + o(r)_y^2 - r^2 \]

\textbf{Compute quadratic cylinder coefficients}:
\begin{verbatim}
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;
\end{verbatim}
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.

\( \text{Cylinder Methods} \)

```cpp
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 \( \phi \))
  (Test cylinder intersection against clipping parameters)
  (Fill in DifferentialGeometry from cylinder hit)
  (Update \( \text{thitp for quadric intersection} \))
  return true;
}
```

\( \text{Partial Cylinders} \)

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

\( \text{Compute cylinder hit point and } \phi \)

```cpp
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.

\( \text{Test cylinder intersection against clipping parameters} \)

```cpp
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 \( \phi \))
  if (Phit.z < zmin || Phit.z > zmax || phi > phiMax)
    return false;
}
```

Again like the sphere the \( u \) value is computed by scaling \( \phi \) to lie between 0 and 1. Straightforward inversion of the parametric equation for the cylinder’s \( z \) value gives us the \( v \) parametric coordinate.
\(\text{Fill in DifferentialGeometry from cylinder hit}\)\

Float \(u = \phi / \phi_{\text{Max}}\);\
Float \(v = (\text{Phit.z} - \text{zmin}) / (\text{zmax} - \text{zmin})\);

\(<\text{Compute cylinder } \partial P / \partial u \text{ and } \partial P / \partial v>\)\
\(<\text{Compute cylinder } \partial N / \partial u \text{ and } \partial N / \partial v>\)

*\(\text{dg} = \text{DifferentialGeometry(}\text{ObjectToWorld(Phit)}, \text{ObjectToWorld(dPdu)},\<\)\
\(<\text{ObjectToWorld(dPdv)}, \text{ObjectToWorld(dNdv)}, \text{ObjectToWorld(dNdv)},\>)\)\
u, v, this);\

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

\[\frac{\partial P}{\partial u} = (-\phi_{\text{Max}}y, \phi_{\text{Max}}x, 0)\]
\[\frac{\partial P}{\partial v} = (0, 0, z_{\text{Max}} - z_{\text{Min}})\]\

\(<\text{Compute cylinder } \partial P / \partial u \text{ and } \partial P / \partial v>\)\

Vector \(\text{dPdu}(-\phi_{\text{Max}} \times \text{Phit.y}, \phi_{\text{Max}} \times \text{Phit.x}, 0)\);
Vector \(\text{dPdv}(0, 0, \text{zmax} - \text{zmin})\);

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

\[\frac{\partial^2 P}{\partial u^2} = -\phi_{\text{Max}}^2 (x, y, 0)\]
\[\frac{\partial^2 P}{\partial u \partial v} = (0, 0, 0)\]
\[\frac{\partial^2 P}{\partial v^2} = (0, 0, 0)\]\

\(<\text{Compute cylinder } \partial N / \partial u \text{ and } \partial N / \partial v>\)\

Vector \(\text{d2Pdu}u = -\phi_{\text{Max}} \times \phi_{\text{Max}} \times \text{Vector(Phit.x, Phit.y, 0)}\);\
Vector \(\text{d2Pduv}(0, 0, 0), \text{d2Pdvv}(0, 0, 0)\);

\(<\text{Compute coefficients for fundamental forms}>\)\
\(<\text{Compute } \partial N / \partial u \text{ and } \partial N / \partial v \text{ from fundamental form coefficients}>\)

\textbf{Surface Area}

A cylinder is just a rolled rectangle. The height of the rectangle is \(z_{\text{Max}} - z_{\text{Min}}\), and the width is \(\phi_{\text{Max}}\): \(\text{Cylinder Methods}\)\+\

Float \text{Cylinder::Area()} const {\
    return (\text{zmax} - \text{zmin}) * \phi_{\text{Max}} * \text{radius};\
}
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 $\phi$ value.

### 3.4 Disks

```c
#include "lrt.h"
#include "shapes.h"

Disk Declarations

``` Disk Methods ```

class Disk : public Shape {
public:
  \[ \phi = u \phi_{\text{max}} \]
  \[ x = r(1 - v) \cos \phi \]
  \[ x = r(1 - v) \sin \phi \]
  \[ z = \text{height} \]

Construction
```
\[ \textit{Disk Methods} \equiv \]
\[ \text{Disk}::\text{Disk} (\text{const Transform } \& o2w, \text{ Float } ht, \text{ Float } r, \text{ Float } tmax) \]
\[ : \text{ Shape} (o2w) \{ \]
\[ \text{height} = ht; \]
\[ \text{radius} = r; \]
\[ \text{phiMax} = \text{Radians} (\text{Clamp} (tmax, 0.0f, 360.0f)); \]
\[ \} \]

\[ \textit{Disk Private Data} \equiv \]
\[ \text{Float } \text{height}, \text{ radius}, \text{ phiMax}; \]

\section*{Bounding}

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

\[ \textit{Disk Methods} \oplus \equiv \]
\[ \text{BBox } \text{Disk}::\text{ObjectBound} () \text{ const } \{ \]
\[ \text{return } \text{BBox} (\text{Point} (-\text{radius}, -\text{radius}, \text{height}), \]
\[ \text{Point} (\text{radius}, \text{radius}, \text{height})); \]
\[ \} \]

\section*{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.

\[ \textit{Disk Methods} \oplus \equiv \]
\[ \text{bool } \text{Disk}::\text{Intersect} (\text{const Ray } \& r, \text{ Float } * \text{thitp}, \]
\[ \text{DifferentialGeometry } * \text{dg} ) \text{ const } \{ \]
\[ \textit{Transform Ray to object space} \]
\[ \textit{Compute plane intersection for disk} \]
\[ \textit{See if hit point is inside disk radius and } \phi_{\text{max}} \]
\[ \textit{Fill in DifferentialGeometry from disk hit} \]
\[ \textit{Update thitp for quadric intersection} \]
\[ \text{return } \text{true}; \]
\[ \} \]

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,

\[ h = o(r)_z + t * \tilde{d}(r)_z \]

So \( t \) is

\[ t = \frac{h - o(r)_z}{\tilde{d}(r)_z} \]
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, \([mint, maxt]\). If not, we can return false.

\[
\text{Compute plane intersection for disk} \equiv \\
\text{if (fabsf(ray.D.z) < 1e-7) return false;}
\text{Float thit = (height - ray.O.z) / ray.D.z;}
\text{if (thit < ray.mint || thit > ray.maxt)}
\text{return false;}
\]

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 \( \text{radius} \). 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 \( \text{height} \).

\[
\text{See if hit point is inside disk radius and } \phi_{max} \equiv \\
\text{Point Phit = ray(thit);}
\text{Float Dist2 = Phit.x * Phit.x + Phit.y * Phit.y;}
\text{if (Dist2 > radius * radius)}
\text{return false;}
\]

\[
\text{Test disk } \phi \text{ value against } \phi_{max} \equiv \\
\text{Float phi = atan2f(Phit.y, Phit.x);}
\text{if (phi < 0) phi += 2. * M_PI;}
\text{if (phi > phiMax)}
\text{return false;}
\]

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 \((0,0,0)\).

\[
\text{Fill in DifferentialGeometry from disk hit} \equiv \\
\text{Float u = phi / phiMax;}
\text{Float v = 1.f - (sqrtf(Dist2) / radius);}
\text{Vector dPdu(-phiMax * Phit.y, phiMax * Phit.x, 0.);}
\text{Vector dPdv(-Phit.x / (1-v), -Phit.y / (1-v), 0.);}
\text{*dg = DifferentialGeometry(ObjectToWorld(Phit), ObjectToWorld(dPdu),}
\text{ObjectToWorld(dPdv), Vector(0,0,0), Vector(0,0,0), u, v, this);}
\]

**Surface Area**

Disks have trivial surface area, since they’re just portions of a circle:
\section{Disk Methods} \(\equiv\)

Float Disk::Area() const {
    return phiMax * 0.5f * radius * radius;
}

\section{Other Quadrics}

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

\texttt{lrl} 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
\[
\begin{align*}
\phi &= u \phi_{\text{max}} \\
x &= r (1 - v) \cos \phi \\
y &= r (1 - v) \sin \phi \\
z &= v \text{height}
\end{align*}
\]

The partial derivatives are:
\[
\begin{align*}
\frac{\partial P}{\partial u} &= (-\phi_{\text{max}} y, \phi_{\text{max}} x, 0) \\
\frac{\partial P}{\partial v} &= (-x/(1 - v), y/(1 - v), \text{height})
\end{align*}
\]

and
\[
\begin{align*}
\frac{\partial^2 P}{\partial u^2} &= -\phi_{\text{max}} (x, y, 0) \\
\frac{\partial^2 P}{\partial u \partial v} &= \phi_{\text{max}} (y, -x, 0) \\
\frac{\partial^2 P}{\partial v^2} &= (0, 0, 0)
\end{align*}
\]

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
\[
\begin{align*}
\phi &= u \phi_{\text{max}} \\
z &= v (z_{\text{max}} - z_{\text{min}}) \\
r &= r_{\text{max}} \sqrt{z/z_{\text{max}}} \\
x &= r \cos \phi \\
y &= r \sin \phi
\end{align*}
\]
The partial derivatives are:

\[
\frac{\partial P}{\partial u} = (-\phi_{\text{max}} y, \phi_{\text{max}} x, 0)
\]

\[
\frac{\partial P}{\partial v} = (z_{\text{max}} - z_{\text{min}})(x/z, y/z, 1)
\]

and

\[
\frac{\partial^2 P}{\partial u^2} = -\phi_{\text{max}}^2 (x, y, 0)
\]

\[
\frac{\partial^2 P}{\partial u \partial v} = \phi_{\text{max}} (z_{\text{max}} - z_{\text{min}})(-y/z, x/z, 0)
\]

\[
\frac{\partial^2 P}{\partial v^2} = -2(z_{\text{max}} - z_{\text{min}})^2(x/z^2, y/z^2, 0)
\]

Finally, the implicit form of the hyperboloid is

\[x^2 + y^2 - z^2 = -1\]

and the parametric form is

\[\phi = u \phi_{\text{max}}\]

\[x_r = (1-v)x_1 + vx_2\]

\[y_r = (1-v)y_1 + vy_2\]

\[x = x_r \cos \phi - y_r \sin \phi\]

\[y = x_r \sin \phi + y_r \cos \phi\]

\[z = (1-v)z_1 + vz_2\]

The partial derivatives are:

\[
\frac{\partial P}{\partial u} = (-\phi_{\text{max}} y, \phi_{\text{max}} x, 0)
\]

\[
\frac{\partial P}{\partial v} = ((x_2 - x_1) \cos \phi - (y_2 - y_1) \sin \phi, (x_2 - x_1) \sin \phi + (y_2 - y_1) \cos \phi, z_2 - z_1)
\]

and

\[
\frac{\partial^2 P}{\partial u^2} = -\phi_{\text{max}}^2 (x, y, 0)
\]

\[
\frac{\partial^2 P}{\partial u \partial v} = \phi_{\text{max}} (-\partial y/\partial v; \partial x/\partial v; 0)
\]

\[
\frac{\partial^2 P}{\partial v^2} = (0, 0, 0)
\]
3.6 Triangles and Meshes

```cpp
#include "lrt.h"
#include "shapes.h"
#include "paramset.h"
```

```cpp
class TriangleMesh: public Shape {
public:
    ... (TriangleMesh Interface)
protected:
    ... (TriangleMesh Data)
};
```

The triangle is one of the most commonly used shapes in computer graphics. rrt 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:

- \( n_t \) Number of triangles in this mesh
- \( n_v \) Number of vertices in this mesh
- \( v_i \) Pointer to an array of vertex indices. For the \( i \)th triangle, its three vertex positions are \( P[v_i[3*i]], P[v_i[3*i+1]], \) and \( P[v_i[3*i+2]] \).
- \( P \) Array of \( n_v \) vertex positions.
- \( u_v \) 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 \( v_i \) and \( P \), since the caller retains ownership of the data being passed in.
\section*{TriangleMesh Methods}

\begin{verbatim}
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\}
}
\end{verbatim}

\section*{TriangleMesh Data}

\begin{verbatim}
int ntris;
int nverts;
int *vertexIndex;
Point *p;
Float *uvs;
\end{verbatim}

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.

\section*{Transform mesh vertices to world space}

\begin{verbatim}
for (int i = 0; i < nverts; ++i)
    p[i] = ObjectToWorld(P[i]);
\end{verbatim}

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 positions back to object space before computing their bound.

\section*{TriangleMesh Methods}

\begin{verbatim}
BBox TriangleMesh::ObjectBound() const {
    BBox bobj;
    for (int i = 0; i < nverts; i++)
        bobj = Union(bobj, WorldToObject(p[i]));
    return bobj;
}
\end{verbatim}
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.

\( \text{TriangleMesh Methods} \) + \( \equiv \)

```cpp
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 TriangleMeshes can not be intersected directly.

\( \text{TriangleMesh Interface} \) + \( \equiv \)

```cpp
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.

\( \text{TriangleMesh Methods} \) + \( \equiv \)

```cpp
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**

\( \text{Triangle Declarations} \) + \( \equiv \)

```cpp
class Triangle : public Shape {
public:
    \( \text{Triangle Interface} \)
    //private:
    \( \text{Triangle Data} \)
};
```

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.
〈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〉≡
#ifndef 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 tri-
angles to approximate the smooth subdivision limit surface—ray intersections are
performed against these triangles, rather than directly against the subdivision sur-
face.
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 \((1, 1)\).

```cpp
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]];  
    }
    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 \(b_2\):

\[
  p(b_1, b_2) = (1 - b_1 - b_2)p_0 + b_1p_1 + b_2p_2
\]

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.

\[
  o(r) + \vec{d}(r) = (1 - b_1 - b_2)p_0 + b_1p_1 + b_2p_2
\]

(3.6.1)

Following the technique described by Möller and Trumbore(MT97), we use the shorthand notation \(\vec{e}_1 = p_1 - p_0, \vec{e}_2 = p_2 - p_0,\) and \(\vec{t} = o(r) - p_0\). We can now
rearrange terms of Equation 3.6.1 to obtain the matrix equation:

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

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 $\begin{vmatrix} \mathbf{a} & \mathbf{b} & \mathbf{c} \end{vmatrix}$ notation—determinant of a 3x3 matrix. XXX

\[
\begin{bmatrix}
t \\
b_1 \\
b_2
\end{bmatrix} = \frac{1}{-\mathbf{d}(\mathbf{r}) \cdot \mathbf{e}_1 \cdot \mathbf{e}_2}
\begin{bmatrix}
\mathbf{t} & \mathbf{e}_1 & \mathbf{e}_2 \\
-\mathbf{d}(\mathbf{r}) & \mathbf{t} & \mathbf{e}_2 \\
-\mathbf{d}(\mathbf{r}) & \mathbf{e}_1 & \mathbf{t}
\end{bmatrix}
\]

This can be rewritten as $\begin{vmatrix} \mathbf{A} & \mathbf{B} & \mathbf{C} \end{vmatrix} = -\begin{vmatrix} \mathbf{A} \times \mathbf{C} \end{vmatrix} \cdot \mathbf{B} = -\begin{vmatrix} \mathbf{C} \times \mathbf{B} \end{vmatrix} \cdot \mathbf{A}$. We can thus rewrite Equation 3.6.3 as:

\[
\begin{bmatrix}
t \\
b_1 \\
b_2
\end{bmatrix} = \frac{1}{\mathbf{d}(\mathbf{r}) \times \mathbf{e}_2 \cdot \mathbf{e}_1} \begin{bmatrix}
\mathbf{t} \times \mathbf{e}_1 & \mathbf{e}_2 \\
\mathbf{d}(\mathbf{r}) \times \mathbf{e}_2 & \mathbf{t} \\
\mathbf{t} \times \mathbf{e}_1 & \mathbf{d}(\mathbf{r})
\end{bmatrix}
\]

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

\[
\begin{bmatrix}
t \\
b_1 \\
b_2
\end{bmatrix} = \frac{1}{\mathbf{s}_1 \cdot \mathbf{e}_1} \begin{bmatrix}
\mathbf{s}_2 \cdot \mathbf{e}_2 \\
\mathbf{s}_1 \cdot \mathbf{t} \\
\mathbf{s}_2 \cdot \mathbf{d}(\mathbf{r})
\end{bmatrix}
\]

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.
In order to compute \( \mathbf{e}_1, \mathbf{e}_2, \) and \( t \) we need 9 subtractions. To compute \( s_1 \) and \( s_2 \), 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.

\(<\text{TriangleMesh Methods}>\)\
\begin{verbatim}
bool Triangle::Intersect(const Ray &ray, float *thitp, 
                          DifferentialGeometry *dg) const {
  \{ \newpage
  \text{Initialize triangle intersection statistics}  
  \text{Update triangle tests count}  
  \text{Compute } s_1 \text{)  
  \text{Compute first barycentric coordinate)  
  \text{Compute second barycentric coordinate}  
  \text{Compute } t \text{ to intersection point)  
  \text{Fill in DifferentialGeometry from triangle hit)  
  }*\text{thitp} = t;  
  return true;
\}
\end{verbatim}
\end{verbatim}

\(<\text{Initialize triangle intersection statistics}>\)\
\begin{verbatim}
static StatsRatio triangleHits("Geometry", "Triangle Ray Intersections");  
\end{verbatim}

\(<\text{Update triangle tests count}>\)\
\begin{verbatim}
triangleHits.add(0, 1);
\end{verbatim}

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.

\(<\text{Compute } s_1>\)\
\begin{verbatim}
\text{Get triangle vertices in } p_1, p_2, \text{ and } p_3)  
Vector E1 = p2 - p1;  
Vector E2 = p3 - p1;  
Vector S_1 = \text{Cross}(ray.D, E2);  
Float divisor = \text{Dot}(S_1, E1);  
if (divisor == 0.)  
  return false;  
Float invDivisor = 1.f / divisor;
\end{verbatim}

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.

\(<\text{Compute first barycentric coordinate}>\)\
\begin{verbatim}
\text{Vector } T = \text{ray.O - p}_1;  
Float b1 = \text{Dot}(T, S_1) * invDivisor;  
if (b1 < 0. || b1 > 1.)  
  return false;
\end{verbatim}
The second barycentric coordinate, $b_2$, is computed in a similar way:

\[\text{Compute second barycentric coordinate}\]

\[
\begin{align*}
\text{Vector } S_2 &= \text{Cross}(T, E_1); \\
\text{Float } b_2 &= \text{Dot}(\text{ray}\cdot D, S_2) * \text{invDivisor}; \\
\text{if } (b_2 < 0. \ || \ b_1 + b_2 > 1.) \\
&\quad \text{return false;}
\end{align*}
\]

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 $\text{mint}$ and $\text{maxt}$ bounds.

\[\text{Compute } t\text{ to intersection point}\]

\[
\begin{align*}
\text{Float } t &= \text{Dot}(E_2, S_2) * \text{invDivisor}; \\
\text{if } (t < \text{ray}\cdot\text{mint} \ || \ t > \text{ray}\cdot\text{maxt}) \\
&\quad \text{return false;}
\end{align*}
\]

\begin{align*}
\text{triangleHits.add}(1, 0);
\end{align*}

We now have all the information we need to compute the $\text{DifferentialGeometry}$ 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 derivatives are also both $(0,0,0)$.

\[\text{Fill in } \text{DifferentialGeometry}\text{ from triangle hit}\]

\[\text{Compute triangle partial derivatives}\]

\[\text{Interpolate } (u,v)\text{ triangle parametric coordinates}\]

\[
\begin{align*}
*\text{dg} &= \text{DifferentialGeometry}(\text{ray}(t), \text{dPdu, dPdv, Vector}(0,0,0), \\
&\quad \text{Vector}(0,0,0), \text{tu, tv, this});
\end{align*}
\]

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:

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

So

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

\(\text{Compute triangle partial derivatives}\)\(\equiv\)

Vector dPdu, dPdv;

Float uvs[3][2];

GetUVs(uvs);

\(\text{Compute deltas for triangle partial derivatives}\)

Float determinant = du1 * dv2 - dv1 * du2;

if (determinant == 0) {
    \(\text{Handle zero determinant for triangle partial derivative matrix}\)
}

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);
}

\(\text{Compute deltas for triangle partial derivatives}\)\(\equiv\)

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..

\(\text{Handle zero determinant for triangle partial derivative matrix}\)\(\equiv\)

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 $v_2$.

\[ \text{Interpolate \ (u,v) \ triangle \ parametric \ coordinates} \equiv \]
\[
\text{Float } b_0 = 1 - b_1 - b_2; \\
\text{Float } t_u = b_0*\text{uvs}[0][0] + b_1*\text{uvs}[1][0] + b_2*\text{uvs}[2][0]; \\
\text{Float } t_v = b_0*\text{uvs}[0][1] + b_1*\text{uvs}[1][1] + b_2*\text{uvs}[2][1];
\]

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 $\frac{1}{2}$ of the area of the parallelogram given by those two vectors—see Figure 3.6.

\[ \text{TriangleMesh Methods} \equiv \]
\[
\text{Float Triangle::Area() const} \\
\quad \text{Get triangle vertices in } p_1, p_2, \text{ and } p_3 \\
\quad \text{return } 0.5f \times \text{Cross}(p_2-p_1, p_3-p_1).\text{Length}();
\]

3.7 Subdivision Surfaces

\[ \text{subdiv.cc*} \equiv \]
\[
\text{Source Code Copyright} \\
\#include "lrt.h" \\
\#include "shapes.h" \\
\#include "paramset.h" \\
\#include "dynload.h" \\
\#include "texture.h" \\
\#include <set> \\
\#include <map> \\
\text{using std::set;} \\
\text{using std::map;} \\
\text{SubdivisionMesh Macros} \\
\text{SubdivisionMesh Local Structures} \\
\text{SubdivisionMesh Declarations} \\
\text{SubdivisionMesh Inline Functions} \\
\text{SubdivisionMesh Methods}
\]
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 recently 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
Subdivision Surfaces

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

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.
\textit{(SubdivisionMesh Methods)}
\begin{verbatim}
LoopSubdiv::LoopSubdiv(const Transform &o2w, int nfaces,
    int nvertices, const int *vertexIndices,
    const Point *P, int nl)
    : Shape(o2w) {
    nLevels = nl;
    \textit{(Allocate LoopSubdiv vertices and faces)}
    \textit{(Set face to vertex pointers)}
    \textit{(Set neighbor pointers in faces)}
    \textit{(Finish vertex initialization)}
}
\end{verbatim}

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

\begin{verbatim}
\textit{(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]);
\end{verbatim}

The \texttt{LoopSubdiv} destructor, which we won’t include here, just deletes all of the faces and vertices allocated above.

\begin{verbatim}
\textit{(LoopSubdiv Private Data)}
int nLevels;
vector<SDVertex *> vertices;
vector<SDFace *> faces;
\end{verbatim}

The Loop subdivision scheme, like most other subdivision schemes, assumes that the control mesh is \textit{manifold}; no more than two faces share any given edge. Such a mesh may be closed or open: a \textit{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 \texttt{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 \textit{valence}. Interior vertices with valence other than six, or boundary vertices with valence other than four are called \textit{extraordinary vertices}; otherwise they are called \textit{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) +≡

```c
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 where 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.)

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) +≡

```c
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) ≡

```c
#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 ith edge is the edge from v[i] to v[(i+1) % 3] such that the neighbor across the ith 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 v0 to v1 is traversed from v0 to v1 by face number one, and from v1 to v0 by face number two. Another way to think of this is in terms of face orientation: all faces’ vertices should be given consistently 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 them. 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.

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

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.

\[ \text{SubdivisionMesh Local Structures} = \]
\[
\text{struct SDEdge} {
\text{SDEdge Constructor}
\text{SDEdge Comparison Function}
\text{SDVertex } *v[2];
\text{SDFace } *f[2];
\text{SDFace **fptr;}
};
\]

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.
\textit{(SDEdge Constructor)}
\begin{verbatim}
SDEdge(SDVertex *v0 = NULL, SDVertex *v1 = NULL) {
    v[0] = min(v0, v1);
    v[1] = max(v0, v1);
    f[0] = f[1] = NULL;
    fptr = NULL;
}
\end{verbatim}

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

\textit{(SDEdge Comparison Function)}
\begin{verbatim}
bool operator<(const SDEdge &e2) const {
    if (v[0] == e2.v[0]) return v[1] < e2.v[1];
    return v[0] < e2.v[0];
}
\end{verbatim}

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 \texttt{set} to store the edges where we’re still looking for the face on the other side of it; the \texttt{set} uses the comparison function above to provide $O(\log n)$ searches in the SDEges.

\textit{(Set neighbor pointers in faces)}
\begin{verbatim}
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
    }
}
\end{verbatim}

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.

\textit{(Update neighbor pointer for edge)}
\begin{verbatim}
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
}
\end{verbatim}

Given an edge that we haven’t seen before, we store the current face’s pointer in the edge object’s \texttt{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.
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 \( v[\text{NEXT}(i)] \). The previous face is defined analogously.

\[\text{Handle new edge} \equiv\]
\[
\begin{align*}
e.f[0] &= f; \\
e.fptr &= &(f->f[\text{edge}]); \\
edges.insert(e); 
\end{align*}
\]

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.

\[\text{Handle previously-seen edge} \equiv\]
\[
\begin{align*}
e &= *\text{edges.find}(e); \\
*e.fptr &= f; \\
f->f[\text{edge}] &= e.f[0]; \\
edges.erase(e); 
\end{align*}
\]

Now that all faces have proper neighbor pointers, we can set the boundary and regular flags in each of the vertices. In order to determine 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 \( v[\text{NEXT}(i)] \) and the previous face as the face across the edge from \( v[\text{PREV}(i)] \) to \( v[i] \).

By successively going to the next face around \( v \), \( f=f->\text{nextFace}(v) \), 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 boundary 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.

```c
(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.

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

```c
(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.

\[ \text{SDFace Methods} \]

\[
\text{int vnum(SDVertex *vert) const \{ \\
\text{for (int i = 0; i < 3; ++i) \\
\text{if (v[i] == vert) return i; \\
Assert(1 == 0); \\
return -1; \\
\}}}
\]

\[ \text{SubdivisionMesh Inline Functions} \]

\[
\text{inline int SDVertex::valence() \{ \\
SDFace *f = startFace; \\
if (!boundary) \{
\text{\{Compute valence of interior vertex\}} \\
\}
else \{
\text{\{Compute valence of boundary vertex\}} \\
\}
\}}
\]

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.

\[ \text{Compute valence of interior vertex} \]

\[
\text{int nf = 1; \\
while ((f = f->nextFace(this)) != startFace) \\
++nf; \\
return nf;}
\]

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.

\[ \text{Compute valence of boundary vertex} \]

\[
\text{int nf = 1; \\
while ((f = f->nextFace(this)) != NULL) \\
++nf; \\
f = startFace; \\
while ((f = f->prevFace(this)) != NULL) \\
++nf; \\
return nf+1;}
\]
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.

\[\text{LoopSubdiv Method Declarations}\] +≡
\begin{align*}
& \text{BBox ObjectBound()} \text{ const;} \\
& \text{BBox WorldBound()} \text{ const;}
\end{align*}

Subdivision

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.

\[\text{SubdivisionMesh Methods}\] +≡
\begin{align*}
& \text{bool LoopSubdiv::CanIntersect()} \text{ const } \\
& \quad \text{return false;}
\end{align*}

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.

\[\text{SubdivisionMesh Methods}\] +≡
\begin{align*}
& \text{void LoopSubdiv::Refine(vector<RefinedShape> \&refined) const } \\
& \quad \text{vector<SDFace \*> } f = \text{faces;} \\
& \quad \text{vector<SDVertex \*> } v = \text{vertices;} \\
& \quad \text{MemoryArena<SDVertex> vertexArena;} \\
& \quad \text{MemoryArena<SDFace> faceArena;} \\
& \quad \text{u_int } i; \\
& \quad \text{for } (i = 0; i < \text{nLevels}; ++i) \\
& \quad \quad \{ \text{// Update } f \text{ and } v \text{ for next level of subdivision} \\
& \quad \quad \} \text{// Push vertices to limit surface} \\
& \quad \quad \{ \text{// Compute vertex tangents on limit surface} \\
& \quad \quad \} \text{// Create TriangleMesh from subdivision mesh} \\
& \quad \}
\end{align*}

Here are the contents the main loop of a subdivision step. We create vectors 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.

\[\text{Update } f \text{ and } v \text{ for next level of subdivision}\]
\begin{verbatim}
vector<SDFace *> newFaces;
vector<SDVertex *> newVertices;
Allocate next level of children in mesh tree
\end{verbatim}
\[\text{Update vertex positions and create new edge vertices}\]
\[\text{Update new mesh topology}\]
\[\text{Prepare for next level of subdivision}\]

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.

\[\text{Allocate next level of children in mesh tree}\]
\begin{verbatim}
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]);
}
\end{verbatim}
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〉
    }
}

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 $\beta$. (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 $\beta$ 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 $\beta$ 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 \( \beta \) 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 \( \beta \) 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.

\[\{\text{LoopSubdiv Private Methods}\}\]
\[
\text{static Float beta(int val) \{} \\
\text{\quad if (val == 3) return 3.f/16.f;}
\text{\quad else return 3.f / (8.f * val);}
\text{\}}
\]

The \( \text{weightOneRing} \) 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 \( \text{oneRing} \) function, defined below, which returns the positions of the vertices around the vertex \( \text{vert} \).

\[\{\text{SubdivisionMesh Methods}\}\]
\[
\text{Point LoopSubdiv::weightOneRing(SDVertex *vert, Float beta) \{} \\
\text{\quad \text{\{Put vert one-ring in Pring\}}} \\
\text{\quad Point P = (1 - val * beta) * vert->P;}
\text{\quad for (int i = 0; i < val; ++i) P += beta * Pring[i];}
\text{\quad return P;}
\text{\}}
\]
\[
\text{\{Put vert one-ring in Pring\}} \\
\text{\quad int val = vert->valence();}
\text{\quad Point *Pring = (Point *)alloca(val * sizeof(Point));}
\text{\quad vert->oneRing(Pring);}
\]
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);

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-1st 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))];
}

SDVertex 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 face $f$, 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], 1.0f/8.0f);
```

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.
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.

(\textit{Subdivision Mesh Methods})

```cpp
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;
}
```

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 \textit{splitEdges} associative array.

(\textit{Compute new odd edge vertices})

```cpp
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 \textit{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 \textit{map}. The \textit{map} is an associative array structure that performs efficient lookups.
Compute odd vertex on kth edge

\[
SDEdge \text{ edge(face->v[k], face->v[NEXT(k)]);}
\]
\[
SDVertex *vert = \text{splitEdges[edge]};
\]
\[
if (!vert) { \\
\hspace{1em} \text{Create and initialize new odd vertex} \\
\hspace{1em} \text{Apply edge rules to compute new vertex position} \\
\hspace{1em} \text{splitEdges[edge] = vert;}
\}
\]

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

\[
\text{vert = new (vertexArena) SDVertex;} \\
\text{newVertices.push_back(vert);} \\
\text{vert->regular = true;} \\
\text{vert->boundary = (face->f[k] == NULL);} \\
\text{vert->startFace = face->children[3];}
\]

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

\[
\text{if (vert->boundary) {} } \text{\hspace{1em} \text{otherVert}} \\
\hspace{2em} \text{vert->P} = \text{0.5f * edge.v[0]->P;} \\
\hspace{2em} \text{vert->P} += \text{0.5f * edge.v[1]->P;}
\]
\[
\text{else { } \text{otherVert}} \\
\hspace{2em} \text{vert->P} = \text{3.f/8.f * edge.v[0]->P;} \\
\hspace{2em} \text{vert->P} += \text{3.f/8.f * edge.v[1]->P;} \\
\hspace{2em} \text{vert->P} += \text{1.f/8.f *} \\
\hspace{3em} \text{face->otherVert(edge.v[0], edge.v[1])->P;} \\
\hspace{2em} \text{vert->P} += \text{1.f/8.f *} \\
\hspace{3em} \text{face->f[k]->otherVert(edge.v[0], edge.v[1])->P;}
\]

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...

\[\begin{align*}
&\text{SDVertex } *\text{otherVert}(\text{SDVertex } *v0, \text{SDVertex } *v1) \{ \\
&\quad \text{for (int } i = 0; i < 3; ++i) \\
&\quad \quad \text{if (} v[i] \neq v0 \&\& v[i] \neq v1) \\
&\quad \quad \quad \text{return } v[i]; \\
&\quad \quad \text{Assert(} 1 == 0); \\
&\quad \text{return NULL;}
&\}
\end{align*}\]

**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 into 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 \texttt{startFace}.
2. Similarly, the odd vertices \texttt{startFace} pointers need to be set.
3. The new faces’ \texttt{f[i]} pointers need to be initialized.
4. The new faces’ \texttt{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.

\[\text{Update new mesh topology}\]
\[\text{Update even vertex face pointers}\]
\[\text{Update face neighbor pointers}\]
\[\text{Update face vertex pointers}\]

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`.

\[\text{Update even vertex face pointers}\]
\[
\text{for (j = 0; j < v.size(); ++j) } \{
\text{SDVertex } *\text{vert = v[j];}
\text{int vertNum = vert->startFace->vnum(vert);}
\text{vert->child->startFace = vert->startFace->children[vertNum];}
\}\]

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.

\[\text{Update face neighbor pointers}\]
\[
\text{for (j = 0; j < f.size(); ++j) } \{
\text{SDFace } *\text{face = f[j];}
\text{for (int k = 0; k < 3; ++k) } \{
\text{\{Update children \& pointers for siblings\}}
\text{\{Update children \& pointers for neighbor children\}}
\}\}
\]

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 \(k\)th edge of the interior face, and the interior face is across the \(k+1\)st edge of the \(k\)th face.

\[\text{Update children \& pointers for siblings}\]
\[
\text{face->children[3]->f[k] = face->children[NEXT(k)];}
\text{face->children[k]->f[NEXT(k)] = face->children[3];}
\]

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 \(\text{PREV}(i)\)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.
Finally, we handle the fourth step in the topological updates: setting the face $v[i]$ vertex pointers.

\section*{Update child vertex pointer to new even vertex}

\begin{verbatim}
\texttt{SDFace} *\texttt{face} = \texttt{f}[j];
\texttt{for} (\texttt{int} \texttt{k} = 0; \texttt{k} < 3; ++\texttt{k}) {
  \texttt{SDFace} *\texttt{f2} = \texttt{face}->\texttt{f}[k];
  \texttt{face}->\texttt{children}[\texttt{k}]->\texttt{f}[k] = 
    \texttt{f2} ? \texttt{f2}->\texttt{children}[\texttt{f2}->\texttt{vnum}(\texttt{face}->\texttt{v}[\texttt{k}])] : \texttt{NULL};
  \texttt{f2} = \texttt{face}->\texttt{f}[\texttt{PREV}(\texttt{k})];
  \texttt{face}->\texttt{children}[\texttt{k}]->\texttt{f}[\texttt{PREV}(\texttt{k})] = 
    \texttt{f2} ? \texttt{f2}->\texttt{children}[\texttt{f2}->\texttt{vnum}(\texttt{face}->\texttt{v}[\texttt{k}])] : \texttt{NULL};
\}
\end{verbatim}

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.

\section*{Update child vertex pointer to new odd vertex}

\begin{verbatim}
\texttt{for} (\texttt{j} = 0; \texttt{j} < \texttt{f}.\texttt{size}(); ++\texttt{j}) {
  \texttt{SDFace} *\texttt{face} = \texttt{f}[\texttt{j}];
  \texttt{for} (\texttt{int} \texttt{k} = 0; \texttt{k} < 3; ++\texttt{k}) {
    \texttt{SDFace} *\texttt{f2} = \texttt{face}->\texttt{f}[\texttt{k}];
    \texttt{face}->\texttt{children}[\texttt{k}]->\texttt{f}[\texttt{k}] = 
      \texttt{f2} ? \texttt{f2}->\texttt{children}[\texttt{f2}->\texttt{vnum}(\texttt{face}->\texttt{v}[\texttt{k}])] : \texttt{NULL};
    \texttt{f2} = \texttt{face}->\texttt{f}[\texttt{PREV}(\texttt{k})];
    \texttt{face}->\texttt{children}[\texttt{k}]->\texttt{f}[\texttt{PREV}(\texttt{k})] = 
      \texttt{f2} ? \texttt{f2}->\texttt{children}[\texttt{f2}->\texttt{vnum}(\texttt{face}->\texttt{v}[\texttt{k}])] : \texttt{NULL};
  \}
}
\end{verbatim}

To update the face vertex pointers to the new odd vertices, we re-use the \texttt{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.

\section*{Update child vertex pointer to new odd vertex}

\begin{verbatim}
\texttt{for} (\texttt{j} = 0; \texttt{j} < \texttt{f}.\texttt{size}(); ++\texttt{j}) {
  \texttt{SDFace} *\texttt{face} = \texttt{f}[\texttt{j}];
  \texttt{for} (\texttt{int} \texttt{k} = 0; \texttt{k} < 3; ++\texttt{k}) {
    \texttt{SDFace} *\texttt{f2} = \texttt{face}->\texttt{f}[\texttt{k}];
    \texttt{face}->\texttt{children}[\texttt{k}]->\texttt{f}[\texttt{k}] = 
      \texttt{f2} ? \texttt{f2}->\texttt{children}[\texttt{f2}->\texttt{vnum}(\texttt{face}->\texttt{v}[\texttt{k}])] : \texttt{NULL};
    \texttt{f2} = \texttt{face}->\texttt{f}[\texttt{PREV}(\texttt{k})];
    \texttt{face}->\texttt{children}[\texttt{k}]->\texttt{f}[\texttt{PREV}(\texttt{k})] = 
      \texttt{f2} ? \texttt{f2}->\texttt{children}[\texttt{f2}->\texttt{vnum}(\texttt{face}->\texttt{v}[\texttt{k}])] : \texttt{NULL};
  \}
}
\end{verbatim}

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.
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, $P_{\text{limit}}$. 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 computes appropriate vertex weights based on the valence of the vertex.
\section*{Push vertices to limit surface}

\begin{verbatim}
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];
\end{verbatim}

\section*{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 computation that gives the precise tangents on the actual limit surface.

\section*{Compute vertex tangents on limit surface}

\begin{verbatim}
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);
    if (!vert->boundary) {
        \{ \textit{Put vert one-ring in Pring} \}
        \{ Compute tangents of interior face \}
    } else {
        \{ Compute tangents of boundary face \}
    }
    Ns.push_back(Normal(Cross(S, T)));
}
\end{verbatim}

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.

\[ \text{Compute tangents of interior face} \equiv \]
\[
\text{for (int } k = 0; k < \text{val; ++k) } \{
\quad S += \cosf(2.0f*M\_PI*k/\text{val}) * \text{Vector(Pring[k])};
\quad T += \sinf(2.0f*M\_PI*k/\text{val}) * \text{Vector(Pring[k])};
\}
\]

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:

<table>
<thead>
<tr>
<th>valence</th>
<th>( w_c )</th>
<th>( w_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-2</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>(0, 1, 0)</td>
</tr>
<tr>
<td>4 (regular)</td>
<td>-2</td>
<td>(-1, 2, 2, -1)</td>
</tr>
</tbody>
</table>

For valences of 5 and higher, \( w_c = 0 \) and

\[
\begin{align*}
    w_0 &= w_{n-1} = \sin \theta \\
    w_i &= (2 \cos \theta - 2) \sin(\theta i)
\end{align*}
\]

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.

\[ \text{Compute tangents of boundary face} \]

\[
S = \text{Pring}[\text{val}-1] - \text{Pring}[0];
\]

\[
\begin{align*}
\text{if } (\text{val} == 2) & \quad T = \text{Vector}((\text{Pring}[0] + \text{Pring}[1] - 2 * \text{vert}\rightarrow P)); \\
\text{else if } (\text{val} == 3) & \quad T = \text{Pring}[1] - \text{vert}\rightarrow P; \\
\text{else if } (\text{val} == 4) & \quad T = \text{Vector}(-1*\text{Pring}[0] + 2*\text{Pring}[1] + 2*\text{Pring}[2] + \text{-1*Pring}[3] + -2*\text{vert}\rightarrow P); \\
\text{else} & \quad \{
\begin{align*}
\text{Float theta} & = \text{M}_\text{PI} / \text{float}((\text{val}-1)); \\
T & = \text{Vector}(\text{sinf}((\text{theta}) * (\text{Pring}[0] + \text{Pring}[\text{val}-1]))); \\
\text{for } (\text{int } k = 1; k < \text{val}-1; ++k) \{} \\
\text{Float wt} & = (2 * \text{cosf}((\text{theta}) - 2) * \text{sinf}((k) * \text{theta}); \\
T & += \text{Vector}(\text{wt} * \text{Pring}[k]); \\
\} \\
T & = -T;
\}
\end{align*}
\]

Finally, the fragment \(\text{Create TriangleMesh from subdivision mesh}\) 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 (HDD94). 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 (ZSD00).

### 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

\[ Ax^2 + By^2 + Cz^2 + 2Dxy + 2Eyz + 2Fxz + 2Hy + 2Iz + J = 0, \]

(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 \( Q \):

\[
\begin{bmatrix}
A & D & F & G \\
D & B & E & H \\
F & E & C & I \\
G & H & I & J
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix} = \begin{bmatrix}
p^T \cdot Q \cdot p = 0
\end{bmatrix}
\]

Given this representation, first show that the matrix \( Q' \) representing a quadric transformed by the matrix \( M \) is:

\[ Q' = (M^T)^{-1}QM^{-1}. \]

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
world-space hit points into object space to test against \( \theta_{\text{max}} \), if it is not \( 2\pi \), etc. How does performance compare to the original scheme?

3.2 Implement a general polygon primitive. \( \text{lrt} \) 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 a 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 \( \text{Shape} \). 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 subdivi-
sion 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 ver-
tex 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

```c
#include "lrt.h"
#include "primitives.h"
#include "light.h"

/*Primitive Methods*/
/*GeometricPrimitive Methods*/
/*PrimitiveList Methods*/
/*Surf Method Definitions*/
```

```c
#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
```
One of the keys to making ray-tracing efficient is having algorithms that reduce the cost of finding intersections of rays with shapes in the scene. Since a ray through 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
Sec. 4.1] Approaches To Reducing Intersections

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.

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 \( p \) and the plane normal \( \hat{n} \). Given a ray \( r \), we’d like to find the parametric point \( t \) along \( r \) that gives the point along \( r \) that lies on the plane. We write an equation that describes the set of points \( p' \) that lie on the plane: this is just the set of all points such that the vector from \( p \) to \( p' \) is perpendicular to \( \hat{n} \). Because perpendicular vectors have a dot product of zero, we have:

\[
((p' - p) \cdot \hat{n}) = 0
\]

Thus, given a ray \( r \) defined by \( r = o(r) + t \vec{d}(r) \), we substitute the ray equation for \( p' \) to find the point where the ray intersects the plane:

\[
((o(r) + t \vec{d}(r) - p) \cdot \hat{n}) = 0.
\]

Using basic definitions of the dot product, we have

\[
((o(r) - p) \cdot \hat{n}) + (t \vec{d}(r) \cdot \hat{n}) = 0
\]

\[
(t \vec{d}(r) \cdot \hat{n}) = -((o(r) - p) \cdot \hat{n})
\]

\[
t = -\frac{((o(r) - p) \cdot \hat{n})}{(\vec{d}(r) \cdot \hat{n})}
\]

As long as \( (\vec{d}(r) \cdot \hat{n}) \) 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
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.

The routine to compute the intersection is called \texttt{IntersectP}. \texttt{IntersectP} is a \textit{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 \texttt{hitt0} and \texttt{hitt1}. Intersections outside of the \texttt{mint/maxt} range of the ray that is passed in are not considered.

\begin{verbatim}
bool BBox::IntersectP(const Ray &ray, Float *hitt0, 
                      Float *hitt1) const {
    \{ 
      \begin{enumerate}
        \item Initialize parametric interval
        \item Check X slab
        \item Check Y slab
        \item Check Z slab
        \end{enumerate}
        \begin{description}
          \item[if \(\text{hitt0}\)] *\texttt{hitt0} = \texttt{t0};
          \item[if \(\text{hitt1}\)] *\texttt{hitt1} = \texttt{t1};
          \item return true;
        \end{description}

    \}

\begin{verbatim}
\end{verbatim}

\begin{verbatim}
Float t0 = ray.mint, t1 = ray.maxt;
\end{verbatim}

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:

\[
t_1 = -\frac{(\mathbf{o}(\mathbf{r}) - (x_1, 0, 0)) \cdot (1, 0, 0)}{(\mathbf{d}(\mathbf{r}) \cdot (1, 0, 0))}
\]
Because the $y$ and $z$ components of the normal are zero, we can use the definition of the dot product to simplify this substantially:

\[
t_1 = \frac{o(r_x) - x_1}{d(r_x)} = \frac{x_1 - o(r_x)}{d(r_x)}
\]

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 $\text{invRayDir}$ will hold an infinite value, either $-\infty$ or $\infty$, and the rest of the algorithm works correctly.

\[\text{Check X slab}\]

```cpp
float invRayDir = 1.f / ray.D.x;
float tNear = (pMin.x - ray.O.x) * invRayDir;
float tFar = (pMax.x - ray.O.x) * invRayDir;
```

\[\text{Update parametric interval}\]

We then swap the two distances, so that $t_{\text{near}}$ holds the closer intersection and $t_{\text{far}}$ 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.

\[\text{Update parametric interval}\]

```cpp
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 $\infty$ is a special value such that any positive number multiplied by $\infty$ gives $\infty$, any negative number multiplied by $\infty$ gives $-\infty$, etc.
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 **Shapes** and **Primitives** 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 first 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 over-
lap 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 GeometricPrimitives, the type of Primitive that we have in
lrt is a PrimitiveList. This is simply a collection of Primitives 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.

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 Primitives. It also main-
tains 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 Primitives 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;
}
### 4.3 Regular Grid Accelerator

Here we will describe the implementation of \texttt{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.
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.
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

\[
XWidth = \frac{\text{diag}.x}{\text{XVoxels}}; \\
YWidth = \frac{\text{diag}.y}{\text{YVoxels}}; \\
ZWidth = \frac{\text{diag}.z}{\text{ZVoxels}}; \\
\text{InvXWidth} = \frac{1}{XWidth}; \\
\text{InvYWidth} = \frac{1}{YWidth}; \\
\text{InvZWidth} = \frac{1}{ZWidth}; \\
\]

int nVoxels = XVoxels * YVoxels * ZVoxels;
voxels = (Voxel **)AllocL1CacheAligned(nVoxels * sizeof(Voxel *));
memset(voxels, 0, nVoxels * sizeof(Voxel *));

Small structure to hold information 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.
To add primitives to the grid, we loop through the primitives in turn, adding each one to the vectors of pointers to MailboxPrims in the cells that its bounding box overlaps.

```
struct MailboxPrim {
    MailboxPrim() {
        primitive = NULL;
        lastMailboxId = -1;
    }
    Reference<Primitive> primitive;
    int lastMailboxId;
};
```

```
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 MailboxPrims allocated above so that the grid's destructor can free this memory.
```

```
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.
<GridAccelerator Method Declarations>

```cpp
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, and add the pointer to the end of it.

<Add primitive to overlapping cells>

```cpp
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>

<GridAccelerator Private Data>

```cpp
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.

```cpp
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.

```cpp
static StatsRatio nPrimitiveVoxels("Acceleration",
    "Voxels covered vs # / primitives");
nPrimitiveVoxels.add((1 + x1-x0) * (1 + y1-y0) * (1 + z1-z0), 1);
```
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.

\[
\text{Update fraction of empty voxels}\equiv
\]

\[
\text{static StatsRatio nEmptyVoxels("Acceleration", "Empty voxels");}
\]
\[
\text{nEmptyVoxels.add(0, XVoxels * YVoxels * ZVoxels);}
\]
\[
\text{for (int i = 0; i < XVoxels * YVoxels * ZVoxels; ++i)}
\]
\[
\text{if (!voxels[i]) nEmptyVoxels.add(1, 0);}
\]

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.

\[
\text{Initialize mailbox} \equiv
\]
\[
\text{curMailboxId = 0;}
\]

\[
\text{GridAccelerator Private Data} +\equiv
\]
\[
\text{mutable int curMailboxId;}
\]

The destructor just has to free up the array of voxels that we made and the array of MailboxPrims 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.

\[
\text{GridAccelerator Method Definitions} +\equiv
\]
\[
\text{GridAccelerator::\text{:GridAccelerator() }\{}
\]
\[
\text{for (u_int i = 0; i < nMailboxes; ++i)}
\]
\[
\text{mailboxes[i].\text{:MailboxPrim();}
\]
\[
\text{FreeCacheAligned(voxels);}
\]
\[
\text{FreeCacheAligned(mailboxes);}
\]
\[
\}
\]

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.)

\[
\text{GridAccelerator Method Definitions} +\equiv
\]
\[
\text{bool GridAccelerator:::Intersect(const Ray &ray,}
\]
\[
\text{Surf *surf) const }\{
\]
\[
\text{\{Check ray against overall grid bounds\}
\]
\[
\text{\{Set up 3D DDA for this ray\}
\]
\[
\text{\{Walk grid\}
\]
\[
\}
\]

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.

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.

(Setup 3D DDA for this ray)≡
(Update statistics for ray inside grid)
(Setup X stepping)
(Setup Y stepping)
(Setup 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 $\text{rayT}$, 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, $\text{NextXCrossing}$, 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 $\text{NextXCrossing}$ by adding a fixed value, the voxel width in $x$ divided by the ray’s $x$ direction, to it.

\begin{verbatim}
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\}
}

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 \text{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 $\Delta X$ and $\text{OutX}$ won’t be used, but we’ll set them to silence over-aggressive compiler warnings about uninitialized variables.
\end{verbatim}
Intersection Acceleration

\[ \text{Handle ray perpendicular to } x \Rightarrow \]
\begin{align*}
\text{NextXCrossing} & = \text{INFINITY}; \\
\text{DeltaX} & = 0; \\
\text{OutX} & = -1;
\end{align*}

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 \( x \), \( \text{NextXCrossing} \) is our parametric starting point, \( \text{rayT} \) 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.

\[ \text{StepX} \] just tells us that when we leave a voxel in the \( x \) direction, we move 1 voxels. \( \text{OutX} \) says that when we reach a voxel with component \( \text{XVoxels} \), we’ve left the grid and are done.

\[ \text{Handle ray with positive } x \text{ direction} \Rightarrow \]
\begin{align*}
\text{NextXCrossing} & = \text{rayT} + (v2x(x+1) - \text{gridIntersect.x})/\text{ray.D.x}; \\
\text{DeltaX} & = \text{XWidth} / \text{ray.D.x}; \\
\text{StepX} & = 1; \\
\text{OutX} & = \text{XVoxels};
\end{align*}

Similar computations compute these values for rays with negative \( x \) components.

\[ \text{Handle ray with negative } x \text{ direction} \Rightarrow \]
\begin{align*}
\text{NextXCrossing} & = \text{rayT} + (v2x(x) - \text{gridIntersect.x})/\text{ray.D.x}; \\
\text{DeltaX} & = -\text{XWidth} / \text{ray.D.x}; \\
\text{StepX} & = -1; \\
\text{OutX} & = -1;
\end{align*}

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, \( \text{hitSomething} \) 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 \( \text{maxt} \) 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.
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.

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.
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.

\textit{Refine primitive if needed:}

\begin{verbatim}
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);
}
\end{verbatim}

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.

\textit{Do mailbox check between ray and primitive:}

\begin{verbatim}
if (mp->lastMailboxId == rayId)
    continue;
\end{verbatim}

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.

\textit{Check for ray-primitive intersection:}

\begin{verbatim}
mp->lastMailboxId = rayId;
rayTests.add(1, 0);
if (mp->primitive->Intersect(ray, surf)) {
    rayHits.add(1, 0);
    hitSomething = true;
}
\end{verbatim}

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: 

\[
\begin{align*}
    \text{if} \ (\text{NextXCrossing} < \text{NextYCrossing} \land \text{NextXCrossing} < \text{NextZCrossing}) \{
        \langle \text{Step in X} \rangle
    \} \\
    \text{else if} \ (\text{NextZCrossing} < \text{NextYCrossing}) \{
        \langle \text{Step in Z} \rangle
    \} \\
    \text{else} \{
        \langle \text{Step in Y} \rangle
    \}
\end{align*}
\]

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 \( \text{maxt} \) is less than the parametric distance at which we enter the next x voxel, \( \text{NextXCrossing} \). Otherwise we update the variable that holds the current voxel address by adding \( \text{StepX} \) (which is either -1 or 1) to it. If we have left the grid \( (x == \text{OutX}) \), then we also break. Otherwise we increment the value of \( \text{NextXCrossing} \) to the \( \text{DeltaX} \) value, so that we know how far we need to go parametrically before stepping in \( x \) again.

\[
\begin{align*}
    \langle \text{Step in X} \rangle \equiv \\
    \text{if} \ (\text{ray.maxt} < \text{NextXCrossing}) \\
        \text{break}; \\
    x += \text{StepX}; \\
    \text{if} \ (x == \text{OutX}) \\
        \text{break}; \\
    \text{NextXCrossing} += \text{DeltaX};
\end{align*}
\]

The cases for stepping in \( y \) and \( z \) are equivalent and are omitted.

We also provide a specialized version of \text{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 \text{GridAccelerator::Intersect()}\) routine, except that it calls the \text{Primitive::IntersectP()}\) method of the primitives, rather than \text{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.

\[
\begin{align*}
    \langle \text{GridAccelerator Method Declarations} \rangle \equiv \\
    \text{bool IntersectP(const Ray &ray) const;}
\end{align*}
\]

### 4.4 Kd Tree

\[
\langle \text{kdtree.cc*} \rangle \equiv \\
\langle \text{Source Code Copyright} \rangle \\
\text{#include "lrt.h"} \\
\text{#include "primitives.h"} \\
\text{#include "geometry.h"} \\
\langle \text{KdTreeAccelerator Declarations} \rangle \\
\langle \text{KdTreeAccelerator Method Definitions} \rangle
\]
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.

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.

\begin{verbatim}
class KdTreeAccelerator : public Primitive {
public:
  \{KdTreeAccelerator Method Declarations\}
private:
  \{KdTreeAccelerator Private Data\}
  \{KdTreeAccelerator Private Methods\}
};
\end{verbatim}

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.

\(\text{KdTreeAccelerator Method Definitions}\) 

\[
\text{KdTreeAccelerator::KdTreeAccelerator(const vector<Reference<Primitive> > &prims) \{}
\]

- \text{Refine all prims until they are intersectable}
- \text{Initialize mailboxes for KdTreeAccelerator}
- \text{Set up memory pools for kd tree nodes}
- \text{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, \text{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.

\(\text{Refine all prims until they are intersectable}\) 

\[
\text{vector<Reference<Primitive> > prefined, todo = prims; while (todo.size() > prefined, todo = prims;}
\]

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.

\(\text{Initialize mailboxes for KdTreeAccelerator}\) 

\[
\text{curMailboxId = 0;}
\]

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.

\(\text{KdTreeAccelerator Private Data}\) 

\[
\text{KdAccelNode *root; MailboxPrim *mailboxPrims; mutable int curMailboxId; BBox bounds;}
\]

\textbf{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)≡

```cpp
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);
```

(KdTreeAccelerator Method Definitions) +≡

```cpp
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;
    }
    buildTree(&root, bounds, mailboxPrims, primBounds, primNums, 0);
}
```

We stop building the tree if we’ve either got a sufficiently small number of primit-
ives 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.

\begin{verbatim}
<Initialize leaf node if termination criteria met>
if (primNums.size() == 1 ||
    (primNums.size() < 5 && depth > 10) ||
    depth > 20) {
    *node = allocNode(depth, primNums, nodeBounds);
    return;
}

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
\end{verbatim}
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 splitting 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.

\begin{verbatim}
(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);
\end{verbatim}

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 classification 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.

```cpp
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);
}
```

Each node of the kd tree—leaf or interior—is represented by a KdAccelNode structure. Assuming floats 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 \texttt{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;
}
```

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;
}
```

\textbf{Cache-friendly memory allocation}

Because acceleration data structure traversal is at the heart of \texttt{lrt}'s inner loop, it's worth going through some effort to ensure cache-friendly layout of \texttt{KdAccelNode}s in memory. Applying the techniques in this section sped up \texttt{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, \(\text{\{Get pointer to new \ KdAccelNode\}}\) to get a pointer to uninitialized memory in `node`; they then use C++'s \textit{placement new operator} to construct a `KdAccelNode` at the given memory location.

\[\text{\{KdTreeAccelerator Method Definitions\}\{\}}\]

\[
\text{KdAccelNode *KdTreeAccelerator::allocNode(int depth, int axis, Float split) \{}
\text{\{Update kd interior node allocation statistics\}}
\text{\{Get pointer to new \ KdAccelNode\}}
\text{new (node) KdAccelNode(axis, split);}
\text{return node;}
\]

\[\text{\{KdTreeAccelerator Method Definitions\}\{\}}\]

\[
\text{KdAccelNode *KdTreeAccelerator::allocNode(int depth, const vector<int> &primNums, const BBox &nodeBound) \{}
\text{\{Update kd leaf node allocation statistics\}}
\text{\{Get pointer to new \ KdAccelNode\}}
\text{new (node) KdAccelNode(mailboxPrims, primNums, nodeBound);}
\text{return node;}
\]

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 \texttt{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.

\[\text{\{Get pointer to new \ KdAccelNode\}\{\}}\]

\[
\text{KdAccelNode *node;}
\text{if (depth < MAX\_TOP\_DEPTH) \{}
\text{\{Allocate kd tree node for top part of tree\}}
\text{\} else \{}
\text{\{Allocate kd tree node for bottom part of tree\}}
\text{\}}
\]
Nodes up to depth \(\text{MAX\_TOP\_DEPTH}\) are allocated from the contiguous chunk of memory for nodes at the top of the tree. \(\text{numTop}\) 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 \(\text{topNodeOffset[]}\) array is used to record how many nodes have been allocated so far at each depth.

\[
\text{Allocate kd tree node for top part of tree} \\
\begin{align*}
\text{int } & \text{offset} = (1 \ll \text{depth}) - 1 + \text{topNodeOffset[depth]}; \\
& \text{Assert}(\text{offset} < (1 \ll \text{MAX\_TOP\_DEPTH}) - 1); \\
& ++\text{topNodeOffset[depth]}; \\
& \text{node} = \text{topNodes} + \text{offset};
\end{align*}
\]

For the bottom levels of the tree, we just hand out nodes as needed from a chunk of memory for \(\text{LOWER\_NODE\_ALLOC\_SIZE}\) nodes. Whenever we need to allocate a new chunk, we add its starting address to the \(\text{allocedNodeBlocks}\) vector, so that we can free all of the allocated memory when we’re done.

\[
\text{Allocate kd tree node for bottom part of tree} \\
\begin{align*}
& \text{if } (\text{lowerNodePos} == \text{LOWER\_NODE\_ALLOC\_SIZE}) \{} \\
& \quad \text{lowerNodePos} = 0; \\
& \quad \text{lowerNodes} = (\text{KdAccelNode} *) \text{AllocL2CacheAligned}(
\quad \text{LOWER\_NODE\_ALLOC\_SIZE}\ast \text{sizeof(\text{KdAccelNode})}); \\
& \quad \text{allocedNodeBlocks}.\text{push}\_\text{back}(\text{lowerNodes}); \\
& \{} \\
& \text{node} = \text{lowerNodes} + \text{lowerNodePos}; \\
& ++\text{lowerNodePos};
\end{align*}
\]

\[
\text{Set up memory pools for kd tree nodes} \\
\begin{align*}
& \text{int } \text{numTop} = (1 \ll \text{MAX\_TOP\_DEPTH}) - 1; \\
& \text{topNodes} = (\text{KdAccelNode} *) \text{AllocL2CacheAligned}(\text{numTop} \ast \text{sizeof(\text{KdAccelNode})}); \\
& \text{allocedNodeBlocks}.\text{push}\_\text{back}(\text{topNodes}); \\
& \text{for } (\text{int } i = 0; i < \text{MAX\_TOP\_DEPTH}; ++i) \\
& \quad \text{topNodeOffset}[i] = 0; \\
& \text{lowerNodes} = \text{NULL}; \\
& \text{lowerNodePos} = \text{LOWER\_NODE\_ALLOC\_SIZE};
\end{align*}
\]

\[
\text{KdTreeAccelerator Private Data} \\
\begin{align*}
& \text{KdAccelNode} \ast \text{topNodes}; \\
& \#\text{define MAX\_TOP\_DEPTH 10} \\
& \text{int } \text{topNodeOffset}[\text{MAX\_TOP\_DEPTH}]; \\
& \text{KdAccelNode} \ast \text{lowerNodes}; \\
& \text{int } \text{lowerNodePos}; \\
& \text{vector<\text{KdAccelNode} \ast > allocedNodeBlocks};
\end{align*}
\]

XXX need to run destructors for the ones that we constructed...

\[
\text{Free memory pools for KdTreeNode} \\
\begin{align*}
& \text{for } (\text{u}\_\text{int } i = 0; i < \text{allocedNodeBlocks}.\text{size}(); ++i) \\
& \quad \text{FreeCacheAligned(allocedNodeBlocks[i])};
\end{align*}
\]
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 $[t_{\text{min}}, t_{\text{max}}]$ 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 $[t_{\text{min}}, t_{\text{split}}]$. 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 $[t_{\text{min}}, t_{\text{split}}]$, 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.

**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.

```cpp
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 its nodes; see Figure 4.11. We start by intersecting the ray with the tree’s overall bounds, giving us initial $t_{\text{min}}$ and $t_{\text{max}}$ values, marked with “x”s in the figure. Because the ray does intersect the top-level bounds, ...
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.

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

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 \( t_{\text{min}} \) 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;
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.

\[\text{Process \(kd\) tree interior node}\]\\
\[\text{Compute distance along ray to split plane}\]
\[\text{Get near and far child pointers for ray}\]
\[\text{Advance to next child node, possibly enqueue far child}\]

The parametric distance to the split plane is computed in the same manner as was done in the ray–bounding box test, for example.

\[\text{Compute distance along ray to split plane}\]\\
Float \(t_{\text{plane}}\);
if (node->axis == SPLIT_X)
  \(t_{\text{plane}} = (\text{node->split} - \text{ray.O.x}) \times \text{invDir.x}\);
else if (node->axis == SPLIT_Y)
  \(t_{\text{plane}} = (\text{node->split} - \text{ray.O.y}) \times \text{invDir.y}\);
else
  \(t_{\text{plane}} = (\text{node->split} - \text{ray.O.z}) \times \text{invDir.z}\);

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 $t_{\text{min}}$ position and thus doesn't enter the far child. The bottom ray is facing away from the splitting plane, indicated by a negative $t_{\text{split}}$ value. On the right, the ray intersects the ray before the ray's $t_{\text{min}}$ value, indicating that the near plane doesn't need processing.

```c
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)
}

(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...)

(b) 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 introduce 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 performance. 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 lighting 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, $\lambda \approx 400\text{nm}$ are the blue-ish colors, the middle wavelengths $\lambda \approx 550\text{nm}$ are the greens, and the upper wavelengths $\lambda \approx 650\text{nm}$ 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, $\lambda$. 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

\[ \langle \text{color.h} \rangle \equiv \]
\[ \langle \text{Source Code Copyright} \rangle \]
\#ifndef COLOR_H
\#define COLOR_H
\#include "lrt.h"
\langle Color Declarations \rangle
\#endif // COLOR_H

\[ \langle \text{color.cc} \rangle \equiv \]
\[ \langle \text{Source Code Copyright} \rangle \]
\#include "color.h"
\langle Spectrum Method Definitions \rangle

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 IRT 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

\[
c_i = \int B_i(\lambda)S(\lambda)\,d\lambda,
\]

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:

\[
B(\lambda)_{a,b} = \begin{cases} 
1 & : a \leq \lambda < b \\
0 & : \text{otherwise}
\end{cases}
\]

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:

\[
\begin{align*}
    c'_i &= \int B_i(\lambda)(kS(\lambda))\,d\lambda \\
    c'_i &= k\int B_i(\lambda)S(\lambda)\,d\lambda \\
    c'_i &= kc_i
\end{align*}
\]

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 \( c^1_i \) and \( c^2_i \), the sum \( S_1(\lambda) + S_2(\lambda) \) can be shown to be

\[
c'_i = \sum c^1_i + c^2_i.
\]
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:

\[
 c_i = \int B_i(\lambda) (S_1(\lambda)S_2(\lambda)) d\lambda \\
 \approx \int B_i(\lambda) \left( \sum_j c^1_j B_j(\lambda) \right) \left( \sum_k c^2_k B_k(\lambda) \right) d\lambda \\
 = \sum_j \sum_k c^1_j c^2_k \int B_i(\lambda)B_j(\lambda)B_k(\lambda) d\lambda
\]

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

\[
 S_{i,j} = \int S_1(\lambda)B_i(\lambda)B_j(\lambda).
\]

Then, multiplication with another SPD is just a matrix-vector multiply with \( S \) and the vector \( c^2_i \), 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 \( i \neq j \),

\[
 \int B_i(\lambda)B_j(\lambda) d\lambda = 0
\]

and

\[
 \int B_i(\lambda)B_i(\lambda) d\lambda = 1.
\]

Under these assumptions, the coefficients for the product of two SPDs is just the produce of their coefficients

\[
 c_i = c^1_i c^2_i,
\]

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:

\[
 \int B_i(\lambda)S_1(\lambda)S_2(\lambda) d\lambda \neq \int B_i(\lambda) \left( \sum_j c^1_j B_j(\lambda) \right) \left( \sum_k c^2_k B_k(\lambda) \right) d\lambda.
\]

This is a natural consequence of both the error introduced in the original transformation to a basis function representation 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.

```
#define COLOR_SAMPLES 3
```

**Color Declarations**

```cpp
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(Float intens = 0.) {
  for (int i = 0; i < COLOR_SAMPLES; ++i)
    c[i] = intens;
}
```

```
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 &operator+=(const Spectrum &s2) {
  for (int i = 0; i < COLOR_SAMPLES; ++i)
    c[i] += s2.c[i];
  return *this;
}
```
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 component-wise. 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.

bool operator==(const Spectrum &sp) const {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        if (c[i] != sp.c[i]) return false;
    return true;
}

bool Black() const {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        if (c[i] != 0.) return false;
    return true;
}

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...

bool Black() const {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        if (c[i] != 0.) return false;
    return true;
}

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 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 \(\text{lrt}\), 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 equilibrium, such that its radiance distribution isn’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 (?).
Radiant flux, $\Phi$, 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, $\lambda$. 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 $\Phi$. 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 \( \Phi \) 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

\[
E = \frac{d\Phi}{dA}.
\]  \hspace{1cm} (5.2.2)
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 \( \theta \)) 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 \( 4\pi \) and a hemisphere subtends \( 2\pi \).

We will use the symbol \( \omega \) 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 \( \omega \) 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 $\bar{\omega}$ to the differential solid angle $d\bar{\omega}$ times the differential projected area of the receiving point.

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

$$L = \frac{d^2\Phi}{d\bar{\omega} dA^\perp}$$  \hspace{1cm} (5.2.4)

where $dA^\perp$ is the projected area of $dA$ on a hypothetical surface perpendicular to $\bar{\omega}$—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 $d\bar{\omega}$ 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 $\Omega$ is

$$E(x) = \int_{\Omega} L(x, \bar{\omega}) \cos \theta d\bar{\omega},$$  \hspace{1cm} (5.2.5)

where $L(x, \bar{\omega})$ denotes the arriving radiance at position $x$ as seen along direction $\bar{\omega}$ (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 $H^2$ or the entire sphere of directions $S^2$.

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
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.

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.

The projected solid angle measure is related to the solid angle measure by $d\omega = \cos \theta d\tilde{\omega}$, so the irradiance-from-radiance integral can be written more simply as

$$E = \int_{\mathbb{S}^2} L(\tilde{\omega}) d\tilde{\omega} \perp.$$ 

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 \((\theta, \phi)\) 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.

**Integrals over spherical coordinates**

It is often convenient to transform integrals over solid angle into integrals over spherical coordinates \((\theta, \phi)\). Recall that an \((x, y, z)\) direction vector can be alternatively written in terms of spherical angles (see Figure 5.9):

\[
\begin{align*}
    x &= \sin \theta \cos \phi \\
    y &= \sin \theta \sin \phi \\
    z &= \cos \theta 
\end{align*}
\]

For convenience, we’ll define two functions that turn \(\theta\) and \(\phi\) values into \((x, y, z)\) direction vectors. The first applies the equations above directly.

\[\text{Geometry Inline Functions}\] 

\[
\begin{verbatim}
inline Vector SphericalDirection(Float sintheta, Float costheta, 
    Float phi) {
    return Vector(sintheta * cosf(phi),
                    sintheta * sinf(phi),
                    costheta);
}
\end{verbatim}
\]

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.

\[\text{Geometry Inline Functions}\] 

\[
\begin{verbatim}
inline Vector SphericalDirection(Float sintheta, Float costheta, 
    Float phi, const Vector &x, const Vector &y, 
    const Vector &z) {
    return sintheta * cosf(phi) * x +
           sintheta * sinf(phi) * y + costheta * z;
}
\end{verbatim}
\]
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, \( d\omega = \sin \theta d\theta d\phi \), is the key to converting between integrals over solid angles and integrals over spherical angles.

The spherical angles for a direction can be found by:

\[
\begin{align*}
\theta &= \arccos z \\
\phi &= \arctan \frac{y}{z}
\end{align*}
\]

Corresponding functions are below. Note that \( \text{SphericalTheta()} \) assumes that the vector \( v \) has been normalized before being passed in.


\begin{verbatim}
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;
}
\end{verbatim}

In order to write an integral over solid angle in terms of an integral over \( (\theta, \phi) \), we need to be able to express the relationship between the differential area of a set of directions \( d\omega \) and the differential area of a \( (\theta, \phi) \) pair—see Figure 5.10. The differential area \( d\omega \) is the product of the differential lengths of the sides of \( d\omega \), \( \sin \theta d\phi \) and \( d\theta \). Therefore,

\[
d\omega = \sin \theta d\theta d\phi.
\]

We can thus see that the irradiance integral over the hemisphere (Equation 5.2.5 with \( \Omega = H^2 \)) 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 $dA \cos \theta / r^2$, where $\theta$ 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 $A$:

$$\Phi = \int_A \int_{\Omega^2} L(x, \vec{d}) \cos \theta \, d\vec{d} \, dA$$

**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

$$d\vec{d} = \frac{dA \cos \theta}{r^2}$$

where $\theta$ 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 $d\vec{d}$, then $d\vec{d} = dA$, $\theta = 0$, 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 $d\vec{d}$, the $r^2$ and $\cos \theta$ terms compensate accordingly to reduce $d\vec{d}$.

Therefore, we can write the irradiance integral for the quadrilateral source as

$$E(x) = \int_A L \cos \theta_i \frac{\cos \theta_o dA}{r^2}$$

where $\theta_i$ is the angle between the surface normal at $x$ and the direction from $x$ to the point $x'$ on the light, and $\theta_o$ is the angle between the surface normal at $x'$ on the light and the direction from $x'$ to $x$ (see Figure 5.12.)
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

$$dL_o(\vec{\omega}_o) \propto dE(\vec{\omega}_i),$$
The constant proportionality for the particular pair of directions $\bar{\omega}_i$ and $\bar{\omega}_o$ is defined to be the surface’s BRDF:

$$f_r(\bar{\omega}_i, \bar{\omega}_o) = \frac{dL(\bar{\omega}_o)}{dE(\bar{\omega}_i)} = \frac{dL(\bar{\omega}_o)}{L(\bar{\omega}_i) \cos \theta_i d\bar{\omega}_i} \quad (5.4.7)$$

Real-world BRDFs have two important qualities:

1. **Reciprocity**: for all pairs of directions $\bar{\omega}_i$ and $\bar{\omega}_o$, $f_r(\bar{\omega}_i, \bar{\omega}_o) = f_r(\bar{\omega}_o, \bar{\omega}_i)$.

2. **Energy conservation**: the total energy of light reflected is less than or equal to the energy of incident light. For all directions $\bar{\omega}$, $\int_{S^2} f(\bar{\omega}_i, \bar{\omega}) \cos \theta_i d\bar{\omega}_i \leq 1$.

The surface’s bidirectional transmittance distribution function (BTDF) can be defined in a similar manner to the BRDF. The BTDF is generally denoted by $f_t(\bar{\omega}_i, \bar{\omega}_o)$, where $\bar{\omega}_i$ and $\bar{\omega}_o$ 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 $f(\bar{\omega}_i, \bar{\omega}_o)$; 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

$$dL_o(\bar{\omega}_o) = L_i(\bar{\omega}_i) f(\bar{\omega}_i, \bar{\omega}_o) \cos \theta_i d\bar{\omega}_i.$$ 

We can integrate this over the sphere of incident directions around $x$ to compute the outgoing radiance in direction $\bar{\omega}_o$ due to the incident illumination at $x$:

$$L_o(\bar{\omega}_o) = \int_{S^2} L_i(\bar{\omega}_i) f(\bar{\omega}_i, \bar{\omega}_o) |\cos \theta_i| d\bar{\omega}_i \quad (5.4.8)$$

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 $H^2$ is being integrated over, or the scattering equation when the sphere $S^2$ 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-186th 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/m²sr 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/m² 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 \( \theta \), 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 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

```c
#include "camera.h"
#include "lrt.h"
#include "color.h"
#include "sampling.h"
#include "geometry.h"
#include "transform.h"
```

```c
Camera Declarations
```

```c
#endif // CAMERA_H
```

```c
#include "lrt.h"
#include "color.h"
#include "sampling.h"
#include "geometry.h"
#include "transform.h"
```

```c
Camera Declarations
```

```c
#endif // CAMERA_H
```
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.

```c
#include "lrt.h"
#include "camera.h"
#include "film.h"
#include "mc.h"
```

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 cameras store a shutter open and shutter close time.
Camera Method Implementations

```cpp
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

```cpp
Transform WorldToCamera, CameraToWorld;
Float ClipHither, ClipYon, invClipHither;
Float ShutterOpen, ShutterClose;
```

Camera Public Data

```cpp
Film *film;
```

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 \texttt{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 \texttt{z} axis is mapped to the viewing direction and the \texttt{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 \texttt{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 \texttt{z} values in
Figure 6.2: A handful of camera-related coordinate spaces help to simplify the implementation of Cameras. 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 $z = \text{hither}$ in camera space. The image plane is at $z = 0$ in raster space, where $x$ and $y$ range from $(0,0)$ to $(x\text{Resolution} - 1, y\text{Resolution} - 1)$. 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 \( (x_{\text{Resolution}} - 1, y_{\text{Resolution}} - 1) \), 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 to 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.

```cpp
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.

```cpp
class ProjectiveCamera : public Camera {
public:
    virtual Float ScreenDepth(const Point &Pworld) const = 0;
};
```

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.

\section*{Camera Method Implementations}

```cpp
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 \texttt{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.

\section*{Compute projective camera transformations}

```cpp
CameraToScreen = proj;
WorldToScreen = CameraToScreen * WorldToCamera;
```

\section*{ProjectiveCamera Options}

```cpp
Transform CameraToScreen, WorldToScreen, RasterToCamera;
```

The only non-trivial one of the precomputed transformations is \texttt{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.

\section*{Compute projective camera screen transformations}

```cpp
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();
```

\section*{ProjectiveCamera Options}

```cpp
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.

```cpp
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 \( z = \text{hither} \) face of the box.

**Orthographic Camera**

```cpp
#include "camera.h"
#include "paramset.h"

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 `Orthographic()` transformation function which will be defined shortly.
Figure 6.4: orthographic ray generation: raster space to ray...

OrthographicCamera Definitions

OrthographicCamera::OrthographicCamera(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) {
  return Scale(1.f, 1.f, 1.f / (zfar-znear)) *
             Translate(Vector(0.f, 0.f, -znear));
}

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 $\text{imagex}$ and $\text{imagex}$ 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 \textit{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);
}

OrthographicCamera Definitions

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);

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 $z' = 1$.

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:

\[
\begin{align*}
    x' &= \frac{x}{z} \\
    y' &= \frac{y}{z} \\
    z' &= \frac{f(z - n)}{z(f - n)}
\end{align*}
\]

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.)
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]\].

\[
\text{Transform Methods} + \equiv
\]

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);
}

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 \(\text{ray.D}\) from the point \(\text{Pcamera}\).

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.
PerspectiveCamera Definitions

```cpp
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-world 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 still 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 infinitesimally small 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.

$$\langle \text{Initialize depth of field parameters} \rangle \equiv$$

```plaintext```
LensRadius = lensr;
FocalDistance = focald;
```

$$\langle \text{ProjectiveCamera Options} \rangle \equiv$$

```plaintext```
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.)
```c
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.

```c
Sample point on lens
Float lu, lv;
ConcentricSampleDisk(sample->lensx, sample->lensy, &lu, &lv);
lu *= LensRadius;
lv *= LensRadius;
```

We next compute 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.

```c
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 direction is set so that the ray still passes through the point on the plane of focus, \( P_{focus} \).

```c
Update ray for effect of lens
ray.O.x += lu;
ray.O.y += lv;
ray.D = Pfocus - ray.O;
```

### 6.3 Environment Camera

```c
#include "camera.h"
#include "film.h"
#include "paramset.h"
```

```c
EnvironmentCamera Definitions
```
Environment Camera

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.

\( \text{EnvironmentCamera} \) Declarations

```cpp
class EnvironmentCamera : public Camera {
public:
    EnvironmentCamera(const Transform &world2cam, Float hither, Float yon, Float sopen, Float sclose, Film *film);
    virtual Float ScreenDepth(const Point &Pworld) const;
private:
    Transform world2cam;
};
```

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 \((\theta, \phi)\) 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 \(\pi\), at the bottom of the image, and phi values range from 0 to \(2\pi\), 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.0 = rayOrigin;
    Generate environment camera ray direction
    Set ray time value
    ray.mint = 0.;
    ray.maxt = INFINITY;
}

To compute the \((\theta, \phi)\) coordinates for this ray, we first compute NDC coordinates from the raster image sample position. These are then scaled up to cover the \((\theta, \phi)\) range and then the spherical coordinate formula is used to compute 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

```c++
#include "film.h"
#include "tonemap.h"
```

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.

```c++
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.

\textit{(Film Method Definitions)}

\begin{verbatim}
Film::Film(int xres, int yres, const Extent2D &cropWindow) {
    xResolution = xres;
    yResolution = yres;
    \textit{(Compute film image extent)}
    xPixelStart = Ceil2Int(xResolution * cropWindow.x0);
    xPixelWidth = Ceil2Int(xResolution * cropWindow.x1) - xPixelStart;
    \textit{(Allocate film image storage)}
    yPixelStart = Ceil2Int(yResolution * cropWindow.y0);
    yPixelWidth = Ceil2Int(yResolution * cropWindow.y1) - yPixelStart;
}
\end{verbatim}

The film constructor starts by initializing a few parameters passed into the constructor. One of the most important parameters is the overall image resolution—\texttt{xResolution} and \texttt{yResolution} hold the total number of pixels in the x and y directions.

\textit{(Film Public Data)}

\begin{verbatim}
int xResolution, yResolution;
\end{verbatim}

The user may have also specified a \texttt{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. \texttt{xPixelStart} and \texttt{yPixelStart} store the pixel position of the upper left corner of the crop window, and \texttt{xPixelWidth} and \texttt{yPixelWidth} give the pixel widths in each direction. Given a pixel \((x,y)\) inside the pixel crop window, the pixel arrays are indexed as

\[(y - yPixelStart) \times xPixelWidth + (x - xPixelStart).
\]

\textit{(Compute film image extent)}

\begin{verbatim}
xPixelStart = Ceil2Int(xResolution * cropWindow.x0);
xPixelWidth = Ceil2Int(xResolution * cropWindow.x1) - xPixelStart;
yPixelStart = Ceil2Int(yResolution * cropWindow.y0);
yPixelWidth = Ceil2Int(yResolution * cropWindow.y1) - yPixelStart;
\end{verbatim}
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];

struct Pixel {
    Pixel() { alpha = 0.; depth = INFINITY; weightSum = 0.; }
    Spectrum L;
    Float alpha, depth, weightSum;
};
Pixel *pixels;

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 representation 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?

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 (CHS93), 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
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 approximation 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 \( \Delta x \) 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 \( \omega_s \) is greater than twice the maximum frequency present in the signal \( \omega_m \), 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

\[
\tilde{f}(x) = \sum_i f(x_i) r(x - x_i)
\]  

(7.1.1)

where \( r(x) \) is the ideal reconstruction filter (also known as the sinc function):

\[
r(x) = \text{sinc} \left( \frac{\omega}{2\pi} x \right)
\]

where \( \omega \) 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 use \textit{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 \(x \in [0, x\text{Resolution})\) and \(y \in [0, y\text{Resolution})\). 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

```c
#ifndef SAMPLING_H
#define SAMPLING_H
#include "lrt.h"

Sampling Constants
Sampling Declarations
#endif // SAMPLING_H
```

```c
#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.
\textit{Sampling Declarations} 

\begin{verbatim}
class Sampler {
public:
    \textit{Sampler Interface}
    \textit{Sampler Options}
};
\end{verbatim}

All Samplers 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 \textit{Filter} to be used to filter the image samples to compute the final pixels. We store these values in member variables for later use.

\textit{Sampler Method Definitions} 

\begin{verbatim}
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;
}
\end{verbatim}

\begin{verbatim}
\textit{Initialize pixel extents from crop window}
\end{verbatim}

\textit{Sampler Options} 

\begin{verbatim}
int xResolution, yResolution;
int xPixelSamples, yPixelSamples;
Filter *filter;
\end{verbatim}

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 \(x_{\text{PixelStart}}\) to \(x_{\text{PixelEnd}}-1\), inclusive, in \(x\) (and analogously in \(y\)) should be generated by the \textit{Sampler}. The fragment that implements \textit{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.

\textit{Sampler Options} 

\begin{verbatim}
int xPixelStart, xPixelEnd, yPixelStart, yPixelEnd;
\end{verbatim}

Samplers need to implement the \textit{GetNextSample()} method, which is here declared as a pure virtual function. The \textit{Scene::Render()} method will call this function until it returns \textit{false}; as long as it keeps returning \textit{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.

\textit{Sampler Interface} 

\begin{verbatim}
virtual bool GetNextSample(Sample *sample) = 0;
\end{verbatim}

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 \textit{TotalSamples()} method returns the total number of samples that the \textit{Sampler} will be returning.
\textbf{Sample Interface}\textsuperscript{+\equiv}

```cpp
int TotalSamples() const {
    return xPixelSamples * yPixelSamples * (xPixelEnd - xPixelStart) *
            (yPixelEnd - yPixelStart);
}
```

\textbf{Sample representation}

The Sample structure

Explain something about different numbers needed at different ray depths, etc...

\textbf{Sampling Declarations}\textsuperscript{+\equiv}

```cpp
struct Sample {
    public:
        \textbf{Sample Method Declarations}
        Float imagex, imagey;
        Float lensx, lensy;
        Float time;
        vector<int> nLightSamples, nBSDFSamples;
        Float **light, **bsdf;
    
    \textbf{Sample Method Definitions}\textsuperscript{=}\textsuperscript{=}
    Sample::Sample(const vector<int> &nLight, const vector<int> &nBSDF) {
        nLightSamples = nLight;
        nBSDFSamples = nBSDF;
        \textit{Compute total number of light and BSDF samples needed}
        \textit{Allocate storage for light and BSDF sample pointers}
        \textit{Allocate storage for light and BSDF sample memory}
    }
    
    \textit{Compute total number of light and BSDF samples needed}\textsuperscript{=}\textsuperscript{=}
    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;
    
    \textit{Allocate storage for light and BSDF sample pointers}\textsuperscript{=}\textsuperscript{=}
    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();
    }
```
Allocate storage for light and BSDF sample memory:

```c
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

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 $x\text{Resolution} \times x\text{PixelSamples}$ 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 $\text{XPos}$ and $\text{YPos}$ variables, which are initialized to point at first stratum in the upper left of the image’s crop window to start out.

### StratifiedSampler Declarations

```cpp
class StratifiedSampler : public Sampler {
public:
    StratifiedSampler Method Declarations
    private:
    StratifiedSampler Private Data
    StratifiedSampler Private Methods
};
```

### StratifiedSampler Method Definitions

```cpp
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
}
```

### StratifiedSampler Private Data

```cpp
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

```cpp
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
Sec. 7.3] Stratified Sampling 203

---

**StratifiedSampler Method Definitions**

```cpp
good distribution–not more and not less....

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;
        }
}

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;
    }
}

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.

```cpp

```cpp
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;
    }
}

```cpp

**Decorrelate sample dimensions**

shuffle2D(lensSamples);
shuffle1D(timeSamples);

---

"
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.

```cpp
bool StratifiedSampler::GetNextSample(Sample *sample) {
  // 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
  }
  return true;
}
```

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.

\begin{verbatim}
Advance to next stratum
if (++XPos == xPixelEnd) {
    XPos = xPixelStart;
    ++YPos;
}
\end{verbatim}

\begin{verbatim}
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];
\end{verbatim}

\begin{verbatim}
Generate stratified lens and BSDF samples
++samplePos;
Need to explain latin hypercube stuff somewhere....
\end{verbatim}

\begin{verbatim}
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);
\end{verbatim}

### 7.4 Low-Discrepancy Sequences

\begin{verbatim}
hammersley.cc*
\end{verbatim}

\begin{verbatim}
Source Code Copyright
#include "sampling.h"
#include "paramset.h"
\end{verbatim}

\begin{verbatim}
HammersleySampler Declarations
HammersleySampler Method Definitions
\end{verbatim}

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**

Mathematicians 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:

\[
B = \{ [0, v_1] \times [0, v_2] \times \cdots \times [0, v_n] \},
\]

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

\[
D_N(B, P) = \sup_{b \in B} \left| \frac{\# \{ x_i \in b \}}{N} - \lambda(b) \right|,
\]

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 \( D_N^* \). (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 \( \# \{ x_i \in B \} = 0 \). This interval (and the intervals \([0, \frac{2}{N}], \ldots, \frac{N-1}{N}, [\frac{1}{N}, 1] \)) 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:

\[ D_N^*(x_i) = \frac{1}{2N} + \max_{1 \leq i \leq N} \left| x_i - \frac{2i-1}{2N} \right| \]

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 \( a_m \ldots a_2 a_1 \) uniquely determined by:

\[ n = \sum_{i=1}^{\infty} a_i b^{-i} \]

Then, the radical inverse function \( \Phi_b \) 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:

\[ \Phi_b(n) = 0.a_1 a_2 \ldots a_m \]

The function `RadicalInverse()` computes the radical inverse for a given number \( n \) in the base \( b \). 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.

```cpp
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.

```c
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.

$$x_i = \Phi_2(i)$$

<table>
<thead>
<tr>
<th>n</th>
<th>base 2</th>
<th>$\Phi_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>.1 = 1/2</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>.01 = 1/4</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>.11 = 3/4</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>.001 = 3/8</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>.101 = 5/8</td>
</tr>
</tbody>
</table>

Figure 7.9: The radical inverse of the first few positive integers, computed in base 2: $\Phi_2(n)$. Notice how successive values of $\Phi_2(n)$ are far from all previous values of $\Phi_2(n)$.

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,

$$D_N^*(P) = O\left(\frac{(\log N)^r}{N}\right).$$

Two well-known low-discrepancy 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

$$D_N(x_i) = O\left(\frac{\log N}{N}\right),$$

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:

$$x_i = \left(\frac{i-\frac{1}{2}}{N}, \Phi_2(i), \Phi_3(i), \ldots, \Phi_p(i)\right),$$

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 $i$th 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 $\Psi_b$. 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 $\Psi_b$ 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 $\text{ib}$ 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.

```c
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

\[
    \text{int digit} = ((n + \text{modOffset}) \mod \text{base});
\]
\[
    n /= \text{base};
\]
\[
    \text{val} += \text{digit} \times \text{scale};
\]
\[
    \text{scale} *= \text{invBase};
\]
\[
    ++\text{modOffset};
\]

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 Hammersley 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 \((\text{xPixelStart}, \text{yPixelStart})\) and scaled by a constant amount in both directions so that it covers the pixels up to \((\text{xPixelEnd}, \text{yPixelEnd})\). Any generated samples that are past \((\text{xPixelEnd}, \text{yPixelEnd})\) 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.

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

\[
    \text{class HammersleySampler : public Sampler {}
    \]
\[
    \text{public:}
    \]
\[
    \text{(HammersleySampler Method Declarations)}
    \]
\[
    \text{private:}
    \]
\[
    \text{(HammersleySampler Private Data)}
    \]
\[
    \text{}};
    \]

The constructor computes the length of a side of the square region samples are generated inside of, \(\text{extent}\), the total number of samples to generate (and its inverse), \(\text{nSamples}\) and \(\text{invNSamples}\), and the sample number \(i\) of the next Hammersley point \(x_i\) to be computed by \text{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 \( \Phi_b \) in the top row, and the folded radical inverse \( \Psi_b \) 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.

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

```cpp
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

```cpp
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-discrepancy light and BSDF samples

For now, Cranley–Patterson stuff. Should do Köllig and Keller stuff...

```cpp
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

```cpp
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

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

```cpp
#include "lrt.h"
#include "sampling.h"
```
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**

```c
#define SQRT_SAMPLE_TABLE_SIZE 64
#define SAMPLE_TABLE_SIZE (SQRT_SAMPLE_TABLE_SIZE * \nSQRT_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**

```c
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**

```c
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**

```c
#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**

```c
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.

```cpp
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.

```cpp
table[0][0] = RandomFloat();
table[0][1] = RandomFloat();
addSampleToGrid(table, 0, grid);
```

A short utility function adds the entryth item in the given table of samples to the given SampleGrid.

```cpp
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 best candidate was, we keep it. At the end of the loop, the remaining candidate is kept.
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();

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.

\[ \text{Update minimum squared distance from cell's samples} \]
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 0.01 and 0.99, respectively. Direct computation will find their distance to be 0.98, though with wrap-around, the actual distance should be 0.02. 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.

\[ \text{Pattern Precomputation Utility Functions} \]
\[
\text{inline Float Wrapped1DDist(Float a, Float b) \{ }
    Float d = fabsf(a - b);
    if (d < .5) return d;
    else return 1 - max(a, b) + min(a, b);
\}
\]

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.

\[ \text{Keep this sample if it is the best one so far} \]
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 \( \text{timeValues} \) array so that the \( i \)'th time sample is a good one for the \( i \)'th image sample.

\[
\langle \text{Compute time samples} \rangle \\
\text{cerr} \ll \text{"Computing time samples: ";} \\
\text{for (int } i = 0; i < \text{SAMPLE_TABLE_SIZE}; ++i) \\
\quad \text{timeSamples}[i] = (i + \text{RandomFloat}) / \text{SAMPLE_TABLE_SIZE}; \\
\text{for (int currentSample = 1; currentSample < \text{SAMPLE_TABLE_SIZE}; ++currentSample) } \\
\quad \text{if ((currentSample } \% \text{ (SAMPLE_TABLE_SIZE/52)) } = = 0) \\
\quad \quad \text{cerr} \ll \text{‘+’;} \\
\langle \text{Select best time sample for current image sample} \rangle \\
\]

\[
\langle \text{Select best time sample for current image sample} \rangle \\
\text{int best = -1;} \\
\langle \text{Find best time relative to neighbors} \rangle \\
\text{Assert(best } != -1);} \\
\text{swap(timeSamples[best], timeSamples[currentSample]);} \\
\]

Given that we’re working on finding a good time for the sample number \( \text{currentSample} \), the elements of \( \text{timeSamples} \) from zero to \( \text{currentSample-1} \) have already been assigned to previous image samples and are unavailable to us. The rest of the times, from \( \text{currentSample} \) to \( \text{SAMPLE_TABLE_SIZE-1} \), are the ones we will choose from.

\[
\langle \text{Find best time relative to neighbors} \rangle \\
\text{Float maxMinDelta = 0.;} \\
\text{for (int } t = \text{currentSample}; t < \text{SAMPLE_TABLE_SIZE}; ++t) \{ \\
\quad \langle \text{Compute min delta for this time} \rangle \\
\quad \langle \text{Update best if this is best time so far} \rangle \\
\} \\
\]
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.

\(\text{Compute min delta for this time}\) 

\[
\begin{align*}
\text{int } gu &= \text{GRID}(\text{pixelSamples}[\text{currentSample}][0]); \\
\text{int } gv &= \text{GRID}(\text{pixelSamples}[\text{currentSample}][1]); \\
\text{Float } \text{minDelta} &= \text{INFINITY}; \\
\text{for (int } du = -1; \ du \leq 1; \ ++du) { \\
& \text{for (int } dv = -1; \ dv \leq 1; \ ++dv) { \\
& \quad \text{Check distance from times of nearby samples} \\
& \text{}} \\
& } \\
\end{align*}
\]

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.

\(\text{Check distance from times of nearby samples}\)

\[
\begin{align*}
\text{Compute } (u,v) \text{ grid cell to check} \\
\text{for } (\text{u_int } g = 0; \ g < \text{pixelGrid}[u][v].\text{size()}; \ ++g) { \\
& \text{int otherSample } = \text{pixelGrid}[u][v][g]; \\
& \text{if (otherSample } < \text{currentSample) } \\
& \quad \text{Float } \text{dt } = \text{Wrapped1DDist}(\text{timeSamples}[\text{otherSample}], \\
& \quad \quad \text{timeSamples}[t]); \\
& \quad \text{minDelta } = \text{min(minDelta, dt);} \\
& \text{}} \\
\end{align*}
\]

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.

\(\text{Update best if this is best time so far}\)

\[
\begin{align*}
\text{if (minDelta } > \text{maxMinDelta) } \\
& \quad \text{maxMinDelta } = \text{minDelta}; \\
& \quad \text{best } = t; \\
& \text{}} \\
\end{align*}
\]

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.
Sampling and Reconstruction

(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)
            
            Assert(best != -1);
            swap(samples[best][0], samples[currentSample][0]);
            swap(samples[best][1], samples[currentSample][1]);
        }
        cerr << endl;
    }

(Select best sample for current image sample)

int best = -1;

(Find best 2D sample relative to neighbors)

As with time, we want to choose the sample from the available ones that maxi-
mizes 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);
    }
  }

(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.
Sampling and Reconstruction [Ch. 7]

Sampling and Reconstruction

§ (Output sample table)

FILE *f = fopen("sampledata.cc", "w");
Assert(f);
fprintf(f, "\n/* Automatically generated \d sample "
"table (%d @ %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, \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\n\nUsing 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 ex-
tent 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

```cpp
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

```cpp
int tableOffset;
Float XTablePos, YTablePos;
Float XTableWidth, YTableWidth;
```

Here we incorporate the precomputed sample data.

BestCandidateSampler Private Data

```cpp
static const Float sampleTable[SAMPLE_TABLE_SIZE][9];
```

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 \(\xi_i\) 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 \(\xi_i\) is updated each time we are about to reuse the table once again.

Update sample shifts

```cpp
for (int i = 0; i < 9; ++i)
   sampleOffsets[i] = RandomFloat();
```

BestCandidateSampler Private Data

```cpp
Float sampleOffsets[9];
```

The GetNextSample() has a similar structure to the one for StratifiedSampler...
Sampling and Reconstruction

\textit{BestCandidateSampler Method Definitions}\textsuperscript{+≡

bool BestCandidateSampler::GetNextSample(Sample *sample) {
    again:
        \textit{Return false if BestCandidateSampler is done}
        \textit{Compute raster sample from table}
        \textit{Advance to next sample table position}
        \textit{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.
\textit{Return false if BestCandidateSampler is done}\textsuperscript{≡

    if (YTablePos >= yPixelEnd)
        return false;

It's just some simple indexing and scaling to compute the raster-space sam-
ple 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 shift-
ning method described above, using the \texttt{WRAP} macro that ensures that the result stays
between 0 and 1.

\textit{Compute raster sample from table}\textsuperscript{≡

    \#define WRAP(x) ((x) > 1 ? ((x)-1) : (x))
    sample->imaginex = 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]);

\textit{BestCandidateSampler Method Definitions}\textsuperscript{+≡

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.

\(\text{Advance to next sample table position}\)

\[
\text{if (++tableOffset == SAMPLE_TABLE_SIZE) } \{
\quad \text{Update sample shifts}
\quad \text{tableOffset} = 0;
\quad \text{XTablePos} += \text{XTableWidth};
\quad \text{if (XTablePos} >= \text{xPixelEnd} \} \{
\quad \text{XTablePos} = \text{xPixelStart};
\quad \text{YTablePos} += \text{YTableWidth};
\}
\]

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.

\(\text{Check sample against crop window, goto again if outside}\)

\[
\text{if (sample->imagex} < \text{xPixelStart || sample->imagex} >= \text{xPixelEnd || sample->imagey} < \text{yPixelStart || sample->imagey} >= \text{yPixelEnd}) \quad \text{goto again;}
\text{sample->imagex} -= .5f;
\text{sample->imagey} -= .5f;
\]

### 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 \(\hat{L}\) from the set of image samples.
2. Prefilter the function \(\hat{L}\) to remove any frequencies past the Nyquist limit for the pixel spacing.
3. Sample \(\hat{L}\) at the pixel locations to compute the final pixel values.

Because we know that we will only be resampling the \(\hat{L}\) 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.
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:

\[
p(x, y) = \frac{\sum_i f(x - x_i, y - y_i) L(x_i, y_i)}{\sum_i f(x - x_i, y - y_i)}
\]

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 \( x\text{width} \) in the \( x \) direction and \( y\text{width} \) 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.

\[\langle\text{Sampling Declarations}\rangle + \equiv\]

```cpp
class Filter {
  public:
    \langle Filter Interface \rangle
    \langle Filter Data \rangle
};
```
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.

```cpp
Filter Interface
Filter(Float xw, Float yw) :
    xWidth(xw), yWidth(yw), halfXWidth(.5f*xw),
    halfYWidth(.5f*yw) {
}
```

```cpp
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.

```cpp
Filter Interface
virtual Float Evaluate(Float x, Float y) const = 0;
```

### Box Filter

```cpp
Box Filter Declarations
class BoxFilter: public Filter {
public:
    BoxFilter(Float xw, Float yw) : Filter(xw, yw) {
    } // Float BoxFilter::Evaluate(Float x, Float y) const {
        return 1.;
    }
};
```

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.

```cpp
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

\[
\text{triangle.cc}^*  \\
\text{(Source Code Copyright)}  \\
\#include "sampling.h"  \\
\#include "paramset.h"  \\
\text{(Triangle Filter Declarations)}  \\
\text{(Triangle Filter Definitions)}
\]

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.

\[
\text{Gaussian Filter}
\]

\[
\text{gaussian.cc}^*  \\
\text{(Source Code Copyright)}  \\
\#include "sampling.h"  \\
\#include "paramset.h"  \\
\text{(Gaussian Filter Declarations)}  \\
\text{(Gaussian Filter Definitions)}
\]
The Gaussian is the first filter in \texttt{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.

\begin{itemize}
\item \textit{Gaussian Filter Declarations}:
\begin{verbatim}
class GaussianFilter : public Filter {
public:
  GaussianFilter(Float xw, Float yw) : Filter(xw, yw) {
    expHalfX = expf(-halfXWidth);
    expHalfY = expf(-halfYWidth);
  }
}
\end{verbatim}
\end{itemize}

In the constructor, we precompute a few terms that will be the same every time we evaluate the filter.

\begin{itemize}
\item \textit{Gaussian Filter Interface}:
\begin{verbatim}
GaussianFilter(Float xw, Float yw) :
  : Filter(xw, yw) {
    expHalfX = expf(-halfXWidth);
    expHalfY = expf(-halfYWidth);
  }
\end{verbatim}
\end{itemize}

\begin{itemize}
\item \textit{Gaussian Filter Definitions}:
\begin{verbatim}
Float GaussianFilter::Evaluate(Float x, Float y) const {
  return Gaussian(x, expHalfX) * Gaussian(y, expHalfY);
}
\end{verbatim}
\end{itemize}

\begin{itemize}
\item \textit{Gaussian Filter Utility Functions}:
\begin{verbatim}
static Float Gaussian(Float t, Float expHalf) {
  return max(0.f, expf(-t) - expHalf);
}
\end{verbatim}
\end{itemize}
Mitchell Filter

\[ (\text{mitchell.cc}) \]

\[ \text{(Source Code Copyright)} \]

\#include "sampling.h"
\#include "paramset.h"

\[ \text{(Mitchell Filter Declarations)} \]

\[ \text{(Mitchell Filter Definitions)} \]

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.

\[ \text{(Mitchell Filter Declarations)} \]

\begin{verbatim}
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;
    }
\end{verbatim}

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 \texttt{lrt} are separable, though this wasn’t made explicit in the previous ones.)

\[ \text{(Mitchell Filter Definitions)} \]

\begin{verbatim}
Float MitchellFilter::Evaluate(Float x, Float y) const {
    return Mitchell1D(x/halfXWidth) * Mitchell1D(y/halfYWidth);
}
\end{verbatim}
\textit{Mitchell Filter Definitions}\textsuperscript{++} 

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;
}

\textit{Sinc Filter}\textsuperscript{*}

\textit{Source Code Copyright}\textsuperscript{229 Filter}
#include "sampling.h"
#include "paramset.h"

\textit{Sinc Filter Declarations}\textsuperscript{229 halfXWidth, 229 halfYWidth, 232 MitchellFilter}

\textit{Sinc Filter Definitions}\textsuperscript{229 Filter}

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.

\textit{Sinc Filter Declarations}\textsuperscript{++}

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.

\textit{Sinc Filter Definitions}\textsuperscript{++}

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.
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).

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...
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 Filters 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 denominator. When all of the samples have been processed, final pixel values can be computed by performing the division.

\[
\text{Sampler Method Definitions} + \equiv
\]

```cpp
void Sampler::AddSample(Film *film, const Point &Praster,
 const Spectrum &radiance, Float alpha) {
  \text{Compute sample’s raster extent}
  \text{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

\[\text{int } x_0 = \text{Ceil2Int}(\text{Praster.x} - \text{filter}->\text{halfXWidth});\]
\[\text{int } x_1 = \text{Floor2Int}(\text{Praster.x} + \text{filter}->\text{halfXWidth});\]
\[\text{int } y_0 = \text{Ceil2Int}(\text{Praster.y} - \text{filter}->\text{halfYWidth});\]
\[\text{int } y_1 = \text{Floor2Int}(\text{Praster.y} + \text{filter}->\text{halfYWidth});\]

Now, given the extent of pixels that are affected by this sample \((x_0, y_0)\) to \((x_1, y_1)\), 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

\[
\text{for (int } y = y_0; y <= y_1; ++y) \\
\text{for (int } x = x_0; x <= x_1; ++x) \\
\text{\{ \text{Evaluate filter value} \}} \\
\text{\{ \text{Update pixel values with filtered sample} \}} \\
\]

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

\[\text{Float } fx = x - \text{Praster.x};\]
\[\text{Float } fy = y - \text{Praster.y};\]
\[\text{Float } \text{filterWt} = \text{filter}->\text{Evaluate}(fx, fy);\]

Now we go ahead and report the weighted sample value to the Film.

Update pixel values with filtered sample

\[
\text{if } (\text{filterWt} > 0.) \\
\quad \text{film}->\text{UpdatePixel}(x, y, \text{radiance}, \text{alpha}, \text{Praster.z}, \text{filterWt});
\]

Computing normalized pixel values

We can now define the fragment \(\text{Apply filter weights}\) for the Film’s imaging pipeline. We divide each pixel sample value by the value of \(\text{weightSum}\) 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

\[
\text{for (int } o = 0; o < \text{xPixelWidth} * \text{yPixelWidth}; ++o) \\
\text{\{ \text{if (pixels[o].weightSum == 0.)} \}} \\
\text{\quad \text{continue;}} \\
\text{\quad \text{Float } invWt = 1.f / \text{pixels[o].weightSum;}} \\
\text{\quad \text{Lout}[o] *= invWt;} \\
\text{\quad \text{AlphaOut[o] *= invWt;}} \\
\]

Sample Crop Extents

We can now also define the fragment that determines 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.

\[
\text{Initialize pixel extents from crop window} = \]

\[
\begin{align*}
\text{int RasterCropLeft} &= \text{Ceil2Int}(x\text{Resolution} \times \text{Crop.x0}); \\
\text{int RasterCropRight} &= \text{Ceil2Int}(x\text{Resolution} \times \text{Crop.x1}); \\
\text{int RasterCropTop} &= \text{Ceil2Int}(y\text{Resolution} \times \text{Crop.y0}); \\
\text{int RasterCropBottom} &= \text{Ceil2Int}(y\text{Resolution} \times \text{Crop.y1}); \\
\text{xPixelStart} &= \text{Floor2Int}(\text{RasterCropLeft} - \text{filter->halfXWidth}); \\
\text{xPixelEnd} &= \text{Ceil2Int}(\text{RasterCropRight} + \text{filter->halfXWidth}); \\
\text{yPixelStart} &= \text{Floor2Int}(\text{RasterCropTop} - \text{filter->halfYWidth}); \\
\text{yPixelEnd} &= \text{Ceil2Int}(\text{RasterCropBottom} + \text{filter->halfYWidth});
\end{align*}
\]

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.

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.

\[
\text{Film Method Definitions} = \]

\[
\begin{align*}
\text{void Film::WriteCoefficients(const string &filename) const} &\{ \\
\text{Compute floating-point pixel SPD coefficients} &\{ \\
\text{TIFFWriteFloat(filename, \((\text{Float *)Lout, AlphaOut,} \\
\text{xPixelWidth, yPixelWidth, COLOR_SAMPLES,} \\
\text{xResolution, yResolution})}; \\
\text{Release temporary image memory} &\}
\}
\end{align*}
\]

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.

\[
\text{Compute floating-point pixel SPD coefficients} = \]

\[
\begin{align*}
\text{Allocate working imaging memory and copy data} &\{ \\
\text{Apply filter weights} &\{ \\
\text{Compute premultiplied alpha color values} &\}
\}
\end{align*}
\]
(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, \(\text{Apply filter weights}\), 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 \textit{premultiplied alpha} (also known as \textit{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.)

(Compute premultiplied alpha color values) ≡
for (int i = 0; i < xPixelWidth * yPixelWidth; ++i)
    Lout[i] *= AlphaOut[i];

(Release temporary image memory) ≡
del[] Lout;
del[] 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 \texttt{WriteDisplayImage()} method that applies each of a series of imaging operations in turn. A structure, \texttt{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 {
    \(\text{Compute floating-point pixel SPD coefficients}\)
    \(\text{Apply display imaging pipeline}\)
    \(\text{Release temporary image memory}\)
The basic display pipeline is shown in Figure 8.1. The fragment \( \langle \text{Apply display imaging pipeline} \rangle \) applies each of the stages in turn; we will describe them in order here.

\[ \langle \text{Apply display imaging pipeline} \rangle \equiv \]
\[ \langle \text{Convert image to XYZ} \rangle \]
\[ \langle \text{Apply tone reproduction to image} \rangle \]
\[ \langle \text{Convert image to display RGB} \rangle \]
\[ \langle \text{Scale and handle out-of-gamut RGB values} \rangle \]
\[ \langle \text{Apply gamma correction} \rangle \]
\[ \langle \text{Map image to display range} \rangle \]
\[ \langle \text{Dither image} \rangle \]
\[ \langle \text{Save display image to disk} \rangle \]

Once we have properly normalized SPD coefficients at each pixel, we will take advantage of a remarkable property of the human visual system that allows us to represent each pixel’s color with just three floating-point numbers. The \textit{tristimulus theory} of color perception says that all visible SPDs can be accurately represented for human observers with three values, \( x_\lambda \), \( y_\lambda \), and \( z_\lambda \).

Given a SPD \( S(\lambda) \), these values are computed by convolution with the \textit{spectral matching curves}, \( X(\lambda) \), \( Y(\lambda) \) and \( Z(\lambda) \) by

\[
x_\lambda = \int_S S(\lambda) X(\lambda) d\lambda
\]
\[
y_\lambda = \int_S S(\lambda) Y(\lambda) d\lambda
\]
\[
z_\lambda = \int_S S(\lambda) Z(\lambda) d\lambda
\]

The three matching curves are graphed in Figure 8.2. These curves were determined by the Commission Internationale de l’Éclairage 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 \( x_\lambda \), \( y_\lambda \), and \( z_\lambda \) values. To the human observer, such SPDs actually appear the
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 $x_\lambda$,

$$
x_\lambda = \int S(\lambda)X(\lambda) d\lambda \\
\approx \int \sum_i c_i B_i(\lambda)X(\lambda) d\lambda \\
= \sum_i c_i \left( \int B_i(\lambda)X(\lambda) d\lambda \right) \\
= \sum_i c_i w_i^x.
$$

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

```c
void XYZ(Float xyz[3]) const {
    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...

```c
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:

```c
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.

```c
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];
```
Given the $\text{XYZ()}$ method in Spectrum, it’s easy for us to convert to an XYZ image.

$$\text{Convert image to XYZ}$$

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 $y_\lambda$ value for the $i$th pixel, etc.

$$\text{ToneMap Declarations}$$

#define X(i) (xyz[3*(i)])
#define Y(i) (xyz[3*(i)+1])
#define Z(i) (xyz[3*(i)+2])

8.4 Tone Mapping

$$\langle \text{tonemap.h} \rangle$$

#include "lrt.h"
#include "film.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 realistic 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 appearance 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

$$Y = \int L(\lambda) V(\lambda) d\lambda.$$  

Fortunately, the CIE $Y(\lambda)$ tristimulus curve was chosen to be proportional to $V(\lambda)$ so that

$$Y = 683 \int L(\lambda) Y(\lambda) d\lambda.$$  

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 $i$th pixel:

```c
#define LUMINANCE(i) (683.f * Y(i))
```

The units of luminance are candelas per meter squared ($\text{cd}/\text{m}^2$), where the candela is the photometric equivalent of radiant intensity. The quantity $\text{cd}/\text{m}^2$ 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 $10 \text{ cd}/\text{m}^2$. Rods give little information about color and are not very good at resolving fine details. Cones handle light ranging from approximately .01 to $10^8 \text{ cd}/\text{m}^2$ (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 $100 \text{ cd}/\text{m}^2$.)

**Basic tone mapping approaches**
Figure 8.3: St. Peter’s Basilica in Rome...
Luminance (cd/m², or nits) | Description
--- | ---
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 scopic 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().

```cpp
class ToneMap {
public:
  virtual void Map(Float *xyz, int xRes, int yRes, const DisplayInfo &di) const = 0;
};
```

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.

```cpp
DisplayInfo Data
ToneMap *toneMap;
Float maxDisplayY, displayAdaptationY;
```

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.

```cpp
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:

```cpp
Apply tone reproduction to image
if (dinfo.toneMap)
  dinfo.toneMap->Map(xyz, xPixelWidth, yPixelWidth, dinfo);
```

**Maximum to white**

```cpp
#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

```cpp
class MaxWhiteOp : public ToneMap {
    void Map(Float *xyz, int xRes, int yRes, const DisplayInfo &di) const;
};
```

MaxWhiteOp Method Definitions

```cpp
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
}
```

Compute maximum luminance of all pixels

```cpp
Float maxLum = 0.;
for (int i = 0; i < xRes * yRes; ++i)
    maxLum = max(maxLum, LUMINANCE(i));
```

Apply scale to all image pixels

```cpp
for (int i = 0; i < xRes * yRes; ++i) {
    X(i) *= scale;
    Y(i) *= scale;
    Z(i) *= scale;
}
```

Contrast-based scale factor

```cpp
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.
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

\[
\Delta Y(Y^a) = 0.0594 \cdot (1.219 + (Y^a)^{0.4})^{2.5}.
\]

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

\[
\Delta Y(Y^a_d) = s \cdot \Delta Y(Y^a_w).
\]
We can substitute Blackwell’s model and solve this for \( s \), giving

\[
  s = \left( \frac{1.219 + (Y_{wa})^{0.4}}{1.219 + (Y_{wd})^{0.4}} \right)^{2.5}
\]

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 \( Y_{wa} \), we compute 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.

\[
\text{Compute world adaptation luminance} \\
\text{Float } Ywa = 0.; \\
\text{for (int } i = 0; i < xRes \times yRes; ++i) \\
\quad \text{if (LUMINANCE}(i) > 0) \ Ywa += \text{logf}(\text{LUMINANCE}(i)); \\
Ywa = \text{expf}(Ywa / (xRes \times yRes));
\]

And the display adaptation luminance, \( Y_{ad} \), is available in the \text{DisplayAdaptionY} field of the \text{DisplayInfo} structure. Because the scale-factor expects luminance values, we scale it by 683 before applying it to the pixels.

\[
\text{Compute contrast-preserving scalefactor} \\
\text{Float scale = powf((1.219f + powf(di.displayAdaptationY, 0.4f)) /} \\
\quad (1.219f + powf(Ywa, 0.4f)), 2.5f); \\
scale *= 683.f / \text{di.maxDisplayY};
\]

### Varying adaptation luminance

\[
\text{highcontrast.cc*} \\
\text{\langle Source Code Copyright\rangle} \\
\text{\#include "tonemap.h"} \\
\text{\#include "mipmap.h"} \\
\text{\langle HighContrastOp Declarations\rangle} \\
\text{\langle HighContrastOp Method Definitions\rangle}
\]

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 \text{ContrastOp} operator defined above.

The main difficulty with methods that compute local adaptation 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 compute 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 preserve 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

\begin{verbatim}
(HighContrastOp Declarations)

class HighContrastOp : public ToneMap {
    public:
        void Map(Float *xyz, int xRes, int yRes, const DisplayInfo &di) const;
    private:
        ...
}

(HighContrastOp Utility Methods)
\end{verbatim}

The tone mapping function that HighContrastOp uses is based on the threshold versus intensity (TVI) function, which gives the just noticeable 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 scotopic 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 auxiliary capacity function \(C(Y)\) is defined as the integral

\[
C(Y) = \int_0^Y \frac{dY}{TVI(Y)},
\]

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

\[
C(Y) = \begin{cases} 
    Y/0.0014 & Y < 0.0034 \\
    2.4483 + \log(L/0.0034)/0.4027 & 0.0034 \leq Y < 1 \\
    16.563 + (Y - 1)/0.4027 & 1 \leq Y < 7.2444 \\
    32.0693 + \log(Y/7.2444)/0.0556 & \text{otherwise}
\end{cases}
\]
Figure 8.6: high contrast, log widths, local contrast...
\textit{HighContrastOp Utility Methods} ≡
\begin{verbatim}
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;
}
\end{verbatim}

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,
\[
C(Y) - C(Y_{\text{min}})
\]
and we can also compute, of all of the JND steps, what fraction of the way through all of them it is:
\[
\frac{C(Y) - C(Y_{\text{min}})}{C(Y_{\text{max}}) - C(Y_{\text{min}})}
\]
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
\[
T(Y) = Y_{\text{max}}^d \frac{C(Y) - C(Y_{\text{min}})}{C(Y_{\text{max}}) - C(Y_{\text{min}})}
\]
Because we want to map final values to [0, 1], the display luminance value \( Y_{\text{max}}^d \) cancels out, saving us the trouble of determining it in the first place.

\textit{HighContrastOp Utility Methods} + ≡
\begin{verbatim}
static Float T(Float y, Float CYmin, Float CYmax) {
    return (C(y) - CYmin) / (CYmax - CYmin);
}
\end{verbatim}

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 \( Y_{\text{max}} \) and \( Y_{\text{min}} \) 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.

\textit{HighContrastOp Method Definitions} ≡
\begin{verbatim}
void HighContrastOp::Map(Float *xyz, int xRes, int yRes,
    const DisplayInfo &di) const {
    \textit{Find minimum and maximum image luminances}
    \textit{Build luminance image pyramid}
    \textit{Apply high contrast tone mapping operator}
}
\end{verbatim}
Find minimum and maximum image luminances

\[
\text{Float minLum} = \text{LUMINANCE}(0), \text{maxLum} = \text{LUMINANCE}(0);
\]

\[
\text{for (int i = 0; i < xRes \times yRes; ++i)} \{
    \text{minLum} = \min(\text{minLum}, \text{LUMINANCE}(i));
    \text{maxLum} = \max(\text{maxLum}, \text{LUMINANCE}(i));
\}
\]

\[
\text{Float CYmin} = C(\text{minLum}), \text{CYmax} = C(\text{maxLum});
\]

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 adaptation 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:

\[
lc(s, x, y) = \frac{B_s(x,y) - B_{2s}(x,y)}{B_s(x,y)}.
\]

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

\[
\text{Float } *\text{Yadapt} = \text{new Float}[\text{xRes } \times \text{yRes}];
\]

\[
\text{for (int i = 0; i < \text{xRes } \times \text{yRes}; ++i)}
    \text{Yadapt}[i] = \text{LUMINANCE}(i);
\]

\[
\text{MIPMap<Float> pyramid(\text{xRes}, \text{yRes}, \text{Yadapt});}
\]

\[
\text{delete[] Yadapt;}
\]

Now we loop over all of the pixels in the image, compute the adaptation luminance, and can then directly apply the tone reproduction operator.
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 0.25 (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.

```c
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
}
```

```c
If maximum contrast is exceeded, compute adaptation luminance
```
```c
    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;
    }
```
```c
```
Increase search region and prepare to compute contrast again

```c
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

```c
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

```c
#include "tonemap.h"

class NonLinearOp : public ToneMap {
    void Map(Float *xyz, int xRes, int yRes, const DisplayInfo &di) const;
};
```

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

\[
    s(x,y) = \frac{1 + \frac{Y(x,y)}{Y_{\max}}}{1 + Y(x,y)}
\]

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.
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;
    }
}

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.

By default, these are initialized to the appropriate weights for the RGB primaries as specified by the HDTV standard.

8.5 Device RGB Conversion and Output
Convert image to display RGB

```c
Float *rgb = new Float[3*nPix];
```

Define RGB access macros

```c
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);
}
```

```c
delete[] xyz;
xyz = NULL;
```

Define RGB access macros

```c
#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.

Scale and handle out-of-gamut RGB values

```c
for (int i = 0; i < nPix; ++i) {
```

```c
    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 \( \gamma \) is generally 2.2.

DisplayInfo Data

```c
Float gain, invGamma;
```
\[\textit{DisplayInfo Constructor Implementation}\]\
\[
\text{invGamma} = 1.f / 2.2f;
\]

\[\textbf{Apply gamma correction}\]
\[
\text{for (int } i = 0; i < \text{nPix}; ++i) {
\text{R}(i) = \text{powf}(\text{R}(i), \text{dinfo.invGamma});
\text{G}(i) = \text{powf}(\text{G}(i), \text{dinfo.invGamma});
\text{B}(i) = \text{powf}(\text{B}(i), \text{dinfo.invGamma});
}\]

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.

\[\textbf{DisplayInfo Data}\]
\[
\text{bool integerFormat;}
\text{int maxDisplayValue;}
\]

\[\textit{DisplayInfo Constructor Implementation}\]
\[
\text{integerFormat} = \text{true;}
\text{maxDisplayValue} = 255;
\]

\[\textbf{Map image to display range}\]
\[
\text{if (dinfo.integerFormat) }\
\text{for (int } i = 0; i < \text{nPix}; ++i) {
\text{R}(i) *= \text{dinfo.maxDisplayValue;}
\text{G}(i) *= \text{dinfo.maxDisplayValue;}
\text{B}(i) *= \text{dinfo.maxDisplayValue;}
\text{AlphaOut}[i] *= \text{dinfo.maxDisplayValue;}
}\]

Finally, we may \textit{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.

\[\textbf{DisplayInfo Data}\]
\[
\text{Float ditherAmount;}
\]

\[\textit{DisplayInfo Constructor Implementation}\]
\[
\text{ditherAmount} = 0.5f;
\]
Further Reading

Tumblin and Rushmeier first introduced 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 scotopic 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).

Exercises

8.1 exercise
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 appearance 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 realizable, 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.
9.1 Basic Interface

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
collection 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, $\omega_i$, and the outgoing viewing direction, $\omega_o$, 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 $\omega_i$ and $\omega_o$ 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 com-
mon 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.

$\langle BxDF\ Declarators \rangle \equiv$

\[
\text{class } BxDF \{
\text{public:}
\quad \langle BxDF\ Interface\ Declarators \rangle
\};
\]

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.

$\langle BxDF\ Declarators \rangle + \equiv$

\[
\text{class } BRDF : \text{ public } BxDF \{
\};
\]

$\langle BxDF\ Declarators \rangle + \equiv$

\[
\text{class } BTDF : \text{ public } BxDF \{
\};
\]

Both BRDFs and BTDFs will provide an $f$ function that returns the value of the
distribution function for the given pair of directions.
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 $\omega_i$ 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.

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:

$$\rho_{dh} = \frac{1}{\pi} \int_{\Omega} f_r(\omega, \omega') |\cos \theta| d\omega'.$$

We will add a method to the BxDF class to compute this value.

$$\rho_{hh} = \int_{\Omega} \int_{\Omega} f_r(\omega_i, \omega_r) |\cos \theta_i \cos \theta_r| d\omega_i d\omega_r$$
**BRDF Interface Declarations**

```cpp
virtual Spectrum rho() const;
```

**BRDF/BTDF Adapter**

It’s handy to 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, possibly switching the \( \omega_i \) direction to lie in the same hemisphere as \( \omega_o \), as the BRDF expects.

**BRDF/BTDF Method Declarations**

```cpp
class BRDFToBTDF : public BTDF {
public:
    BRDFToBTDF(BRDF *b) { brdf = b; }
    ~BRDFToBTDF() { delete brdf; }
private:
    BRDF *brdf;
};
```

```cpp
static Vector otherHemisphere(const Vector &w) { return Vector(w.x, w.y, -w.z); }
```

```cpp
bool IsSpecular() { return brdf->IsSpecular(); }
Spectrum rho(const Vector &w) const { return brdf->rho(otherHemisphere(w)); }
Spectrum rho() const { return brdf->rho(); }
```

```cpp
Spectrum BRDFToBTDF::f(const Vector &wo, const Vector &wi) const { return brdf->f(wo, otherHemisphere(wi)); }
```

```cpp
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 $\omega_i$ 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 $\theta_t$ to the angle of the incident ray, $\theta_i$. 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, $\eta$, 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:

$$r_{||} = \frac{\eta_t(N \cdot \bar{\omega}_t) + \eta_i(N \cdot \bar{\omega}_i)}{\eta_t(N \cdot \bar{\omega}_t) - \eta_i(N \cdot \bar{\omega}_i)}$$

$$r_{\perp} = \frac{\eta_t(N \cdot \bar{\omega}_t) + \eta_i(N \cdot \bar{\omega}_i)}{\eta_t(N \cdot \bar{\omega}_t) - \eta_i(N \cdot \bar{\omega}_i)}$$

where $r_{||}$ is the Fresnel reflectance for parallel polarized light and $r_{\perp}$ is the reflectance for perpendicular polarized light. $\eta_i$ and $\eta_t$ are the indices of refraction for the incident and transmitted media, and $\bar{\omega}_i$ and $\bar{\omega}_t$ are the incident and transmitted directions, where $\bar{\omega}_t$ was computed with Snell’s law. For light with random polarization (the usual assumption in graphics),

$$r = \frac{1}{2}(r_{||}^2 + r_{\perp}^2).$$
Figure 9.2: Basic setting for specular reflection and transmission. The reflected direction is the direction $\omega_o$ opposite the incident direction $\omega_i$ that makes the same angle $\theta_i$ with the surface normal as the incident ray. The transmitted direction makes an angle $\theta_t$ with the negated surface normal, where $\theta_t$ is given by Snell’s law, which depends on the indices of refraction of the incident and transmitted media, $\eta_i$ and $\eta_t$, respectively.

<table>
<thead>
<tr>
<th>Medium</th>
<th>Index of refraction $\eta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vacuum</td>
<td>1.0</td>
</tr>
<tr>
<td>Air at sea level</td>
<td>1.00029</td>
</tr>
<tr>
<td>Ice</td>
<td>1.31</td>
</tr>
<tr>
<td>Water ($20^\circ$ C)</td>
<td>1.333</td>
</tr>
<tr>
<td>Fused Quartz</td>
<td>1.46</td>
</tr>
<tr>
<td>Glass</td>
<td>1.5 - 1.6</td>
</tr>
<tr>
<td>Sapphire</td>
<td>1.77</td>
</tr>
<tr>
<td>Diamond</td>
<td>2.42</td>
</tr>
</tbody>
</table>

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.
Figure 9.4: Representative measured values of $\eta$ and $k$ for a few conductors (data from Hall.)

<table>
<thead>
<tr>
<th>Object</th>
<th>$\eta$</th>
<th>$k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold</td>
<td>0.37</td>
<td>2.82</td>
</tr>
<tr>
<td>Silver</td>
<td>0.177</td>
<td>3.638</td>
</tr>
<tr>
<td>Copper</td>
<td>0.617</td>
<td>2.63</td>
</tr>
<tr>
<td>Steel</td>
<td>2.485</td>
<td>3.433</td>
</tr>
</tbody>
</table>

Due to conservation of energy, the energy transmitted by a dielectric is $1-F_r$, if $F_r$ is the Fresnel reflectance.

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 $\eta$, the index of refraction of the conductor, and $k$, its absorption coefficient. Values for $\eta$ 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

$$
\begin{align*}
F_r^\parallel &= \frac{(\eta^2 + k^2)(N \cdot \hat{\omega}_i)^2 - 2\eta(N \cdot \hat{\omega}_i) + 1}{(\eta^2 + k^2)(N \cdot \hat{\omega}_i)^2 + 2\eta(N \cdot \hat{\omega}_i) + 1} \\
F_r^\perp &\approx \frac{(\eta^2 + k^2) - 2\eta(N \cdot \hat{\omega}_i) + (N \cdot \hat{\omega}_i)^2}{(\eta^2 + k^2) + 2\eta(N \cdot \hat{\omega}_i) + (N \cdot \hat{\omega}_i)^2}
\end{align*}
$$

For many conductors, values for $\eta$ and/or $k$ aren’t known–less work has gone into measuring these values for real surfaces than has been done for measuring $\eta$ 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 $\eta$ 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 $\eta$, assuming that the absorp-
tion coefficient is equal to zero.

\[
\langle BxDF Utility Functions \rangle \equiv
\]
\[
\text{Spectrum FresnelApproxEta(const Spectrum &intensity) } \{
\text{ return (Spectrum(1.) + intensity.Sqrt()) /}
\text{ (Spectrum(1.) - intensity.Sqrt());}
\}
\]

And the second technique computes an approximate value of $k$ assuming that $\eta = 1$.

\[
\langle BxDF Utility Functions \rangle \equiv
\]
\[
\text{Spectrum FresnelApproxK(const Spectrum &intensity) } \{
\text{ return 2.f * (intensity / (Spectrum(1.) - intensity)).Sqrt();}
\}
\]

For convenience, we will define an abstract Fresnel class that defines an in-
terface 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.

\[
\langle BxDF Declarations \rangle \equiv
\]
\[
\text{class Fresnel { public:}
\text{ \langle Fresnel Interface \rangle}
\text{ };}
\]

\[
\langle Fresnel Interface \rangle \equiv
\]
\[
\text{virtual Spectrum evaluate(Float cosi) const = 0;}
\]

\[
\langle BxDF Declarations \rangle \equiv
\]
\[
\text{class FresnelConductor : public Fresnel { public:
\text{ \langle FresnelConductor Interface \rangle}
\text{ private:
\text{ \langle FresnelConductor Private Data \rangle}
\};
\]

\[
\langle FresnelConductor Interface \rangle \equiv
\]
\[
\text{FresnelConductor(const Spectrum &e, const Spectrum &kk)
\text{ : eta(e), k(kk) } }
\]

\[
\langle FresnelConductor Private Data \rangle \equiv
\]
\[
\text{Spectrum eta, k;}
\]
The evaluation routine for FresnelConductor is simple; it just calls the appropriate utility function.

\[\text{BxDF Method Definitions}\] + \equiv
\[
\text{Spectrum FresnelConductor::evaluate(Float cosi) const } \{
    \text{return FrCond(fabsf(cosi), eta, k);}
}\]

\[\text{BxDF Declarations}\] + \equiv
\[
\text{class FresnelDielectric : public Fresnel } \{ \\
\text{public: } \{ \\
\text{\quad FresnelDielectric Interface } \\
\text{private: } \{ \\
\text{\quad FresnelDielectric Private Data } \\
\}\};
\]

\[\text{FresnelDielectric Interface}\] + \equiv
\[
\text{FresnelDielectric(Float ei, Float et) } \{ \\
  \text{eta_i = ei;} \\
  \text{eta_t = et;}
\}
\]

\[\text{FresnelDielectric Private Data}\] + \equiv
\[
\text{Float eta_i, eta_t;}
\]

\[\text{BxDF Method Definitions}\] + \equiv
\[
\text{Spectrum FresnelDielectric::evaluate(Float cosi) const } \{ \\
\text{\quad Compute Fresnel reflectance for dielectric } \\
\}
\]

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 \(\sin^2 x + \cos^2 x = 1\).

\[\text{Compute Fresnel reflectance for dielectric}\] + \equiv
\[
\text{cosi = Clamp(cosi, -1.f, 1.f); } \\
\text{\quad Compute indices of refraction for dielectric } \\
\text{Float sint = ei/et * sqrtf(max(0.f, 1.f - cosi*cosi));} \\
\text{if (sint > 1.) } \{ \\
\text{\quad Handle total internal reflection } \\
\} \\
\text{else } \{ \\
\text{\quad Float cost = sqrtf(max(0.f, 1.f - sint*sint));} \\
\text{\quad return FrDiel(fabsf(cosi), cost, ei, et);}
\}
\]

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
the variables $e_i$ and $e_t$ such that $e_i$ has the index of refraction of the medium the incident ray is in.

(Compute indices of refraction for dielectric)
bool entering = cosi > 0.;
Float $e_i = \eta_i$, $e_t = \eta_t$;
if (!entering)
    swap($e_i$, $e_t$);

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 $\theta_i$ 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 {
        return 1.;
    }
};

(BxDF Method Definitions) +
Spectrum FresnelNoOp::evaluate(Float cosi) const {
    return 1.;
}
We can now implement the \texttt{SpecularReflection} class, our first specular \texttt{BxDF}. It describes perfect reflection. First, we will derive the \texttt{BRDF} for a specular reflector. If the Fresnel equations say that the fraction of light reflected is $F_r(\overrightarrow{\omega}_o)$, then we need a \texttt{BRDF} such that

$$L_o(\overrightarrow{\omega}_o) = F_r(\overrightarrow{\omega}_o) L_i(\overrightarrow{\omega}_i)$$

where $\overrightarrow{\omega}_i$ is the reflection vector for $\overrightarrow{\omega}_o$ about the surface normal.

Such a \texttt{BRDF} can be constructed using the \textit{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

$$\int f(x) \delta(x - x_0) \, dx = f(x_0) \quad (9.2.1)$$

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 \texttt{BxDF}s as well as some of the light sources to be defined in Chapter 12.

Using the definition of the delta distribiton in conjunction with the scattering equation, 5.4.8, we can find that the \texttt{BRDF} for perfect specular reflection is

$$f_r(\overrightarrow{\omega}_i, \overrightarrow{\omega}_o) = F_r(\overrightarrow{\omega}_o) \frac{\delta(\overrightarrow{\omega}_i - R(\overrightarrow{\omega}_o, N))}{|\cos \theta_i|}$$

if $R(\overrightarrow{\omega}_o, N)$ is the specular reflection vector for $\overrightarrow{\omega}_o$ reflected about the surface normal $N$.

\textit{BxDF Declarations} +\Xi

```cpp
class SpecularReflection : public BRDF {
  public:
    \{
      \textit{(SpecularReflection Methods)}
    \}
  private:
    \{
      \textit{(SpecularReflection Private Data)}
    \};
}
```

The \texttt{SpecularReflection} \texttt{BxDF} takes a \texttt{Fresnel} object to describe dielectric or conductor Fresnel properties and an additional spectrum, which is further used to scale the reflected color.

\textit{BxDF Declarations} +\Xi

```cpp
SpecularReflection(const Spectrum &r, Fresnel *f)
  : R(r), fresnel(f) \{
      \}
```
Figure 9.6: Given an incident direction that makes an angle $\theta$ with the surface normal and an angle $\phi$ with the $x$ axis, the reflected ray about the normal makes an angle $\theta$ 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 $(−1, −1, 1)$.

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.

However, we do implement the $f$\_delta() method, which selects an appropriate direction according to the delta function.

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 $\pi$ radians about $N$—see Figure 9.6.
Recall the transformation matrix from Chapter 2 for a rotation around the \( z \) axis; if the angle of rotation is \( \pi \) radians, it is:

\[
\begin{pmatrix}
-1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

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.

\[
\text{Compute perfect specular reflection direction} \quad \Xi \\
\quad *\mathbf{w}_i = \mathbf{Vector}(-\mathbf{w}_o.x, -\mathbf{w}_o.y, \mathbf{w}_o.z);
\]

**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 \( \eta_i \) and \( \eta_t \) for the incident and transmitted media, respectively—see Figure 9.7. We will denote by \( \tau \) the fraction of incident energy that is transmitted (\( \tau \) 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

\[
(L_t \cos \theta_t \ dA \ d\phi_t) = \tau (L_i \cos \theta_i \ dA \ d\phi_i).
\]

Expanding the solid angles to spherical angles, we have

\[
(L_t \cos \theta_t \ dA \sin \theta_t \ d\theta_t \ d\phi_t) = \tau (L_i \cos \theta_i \ dA \sin \theta_i \ d\theta_i \ d\phi_i). \quad (9.2.2)
\]
We can now differentiate Snell’s law with respect to $\theta$, 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

$$L_t \eta_t^2 d\phi_t = \tau L_i \eta_i^2 d\phi_i.$$  

Because $\phi_t = \phi_i + \pi$, $d\phi_t = d\phi_i$, so

$$L_t = \tau L_i \frac{\eta_t^2}{\eta_i^2}. \quad (9.2.3)$$  

The BTDF for specular transmission is thus

$$f_t(\vec{\omega}_o, \vec{\omega}_t) = \frac{\eta_t^2}{\eta_i^2} \frac{\delta(\vec{\omega}_o - \vec{T}(\vec{\omega}_t))}{|\cos \theta_t|}.$$  

The SpecularTransmission class is almost exactly the same as SpecularReflection except that the sampled direction is the direction for perfect specular transmission.

```cpp
(BxDF Declarations) +\equiv
class SpecularTransmission : public BTDF {
public:
   (SpecularTransmission Methods)
private:
   (SpecularTransmission Private Data)
};

(SpecularTransmission Methods)\equiv
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)\equiv
Spectrum T;
Float etai, etat;
FresnelDielectric fresnel;

(SpecularTransmission Methods) +=\equiv
Spectrum f(const Vector &s, const Vector &) const {
   return Spectrum(0.);
}
```
Figure 9.8: The specularly transmitted direction make an angle $\theta_t$ with the negated surface normal, $-z$. Like specular reflection, the angle it makes with the $x$ axis is $\pi$ greater than the incident ray’s angle.

```cpp
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 $\theta_i$ given by Snell’s law.

```cpp
Spectrum SpecularTransmission::f_delta(const Vector &wo, Vector *wi) const {
    // Figure out which $\eta$ 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 $\omega_o$ direction is greater than zero, the incident ray is coming from outside of the object.

```cpp
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 $\sin^2 \theta$ and $\sin^2 \theta$, 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 $\vec{\omega}_o$, $(\sin \theta_i)^2$ can be computed directly from $(\sin \theta_i)^2$ using Snell’s law.
We then apply the trigonometric identity \( \sin^2 \theta + \cos^2 \theta = 1 \) to compute \( \cos \theta \), from \( \sin \theta \); this directly gives us the \( z \) component of the transmitted direction. To compute the \( x \) and \( y \) components, we first mirror \( \vec{\omega}_i \) about the normal, as we did for specular reflection, but then scale it by the ratio \( \sin \theta_i / \sin \theta_t \) to give it the proper magnitude. From Snell’s law, this ratio is just \( \eta_i / \eta_t \), though, which we happen to have computed previously.

\[
\begin{align*}
\text{Compute transmitted ray direction} \equiv \\
\text{Handle total internal reflection for transmission} \equiv \\
\text{if (sint2 > 1.) return 0.;}
\end{align*}
\]

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.
Sec. 9.4] Microfacet Models 281

class Lambertian : public BRDF {
    public:
        \{Lambertian Methods\}
        Lambertian(const Spectrum &reflectance)
        : R(reflectance), RoverPI(reflectance * INV_PI) \{
            \}
        \{Lambertian Private Data\}
        Spectrum R, RoverPI;

        \{Lambertian Methods\}
        Spectrum rho(const Vector &w) const { return R; }
        Spectrum rho() const { return R; }

    \};

    \{Lambertian Methods\}
    Spectrum Lambertian::f(const Vector &wo, const Vector &wi) const { return RoverPI;
        \}

    \{BxDF Method Definitions\}
    The reflection distribution function for Lambertian is quite straightforward,
    since its value is constant; we just return the overall reflectance divided by \pi. The
    need for the normalization by \frac{1}{\pi} can be understood with the scattering equation.
    If a point with Lambertian reflectance \rho is being uniformly illuminated by a con-
    stant amount of radiance from all directions, \( L_i(\vec{\omega}_o) = c \), then outgoing radiance in
    any direction should be \rho c. Substituting into the scattering equation and simplify-
    ing, we find that \( f_i = \frac{c}{\pi} \).

    \{BxDF Method Definitions\}
    \}

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, dA, 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 $N_f$ 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 $\sigma$ 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:

$$f_r(\vec{\omega}_i, \vec{\omega}_o) = \frac{\rho}{\pi} (A + B \max(0, \cos(\phi_i - \phi_o)) \sin \omega_\pm \tan \omega_\pm)$$

where if $\sigma$ is in radians,

$$A = 1 - \frac{\sigma^2}{2(\sigma^2 + 0.33)}$$

$$B = \frac{0.45\sigma^2}{\sigma^2 + 0.09}$$

$$\omega_+ = \max(\theta_i, \theta_o)$$

$$\omega_- = \min(\theta_i, \theta_o)$$

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.

\[ \text{OrenNayar Methods} \]
\begin{verbatim}
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);
  }
\end{verbatim}

\[ \text{OrenNayar Private Data} \]
\begin{verbatim}
Spectrum R;
Float A, B;
\end{verbatim}

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 \( \sin \theta \).

\begin{align*}
\text{\{BxDF Method Definitions\}} \ &= \ \phi \\
\text{Spectrum OrenNayar::f} & \text{(const Vector \&wo,} \\
& \text{ const Vector \&wi) const} \\
& \hspace{1cm} \text{Float sinhetai = sqrtf(wi.x*wi.x + wi.y*wi.y);} \\
& \hspace{1cm} \text{Float sinthetao = sqrtf(wo.x*wo.x + wo.y*wo.y);} \\
& \hspace{1cm} \text{\{Compute cosine term of Oren–Nayar model\}} \\
& \hspace{1cm} \text{\{Compute sine and tangent terms of Oren–Nayar model\}} \\
& \hspace{1cm} \text{return R * INV_PI * (A + B * maxcos * sinalpha * tanbeta);} \\
& \}\end{align*}

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 \( \phi_i \) and \( \phi_o \). The geometric setting for this is shown in Figure 9.12. In the plane of the point being shaded, the vector \( \mathbf{\omega} \) has coordinates \( (x, y) \), which are given by \( r \cos \phi \) and \( r \sin \phi \), respectively. The radius \( r \) is just \( \sin \theta \), so

\[
\begin{align*}
\cos \phi &= \frac{x}{r} = \frac{x}{\sin \theta} \\
\sin \phi &= \frac{y}{r} = \frac{y}{\sin \theta}.
\end{align*}
\]

\begin{align*}
\text{\{Compute cosine term of Oren–Nayar model\}} \ &= \ \phi \\
\text{Float sinhii = wi.y / sinhetai;} \\
\text{Float cosphii = wi.x / sinhetai;} \\
\text{Float sinhio = wo.y / sinthetao;} \\
\text{Float cosphio = wo.x / sinthetao;} \\
\text{Float dcos = cosphii * cosphio + sinhii * sinhio;} \\
\text{Float maxcos = \max(0.f, dcos);} \\
\end{align*}
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 $\theta$. Given the knowledge of which angle is smaller, we can set $\sin \alpha$ just be computed using the identity $\tan \alpha = \sin \alpha / \cos \alpha$.

\[\text{(Compute sine and tangent terms of Oren–Nayar model)}\]

```plaintext
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 $\theta$ (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,

\[\vec{\omega}_h = \vec{\omega}_i + \vec{\omega}_o\]

cause perfect specular reflection from $\vec{\omega}_i$ to $\vec{\omega}_o$ (and vice-versa). (See Figure 9.13.)
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 $\omega_h$ for directions $\omega_i$ and $\omega_o$, $d^2\Phi_h$. From the definition of radiance, Equation 5.2.4, it is

$$d^2\Phi_h = L_i(\bar{\omega}_i) d\bar{\omega}_h dA \frac{1}{4}(\bar{\omega}_h) = L_i(\bar{\omega}_i) d\bar{\omega}_h \cos \theta_h dA(\bar{\omega}_h),$$

where we have written $dA(\bar{\omega}_h)$ for the area measure of the microfacets with orientation $\bar{\omega}_h$ and $\cos \theta_h$ for the cosine of the angle between $\bar{\omega}_i$ and $\bar{\omega}_h$ (see Figure 9.14.)

The differential area of microfacets with orientation $\bar{\omega}_h$ is just

$$dA(\bar{\omega}_h) = D(\bar{\omega}_h) d\bar{\omega}_h dA.$$

The first two terms describe the differential area of facets per unit area that have the proper orientation, and the $dA$ term converts this to differential area.

Therefore,

$$d^2\Phi_h = L_i(\bar{\omega}_i) d\bar{\omega}_h \cos \theta_h D(\bar{\omega}_h) d\bar{\omega}_h dA(\bar{\omega}_h). \quad (9.4.4)$$

If we assume that the microfacets individually reflect light according to Fresnel’s law, the outgoing flux is

$$d^2\Phi_o = F(\bar{\omega}_i, \bar{\omega}_o)d^2\Phi_h. \quad (9.4.5)$$

Again using the definition of radiance, the reflected outgoing radiance is

$$L(\bar{\omega}_o) = \frac{d^2\Phi_o}{d\bar{\omega}_o \cos \theta_o dA}.$$

If we substitute Equation 9.4.5 into this and then Equation 9.4.4 into the result, we have

$$L(\bar{\omega}_o) = \frac{F(\bar{\omega}_i, \bar{\omega}_o) L_i(\bar{\omega}_i) d\bar{\omega}_h D(\bar{\omega}_h) d\bar{\omega}_h dA \cos \theta_h}{d\bar{\omega}_o dA \cos \theta_o}.$$

In Section 14.3, we will derive an important relation between $d\bar{\omega}_h$ and $d\bar{\omega}_o$; it is

$$d\bar{\omega}_h = \frac{d\bar{\omega}_o}{4 \cos \theta_h}.$$
We can substitute this into the previous equation and simplify, giving

\[ L(\bar{\phi}_o) = \frac{F(\bar{\phi}_i, \bar{\phi}_o)L_i(\bar{\phi}_o)D(\bar{\phi}_h)\,d\bar{\phi}_i}{4 \cos \theta_o}. \]

We can now apply the definition of the BRDF, Equation 5.4.7, giving us the Torrance–Sparrow BRDF:

\[ f_r(\bar{\phi}_i, \bar{\phi}_o) = \frac{D(\bar{\phi}_h)F(\bar{\phi}_i, \bar{\phi}_o)}{4 \cos \theta_i \cos \theta_o}. \]

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 \( \bar{\phi}_i \) and \( \bar{\phi}_o \). This \( G \) term can just be included in the derivation as the Fresnel term was above. The full model, then, is

\[ f_r(\bar{\phi}_i, \bar{\phi}_o) = \frac{D(\bar{\phi}_h)G(\bar{\phi}_i, \bar{\phi}_o)F(\bar{\phi}_i, \bar{\phi}_o)}{4 \cos \theta_i \cos \theta_o}. \]  \( 9.4.6 \)

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 cannot be easily substituted: the relationship between \( d\bar{\phi}_h \) and \( d\bar{\phi}_o \) used in its derivation depends on the specular reflection assumption.

We can now define a general microfacet-based BRDF. It takes a pointer to an abstract \( \text{MicrofacetDistribution} \) class, which provides routines to compute the \( D \) term of the Torrance–Sparrow model. Here is the pure virtual function that \( \text{MicrofacetDistribution} \) must implement; it gives the probability density for microfacets to be oriented with normal \( \bar{\phi}_h \).

\[ \text{BxDF Declarations} \quad + \equiv \]

\[
\text{class MicrofacetDistribution} \{
\text{public:}
\quad \text{\langle MicrofacetDistribution Interface \rangle}
\};
\]

\[ \text{\langle MicrofacetDistribution Interface \rangle} + \equiv \]

\[
\text{virtual Float D(const Vector \&wh) const } = 0;
\]

The Microfacet \( \text{BRDF} \), then, just takes a pointer to a distribution, the reflectance of the object, and a Fresnel function.

\[ \text{BxDF Declarations} + \equiv \]

\[
\text{class Microfacet : public BRDF} \{
\text{public:}
\quad \text{\langle Microfacet Methods \rangle}
\text{private:}
\quad \text{\langle Microfacet Private Data \rangle}
\};
\]
Reflection Models

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 \( \theta_h \) is the same between \( \omega_h \) and both \( \omega_i \) and \( \omega_o \), 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);
}

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
Microfacet Models

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.

The Blinn model is

$$D(\omega_h) = c(\omega_h \cdot N)^e$$

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.
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 computed by finding the radius $e$ of the super-ellipse for the actual orientation angle $\phi$.

**Anisotropic microfacet model**

Ashikhmin and Shirley have developed a microfacet distribution function for modeling the appearance 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 perpendicular to the surface normal. Brushed metals and some types of fabric exhibit anisotropy.

Their model is physically-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 parameters: $e_x$, which gives an exponent for the distribution function for half-angle vectors with an azimuthal angle that orients them exactly along the $x$ axis, and $e_y$, an exponent for microfacets oriented along the $y$ axis. Exponents for intermediate orientations are found by considering these two values as the lengths of the axes 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

$$D(\tilde{\omega}_h) = \sqrt{(e_x + 1) \cdot (e_y + 1)(\tilde{\omega}_h \cdot N)^{e_x \cos^2 \phi + e_y \sin^2 \phi}}.$$
class Anisotropic : public MicrofacetDistribution {
public:
    Anisotropic(Float x, Float y) { ex = x; ey = y; }
private:
    Float ex, ey;
};

The terms of the distribution function can be computed quite efficiently. Recall from the Oren–Nayar BRDF that \( \cos \phi = \frac{x}{\sin \theta} \) and \( \sin \phi = \frac{y}{\sin \theta} \). 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

\[
\begin{align*}
\cos^2 \phi &= \frac{x^2}{1 - z^2} \\
\sin^2 \phi &= \frac{y^2}{1 - z^2}.
\end{align*}
\]

The Lafortune model avoids these shortcomings. 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 reciprocal or energy-conserving—that the Lafortune model avoids.

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

\[
\begin{align*}
\omega_i &\rightarrow \omega_i R(N) \\
\omega_o &\rightarrow \omega_o R(N)
\end{align*}
\]

where \( R(N) \) is the operator that reflects the vector \( \omega \) 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

\[
\begin{align*}
f_r(\vec{\omega}_i, \vec{\omega}_o) &= (\vec{\omega}_i \cdot R(\vec{\omega}_o, N))^e = (\vec{\omega}_o \cdot R(\vec{\omega}_i, N))^e.
\end{align*}
\]

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 re-oriented incident vector is no longer necessarily of unit length and its length 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:

\[
f_r(\vec{o}_i, \vec{o}_o) = \frac{\rho_d}{\pi} + \sum_{i} \rho_i \cdot (\vec{o}_i \cdot (\vec{o}_o \times o_i))^e_i,
\]

where \(\rho_d\) 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 function 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.

\[
\langle BxDF Declarations \rangle + \equiv
\]

```cpp
class Lafortune : public BRDF {
public:
    \langle Lafortune Methods \rangle
definitions
private:
    \langle Lafortune Private Data \rangle
};
```

Our implementation limits the number of separate lobes to the compile-time constant `MAX_LOBES`, in order to avoid additional run-time memory allocation.
\(\text{BxDF Method Definitions}\)\+\(\equiv\)

\[
\text{Lafortune::Lafortune}(\text{const Spectrum } &r, \text{int } nL, \text{const Spectrum } *xx, \text{const Spectrum } *yy, \text{const Spectrum } *zz, \text{const Spectrum } *e) \\
: R(r) \\
\text{nLobes} = nL; \\
\text{Assert}(\text{nLobes} \leq \text{MAX_LOBES}); \\
\text{for } (\text{int } i = 0; i < \text{nLobes}; ++i) \{ \\
\text{x}[i] = xx[i]; \\
\text{y}[i] = yy[i]; \\
\text{z}[i] = zz[i]; \\
\text{exponent}[i] = e[i]; \\
\}
\]

\(\text{Lafortune Private Data}\)\(\equiv\)

\[
\text{Spectrum } R; \\
\text{\#define MAX_LOBES } 3 \\
\text{Spectrum } x[\text{MAX_LOBES}], y[\text{MAX_LOBES}], z[\text{MAX_LOBES}]; \\
\text{Spectrum exponent[\text{MAX_LOBES}];} \\
\text{int } \text{nLobes};
\]

\(\text{BxDF Method Definitions}\)\+\(\equiv\)

\[
\text{Spectrum } \text{Lafortune::f}(\text{const Vector } &wo, \text{const Vector } &wi) \text{ const}\+16 \\
\text{Spectrum } \text{ret} = \text{R} / \text{M_PI}; \\
\text{for } (\text{int } i = 0; i < \text{nLobes}; ++i) \{ \\
\text{\langle Evaluate Lafortune model for ith lobe\rangle} \\
\}
\]

\[
\text{return } \text{ret};
\]

The paper 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 \(\vec{o}_e\) vector by multiplying its \(x, y,\) and \(z\) coefficients with the appropriate spectral orientation coefficients and compute the dot product of the result with \(\vec{o}_i\), giving a spectral result which is itself then raised to the spectral exponent provided.

\[
\text{\langle Evaluate Lafortune model for ith lobe\rangle}\equiv \\
\text{Spectrum } v = x[i] * wo.x * wi.x + y[i] * wo.y * wi.y + \\
\quad z[i] * wo.z * wi.z; \\
\text{ret} += v.\text{Pow}(\text{exponent}[i]);
\]
Figure 9.17: The FresnelBlend BRDF models the effect of a surface with a glossy layer on top of a diffuse substrate. As the angle of incidence of the direction vectors \( \hat{\omega}_i \) and \( \hat{\omega}_o \) 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

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 above 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.

\[
\langle BxDF \textit{ Declarations} \rangle \equiv
\]

```c++
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.
Fresnel Blend

Fresnel Blend::Fresnel Blend(const Spectrum &d, const Spectrum &s, MicrofacetDistribution *dist) : Rd(d), Rs(s) {
    distribution = dist;
}

Fresnel Blend 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

\[ f_r(\vec{\omega}_i, \vec{\omega}_o) = \frac{D(\vec{\omega}_h)}{8\pi(\vec{\omega}_h \cdot \vec{\omega}_i)(\max((N \cdot \vec{\omega}_i), (N \cdot \vec{\omega}_o)))} F(\vec{\omega}_i, \vec{\omega}_o), \]

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:

\[ f_r(\vec{\omega}_i, \vec{\omega}_o) = \frac{28R_d}{23\pi}(1 - R_s) \left( 1 - \left( 1 - \frac{(N \cdot \vec{\omega}_i)}{2} \right)^5 \right) \left( 1 - \left( 1 - \frac{(N \cdot \vec{\omega}_o)}{2} \right)^5 \right) \]

BxDF Method Definitions

Spectrum Fresnel Blend::Schlick Fresnel (Float costheta) const {
    return Rs + powf(1 - costheta, 5.f) * (Spectrum(1.) - Rs);
}

BxDF Method Definitions
Spectrum Fresnel Blend::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))) *
    Schlick Fresnel(Dot(wi, H));
    return diffuse + specular;
}
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 $\theta_o$, $\theta_i$, $d\phi$...

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 BRDFs and BTDFs 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 BRDFs and BTDFs, allowing the rest of the system to work with composite BSDFs 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.

\[ \text{BSDF Declarations} \]

\begin{verbatim}
class BSDF {
public:
    \text{(BSDF Method Declarations)}
private:
    \text{(BSDF Member Variables)}
};
\end{verbatim}

The BSDF constructor takes two pieces of DifferentialGeometry: \( \text{dgS} \), 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 \( \text{dgG} \), 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.

\[ \text{BSDF Method Definitions} \]

\begin{verbatim}
BSDF::BSDF(const DifferentialGeometry &dgS,
            const DifferentialGeometry &dgG) {
    Ng = dgG.Nn;
    \text{(Orient shading normal to match geometric normal)}
}
\end{verbatim}

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);
}

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.
\(\text{\textit{BSDF Inline Methods}}\)+≡

```cpp
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 \text{BSDF}s (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 \text{BSDF} equations in Chapter 9. These transformations are computed in the same way as the \text{DifferentialGeometry} methods 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 \text{BSDF} implementations depend on this. However, we don’t normalize directions in world space, since there’s not a corresponding assumption for world-space rays.

\(\text{\textit{BSDF Method Declarations}}\)+≡

```cpp
Vector WorldToLocal(const Vector &v) const {
    return Vector(Dot(v, Ss), Dot(v, Ts), Dot(v, Ns)).Hat();
}
```

```cpp
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 \textit{light leak}: the geometric normal indicates that \(\mathbf{\omega}_i\) and \(\mathbf{\omega}_o\) lie on opposite 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 \(\mathbf{\omega}_i\). Thus, we can see that \(N_g\) can’t just be used as a direct replacement for \(N_s\) 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 indicates that they are in the same hemisphere. Direct use of \(N_s\) would cause ugly black spots on the surface where this situation happens.

Fortunately, there is an elegant solution to these problems. When evaluating the \text{BSDF}, we use the geometric normal to decide if we should be evaluating reflection or transmission: if \(\mathbf{n}_g\) and \(\mathbf{n}_s\) lie in the same hemisphere with respect to \(N_s\), we evaluate the \text{BRDFs}, and otherwise we evaluate the \text{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 backside 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 BSDFs in lrt should evaluate their values without regard to whether \( \omega_i \) and \( \omega_o \) 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.

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.

```cpp
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 \( f_{\text{delta}} \) 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>\(\equiv\)

```cpp
inline Spectrum BSDF::f_delta(int component, const Vector &w, Vector *wi) const {
    BxDF *spec = NULL;
    Float weight = 0.;
    <Find the component\(\text{th}\) specular component>
    Vector wo = WorldToLocal(w);
    Spectrum f = weight * spec->f_delta(wo, wi);
    *wi = LocalToWorld(*wi);
    return f;
}
```

<Find the component\(\text{th}\) specular component>\(\equiv\)

```cpp
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` is `NULL`, look for specular reflection in BTDFs.

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>\(\equiv\)

```cpp
Spectrum rho() const;
Spectrum rho(const Vector &wo) const;
```

10.2 Material Interface and Bump Mapping

<materials.h>\(\equiv\)

```cpp
#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>
```cpp
#endif // MATERIALS_H
```
There are two main functions that Materials 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.

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.

Filter Regions for Anti-Aliasing

The various Materials 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 worthwhile 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.
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 assumption and do some form of anti-aliasing than 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 to 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 $d = -(N \cdot P)$. 
Next, we intersect the auxiliary rays $r_x$ and $r_y$ with this plane (Figure 10.4). Given their hit positions, we would like to find the amount of variation in position on the surface and variation in parametric $(u,v)$ coordinates between adjacent camera ray samples; these give us the sampling rate in texture parameter space, which individual textures can use to determine their maximum allowed frequency content.

**DifferentialGeometry Data**

```cpp
data class DifferentialGeometry {
    mutable Vector dPdx, dPdy;
    mutable Float dudx, dvdx, dudy, dvdy;
}
```

**Initialize DifferentialGeometry from parameters**

```cpp
    dudx = dux;
    dvdx = dvx;
    dudy = duy;
    dvdy = dvy;
```

**DifferentialGeometry Method Declarations**

```cpp
    void ComputeDifferentials(const RayDifferential &r) const;
```
DifferentialGeometry Method Definitions

void DifferentialGeometry::ComputeDifferentials(const RayDifferential &ray) const {
    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 \( \text{dgGeom}.Nn \).

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 that the 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

\[
(P' - P) = (\partial P/\partial u \partial P/\partial v)(\begin{vmatrix}
    du \\
    dv
\end{vmatrix})
\]

or

\[
\begin{pmatrix}
    P' - P_x \\
    P' - P_y \\
    P' - P_z
\end{pmatrix} = \begin{pmatrix}
    \partial P/\partial u_x & \partial P/\partial v_x \\
    \partial P/\partial u_y & \partial P/\partial v_y \\
    \partial P/\partial u_z & \partial P/\partial v_z
\end{pmatrix}(\begin{vmatrix}
    du \\
    dv
\end{vmatrix})
\]

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.
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 \( \mathbf{Nn} \).

\[
\langle \text{Compute } (u, v) \text{ offsets at auxiliary points} \rangle \equiv 
\langle \text{Initialize } A, Bx, \text{ and } By \text{ matrices for offset computation} \rangle
\]

\[
\text{SolveLinearSystem2x2} (A, Bx, x);
\]

\[
dudx = x[0];
\]

\[
dvdx = x[1];
\]

\[
\text{SolveLinearSystem2x2} (A, By, x);
\]

\[
dudy = x[0];
\]

\[
dvdy = x[1];
\]

\[
\langle \text{Initialize } A, Bx, \text{ and } By \text{ matrices for offset computation} \rangle \equiv 
\]

\[
\text{Float } A[2][2], Bx[2], By[2], x[2];
\]

\[
\text{if } (\text{fabsf}(\mathbf{Nn}.x) > \text{fabsf}(\mathbf{Nn}.y) \&\& \text{fabsf}(\mathbf{Nn}.x) > \text{fabsf}(\mathbf{Nn}.z)) \{
\langle \text{Project onto } yz \text{ plane to initialize matrices} \rangle
\}
\]

\[
\text{else if } (\text{fabsf}(\mathbf{Nn}.y) > \text{fabsf}(\mathbf{Nn}.z)) \{
\langle \text{Project onto } xz \text{ plane to initialize matrices} \rangle
\}
\]

\[
\text{else } \{
\langle \text{Project onto } xy \text{ plane to initialize matrices} \rangle
\}
\]

\[
\langle \text{Project onto } yz \text{ plane to initialize matrices} \rangle \equiv 
\]

\[
\text{#define } C1 y
\]

\[
\text{#define } C2 z
\]

\[
\langle \text{Initialize matrices for chosen projection plane} \rangle
\]

\[
\text{#undef } C1
\]

\[
\text{#undef } C2
\]

\[
\langle \text{Initialize matrices for chosen projection plane} \rangle \equiv 
\]

\[
A[0][0] = \text{dPdu}.C1;
\]

\[
A[0][1] = \text{dPdu}.C2;
\]

\[
A[1][0] = \text{dPdv}.C1;
\]

\[
A[1][1] = \text{dPdv}.C2;
\]

\[
Bx[0] = \text{Px}.C1 - P.C1;
\]

\[
Bx[1] = \text{Px}.C2 - P.C2;
\]

\[
By[0] = \text{Py}.C1 - P.C1;
\]

\[
By[1] = \text{Py}.C2 - P.C2;
\]

\[
\text{Bump mapping}
\]

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
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.

\[
\text{Material Interface} + \equiv \\
\text{Material}(\text{Texture<Float> } \ast \text{disp}) \{ \\
\quad \text{displace} = \text{disp}; \\
\}
\]

\[
\text{Material Private Data} \equiv \\
\text{Texture<Float> } \ast \text{displace};
\]

The second important Material method, \text{bump}, is responsible for computing the effect of bump mapping at the point being shaded. Two instances of \text{DifferentialGeometry} are stored in \text{Surfs}; the first, \text{dgGeom}, represents the geometric differential geometry at the hit point—the true geometry of the intersection. The second, \text{dgShading}, represents the shading geometry; by default, it is the same as \text{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 \( \overline{v_u} = x'_u - x \) and \( \overline{v_v} = x'_v - x \) 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 \( x + \partial P / \partial u \ast du \) 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.

```cpp
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 displacement and compute bumped values
    }
}

Will expect these guys to be in world space already...

Evaluate texture for shading normal and tangent

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

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.

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 \( \partial P/\partial v \).

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

```cpp
#include "lrt.h"
#include "materials.h"

Matte Class Declarations

class Matte : public Material {
public:
  Matte(Texture<Spectrum> *kd, Texture<Float> *sig,
      Texture<Float> *disp)
    : Material(disp) {
    Kd = kd;
    sigma = sig;
  }

Matte Interface

Matte::Kd Texture<Spectrum> *kd, Texture<Float> *sig,
    Texture<Float> *disp)
  : Material(disp) {
    Kd = kd;
    sigma = sig;
  }

Matte Private Data

Texture<Spectrum> *Kd;
Texture<Float> *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 eval-
uated 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.
10.4 Plastic

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, $K_d$ and $K_s$, 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

```cpp
Plastic(Texture<Spectrum> *kd, Texture<Spectrum> *ks,
        Texture<Float> *rough, Texture<Float> *disp)
    : Material(disp) {
    Kd = kd;
    Ks = ks;
    roughness = rough;
}
```

Plastic Private Data

```cpp
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 Class Declarations

class Translucent : public Material {
    public:
        Translucent Interface
    private:
        Translucent Private Data
};
(Translucent Method Definitions)\(\equiv\)

```cpp
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;
}
```

10.6 Glass

(glass.cc*)\(\equiv\)

(Source Code Copyright)

#include "lrt.h"
#include "materials.h"

(Glass Class Declarations)

(Glass Method Definitions)

(Glass Class Declarations)\(\equiv\)

```cpp
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.

\[ \text{Glass Interface} \]
\[
\text{Glass(} \text{Texture}<\text{Spectrum}> *r, \text{Texture}<\text{Spectrum}> *t, \\
\text{Texture}<\text{Float}> *i, \text{Texture}<\text{Float}> *\text{disp}) \\
: \text{Material}(\text{disp}) \{
\text{Kr} = r;
\text{Kt} = t;
\text{index} = i;
\}
\]

\[ \text{Glass Private Data} \]
\[
\text{Texture}<\text{Spectrum}> *\text{Kr}, *\text{Kt}; \\
\text{Texture}<\text{Float}> *\text{index};
\]

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.

\[ \text{Glass Method Definitions} \]
\[
\text{BSDF } *\text{Glass::GetBSDF(const Surf } *\text{surf) const } \{
\text{Spectrum } R = \text{Kr->Evaluate(} \text{surf->dgShading});
\text{Spectrum } T = \text{Kt->Evaluate(} \text{surf->dgShading});
\text{Float } ior = \text{index->Evaluate(} \text{surf->dgShading});
\text{BSDF } *\text{ret } = \text{new BSDF(} \text{surf->dgShading, surf->dgGeom});
\text{if } (!\text{R.Black()})
\text{ret->Add(new SpecularReflection}(R, \\
\text{new FresnelDielectric}(1., ior)));
\text{if } (!\text{T.Black()})
\text{ret->Add(new SpecularTransmission}(T, 1., ior));
\text{return ret;}
\}
\]

10.7 Shiny Metal

\[ \text{shinymetal.cc}\]
\[
\langle \text{Source Code Copyright} \rangle
#include "lrt.h"
#include "materials.h"
\langle \text{ShinyMetal Class Declarations} \rangle
\langle \text{ShinyMetal Method Definitions} \rangle
\]

\[ \text{ShinyMetal Class Declarations} \]
\[
\text{class ShinyMetal : public Material } \{
\text{public:}
\langle \text{ShinyMetal Interface} \rangle
\text{private:}
\langle \text{ShinyMetal Private Data} \rangle
\};
\]
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 $K_s$, and perfect mirror specular reflection, with reflectance $K_r$.

### ShinyMetal Interface

```cpp
ShinyMetal(Texture<Spectrum> *ks, Texture<Float> *rough,
            Texture<Spectrum> *kr, Texture<Float> *disp)
  : Material(disp) {
  Ks = ks;
  roughness = rough;
  Kr = kr;
}
```

### ShinyMetal Private Data

```cpp
Texture<Spectrum> *Ks, *Kr;
Texture<Float> *roughness;
```

### ShinyMetal Method Definitions

```cpp
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);
  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;
}
```

## 10.8 Diffuse Substrate

XXX need a better name

```cpp
#include "lrt.h"
#include "materials.h"
```

```cpp
class Substrate : public Material {
public:
  Substrate Interface
private:
  Substrate Private Data
};
```
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 interacted 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.

\begin{align*}
\text{Substrate Interface} & \equiv \\
& \text{Substrate}(\text{Texture}<\text{Spectrum}> \ast \text{Kd}, \text{Texture}<\text{Spectrum}> \ast \text{Ks}, \\
& \quad \text{Texture}<\text{Float}> \ast \text{nu}, \text{Texture}<\text{Float}> \ast \text{nv}, \text{Texture}<\text{Float}> \ast \text{disp}) \\
& \quad : \text{Material}(\text{disp}) \\
& \quad \text{Kd} = \text{kd}; \\
& \quad \text{Ks} = \text{ks}; \\
& \quad \text{nu} = \text{u}; \\
& \quad \text{nv} = \text{v};
\end{align*}

\begin{align*}
\text{Substrate Private Data} & \equiv \\
& \text{Texture}<\text{Spectrum}> \ast \text{Kd}, \ast \text{Ks}; \\
& \text{Texture}<\text{Float}> \ast \text{nu}, \ast \text{nv};
\end{align*}

\begin{align*}
\text{Substrate Method Definitions} & \equiv \\
& \text{BSDF} * \text{Substrate::GetBSDF}(\text{const Surf} * \text{surf}) \text{ const} \\
& \quad \{ \\
& \quad \quad \text{Spectrum} \ d = \text{Kd->Evaluate}(<\text{surf->dgShading}>); \\
& \quad \quad \text{Spectrum} \ s = \text{Ks->Evaluate}(<\text{surf->dgShading}>); \\
& \quad \quad \text{Float} \ u = \text{nu->Evaluate}(<\text{surf->dgShading}>); \\
& \quad \quad \text{Float} \ v = \text{nv->Evaluate}(<\text{surf->dgShading}>); \\
& \quad \quad \text{BSDF} * \text{ret} = \text{new BSDF}(\text{surf->dgShading}, \text{surf->dgGeom}); \\
& \quad \quad \text{ret}->\text{Add}(\text{new FresnelBlend}(\text{d, s, new Anisotropic}((1.f/u, 1.f/v)))); \\
& \quad \quad \text{return} \ \text{ret}; \\
& \quad \}
\end{align*}
10.9 Measured Data

\(\text{clay.cc}^*\) ≡

(Source Code Copyright)
#include "irt.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;
}

(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] = { 0.679314f, 0.676108f, 0.726031f };
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 infinitesimal 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 appearance of smooth surfaces from polygonal meshes (PC75). Blinn later developed the bump-mapping technique to give the appearance 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...
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 the 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
that will be used in \texttt{lrt} throughout the rest of this chapter for implementing various
anti-aliasing approaches.

\begin{verbatim}
#pragma once

#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

// Texture Forward Declarations

// Texture Cache Data

// Texture Cache Methods

// Perlin Noise Data

// Texture Method Definitions
\end{verbatim}
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.

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.

\begin{verbatim}
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 them-
  selves will use the differential values dPdx, dPdy, dudx, dvdx, dudy, and dtdy to
do so...

  \{Texture Interface\}{
    virtual T Evaluate(const DifferentialGeometry &) const = 0;
  }
\end{verbatim}
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 ConstantTextures 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.

\[\text{Texture Class Declarations}\] +\equiv

```cpp
template <class T>
class ConstantTexture: public Texture<T> {
public:
    ConstantTexture(const T &v) { value = v; }
    T Evaluate(const DifferentialGeometry &d) const;
private:
    T value;
};
```

\[\text{Texture Template Method Definitions}\] +\equiv

```cpp
template <class T>
T ConstantTexture<T>::Evaluate(const DifferentialGeometry &dg) const {
    return value;
}
```

Scale

One of the most useful things that can be done with the Textures in this chapter is to compose them together, feeding the output of one texture into the input of another. The ScaleTexture 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.

\[\text{Texture Class Declarations}\] +\equiv

```cpp
template <class T1, class T2>
class ScaleTexture: public Texture<T2> {
public:
    ScaleTexture(Texture<T1> *s, Texture<T2> *v) {
        scale = s;
        value = v;
    }

    \[\text{ScaleTexture Methods}\]
private:
    Texture<T1> *scale;
    Texture<T2> *value;
};
```

\[\text{Texture Template Method Definitions}\] +\equiv

```cpp
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 ScaleTextures 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.

\[\textit{ScaleTexture Methods}\] +≡

\texttt{\textbar ScaleTexture(\textbar) \{}
\texttt{\hspace{1em} delete scale;}
\texttt{\hspace{1em} delete value;}
\texttt{\}}

\textbf{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.

\[\textit{Texture Class Declarations}\] +≡

\texttt{template <class T>}
\texttt{class MixTexture : public Texture\langle T\rangle\ {}
\texttt{public:
\hspace{1em} MixTexture(Texture\langle T\rangle *t1, Texture\langle T\rangle *t2,
\hspace{1.5em} Texture\langle Float\rangle *amt) \{
\hspace{2em} tex1 = t1;
\hspace{2em} tex2 = t2;
\hspace{2em} amount = amt;
\hspace{1em}\}}
\texttt{\langle MixTexture Interface \rangle}
\texttt{private:
\hspace{1em} Texture\langle T\rangle *tex1, *tex2;
\hspace{1em} Texture\langle Float\rangle *amount;
\hspace{1em}\};}

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 \textit{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 \textit{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.

\[\textit{Texture Template Method Definitions}\] +≡

\texttt{template <class T>
\hspace{1em} T MixTexture\langle T\rangle::Evaluate(const DifferentialGeometry &dg) const \{
\hspace{2em} T t1 = tex1->Evaluate(dg), t2 = tex2->Evaluate(dg);
\hspace{2em} Float amt = amount->Evaluate(dg);
\hspace{2em} return (1. - amt) * t1 + amt * t2;
\hspace{1em}\}}
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 introduce 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.

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 dtty.

```
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;
};
```
IdentityMapping2D::IdentityMapping2D(Float _su, Float _sv,
   Float _du, Float _dv) {
   su = _su; sv = _sv;
   du = _du; dv = _dv;
}

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.

The SphericalMapping2D object stores a transformation that is applied to points before this mapping is performed; this effectively allows the sphere to be positioned and oriented with respect to the object.

XXX compute the differentials by just applying the mapping to all three points and taking the differences...
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;
}

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);
}

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.

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;
};
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;
}

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;
}

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 paramet-
erization 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 $\vec{v}_u$ and $\vec{v}_v$ 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
$\vec{v}_u$ and $\vec{v}_v$ and then adding the offset. For the example in the previous paragraph,
we’d have $\vec{v}_u = (1,0,0)$, $\vec{v}_v = (0,1,0)$, and $du = dv = 0$.

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;
};
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

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.
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.

```
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.

\[ \text{Texture Class Declarations} \]

```cpp
template <class T>
class VertexTexture : public Texture<T> {
public:
    \{ \text{VertexTexture Interface} \}
    T Evaluate(const DifferentialGeometry &) const;

private:
    \{ \text{VertexTexture Private Data} \}
};
```

\[ \text{Texture Template Method Definitions} \]

```cpp
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];
}
```

\[ \text{VertexTexture Private Data} \]

```cpp
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.

\[ \text{Texture Template Method Definitions} \]

```cpp
template <class T>
T VertexTexture<T>::Evaluate(const DifferentialGeometry &dg) const {
    \{ \text{Find three vertex texture values, v0, v1, and v2} \}
    \{ \text{Compute barycentric coordinates for point} \}
    return b[0] * v0 + b[1] * v1 + b[2] * v2;
}
```

\[ \text{Find three vertex texture values, v0, v1, and v2} \]

```cpp
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.

\[
\begin{align*}
    u &= b_0 u_0 + b_1 u_1 + b_2 u_2 \\
    v &= b_0 v_0 + b_1 v_1 + b_2 v_2
\end{align*}
\]
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 substitute 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];
```

```
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
```

### 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 `ImageTextures`, in a few pages. The caller provides the filename of a TIFF texture, and we read it into an array of `Spectrum` objects. 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.

`ImageTextures` implement textures from 2D bitmaps stored on disk. Careful filtering of the bitmap values is essential for anti-aliasing...
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...

\[\text{Texture Method Definitions}\] +É

```cpp
void ImageMap::Lookup(Float s, Float t, Float dsdx, Float dtdx, 
                      Float dsdy, Float dtty, Spectrum *val) const {
    if (!mipmap)
        *val = Spectrum(1.f);
    else
        *val = mipmap->Lookup(s, t, dsdx, dtdx, dsdy, dtty);
}
```

\[\text{Texture Method Definitions}\] +É

```cpp
void ImageMap::Lookup(Float s, Float t, Float dsdx, Float dtdx, 
                      Float dsdy, Float dtty, Float *val) const {
    Spectrum sp;
    Lookup(s, t, dsdx, dtdx, dsdy, dtty, &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...

\[\text{mipmap.h*}\] ≡

\[\text{Source Code Copyright}\]

```cpp
#ifndef MIPMAP_H
#define MIPMAP_H 1
#include "lrt.h"

\text{MIPMap Declarations}
\text{MIPMap Method Definitions}
```

# endif // MIPMAP_H
\[ \text{MIPMap Declarations} \equiv \]
\[
\quad \text{template <class T> class MIPMap} \{
\quad \quad \text{public:}
\quad \quad \text{\{MIPMap Public Interface\}}
\quad \quad \text{private:}
\quad \quad \text{\{MIPMap Private Methods\}}
\quad \quad \text{\{MIPMap Private Data\}}
\quad \};
\]

\[ \text{MIPMap Method Definitions} \equiv \]
\[
\quad \#define BLOCK_SIZE 4
\quad \#define LOG_BLOCK_SIZE 2
\quad \#define UP(x) (((x)+BLOCK_SIZE-1) & (!BLOCK_SIZE-1))
\]

\[ \text{MIPMap Method Definitions} + \equiv \]
\[
\quad \text{template <class T>}
\quad \text{MIPMap\<T\>::MIPMap(int ur, int vr, T \*d) \{}\]
\[
\quad \quad \text{nLevels = Float2Int(log(max(ur, vr)) / log(2.));}
\quad \quad \text{uRes = new int[nLevels];}
\quad \quad \text{vRes = new int[nLevels];}
\quad \quad \text{data = new T *[nLevels];}
\quad \quad \text{Initialize most detailed level of MIPMap}
\quad \quad \text{for (int i = 1; i < nLevels; ++i) \{}\]
\[
\quad \quad \quad \text{Initialize ith MIPMap level from i \text{-} 1st level}\}
\quad \quad \text{Initialize MIPMap filter weights if needed}\}
\]

\[ \text{MIPMap Private Data} \equiv \]
\[
\quad \quad T **data;
\quad \quad int nLevels;
\quad \quad int *uRes, *vRes;
\]

\[ \text{Initialize most detailed level of MIPMap} \equiv \]
\[
\quad \quad \text{uRes[0] = ur;}
\quad \quad \text{vRes[0] = vr;}
\quad \quad \text{data[0] = (T *)AllocL2CacheAligned(UP(ur) * UP(vr) * sizeof(T));}
\quad \quad \text{for (int v = 0; v < vr; ++v)\}
\quad \quad \quad \text{for (int u = 0; u < ur; ++u)\}
\quad \quad \quad \quad \text{texel(0, u, v) = d[u + v * ur];}
\]

\[ \text{Initialize ith MIPMap level from i \text{-} 1st level} \equiv \]
\[
\quad \quad \text{uRes[i] = max(1, uRes[i-1]/2);}
\quad \quad \text{vRes[i] = max(1, vRes[i-1]/2);}
\quad \quad \text{data[i] = (T *)AllocL2CacheAligned(UP(uRes[i]) * UP(vRes[i]) * sizeof(T));}
\]

\[ \text{Filter four texels from finer level of pyramid} \]
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

\[
\text{template } \text{<class } T \text{>}
\]
\[
\text{const } T & \text{MIPMap<T>::texel(int level, int u, int v) const } \{
\text{Return } (u,v) \text{ texel from level}
\}
\]

Return \((u,v)\) texel from level

\[
u = \text{Clamp}(u, 0, uRes[level]-1);
\]
\[
v = \text{Clamp}(v, 0, vRes[level]-1);
\]
\[
\text{int bu = } (u >> \text{LOG_BLOCK_SIZE}), \text{ bv = } (v >> \text{LOG_BLOCK_SIZE});
\]
\[
\text{int ou = } u \& (\text{BLOCK_SIZE}-1), \text{ ov = } v \& (\text{BLOCK_SIZE}-1);
\]
\[
\text{int offset = BLOCK_SIZE } \times \text{ BLOCK_SIZE } \times
\]
\[
\left( (\UP(uRes[level]) >> \text{LOG_BLOCK_SIZE}) \times \text{bv + bu};
\right)
\]
\[
\text{offset } = \text{ BLOCK_SIZE } \ast \text{ ov + ou};
\]
\[
\text{return data[level][offset];}
\]

elliptically weighted average([Hec86]).

MIPMap Private Data

\[
\text{#define WEIGHT_LUT_SIZE 256}
\]
\[
\text{static Float weightLut[WEIGHT_LUT_SIZE];}
\]
\[
\text{static bool weightsInitialized;}
\]

MIPMap Method Definitions

\[
\text{template } \text{<class } T \text{> Float MIPMap<T>::weightLut[WEIGHT_LUT_SIZE];}
\]
\[
\text{template } \text{<class } T \text{> bool MIPMap<T>::weightsInitialized = false;}
\]

Initialize MIPMap filter weights if needed

\[
\text{if (!weightsInitialized) } \{
\text{for } (\text{int } i = 0; i < \text{WEIGHT_LUT_SIZE}; ++i) \{
\text{Float alpha = 2;}
\text{Float r2 = float(i) } / \text{ float(WEIGHT_LUT_SIZE - 1);}
\text{weightLut[i] } = \text{expf(-alpha } \ast \text{ r2);} \}
\text{weightsInitialized } = \text{true;}
\}
\]
template <class T>
MIPMap<T>::MIPMap() {
    for (int i = 0; i < nLevels; ++i)
        FreeCacheAligned(data[i]);
    delete[] data;
    delete[] uRes;
    delete[] vRes;
}

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 (width,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.
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);
  }

(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);

Figure 11.5: Those 0.5 texel offsets...
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 10 = Floor2Int(level);
        Float delta = level - 10;
        return (1.f-delta) * bilerp(10, u, v) +
            delta * bilerp(10+1, u, v);
    }
}

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
    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]);

    // Clamp ellipse eccentricity if too large
    if (lod >= nLevels)
        return texel(nLevels-1, 0, 0);
    else
        return ewaLod(u, v, du0, dv0, du1, dv1, lod);
}
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...

\textit{Clamp ellipse eccentricity if too large}\
\begin{verbatim}
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;
}
\end{verbatim}

Pick a lod such that we’re looking at somewhere around 3-9 texels in the minor
axis direction.

\textit{Choose level of detail for EWA lookup}\
\begin{verbatim}
int lod = max(0, nLevels - 1 - Log2Int(5.f * invMinorLength));
\end{verbatim}

\textit{MIPMap Method Definitions} + \equiv
\begin{verbatim}
template <class T>
T MIPMap<T>::ewaLod(Float u, Float v, Float du0, Float dv0,
    Float du1, Float dv1, int level) const {
    \textit{Convert EWA coordinates to appropriate scale for level}
    \textit{Compute ellipse coefficients to bound EWA filter region}
    \textit{Compute the ellipse’s (s,t) bounding box in texture space}
    \textit{Scan over ellipse bound and compute quadratic equation}
}
\end{verbatim}

\textit{Convert EWA coordinates to appropriate scale for level}\
\begin{verbatim}
u = u * uRes[level]; // - 0.5f;
v = v * vRes[level]; // - 0.5f;
du0 *= uRes[level];
dv0 *= vRes[level];
du1 *= uRes[level];
dv1 *= vRes[level];
\end{verbatim}

\textit{Compute ellipse coefficients to bound the region: } A*x*x + B*x*y + C*y*y = F.

\textit{Compute ellipse coefficients to bound EWA filter region}\
\begin{verbatim}
    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;
\end{verbatim}
\textit{Compute the ellipse's \((s,t)\) bounding box in texture space}\: 

\begin{align*}
\text{Float } \det &= -B^2 + 4.f^*A^*C; \\
\text{Float } \text{invDet} &= 1.f / \det; \\
\text{Float } u\text{Sqrt} &= \sqrt{\text{det} * C}, \quad v\text{Sqrt} = \sqrt{A * \text{det}}; \\
\text{int } u0 &= \text{Ceil2Int} (u - 2.f * \text{invDet} * u\text{Sqrt}); \\
\text{int } u1 &= \text{Floor2Int} (u + 2.f * \text{invDet} * u\text{Sqrt}); \\
\text{int } v0 &= \text{Ceil2Int} (v - 2.f * \text{invDet} * v\text{Sqrt}); \\
\text{int } v1 &= \text{Floor2Int} (v + 2.f * \text{invDet} * v\text{Sqrt}); \\
\text{static StatsRatio } \text{ewaTexels} ("\text{Texture}", "\text{Texels per EWA lookup}", \text{false}); \\
\text{ewaTexels}.\text{add} ((1+u1-u0) * (1+v1-v0), 1);
\end{align*}

\textit{Scan over ellipse bound and compute quadratic equation}\: 

\begin{align*}
\text{T num}(0.); \\
\text{Float } \text{den} &= 0; \\
\text{for } (\text{int } iv = v0; \ iv <= v1; ++iv) \{ \\
\text{Float } V &= iv - v; \\
\text{for } (\text{int } iu = u0; \ iu <= u1; ++iu) \{ \\
\text{Float } U &= iu - u; \\
\text{Float } r2 &= A^*U^*U + B^*U^*V + C^*V^*V; \\
\text{if } (r2 < 1.) \{ \\
\text{Add EWA sample for current texel} \\
\}
\}
\}
\text{return num / den;}
\end{align*}

\textit{Add EWA sample for current texel}\: 

\begin{align*}
\text{Float } \text{weight} &= \text{weightLut[Float2Int(r2 * (WEIGHT_LUT_SIZE - 1))]}; \\
\text{num} &= \text{num + texel(level, iu, iv) * weight; } \\
\text{den} &= \text{den + weight;}
\end{align*}

\textbf{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.

\textit{Texture Cache Data}\: 

\begin{align*}
\text{static StringHashTable } \text{textures};
\end{align*}

\textit{Texture Cache Methods}\: 

\begin{align*}
\text{ImageMap *GetTexture(const string &filename) \{} \\
\quad \text{ImageMap *ret} = (\text{ImageMap *})\text{textures.\text{Search(filename)};} \\
\quad \text{if } (!\text{ret}) \{ \\
\text{static StatsCounter } \text{texLoaded} ("\text{Texture}", \\
\quad "\text{Number of image maps loaded}"); \\
\text{++texLoaded;}
\quad \text{ret} = \text{new ImageMap(filename);} \\
\quad \text{textures.\text{Add(filename, ret);} \\
\quad \}
\quad \text{return ret;}
\}
Image Texture Maps

We now provide the Texture subclass that uses a ImageMap for image mapping.

\[\text{Texture Class Declarations}\] +≡

\[
\text{template <class T>}
\text{class ImageTexture : public Texture<T> \{}
\text{public:
\hspace{1em} ImageTexture Interface
\text{private:
\hspace{1em} ImageTexture Private Data
\};}
\]

\[\text{Texture Template Method Definitions}\] +≡

\[
\text{template <class T>
ImageTexture<T>::ImageTexture(TextureMapping2D *m,
const string &filename) \{
\hspace{1em} mapping = m;
\hspace{1em} imageMap = GetTexture(filename);
\}}
\]

\[\text{ImageTexture Private Data}\]

\[\text{ImageMap *imageMap;}
\text{TextureMapping2D *mapping;}
\]

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++.

\[\text{Texture Template Method Definitions}\] +≡

\[
\text{template <class T>
T ImageTexture<T>::Evaluate(const DifferentialGeometry &dg) const \{}
\hspace{1em} Float s, t, dsdx, dtdx, dsdy, dtdy;
\hspace{1em} mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
\hspace{1em} T val;
\hspace{1em} imageMap->Lookup(s, t, dsdx, dtdx, dsdy, dtdy, &val);
\hspace{1em} return val;
\}
\]

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.

```cpp
Texture Class Declarations

class TextureMapping3D {
public:
    virtual ~TextureMapping3D() {}  // Texture Mapping 3D (Ch. 11)
    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.

```cpp
Texture Class Declarations

class IdentityMapping3D : public TextureMapping3D {
public:
    IdentityMapping3D(const Transform &x) : xform(x) {}  // Identity Mapping
    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 artist, 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**

```cpp
class UVTexture : public Texture<Spectrum> {  
public:
    UVTexture(TextureMapping2D *m) {  
        mapping = m;
    }

private:
    TextureMapping2D *mapping;
};
```

**Texture Method Definitions**

```cpp
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**

```cpp
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

\[
\begin{align*}
\text{T UVCheckerboard<T>::Evaluate(} & \text{const DifferentialGeometry &dg)} \text{ const } \\
& \text{Float } u, v, dsdx, dtdx, dsdy, dtdy; \\
& \text{mapping->Map(dg, &u, &v, &dsdx, &dtdx, &dsdy, &dtdy);} \\
& \text{if } ((\text{Round(u)} + \text{Round(v)}) \% 2 == 0) \\
& \text{return tex1->Evaluate(dg);} \\
& \text{return tex2->Evaluate(dg);} \\
\end{align*}
\]

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.

Solid Checkerboard Interface

SolidCheckerboard(TextureMapping3D *m, Texture<T> *c1,
                   Texture<T> *c2) {
  mapping = m;
  tex1 = c1;
  tex2 = c2;
}
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 lake, but we don’t want them to be the same (as they would be if they were constructed from a sine wave, 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.

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 \((x,y,z)\) lattice coordinate, then, we look up a value in the permutation table as:

\[
\text{NoisePerm}[\text{NoisePerm}[\text{NoisePerm}[ix]+iy]+iz];
\]

where \(ix = x \% \text{NOISE\_PERM\_SIZE}\), and so forth. By doing three permutations in this way, we avoid regularity that might be present if we used \(\text{NoisePerm}[ix+iy+iz]\), 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 \([(\text{NoisePerm}[ix]+iy)\]

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.

```cpp
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.

```cpp
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

```cpp
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.

Perlin Noise Data

```cpp
#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

```cpp
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

```cpp
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.

\[\text{Texture Method Definitions}\]

\[
\begin{align*}
\text{inline Float NoiseWeight(Float t) } & \{ \\
\text{Float } t3 &= t*t*t; \\
\text{Float } t4 &= t3*t; \\
\text{return } 6*t4*t - 15*t4 + 10&t3; \\
\}
\end{align*}
\]

\[\text{Compute trilinear interpolation of weights}\]

\[
\begin{align*}
\text{Float } wx &= \text{NoiseWeight}(dx); \\
\text{Float } wy &= \text{NoiseWeight}(dy); \\
\text{Float } wz &= \text{NoiseWeight}(dz); \\
\text{Float } x00 &= \text{Lerp}(wx, w000, w100); \\
\text{Float } x10 &= \text{Lerp}(wx, w010, w110); \\
\text{Float } x01 &= \text{Lerp}(wx, w001, w101); \\
\text{Float } x11 &= \text{Lerp}(wx, w011, w111); \\
\text{Float } y0 &= \text{Lerp}(wy, x00, x10); \\
\text{Float } y1 &= \text{Lerp}(wy, x01, x11); \\
\text{return } \text{Lerp}(wz, y0, y1);
\end{align*}
\]

**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.

\text{PolkaDots} takes the usual 2D mapping function, as well as two \text{Textures}, one for the regions of the surface outside of the dots and one for the regions inside.

\[\text{PolkaDots Interface}\]

\[
\begin{align*}
\text{PolkaDots(TextureMapping2D } &\text{m, Texture<T> } *c1, \text{ Texture<T> } *c2) \{ \\
\text{mapping } &= \text{m}; \\
\text{outsideDot } &= \text{c1}; \\
\text{insideDot } &= \text{c2}; \\
\}
\end{align*}
\]

\[\text{PolkaDots Private Data}\]

\[
\begin{align*}
\text{Texture<T> } *\text{outsideDot}, *\text{insideDot}; \\
\text{TextureMapping2D } *\text{mapping};
\end{align*}
\]

The evaluation function is pretty straightforward. We start by taking the \((s,t)\) texture coordinates and computing integer \(u\text{Cell}\) and \(v\text{Cell}\) values, which give us the coordinates of the cell that we’re in. (See Figure 11.8.)
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);
}

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—\( u_{Cell}+.5, v_{Cell}+.5 \). 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 \( u_{Cell}, v_{Cell} \). Although the 3D noise function would actually be evaluating noise at \( u_{Cell}, v_{Cell}, .5 \), 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.
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**

```c
FBmTexture(int oct, Float roughness, TextureMapping3D *map) {
    omega = roughness;
    octaves = oct;
    mapping = map;
}
```
FBmTexture Private Data

```cpp
int octaves;
float omega;
TextureMapping3D *mapping;
```

Texture Method Definitions

```cpp
float FBm(const Point &P, float omega, int octaves) {
    float sum = 0.0, lambda = 1.0, o = 1.0;
    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

```cpp
float FBmTexture::Evaluate(const DifferentialGeometry &dg) const {
    Point dPdx, dPdy;
    Point P = mapping->Map(dg, &dPdx, &dPdy);
    return FBm(P, omega, octaves);
}
```

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

```cpp
Windy(TextureMapping3D *map) {
    mapping = mapping;
}
```

Windy Private Data

```cpp
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

\[
\text{Float Windy::Evaluate(const DifferentialGeometry &dg) const \{}
\begin{align*}
\text{Point } & \text{ dPdx, dPdy; } \\
\text{Point } & \text{ P = mapping->Map(dg, \&dPdx, \&dPdy); } \\
\text{Point } & \text{ Ps = Point(.1f*P.x, .1f*P.y, .1f*P.z); } \\
\text{Float windStrength = FBm(Ps, .5f, 3); } \\
\text{Float waveHeight = FBm(P, .5f, 6); } \\
\text{return fabsf(windStrength) * waveHeight;} \\
\end{align*}
\]

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-varying filter function, instead of using a box filter (FLC80).

Norton et al (NRS82).

Crow summed area tables (Cro84).


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 (Pea85). 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 \( \omega_h \) inside the spherical triangle–either \((0,0,1)\), at a vertex, or along an edge? Can we be sure that all \( \omega_h \) 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

```
#include "light.h"
#include "scene.h"
```

```c
#include "light.h"
#include "lrt.h"
#define LIGHT_H
#include "geometry.h"
#include "transform.h"
#include "color.h"
#include "paramset.h"
#include "mc.h"
```

```c
#include "light.cc"
#include "scene.h"
```

```c
#include "light.cc"
#include "scene.h"
```
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.

```cpp
class Light {
public:
    // Light Interface
protected:
    // Light Protected Data
};
```

So that the Integrators can compute light reflection at a point on a surface, Lights 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.
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 \( \omega \); therefore, the caller passes a pointer to the direction vector, which the light must fill in.

virtual Spectrum dE(const Point &P, const Normal &N, Vector *w, VisibilityTester *vis) const = 0;

Visibility Testing

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;
};

bool VisibilityTester::Unoccluded(const Scene *scene) const {
    // Update shadow ray statistics
    return !scene->IntersectP(r);
}

Spectrum VisibilityTester::Transmittance(const Scene *scene) const {
    return scene->Transmittance(r);
}
Since shadow rays may represent a significant fraction of overall rendering time, it’s useful to keep track of the total number of shadow rays traced.

\[\text{Update shadow ray statistics}\]≡

```cpp
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

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`.

```cpp
class PointLight : public Light {
public:
    \[PointLight Methods\]
private:
    \[PointLight Private Data\]
};
```

PointLights 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.
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 \( \omega \) 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.

\[
\text{SpotLight Method Definitions}
\]

\[
\text{SpotLight Class Definitions}
\]

\[
\text{SpotLight Classes}
\]

\[
\text{SpotLight Method Definitions}
\]

\[
\text{Spotlight.cc*}
\]
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.

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.

```c++
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));
}
```

```c++
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):

\[
\cos \theta = \frac{p - (0,0,0) \cdot (0,0,1)}{p_z/\|\vec{p}\|}
\]

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.

\textit{SpotLight Method Definitions}+

\begin{verbatim}
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);
}
\end{verbatim}

\textit{SpotLight Method Definitions}+

\begin{verbatim}
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)
}
\end{verbatim}

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.

\textit{Compute falloff inside spotlight cone}=

\begin{verbatim}
Float delta = (costheta - cosTotalWidth) / (cosFalloffStart - cosTotalWidth);
return delta*delta*delta*delta;
\end{verbatim}

\textbf{Texture Projection Light}

\textit{projectionlight.cc}*=

\textit{Source Code Copyright}

#include "lrt.h"
#include "light.h"
#include "shapes.h"
#include "texture.h"

\textit{ProjectionLight Classes}=

\textit{ProjectionLight Method Definitions}
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.

Because ProjectionLight also inherits from PointLight, we just need to override its \( I \) method.

```cpp
ProjectionLight::dE(const Point &P, const Normal &N, Vector *w, VisibilityTester *vis) 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 onto 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.

\section{ProjectionLight Method Definitions}

\begin{verbatim}
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 \}
    \}
}
\end{verbatim}

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, 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.

\begin{verbatim}
if (wl.z < hither) return 0.;
\end{verbatim}

After projecting the point to the projection plane, points with coordinate values between \( \pm 1 \) 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.

\begin{verbatim}
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;
\end{verbatim}

We can now just go ahead and evaluate the texture map, setting up a fake DifferentialGeometry for it to use.

\begin{verbatim}
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);
\end{verbatim}
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 $\omega$ is found by interpolating the intensities of the adjacent samples.

Goniometric diagram lights

```cpp
#include "lrt.h"
#include "light.h"
#include "shapes.h"
#include "scene.h"
#include "texture.h"

GoniometricLight Classes

GoniometricLight Method Definitions

The goniometric diagram describes the distribution of luminance from 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.

```cpp
class GoniometricLight : public Light {
public:
  GoniometricLight(bool shadows, const Transform &light2world,
                   const Spectrum &, const string &texname);

GoniometricLight Member Functions

GoniometricLight Private Data

};
```

```cpp
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 coordinates functions...

\[ \text{Goniometric Light Member Functions} \]

```cpp
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

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.
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.

```
#include "light.h"
#include "primitives.h"

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();
}
```

### 12.4 Area Lights

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 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 \( \pi \), the area of the hemisphere with projected solid angle measure, gives radiance in a particular direction.

AreaLight Interface

```cpp
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

```cpp
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

```cpp
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.
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 Integrators 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;
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 \((\theta, \phi)\) directions from there to \((u, v)\) texture coordinates; the transformation describes which direction is “up”. 

```cpp
#include "lrt.h"
#include "light.h"
#include "texture.h"
#include "shapes.h"
#include "scene.h"

class InfiniteAreaLight : public Light {
public:
    InfiniteAreaLight();

private:
    void transformTexture(spherical_to_texture transforms) const;
}
```

Figure 12.5: Uffizi latlong map
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
Further Reading 373

\[ \text{Light Method Definitions} \]
\[
\text{Spectrum Light::Le(const RayDifferential &r) const} \\
\text{\{ return Spectrum(0.); \}}
\]

The InfiniteAreaLight’s implementation of this can reuse the fragment from its \( d\text{E} \) method.

\[ \text{InfiniteAreaLight Function Definitions} \]
\[
\text{Spectrum InfiniteAreaLight::Le(const RayDifferential &r) const} \\
\text{\{ Vector w = r.D; \}} \\
\text{\{ Compute infinite light radiance for this direction \}} \\
\text{\{ return L; \}}
\]

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 illumination for artistic effect (Bjorke 01 renderman course notes). (The Barzel and Bjorke approaches are not physically based, however.)

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 \text{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 VolumeRegions will be used by VolumeIntegrators, which account for light interactions in participating media and will be described and implemented in Section 15.9.
There are 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 characteristics 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, \( \sigma_a \); 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 \( \hat{\omega} \), 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 \( L + dL \) is the amount that exits.

The change in radiance along differential ray length \( dr \) is described by the differential equation

\[
dL(x, \hat{\omega}) = -\sigma_a(x, \hat{\omega})L(x, \hat{\omega})dr.
\]

For a ray passing along a non-differential distance between two points \( x \) and \( x' \) in direction \( \hat{\omega} = \hat{x'} - \hat{x} \) with distance \( d \) between \( x \) and \( x' \), the differential equation can be solved to give the integral equation

\[
e^{-\int_0^d \sigma_a(x + r\hat{\omega}, \hat{\omega})dr},
\]
Emission

In this section, 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 \( \omega \) by \( L_e(x, \omega) \), then the change in radiance due to emission is

\[
\frac{dL}{dt}(x, \omega) = -\sigma_s(x, \omega)L_e(x, \omega)dt.
\]

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, \( \sigma_s \). Similarly to the attenuation coefficient, the change in radiance along a differential length \( dt \) is given by

\[
\frac{dL}{dt}(x, \omega) = -\sigma_s(x, \omega)L(x, \omega)dt.
\]

The total reduction in radiance due to absorption and out-scattering, then, is given by \( \sigma_a + \sigma_s \)

\[
\frac{dL}{dt}(x, \omega) = -(\sigma_a(x, \omega) + \sigma_s(x, \omega))L(x, \omega)dt.
\]

The sum of these terms is denoted by the attenuation coefficient \( \sigma_t \)

\[
\sigma_t(x, \omega) = \sigma_a(x, \omega) + \sigma_s(x, \omega)
\]
Given the attenuation coefficient $\sigma_t$, the differential equation can be solved to find the beam transmittance

$$T_r(x \rightarrow x') = e^{-\int_0^d \sigma_t(x + t\bar{\omega})dt},$$

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 $L(x, \bar{\omega})$, after accounting for extinction, the radiance at another point $x' = x + d\bar{\omega}$ in direction $\bar{\omega}$ is

$$T_r(x \rightarrow x')L(x, \bar{\omega}).$$

(See Figure 13.4.)

Two useful basic properties of beam transmittance are $T_r(x \rightarrow x) = 1$, and in a vacuum, $T_r(x \rightarrow x') = 1$ for all $x'$. Another important property, true in all media, is

$$T_r(x \rightarrow x'') = T_r(x \rightarrow x')T_r(x' \rightarrow x''),$$

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 $\tau$:

$$\tau(x \rightarrow x') = \int_0^d \sigma_t(x + t\bar{\omega})dt.$$

In a homogeneous medium, $\sigma_t$, is a constant, and Beer’s law describes the attenuation.

$$T_r = e^{-\sigma_t d},$$
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* $\alpha$, 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*, $p(\vec{\omega} \rightarrow \vec{\omega}')$, 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 $\vec{\omega}$,

\[
\frac{1}{4\pi} \int_{S^2} p(\vec{\omega} \rightarrow \vec{\omega}') d\vec{\omega}' = 1. \tag{13.1.1}
\]

In most naturally-occurring 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 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.
The change in radiance due to in-scattering is given by the source term,

\[ S(x, \omega) = \sigma_s(x, \omega) \int_{S^2} p(\omega' \rightarrow \omega) L_i(x, \omega') d\omega'. \]

#### 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

\[ p_{\text{isotropic}}(\omega \rightarrow \omega') = \frac{1}{4\pi}. \]
If the particles have radius $r$ that is smaller than the wavelength of light $\lambda$, the Rayleigh model is a good fit if $r/\lambda < 0.05$.

$$p_{\text{rayleigh}}(\vec{\omega} \rightarrow \vec{\omega}') = p_{\text{rayleigh}}(\vec{\omega} \cdot \vec{\omega}') = p_{\text{rayleigh}}(\cos \theta) = \frac{3}{16\pi} (1 + \cos^2 \theta)$$

Wavelength-dependent Rayleigh scattering is what makes the sky blue and sunsets red.

\[
\text{Mie scattering is when } r \approx \lambda. \text{ Water droplets, fog. Nishita et al suggest two approximations for hazy and murky atmospheric conditions}
\]

\[
p_{\text{Mie-hazy}}(\cos \theta) = 1 + \frac{9}{4\pi} \left( \frac{1 + \cos \theta}{2} \right)^8
\]

and

\[
p_{\text{Mie-murky}}(\cos \theta) = 1 + \frac{50}{4\pi} \left( \frac{1 + \cos \theta}{2} \right)^{32}
\]

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.

\[
p_{\text{HG}}(\cos \theta) = \frac{1}{4\pi (1 + g^2 - 2g\cos \theta)^{3/2}}
\]
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.

$$g = \frac{1}{2} \int_{S^2} p(\vec{\omega} \rightarrow \vec{\omega}') (\vec{\omega} \cdot \vec{\omega}') d\vec{\omega}'$$

Thus, isotropic scattering corresponds to a $g$ of zero. Any number of phase functions can satisfy this equation; 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:

$$p(\vec{\omega} \rightarrow \vec{\omega}') = \sum_{i=1}^{n} w_i p_i(\vec{\omega} \rightarrow \vec{\omega}')$$

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

$$p_{Schlick}(\cos \theta) = \frac{1}{4\pi} \frac{1 - g^2}{(1 - g \cos \theta)^2}$$

$$p_{Schlick}(\cos \theta) = \frac{1}{4\pi} \frac{1 - g^2}{(1 - g \cos \theta)^2}$$

Float PhaseSchlick(const Vector &w, const Vector &wp, Float g) {
    Float gcostheta = g * Dot(w, wp);
    return 1.f / (4.f * M_PI) * (1.f - g*g) /
           powf(1.f + g*g - 2.f * g * gcostheta, 1.5f);
}

}
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 VolumeRegions 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.

```cpp
#include "volume.h"
```

All VolumeRegions 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

```cpp
virtual BBox WorldBound() const = 0;
virtual bool Intersect(const Ray &ray, Float *t0, Float *t1) const = 0;
```

There are four basic functions that allow VolumeRegions 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.

```cpp
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...

 sandbox/VolumeScattering 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.)

HomogeneousRegion Methods

<table>
<thead>
<tr>
<th>Homogeneous Region</th>
</tr>
</thead>
</table>

The simplest volume representation, HomogeneousRegion, describes a region of space bounded by a BBox with homogeneous scattering properties throughout it. Values for \( \sigma_a, \sigma_s \), 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.

<table>
<thead>
<tr>
<th>homogeneous.cc</th>
</tr>
</thead>
</table>

HomogeneousRegion Declarations

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Declarations

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>

HomogeneousRegion Definitions

<table>
<thead>
<tr>
<th>HomogeneousRegion Definitions</th>
</tr>
</thead>
</table>
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.IntersecP(ray, t0, t1);
}

Implementation of the rest of the VolumeRegion interface methods is straight-
forward; 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 : 0.0;
}

HomogeneousRegion Methods

Spectrum sigma_s(const Point &p, const Vector &) const {
    return extent.Inside(WorldToVolume(p)) ? sig_s : 0.0;
}

HomogeneousRegion Methods

Spectrum sigma_t(const Point &p, const Vector &) const {
    return extent.Inside(WorldToVolume(p)) ? (sig_a + sig_s) : 0.0;
}

HomogeneousRegion Methods

Spectrum Le(const Point &p, const Vector &) const {
    return extent.Inside(WorldToVolume(p)) ? le : 0.0;
}

HomogeneousRegion Methods

Float phase(const Point &p, const Vector &wi, const Vector &wo) const {
    if (!extent.Inside(WorldToVolume(p))) return 0.0;
    return PhaseHG(wi, wo, g);
}

HomogeneousRegion Methods

Spectrum tau(const Ray &r) const {
    Ray ray = WorldToVolume(r);
    Float t0, t1;
    if (!extent.IntersecP(ray, &t0, &t1)) return 0.0;
    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.

```cpp
class DensityRegion : public VolumeRegion {
public:
    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;
    }

protected:
    Transform WorldToVolume;
    Spectrum sig_a, sig_s, le;
    Float g;

public:
    virtual Float density(const Point &Pobj) const = 0;
    Spectrum sigma_a(const Point &p, const Vector &) const {
        return density(WorldToVolume(p)) * sig_a;
    }
    Spectrum sigma_s(const Point &p, const Vector &) const {
        return density(WorldToVolume(p)) * sig_s;
    }
    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
$\sigma_a, \sigma_s$, etc., all of which are just scaled by the local density at the point of interest.
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

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); }

bool Intersect(const Ray &r, Float *t0, Float *t1) const {
  Ray ray = WorldToVolume(r);
  return extent.IntersectP(ray, t0, t1);
}
VolumeGrid Definitions

```cpp
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

```cpp
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

```cpp
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);
```

Exponential Mist

Density varies linearly as a function of $z$

$$d = ae^{-bP_z}$$

```cpp
#include "volume.h"
ExponentialMist Definitions
```

Exponential Mist Declarations

```cpp
class ExponentialMist : public DensityRegion {
public:
    // ExponentialMist Methods
private:
    // ExponentialMist Private Data
};
```
\(\langle \text{ExponentialMist Methods} \rangle \equiv \)
\[
\text{ExponentialMist} \left( \text{const Spectrum } \&sa, \text{ const Spectrum } \&ss, \text{ Float } gg, \\
\text{ const Spectrum } \&\text{emit}, \text{ const } \text{BBox } \&e, \text{ const } \text{Transform } \&v2w, \\
\text{ Float } a, \text{ Float } b \right) \\
: \text{DensityRegion}(sa, ss, gg, emit, v2w) \{ \\
\text{extent} = e; \\
A = a; \\
B = b; \\
\}
\]

\(\langle \text{ExponentialMist Private Data} \rangle \equiv \)
\[
\text{BBox extent;} \\
\text{Float A, B;} \\
\]

\(\langle \text{ExponentialMist Methods} \rangle \equiv \)
\[
\text{BBox WorldBound() const } \{ \text{return WorldToVolume.GetInverse()}(extent); \} \\
\]

\(\langle \text{ExponentialMist Methods} \rangle \equiv \)
\[
\text{bool Intersect(const Ray } \&r, \text{ Float } *t0, \text{ Float } *t1) \text{ const } \{ \\
\text{Ray ray} = \text{WorldToVolume}(r); \\
\text{return extent.IntersectP(ray, t0, t1);} \\
\}
\]

\(\langle \text{ExponentialMist Methods} \rangle \equiv \)
\[
\text{Float density(const Point } \&\text{Pobj) const } \{ \\
\text{return } A * \exp(\text{-B } \text{Pobj.z}); \\
\}
\]

\[
d_{\text{total}} = \int_0^1 \int_0^1 d(\alpha) d\alpha' = |P_1 - P_0| \int_0^1 e^{-\alpha} dz = \frac{A|P_1 - P_0|}{Bd(r)z} \left( e^{-\alpha(r)z} + d'(r)z|P_1 - P_0| e^{-\alpha(r)z} \right)
\]

\(\langle \text{ExponentialMist Methods} \rangle \equiv \)
\[
\text{Spectrum tau(const Ray } \&r) \text{ const } \{ \\
\text{Ray ray} = \text{WorldToVolume}(r); \\
\text{Float t0, t1;} \\
\text{if (!extent.IntersectP(ray, } \&t0, \&t1)) \text{ return 0.;} \\
\text{Float dist} = \text{Distance}(ray(t0), \text{ray(t1)}); \\
\text{return } (A * \text{dist})/(B * \text{ray.D.z}) * \\
\left( \exp(-\text{ray(t1).z}) - \exp(-\text{ray(t0).z}) \right) * \\
\left( \text{sig_a + sig_s} \right); \\
\}
\]
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

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 $S(x', \omega_t, x, \omega_o)$ that describes the proportion of outgoing differential radiance at point $x$ in direction $\omega_o$ due to differential irradiance at $x'$ from direction $\omega_t$.

The scattering equation for the BSSRDF requires integration over surface area $A$ and incoming direction $\omega_t$; it is substantially more complex than Equation 5.4.8.

$$L_o(x, \omega_o) = \int_A \int_{S^2} S(x', \omega_t, x, \omega_o) \cos \theta_i d\omega_i dA$$

Fortunately, points $x'$ that are far away from $x$ generally contribute little to $L_o(x, \omega_o)$. 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 simulate 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 (DEL99). 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 than 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).


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 $\int f(x) dx$; 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.
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 $\xi$ 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 $\xi$, 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.

\[
p(x) = \frac{dP(x)}{dx}
\]

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 \( \xi \), \( P(x) = x \) 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.

\[
P(x \in [a, b]) = \int_a^b p(x) dx
\]

**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.

\[
E[f(x)] = \int f(x)p(x)dx
\] (14.1.1)

Consider finding the expected value of the cosine function between 0 and \( \pi \), where \( p \) is uniform. Because \( p(x) \) must integrate to one over the domain, we have \( p(x) = 1/\pi \) and

\[
E[\cos(x)] = \int_0^\pi \frac{\cos x}{\pi} dx
\]

\[
= \frac{1}{\pi}(-\sin \pi + \sin 0)
\]

\[
= 0
\]

Which is precisely what we expect.

The expected value can be estimated with the sum

\[
E[f(x)] \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i), \quad (14.1.2)
\]

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:

\[
\int f(x) \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \quad (14.1.3)
\]
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:

\[
E[\int f(x)] = \frac{1}{N} \sum_{i=1}^{N} f(x_i)w(x_i)
\]  

(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 \( \xi_i \) 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, \( \xi_1 \) and \( \xi_2 \). If \( \xi_2 < f(\xi_1)/M \), then we accept \( \xi_1 \) 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 \( \xi \), we need to solve the equation

\[
\xi = \int_{-\infty}^{x} p(x)dx
\]

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 \( \xi_i \) 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 $\xi$ and determine which value of $x \in [0, 1]$ it maps to:

$$\xi = (n + 1) \int_0^x x^a \, dx = x^{n+1}$$

$$x = \sqrt[n+1]{\xi}$$

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:

$$p_x(x) = \int_{-\infty}^{\infty} p(x, y) \, dy$$

$$p_y(y|x) = \frac{p(x, y)}{\int_{-\infty}^{\infty} p(x, y') \, dy'}$$

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 $x_i = i \ast \Delta$, 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

$$f(x) = \begin{cases} v_0 & x_0 \leq x < x_1 \\ v_1 & x_1 \leq x < x_2 \\ \vdots & \vdots \end{cases}$$

The integral of $\int f(x) \, dx$ is

$$c = \int_0^1 f(x) \, dx = \sum_{i=0}^{N-1} \Delta f_i = \sum_{i=0}^{N-1} \frac{f_i}{N}, \quad (14.1.6)$$

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

$$F(x_0) = 0$$

$$F(x_1) = \int_{x_0}^{x_1} p(x) \, dx = \frac{v_0}{(N \ast c)} = F(x_0) + \frac{v_0}{(N \ast c)}$$

$$F(x_2) = \int_{x_0}^{x_2} p(x) \, dx = \int_{x_0}^{x_1} p(x) \, dx + \int_{x_1}^{x_2} p(x) \, dx = F(x_1) + \frac{v_1}{(N \ast c)}$$

$$\vdots \quad \vdots$$

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

$$\xi = \int_0^{x'} p(x) \, dx = F(x').$$

Because the cdf is monotonically increasing, the value of $x'$ must be between the $x_i$ and $x_{i+1}$ such that $F(x_i) \leq \xi$ and $\xi \leq F(x_{i+1})$.

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 $c$.

\[
\text{MC Function Definitions} \equiv \\
\text{void ComputeStep1dCDF(Float *f, int nSteps, Float *c, Float *cdf) {} \\
\quad \langle \text{Compute integral of step function at } x_i \rangle \\
\quad \langle \text{Transform step function integral into cdf} \rangle 
\]
We start by computing the integral of \( f(x) \), using Equation 14.1.6. We will store the result in the \( \text{cdf} \) array for now so that we don’t need to allocate additional temporary space for it. We allocate \( n\text{Steps}+1 \) floats for the \( \text{cdf} \) array because if \( f(x) \) has \( N \) step values, then we need to store the value of the \( \text{cdf} \) at each of the \( N+1 \) values of \( x_i \).

\[
\text{Compute integral of step function at } x_i \equiv
\begin{align*}
\text{int } i; \\
cdf[0] &= 0.; \\
\text{for } (i = 1; i < n\text{Steps}+1; ++i) \\
\text{cdf}[i] &= cdf[i-1] + f[i-1] / n\text{Steps};
\end{align*}
\]

Now that the value of the integral over all of \([0, 1]\) is stored in \( \text{cdf}[n\text{Steps}] \), we can normalize the \( \text{cdf} \) by dividing through by this value.

\[
\text{Transform step function integral into cdf} \equiv
\begin{align*}
*\text{c} &= \text{cdf}[n\text{Steps}]; \\
\text{for } (i = 1; i < n\text{Steps}+1; ++i) \\
\text{cdf}[i] &= \text{cdf}[i]/*\text{c};
\end{align*}
\]

Sampling the function from the \( \text{cdf} \) is handled by the \texttt{SampleStep1d} function.

\[
\text{MC Function Definitions} \equiv
\begin{align*}
\text{Float SampleStep1d}(\text{Float } \ast f, \text{Float } \ast \text{cdf}, \text{Float } c, \\
\text{int } n\text{Steps}, \text{Float } u, \text{Float } \ast \text{weight}) 
\begin{cases}
\text{Find surrounding cdf segments} \\
\text{Return offset along current cdf segment}
\end{cases}
\end{align*}
\]

First, we need to find the pair of \( \text{cdf} \) values that straddle \( \xi \). Because the \( \text{cdf} \) array is monotonically increasing (and is thus a sorted array), we can use a binary search function from the C++ standard library: \texttt{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.

\[
\text{Find surrounding cdf segments} \equiv
\begin{align*}
\text{Float } \ast \text{ptr} &= \text{std::lower_bound(cdf, cdf+n\text{Steps}+1, u);} \\
\text{int } \text{offset} &= (\text{int})(\text{ptr}-\text{cdf}-1);
\end{align*}
\]

Now that we know the pair of \( \text{cdf} \) values, we can compute \( x' \). First, we determine how far \( \xi \) is between \( \text{cdf}[\text{offset}] \) and \( \text{cdf}[\text{offset}+1] \). Because the \( \text{cdf} \) 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.

\[
\text{Return offset along current cdf segment} \equiv
\begin{align*}
u &= (u - \text{cdf}[\text{offset}]) / (\text{cdf}[\text{offset}+1] - \text{cdf}[\text{offset}]); \\
*\text{weight} &= f[\text{offset}] / (c \times n\text{Steps}); \\
\text{return } (\text{offset} + u) / n\text{Steps};
\end{align*}
\]

**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

$$f_x(x) = \int_0^1 f(x,y)dy = \sum_{i=0}^{N_y-1} f(x,y)\Delta_y$$

Now, to generate a sample from $f$ given uniform random numbers $\xi_1$ and $\xi_2$, 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.

```cpp
MC Function Definitions

Constant2DSampler::Constant2DSampler(Float *f, int _nx, int _ny) {
    #if 0
        f = f;
        nx = _nx;
        ny = _ny;
        (Compute $f_x(x)$ cdf)
        (Compute y column cdfs)
    #endif
}

Constant2DSampler Private Data

Float *f, *fx;
int nx, ny;
Float *xcdf;
Float **ycdfs;
Float xc, *ycs;
```
\( \text{Compute } f_x(x) \text{ cdf} \equiv \)
\[ fx = \text{new Float}[nx]; \]
\[ \text{for (int } x = 0; x < nx; ++x) \{ \]
\[ \quad fx[x] = 0.; \]
\[ \quad \text{for (int } y = 0; y < ny; ++y) \]
\[ \quad \quad fx[x] += f[y*nx + x]; \]
\[ \} \]
\[ \text{xcdf} = \text{new Float}[nx]; \]
\[ \text{ComputeStep1dCDF}(fx, nx, \&xc, xcdf); \]

\( \text{Compute } y \text{ column cdfs} \equiv \)
\[ \text{ycdfs} = \text{new Float *}[nx]; \]
\[ \text{ycs} = \text{new Float}[nx]; \]
\[ \text{float *ftmp} = \text{new Float}[ny]; \]
\[ \text{for (int } x = 0; x < nx; ++x) \{ \]
\[ \quad \text{for (int } y = 0; y < ny; ++y) \]
\[ \quad \text{ftmp}[y] = f[y*nx + x]; \]
\[ \quad \text{ycdfs}[x] = \text{new Float}[ny]; \]
\[ \quad \text{ComputeStep1dCDF(ftmp, ny, } \&\text{ycs}[x], \text{ycdfs}[x]); \]
\[ \} \]
\[ \text{delete[] ftmp}; \]

\( \text{MC Function Definitions} \equiv \)
\[ \text{void Constant2DSampler::Sample(Float } u1, \text{ Float } u2, \text{ Float } *x, \text{ Float } *y, \text{ Float } *weight) \text{ const } \{ \]
\[ \quad \text{#if 0} \]
\[ \quad \quad \text{Float xweight, yweight;} \]
\[ \quad \quad \quad *x = \text{SampleStep1d}(fx, xcdf, nx, u1, \&xweight); \]
\[ \quad \quad \quad \text{int xoffset} = \text{int}(*x * nx); \]
\[ \quad \quad \quad *y = \text{SampleStep1d}(\text{ycdfs[xoffset]}, ny, u2, \&yweight); \]
\[ \quad \quad \quad \text{*weight} = \text{xweight} * *yweight; \]
\[ \quad \text{#endif} \]
\[ \} \]

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 \( \Delta: x_i = i \Delta \) 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 \( F(i\Delta) \).
To compute the normalization constant, we first need to compute the integral \( \int f(x) \, dx \). We will incrementally compute the integral, storing the value \( \int_{x_0}^{x_1} f(x) \, dx \) in the \( i \)th element of the \( \text{cdf} \) array. It is easy to see that the area of the shaded region (and thus the value of \( \int_{x_0}^{x_1} f(x) \, dx \)) in Figure 14.3 is

\[
\left( \frac{1}{2} v_0 + \frac{1}{2} v_1 \right) \Delta.
\]

Similarly, the area of the region next to it is

\[
\left( \frac{1}{2} v_1 + \frac{1}{2} v_2 \right) \Delta.
\]

Since

\[
\int_{x_0}^{x_2} f(x) \, dx = \int_{x_0}^{x_1} f(x) \, dx + \int_{x_1}^{x_2} f(x) \, dx,
\]

we can compute successive values of the integral from previous ones:

\[
\text{(Compute integral of piecewise linear function)}
\]

```cpp
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, \( \text{cdf}[nValues-1] \) 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.

\[
\text{(Compute piecewise linear function’s cdf)}
\]

```cpp
Float c = 1.f / cdf[nValues-1];
for (i = 1; i < nValues; ++i)
    cdf[i] *= c;
cdf[nValues-1] = 1.;
```
To sample this function, we need to take a point $\xi$ and compute the offset $x'$ such that

$$
\xi = \int_{x_0}^{x'} F(x) \, dx.
$$

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

```cpp
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 is a monotonically 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 $\xi$; 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

$$
G(x) = \frac{v_i x + \frac{1}{2}(v_{i+1} - v_i) x^2}{\frac{1}{2}(v_i + v_{i+1})}
$$

We remap $\xi$ to $\xi'$, 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 $\Delta$ between 0 and 1. This offset tells us how far between $x_i$ and $x_{i+1}$ $x'$ lies.
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 \textit{stratify} the random variables \( \xi \) 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 \texttt{JitterSampler} in Chapter 7 is doing.

\( \text{XXX} \) 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 \( 2 \times \text{rootSamples} \times \text{rootSamples} \) Floats; successive pairs of them hold the resulting sample pattern.

\begin{verbatim}
\begin{verbatim}

define (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;
            }
    }
\end{verbatim}

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 \textit{Latin Hypercube} sampling (in graphics, sometimes this is called \textit{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 independently permute the sample points in that dimension.
Monte Carlo Integration

\[ MC \text{ Function Definitions} \]

\[
\begin{align*}
\text{void LatinHypercube(Float *samples, int nSamples, int nDim) } & \{ \\
& \quad \text{int i, j;} \\
& \quad \text{\langle Generate samples along diagonal \rangle} \\
& \quad \text{\langle Permute in each dimension \rangle} \\
& \} \\
\end{align*}
\]

\[
\text{\langle Generate samples along diagonal \rangle} \equiv \\
\text{Float delta = } 1.f / \text{nSamples;} \\
\text{for (i = 0; i < nSamples; ++i)} \\
\quad \text{for (j = 0; j < nDim; ++j)} \\
\quad \quad \text{samples[nDim * i + j] = (i + RandomFloat()) * delta;} \\
\]

To do the permutation, we loop over the samples, processing one dimension at a time. We use a utility function, 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.

\[
\text{\langle Permute in each dimension \rangle} \equiv \\
\quad \text{for (i = 0; i < nDim; ++i)} \{ \\
\qquad \text{int *permuteTable = (int *)alloca(nSamples * sizeof(int));} \\
\qquad \text{Permute(permuteTable, nSamples);} \\
\qquad \text{for (j = 0; j < nSamples; ++j)} \{ \\
\qquad \quad \text{int other = permuteTable[j];} \\
\qquad \quad \text{swap(samples[nDim * j + i], samples[nDim * other + i]);} \\
\qquad \} \\
\}
\]

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.

\[
\text{\langle MC \text{ Function Definitions} \rangle} \equiv \\
\text{void Permute(int *table, int n) } \{ \\
& \quad \text{int i;} \\
& \quad \text{for (i = 0; i < n; ++i)} \\
& \quad \quad \text{table[i] = i;} \\
& \quad \text{for (i = 0; i < n; ++i)} \\
& \quad \quad \text{swap(table[i], table[RandomInt() % n]);} \\
& \} \\
\]

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:

\[
p(x, y) = \int_x \int_y f(x - x', y - y') L(x', y') \, dx' \, dy' \tag{14.1.7}
\]

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 denominator serves to normalize the filter function. For simplicity, we assume here that the pixel filter function has been normalized so that

\[
\int_x \int_y f(x', y') \, dx' \, dy' = 1.
\]

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

\[ |I - \hat{I}| \leq V(f) D_N^s(P). \tag{14.2.8} \]

Thus, the error is split into a component \( V(f) \) that depends only on the function being integrated and a component \( D_N^s(P) \) 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

\[ D_N^s(P) = O\left(\frac{\log N}{N}\right)^{s-1}. \]

In particular, note that for \( s = 1 \),

\[ D_N^1(P) = O\left(\frac{1}{N}\right). \]

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 \( O(N^{-\frac{s}{2}}) \) 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:

$$E = \prod_{i=1}^{s} [a_i b^{-d_i}, (a_i + 1)b^{-d_i}], \quad d_i \geq 0, \quad 0 \leq a_i \leq b^{d_i}.$$ 

Where $\prod$ denotes a product of intervals over all dimensions.

For example, with $b = 5$ and $s = 3$, valid boxes include $1 \times \frac{1}{5} \times 1$, $\frac{1}{125} \times \frac{1}{125} \times 1$, 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 $\lambda(E) = b^{-m}$, then

$$\|\{x_i \in E\} = b^t.$$ 

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 $\omega_o$.

$$L_o(x, \omega_o) = \int_{\Omega} f_r(x, \omega_i \rightarrow \omega_o) L_i(\omega_i) \cos \theta_i d\omega_i. \quad (14.3.9)$$

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.

```
Vector RejectionSampleHemisphere() {
    Vector wo;
    Sample BxDF hemisphere uniformly to compute wo
    return wo;
}
```

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

\[
1 = \int_{\Omega} p(\omega) d\omega = \int_{\Omega} c d\omega = \int_0^{2\pi} \int_{\phi=0}^{\pi/2} c \sin \theta d\theta d\phi = 2\pi c
\]

so \(p(\alpha) = 1/2\pi\).
Sec. 14.3] Sampling Reflection Functions

513

[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 $\xi_1$ and $\xi_2$ to uniform directions $(\theta, \phi)$ 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 $\theta$ and $\phi$ separately.

For $\theta$, we need to solve $\xi_1 = \int_0^\phi \sin \theta d\theta$ for $\theta$. Some algebra gives us:

$$\theta = \arccos(1 - \xi_1)$$

To sample $\phi$, we just have $\phi = 2\pi \xi_2$.

To compute a vector direction, we use the spherical angle formula, which gives us:

\[
\begin{align*}
    x &= \sqrt{1 - z^2} \cos \phi \\
    y &= \sqrt{1 - z^2} \sin \phi \\
    z &= \cos \theta = 1 - \xi_1
\end{align*}
\]

[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:

\[
\begin{align*}
    x &= 2r \sqrt{\xi_1(1 - \xi_1)} \cos \phi \\
    y &= 2r \sqrt{\xi_1(1 - \xi_1)} \sin \phi \\
    z &= 1 - 2\xi_1
\end{align*}
\]
We would like to sample a direction $\theta$ uniformly over the cone of directions around the center direction $\omega_c$ up to that maximum angle.

$$1 = c \int_0^{\theta_{\text{max}}} \sin \theta d\theta$$

$$= c (\cos \theta_{\text{max}} + 1)$$

So $p(\theta) = c/ (1 - \cos \theta_{\text{max}})$ and the weighting function $w(\theta) = 1 - \cos \theta_{\text{max}}$.

To sample a particular offset angle,$$
\xi = \frac{1}{(1 - \cos \theta_{\text{max}})} \int_0^{\theta'} \sin \theta d\theta$$

$$\xi (1 - \cos \theta_{\text{max}}) = 1 - \cos \theta'$$

$$\cos \theta' = 1 - \xi (1 - \cos \theta_{\text{max}})$$

$$\theta' = \arccos(1 - \xi (1 - \cos \theta_{\text{max}}))$$

Actually $\cos \theta'$ is what we want anyway for spherical angles centered around $\omega_c$.

```
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);
}

MC Function Definitions}+

MC Function Definitions}+

Float UniformConeWeight(Float cosThetaMax) {
    return 1.f / (2.f * M_PI * (1.f - cosThetaMax));
}
```
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 \( \omega_i \) 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.

\[
\text{Figure 14.6: Malley's method: to sample direction vectors from a cosine-weighted distribution, uniformly sample points on the unit disk and project them up to the unit sphere.}
\]

\[
\text{BxDF method definitions}
\]

\[
\text{inline Vector CosineSampleHemisphere(Float u1, Float u2) {}
\]

\[
\text{Vector ret;}
\text{ConcentricSampleDisk(u1, u2, &ret.x, &ret.y);}
\text{ret.z = sqrtf(max(0.f, 1.f - ret.x*ret.x - ret.y*ret.y));}
\text{return ret;}
\]

\[
\text{We will make this the default BxDF sampling method; only the BxDFSs where a more effective sophisticated approach can be derived need to override this. As with the method that sampled uniformly over the hemisphere, this one also has non-zero probability of sampling all directions, so can be used for any reflective BxDF.}
\]

\[
\text{MC utility declarations}
\]

\[
\text{inline Spectrum BxDF::sample_f(const Vector &wi, Vector *wo,}
\text{Float u1, Float u2, Float *wt) const {
\text{Float x, y;
\text{ConcentricSampleDisk(u1, u2, &x, &y);
\text{\{Compute final direction with Malley's method\}\}
\text{*wt = Weight(wi, *wo);
\text{return f(wi, *wo);}}
\]

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 \( \xi_1 \) and \( \xi_2 \) 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 \( \xi_1 \) and \( \xi_2 \) 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:

\[
1 = c \int_0^1 2\pi r dr
\]

so the normalization constant is \(c = 1/\pi\). Given a uniform random variable \(\xi\), the radius we sample should be

\[
\xi = \frac{1}{\pi} \int_0^r 2\pi r dr
\]

so \(r = \sqrt{\xi}\). Given this radius, we then sample uniformly in direction \(\theta\), picking all points around the circle with equal probability.

\[
\text{MC Function Definitions}
\]

\[
\text{void UniformSampleDisk(Float u1, Float u2, Float *x, Float *y) \{}
\text{Float r = sqrtf(u2);}
\text{Float theta = 2.0f * M_PI * u1;}
\text{*x = r * cosf(theta);}
\text{*y = r * sinf(theta);}
\text{\}}
\]

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 to nearby points on the disk (e.g. \((.5, .01)\) and \((.5, .99)\)). A number of researchers
Sec. 14.3] Sampling Reflection Functions

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 $(r, \theta)$ 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 concentric 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 $(r, \theta)$ by

\[
\begin{align*}
    r &= x \\
    \theta &= \frac{y}{x}
\end{align*}
\]

See Figure 14.8. The other four quadrants are handled analogously.

\[\text{MC Function Definitions} + \equiv\]

\[
\text{void ConcentricSampleDisk(Float u1, Float u2,} \nonumber \\
\text{Float } \ast dx, \text{ Float } \ast dy) \{ 
\]

\[
\text{Float r, theta;} \nonumber \\
\text{\langle Map uniform random numbers to } [-1, 1]^2 \boxed{\rangle} \nonumber \\
\text{\langle Map square to } (r, \theta) \boxed{\rangle} \nonumber \\
\text{\ast dx = r*cosf(theta);} \nonumber \\
\text{\ast dy = r*sinf(theta);} 
\}
\]

\[\text{\langle Map uniform random numbers to } [-1, 1]^2 \equiv \]

\[
\text{Float sx = 2 * u1 - 1;} \\
\text{Float sy = 2 * u2 - 1;} 
\]
\[
\langle \text{Map square to } (r, \theta) \rangle \equiv \\
\langle \text{Handle degeneracy at the origin} \rangle \\
\text{if } (sx >= -sy) \{ \\
\phantom{=} \text{if } (sx > sy) \{ \\
\phantom{=} \langle \text{Handle first region} \rangle \\
\phantom{=} \} \\
\phantom{=} \text{else } \{ \\
\phantom{=} \langle \text{Handle second region} \rangle \\
\phantom{=} \} \\
\} \\
\text{else } \{ \\
\phantom{=} \text{if } (sx <= sy) \{ \\
\phantom{=} \langle \text{Handle third region} \rangle \\
\phantom{=} \} \\
\phantom{=} \text{else } \{ \\
\phantom{=} \langle \text{Handle fourth region} \rangle \\
\phantom{=} \} \\
\} \\
\theta * = \frac{\pi}{4}; \\
\]

\[
\langle \text{Handle degeneracy at the origin} \rangle \equiv \\
\text{if } (sx == 0.0 \&\& sy == 0.0) \{ \\
\phantom{=} *dx = 0.0; \\
\phantom{=} *dy = 0.0; \\
\phantom{=} \text{return; } \\
\} \\
\]

\[
\langle \text{Handle first region} \rangle \equiv \\
\phantom{=} r = sx; \\
\phantom{=} \text{if } (sy > 0.0) \\
\phantom{=} \phantom{=} \theta = sy/r; \\
\phantom{=} \text{else} \\
\phantom{=} \phantom{=} \phantom{=} \phantom{=} \theta = 8.0f + sy/r; \\
\text{The remaining cases are analogous and are elided.} \\
\text{Once we have a point on the unit disk, it is straightforward to compute the } z \text{ value of this point on the sphere for Malley's method, since we know that } x^2 + y^2 + z^2 = 1. \text{ We wrap up by setting the output } wo \text{ variable and then flipping the sampled direction so that it lies on the same hemisphere (above or below the surface as the incident direction.)} \\
\langle \text{Compute final direction with Malley's method} \rangle \equiv \\
\phantom{=} \text{Float z = sqrtf(max(0.f, 1.f - x*x - y*y)); } \\
\phantom{=} \phantom{=} *wo = \text{Vector}(x, y, z); \\
\phantom{=} \phantom{=} \text{if } (wi.z * wo->z < 0.) *wo = -*wo; \\
\text{We know that Malley's method generates samples in a cosine distribution, so } \rho(\omega) \propto \cos \theta. \text{ We need to normalize this distribution so that it's a valid pdf.}
Thus \( p(\omega) = \cos \omega / \pi \) and the weight method is:

\[
\text{BxDF Method Definitions}\] + \Xi
\[
\text{Float BxDF::Weight(const Vector &wi, const Vector &wo) const } \begin{align*}
&\text{const } \\
&\text{return fabsf(wo.Hat().z) * INV_PI;}
\end{align*}
\]

This sampling method is a fine one for Lambertian reflectors, so we won’t over-ride the method for Lambertian or OrenNayar BxDFs.

\[
\text{BxDF Method Definitions}\] + \Xi
\[
\text{Spectrum BRDFToBTDF::sample_f(const Vector &wi, Vector *wo, Float u1, Float u2, Float *wt) const } \begin{align*}
&\text{const } \\
&\text{Spectrum f = brdf->sample_f(wi, wo, u1, u2, wt);} \\
&\text{*wo = -*wo;} \\
&\text{return f;}
\end{align*}
\]

\[
\text{BxDF Method Definitions}\] + \Xi
\[
\text{Float BRDFToBTDF::Weight(const Vector &wi, const Vector &wo) const } \begin{align*}
&\text{const } \\
&\text{return brdf->Weight(wi, -wo);}
\end{align*}
\]

**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 must implement sampling and weight functions, each with the same signature as the corresponding BxDF function.

\[
\text{MicrofacetDistribution Interface}\] + \Xi
\[
\text{virtual void sample_f(const Vector &wi, Vector *wo, Float u1, Float u2) const = 0;}
\]

Microfacet BxDFs, then, just forward on the sampling and weight requests to their distribution function.
Monte Carlo Integration

\( \text{BxDF Method Definitions} \)

\[ \text{Spectrum Microfacet}::\text{sample}_f(\text{const Vector } \&\text{wi}, \text{Vector } *\text{wo}, \]  
\[ \text{Float } u1, \text{Float } u2, \text{Float } *\text{wt} \text{ const } \]  
\[ \{ \]  
\[ \text{distribution->sample}_f(\text{wi}, \text{wo}, u1, u2); \]  
\[ *\text{wt} = \text{distribution->Weight}(\text{wi}, *\text{wo}); \]  
\[ \text{return } f(\text{wi}, *\text{wo}); \]  
\[ \} \]

\[ \text{BxDF Method Definitions} \]

\[ \text{Float Microfacet}::\text{Weight}(\text{const Vector } \&\text{wi}, \]  
\[ \text{const Vector } \&\text{wo} \text{ const } \]  
\[ \{ \]  
\[ \text{return } \text{distribution->Weight}(\text{wi}, \text{wo}); \]  
\[ \} \]

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 \( \phi \) 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 \( \theta_H \), we must first normalize it, so that \( \int_0^{\pi/2} p(\theta)d\omega = 1 \).

\[ 1 = c \int_0^{\pi/2} (n+1)\cos^n \theta_H \sin \theta_H d\theta_H \]
\[ c = 2\pi \]

Thus, our pdf \( p_h(\theta) \) is \( (n+1)\cos^n \theta_H \). To sample from the distribution given a uniform random number \( \xi \), we solve:

\[ \xi = \int_0^\theta (n+1)\cos^n \theta_H \sin \theta_H d\theta_H \]
\[ \xi = \cos^{n+1} \theta - \cos^{n+1} 0 \]
\[ \sqrt{1-\xi} = \cos \theta \]

Since \( \xi \) is a uniform random number, so is \( 1-\xi \), so we can simplify this to \( \cos \theta = \sqrt{\xi} \). Since the value \( \phi \) 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:

\[ \int_{\Omega} f_r(\omega_r) L(\omega_i) \cos \theta_i d\omega_i \]

In order to convert to an integral over solid angle, we must multiply by the Jacobian \( \partial \omega_r / \partial \omega_H \):

\[ \int_{\Omega} f_r(\omega_i, \omega_r) L(\omega_i) \cos \theta_i d\omega_H \frac{\partial \omega_i}{\partial \omega_H} = \int_{\Omega} f_r(\omega_i, \omega_r) L(\omega_i) \cos \theta_i d\omega_i = L_o(x, \omega_r) \]
Figure 14.9: The adjustment for change of variable from sampling from the half-angle distribution to sampling from the incident direction distribution can be derived with an observation about the relative angles involved.

Consider the spherical coordinate system oriented about $\omega_o$—see Figure 14.9. The differential solid angles $d\omega_i$ and $d\omega_H$ are $\sin \theta_id\theta_id\phi_i$ and $\sin \theta_Hd\theta_Hd\phi_H$, respectively.

$$\frac{d\omega_i}{d\omega_H} = \frac{\sin \theta_id\theta_id\phi_i}{\sin \theta_Hd\theta_Hd\phi_H}$$

Because $\omega_i$ is computed by reflecting $\omega_o$ about $\omega_H$, $\theta_i = 2\theta_H$. Thus,

$$\frac{d\omega_i}{d\omega_H} = \frac{\sin 2\theta_H2d\theta_Hd\phi_H}{\sin \theta_Hd\theta_Hd\phi_H} = \frac{4\cos \theta_H \sin \theta_H}{\sin \theta_H} = 4\cos \theta_H = 4(\omega_i \cdot H) = 4(\omega_o \cdot H)$$

Therefore, the pdf is $p(\theta) = p_h(\theta)4(\omega_o \cdot H)$.

After all that work, the sampling function is actually quite straightforward. We sample a $\cos \theta$ and a $\phi$ 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 $\omega_i$ is in lower hemisphere XXX

\textbf{BxDF Method Definitions}

```cpp
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 $\omega_o$ 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 $\omega_o$ onto $H$ (which is given by their dot product) to give us the direction $\omega_i$ above the surface.

\[ \text{Compute incident direction by reflecting about } H \equiv \]  
*wo = -wi + 2.f * Dot(wi, H) * H;

The weighting function is also straightforward.

\langle BxDF Method Definitions \rangle \equiv \]  
\text{Float Blinn::Weight(const Vector &wi, const Vector &wo) const { \} }  
\text{if (wi.z * wo.z < 0.) return 0;}  
\text{Vector H = (wi + wo).Hat(); \} }  
\text{Float costheta = fabsf(H.z);}  
\text{return ((exponent + 1.f) * powf(costheta, exponent)) / (4.f * Dot(wo, H));}  

\textbf{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, $\phi \in [0, \pi/2]$:

\[ \phi = \arctan \left( \sqrt{\frac{n_n + 1}{n_v + 1}} \tan \left( \frac{\pi \xi_1}{2} \right) \right) \]

and then

\[ \cos \theta = \xi_2^{(n_n \cos^2 \phi + n_v \sin^2 \phi + 1)} \]

More generally, see if $\xi_1 \in [0, .25), [.25, .5), [.5, .75), \text{or } [.75, 1)$. Then remap it to $[0, 1]$, sample as above, and add $0, \pi/2, \pi$, or $3\pi/2$ to $\phi$. 
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;
}

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));
}

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...
Monte Carlo Integration

\[ \text{BxDF Method Definitions} \]

\begin{verbatim}
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);
  } else {
    \{Sample lobe comp for Lafortune BRDF\}
  }
  *wt = Weight(wi, *wo);
  return f(wi, *wo);
}

\begin{verbatim}
Spectrum Lafortune::Weight(const Vector &wi, const Vector &wo) const {
  Float pdfSum = fabsf(wo.z) / M_PI;
  for (int i = 0; i < nLobes; ++i) {
    Float x, y, z;
    Vector lobeCenter = Vector(wi.x * x[i], wi.y * y[i], wi.z * z[i]).Hat();
    pdfSum += (e[i] + 1.f) * powf(max(0.f, Dot(wo, lobeCenter)), e[i]);
  }
  \{balance heuristic\}
  return pdfSum / (1.f + nLobes);
}
\end{verbatim}

\begin{verbatim}
XXX and now do FresnelBlend sampling stuff...
\end{verbatim}

\begin{verbatim}
\end{verbatim}
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:

$$\rho_{dh}(\omega) = \frac{1}{\pi} \int_{\Omega} f(\omega, \omega') d\omega'.$$

To estimate its value for a particular BSDF, 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.)

Computing the estimate is straightforward; we just importance sample the BSDF and apply the Monte Carlo estimator.
Estimate one term of $\rho_{dh}$

Vector $\mathbf{w}_i$;
Float $w_t$;
Spectrum $f = \text{sample}_f(w, &w_i, \text{samples}[2*i], \text{samples}[2*i+1], &w_t)$;
if ($w_t > 0.$) $r += f * \text{fabs}(w_i.z) / w_t$;

The hemispherical-hemispherical reflectance can be estimated similarly. Given

$$\rho_{hh} = \frac{1}{\pi} \int_{\Omega} \int_{\Omega} f(\omega_i, \omega_o) d\omega_i d\omega_o,$$

to estimate a term of $\rho_{hh}$, we need to sample two vectors, $\omega_i$ and $\omega_o$. We first sample $\omega_o$ 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)

```cpp
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 $\rho_{hh}$
    }
    return r / (M_PI*nSamples);
}
```

(Estimate one term of $\rho_{hh}$)

Vector $\mathbf{w}_i$, $\mathbf{w}_o$;
$\mathbf{w}_o = \text{UniformSampleHemisphere}(\text{samples}[4*i], \text{samples}[4*i+1])$;
Float $\text{weight}_o = 2.f * \text{M_PI}$, $\text{weight}_i$;
Spectrum $f = \text{sample}_f(\mathbf{w}_o, &\mathbf{w}_i, \text{samples}[4*i+2], \text{samples}[4*i+3], &\text{weight}_i)$;
if ($\text{weight}_i > 0.$) $r += f * \text{fabs}(w_i.z * w_o.z) / (\text{weight}_o * \text{weight}_i)$;

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.
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, $\int f(x)g(x)dx$, 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

$$E[\int f(x)g(x)] = \frac{1}{N} \left( \frac{N/2}{f(x_i)g(x_i)w_f(x_i)} + \sum_{i=N/2+1}^{N} f(x_i)g(x_i)w_g(x_i) \right)$$  \hspace{1cm} (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

$$E[\int f(x)g(x)] = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)g(x_i)}{w_c(x_i)}$$

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

```cpp
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...

\[
\begin{aligned}
\text{SpecularReflection Methods} &+&
\text{Spectrum sample}_f(\text{const Vector} & \text{wi}, \text{Vector} * \text{wo}, \text{Float} u1, \text{Float} u2, \\
& & \text{Float} * \text{wt}) \text{ const } \\
& & \text{const } \\
& & \{ \\
& & \text{*} \text{wt} = 1.; \\
& & \text{return } f_{\text{delta}}(\text{wi}, \text{wo});
\}
\end{aligned}
\]

\[
\begin{aligned}
\text{SpecularReflection Methods} &+&
\text{Float Weight(\text{const Vector} \& \text{wi}, \text{const Vector} \& \text{wo}) \text{ const } \\
& & \text{const } \\
& & \{ \\
& & \text{return } 0.;
\}
\end{aligned}
\]

\[
\begin{aligned}
\text{SpecularTransmission Methods} &+&
\text{Spectrum sample}_f(\text{const Vector} & \text{wi}, \text{Vector} * \text{wo}, \text{Float} u1, \text{Float} u2, \\
& & \text{Float} * \text{wt}) \text{ const } \\
& & \text{const } \\
& & \{ \\
& & \text{*} \text{wt} = 1.; \\
& & \text{return } f_{\text{delta}}(\text{wi}, \text{wo});
\}
\end{aligned}
\]

\[
\begin{aligned}
\text{SpecularTransmission Methods} &+&
\text{Float Weight(\text{const Vector} \& \text{wi}, \text{const Vector} \& \text{wo}) \text{ const } \\
& & \text{const } \\
& & \{ \\
& & \text{return } 0.;
\}
\end{aligned}
\]

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

\[
\begin{aligned}
\text{Light Interface} &+&
\text{virtual Spectrum Sample}_L(\text{const Point} & \text{P}, \text{Float} u1, \text{Float} u2, \text{Vector} * \text{wo}, \text{Float} * \text{weight}, \\
& & \text{bool} * \text{deltaLight}, \text{VisibilityTester} *) \text{ const } = 0;
\end{aligned}
\]

\[
\begin{aligned}
\text{Light Interface} &+&
\text{virtual Spectrum Sample}_L(\text{const Point} & \text{P}, \text{const Normal} & \text{N}, \text{Float} u1, \text{Float} u2, \\
& & \text{Vector} * \text{wo}, \text{Float} * \text{weight}, \text{bool} * \text{deltaLight}, \\
& & \text{VisibilityTester} *) \text{ const };
\end{aligned}
\]

Sample a ray for shooting energy from the light

\[
\begin{aligned}
\text{Light Interface} &+&
\text{virtual Spectrum Sample}_L(\text{const Scene} * \text{scene}, \text{Float} u1, \text{Float} u2, \\
& & \text{Float} u3, \text{Float} u4, \text{Ray} * \text{ray}, \text{Float} * \text{weight}, \\
& & \text{bool} * \text{deltaLight}) \text{ const } = 0;
\end{aligned}
\]
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.

\textit{Light Interface}:

\begin{verbatim}
virtual Float Weight(const Point &P, const Normal &N, const Vector &w) const {
  return Weight(P, w);
}
\end{verbatim}

\textit{Light Method Definitions}:

\begin{verbatim}
Spectrum Light::Sample_L(const Point &P, const Normal &N, 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;
}
\end{verbatim}

\textit{PointLight Method Definitions}:

\begin{verbatim}
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;
}
\end{verbatim}

\textbf{Delta function lights}

XXX Actually, this is wrong, since pointlight is based on intensity, not radiance. Effect is that radiance should be multiplied by a delta function. But then the weight should have one over a delta function, so it all cancels out... XXX

\textit{PointLight Method Definitions}:

\begin{verbatim}
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;
}
\end{verbatim}
\textbf{PointLight Method Definitions}\footnote{Float PointLight::Weight(const Point &, const Vector &) const {
 \hspace{10pt} return 0.;
}}

Spotlight is similar; just need to compute outgoing “radiance” differently. Plus, for sampling a direction for shooting, we can be clever and only sample directions in the spotlight cone.

\textbf{SpotLight Method Definitions}\footnote{Spectrum Spotlight::Sample_L(const Point &P, Float u1, Float u2, Vector *wo, Float *weight, bool *deltaLight, VisibilityTester *visibility) const {
 \hspace{10pt} *wo = (lightPos - P).Hat();
 \hspace{10pt} *weight = DistanceSquared(lightPos, P);
 \hspace{10pt} *deltaLight = true;
 \hspace{10pt} visibility->SetSegment(P, lightPos, CastsShadows);
 \hspace{10pt} return Intensity * Falloff(-*wo);
}}

\textbf{SpotLight Method Definitions}\footnote{Spectrum Spotlight::Sample_L(const Scene *scene, Float u1, Float u2, Float u3, Float u4, Ray *ray, Float *weight, bool *deltaLight) const {
 \hspace{10pt} ray->O = lightPos;
 \hspace{10pt} Vector v = UniformSampleCone(u1, u2, cosTotalWidth);
 \hspace{10pt} ray->D = LightToWorld(v);
 \hspace{10pt} *weight = UniformConeWeight(cosTotalWidth);
 \hspace{10pt} *deltaLight = true;
 \hspace{10pt} return Intensity * Falloff(ray->D);
}}

\textbf{GoniometricLight Method Definitions}\footnote{Spectrum GoniometricLight::Sample_L(const Point &P, Float u1, Float u2, Vector *wo, Float *weight, bool *deltaLight, VisibilityTester *visibility) const {
 \hspace{10pt} *wo = (lightPos - P).Hat();
 \hspace{10pt} *weight = DistanceSquared(lightPos, P);
 \hspace{10pt} *deltaLight = true;
 \hspace{10pt} visibility->SetSegment(P, lightPos, CastsShadows);
 \hspace{10pt} return Intensity * Scale(-*wo);
}}

\textbf{Same issues for ProjectionLight as for SpotLight...}
### GoniometricLight Method Definitions

```cpp
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);
}
```

### InfinitePointLight Method Definitions

```cpp
Float InfinitePointLight::Weight(const Point &, const Vector &) const {
  return 0.;
}
```

Distant light source.

```cpp
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...

```cpp
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

```cpp
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);
```
(Set ray origin and direction for infinite light ray)\(\equiv\)
\[
\text{ray->O = Pdisk + worldRadius * lightDir;}
\]
\[
\text{ray->D = -lightDir;}
\]

\(\text{InfinitePointLight Method Definitions}\)\(\equiv\)
\[
\text{Float InfinitePointLight::Weight(const Point &, const Vector &) const {}
\]
\[
\text{\quad return 0.;}
\]
\[
\text{}}
\]

\(\text{Infinite Area Lights}\)

Sample according to cosine weighting, but over the entire sphere around the hit point.

\(\text{InfiniteAreaLight Function Definitions}\)\(\equiv\)
\[
\text{Spectrum InfiniteAreaLight::Sample_L(const Point &P,}
\]
\[
\text{\quad \text{const Normal} &N, \text{Float} u1, \text{Float} u2,}
\]
\[
\text{\quad \text{Vector} *wo, \text{Float} *weight, \text{bool} *deltaLight,}
\]
\[
\text{\quad \text{VisibilityTester} *visibility) const {}
\]
\[
\text{\quad \text{Sample cosine-weighted direction on unit sphere}}
\]
\[
\text{\quad \text{Compute weight for cosine-weighted infinite light direction}}
\]
\[
\text{\quad \text{Transform direction to world space}}
\]
\[
\text{\quad \text{\quad *deltaLight = false;}}
\]
\[
\text{\quad \text{\quad // XXX yuck}}
\]
\[
\text{\quad \quad \text{visibility->SetRay(P, *wo, CastsShadows);}}
\]
\[
\text{\quad \quad return Le(Ray(P, *wo));}
\]
\[
\text{\quad \}}
\]

\(\text{Sample cosine-weighted direction on unit sphere}\)\(\equiv\)
\[
\text{Float x, y, z;}
\]
\[
\text{ConcentricSampleDisk(u1, u2, \&x, \&y);}\]
\[
\text{z = sqrtf(max(0.f, 1.f - x*x - y*y));}
\]
\[
\text{if (RandomFloat() < .5) z *= -1;}
\]
\[
\text{\quad *wo = Vector(x, y, z);}
\]

This is just like for cosine-weighted hemisphere sampling, except we’re doing cosine-weighted sphere...

\(\text{Compute weight for cosine-weighted infinite light direction}\)\(\equiv\)
\[
\text{\quad \quad *weight = fabsf(wo->z) / (2.f * M_PI);}
\]

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...

\(\text{Transform direction to world space}\)\(\equiv\)
\[
\text{Vector v1, v2;}
\]
\[
\text{CoordinateSystem(Vector(N).Hat(), \&v1, \&v2);}\]
\[
\text{\quad *wo = Vector(v1.x * wo->x + v2.x * wo->y + N.x * wo->z,}
\]
\[
\text{\quad \quad v1.y * wo->x + v2.y * wo->y + N.y * wo->z,}
\]
\[
\text{\quad \quad v1.z * wo->x + v2.z * wo->y + N.z * wo->z);}\]
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 $4\pi$, so...

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));
}

Float InfiniteAreaLight::Weight(const Point &, const Vector &) const {
  return 1.f / (4.f * M_PI);
}

XXX 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?

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 $p_1$ and $p_2$ 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;
}

Point worldCenter;
Float worldRadius;
scene->BoundingSphere(&worldCenter, &worldRadius);
Point p1 = worldCenter + worldRadius * UniformSampleSphere(u1, u2);
Point p2 = worldCenter + worldRadius * UniformSampleSphere(u3, u4);
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 Shapes to sample random points on their surfaces.

Sampling Disk Shapes 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...

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 Shapes to compute one over the probability density for choosing to sample a particular direction.

These two are with respect to solid angle...
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.

\( \text{Intersect sample ray with area light geometry} \) 

\[
\text{Weight}(P, \text{dir}) = \text{Area}() * \text{DistanceSquared}(P, \text{ray}(\text{ray.maxt})) / \text{fabsf}(<\text{dot}(\text{dgLight.Nn}, -\text{dirHat})>); 
\]

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:

\[
\frac{\partial \omega}{\partial A} = \frac{r^2}{\cos \theta_o} 
\]

where \( \theta_o \) 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.

\( \text{Convert light sample weight to solid angle measure} \) 

\[
\text{weight} = \text{Area}() * \text{DistanceSquared}(P, \text{ray}(\text{ray.maxt})) / \text{fabsf}(<\text{dot}(\text{dgLight.Nn}, -\text{dirHat})>); 
\]

With respect to area. Default assumes uniform sampling.
Uniform sampling on cylinders is straightforward; just pick a height and a \( \phi \) value uniformly, compute the corresponding point, and compute the

\begin{verbatim}
\textit{Cylinder Methods}\texttt{+}\texttt{=} \\
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);
}
\end{verbatim}

XXX handle partial spheres XXX

\begin{verbatim}
\textit{Sphere Methods}\texttt{+}\texttt{=} \\
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);
}
\end{verbatim}

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 \( \theta \) from the center vector \( \omega_c \) and then sampling a rotation angle \( \phi \) around the vector.

Figure 14.11: To sample points on a spherical light source, we can uniformly sample within the cone of directions around a central vector \( \omega_c \) with a angular spread of up to \( \theta \).
As seen from point being shaded, the sphere subtends an angle of

\[ \theta_{\text{max}} = \arcsin \left( \frac{r}{|P - c|} \right) = \arccos \sqrt{1 - \left( \frac{r}{|P - c|} \right)^2} \]

where \( r \) is the radius of the sphere and \( c \) is its center—see Figure 14.11.

\(<\text{Sphere Methods}\>++
Point Sphere::Sample(const Point &P,
    Float u1, Float u2, Normal *Ns) const {
  \(<\text{Compute coordinate system for sphere sampling}\>
  \(<\text{Sample uniformly on sphere if } P \text{ is inside it}\>
  \(<\text{Sample sphere uniformly inside subtended cone}\>
}

\(<\text{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);

\(<\text{Sample uniformly on sphere if } P \text{ is inside it}\>++
if (DistanceSquared(P, Pcenter) < radius*radius)
    return Sample(u1, u2, Ns);

\(<\text{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.

\(<\text{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.
\textit{AreaLight Function Definitions}\(\equiv\)

\begin{verbatim}
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);
}
\end{verbatim}

\textit{AreaLight Function Definitions}\(\equiv\)

\begin{verbatim}
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);
}
\end{verbatim}

\textit{AreaLight Function Definitions}\(\equiv\)

\begin{verbatim}
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);
}
\end{verbatim}

\textit{AreaLight Function Definitions}\(\equiv\)

\begin{verbatim}
Float AreaLight::Weight(const Point &P, const Normal &N, const Vector &w) const {
    return shape->Weight(P, N, w);
}
\end{verbatim}

\textit{AreaLight Function Definitions}\(\equiv\)

\begin{verbatim}
Float AreaLight::Weight(const Point &P, const Vector &w) const {
    return shape->Weight(P, w);
}
\end{verbatim}
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;

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:

\[
1 = c \int_{0}^{d} e^{-\alpha x} \, dx
\]

\[
= \frac{c}{\alpha} \left( e^{-\alpha d} - 1 \right)
\]

\[
= \frac{c}{\alpha} (1 - e^{-\alpha d})
\]

So

\[
c = \frac{\alpha}{1 - e^{-\alpha d}}.
\]

Following similar steps, we can now determine how to sample a distance \( d' \) given a uniform random number \( \xi \):

\[
\xi = \frac{\alpha}{1 - e^{-\alpha d}} \int_{0}^{d'} e^{-\alpha x} \, dx
\]

\[
= \frac{1 - e^{-\alpha d'}}{1 - e^{-\alpha d}}
\]

\[
\xi (1 - e^{-\alpha d}) = 1 - e^{-\alpha d'}
\]

\[
e^{-\alpha d'} = 1 - \xi (1 - e^{-\alpha d})
\]

\[
-\alpha d' = \log(1 - \xi (1 - e^{-\alpha d}))
\]

\[
d' = \frac{\log(1 - \xi (1 - e^{-\alpha d}))}{\alpha}
\]

To sample Henyey-Greenstein, Equation ??, it’s just:

\[
\cos \theta = -\frac{1}{2g} \left( 1 + g^2 - \left( \frac{1 - g^2}{1 - g + 2g \xi} \right)^2 \right)
\]

If \( g \neq 0 \), otherwise \( \cos \theta = 1 - 2\xi \)

XXX put it all together, show how you sample that, then sample \( \phi \), make a little coordinate system and you’re off....

\[
\text{(Foo)} \equiv
\]

\[
\text{double evalHG(double g, double costheta) \{ }
\]

\[
\quad \text{return (1 - g*g) / powf(1 + g*g - 2*g*costheta, 1.5); }
\]

\[
\text{\}}
\]

\[
\text{(Foo)} +\equiv
\]

\[
\text{double sampleHG(double g, double u, double *pdf) \{ }
\]

\[
\quad \text{if (fabsf(g) < 1e-5) \{ }
\]

\[
\quad \quad *pdf = 1.;
\]

\[
\quad \text{return 1.f - u * 2.; }
\]

\[
\quad \}
\]

\[
\quad \text{double cost = -1.f / (2.0 * g) * (1 + g*g - sqrt((1 - g*g)/(1-g+2*g*u))));}
\]

\[
\quad *pdf = evalHG(g, -cost);
\]

\[
\quad \text{return cost; }
\]

\[
\}
\]
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 compendium


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

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 SurfaceIntegrators, each providing a different level of accuracy in its modelling of light transport.

```c
#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"
#endif // TRANSPORT_H
```

This code is part of the source code that implements the integrator class. The integrator is responsible for evaluating the rendering equation along the ray.
There is just a single function that SurfaceIntegrators must implement, \( L \), which returns the radiance along the ray. The parameters are the following:

1. \textit{scene} : a pointer to the \texttt{Scene} being rendered. The integrator will query the scene for information about the lights and geometry present, etc.

2. \textit{ray} : the ray along which the scattered radiance should be evaluated.

3. \textit{sample} : a pointer to a \texttt{Sample} generated by the \texttt{Sampler} for this ray; some integrators will use some of its entries for Monte Carlo sampling.

4. \textit{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.

\( L \) returns a \texttt{Spectrum} that holds the radiance along the ray.

\begin{verbatim}
virtual Spectrum L(const Scene *scene,
    const RayDifferential &ray, const Sample *sample,
    Float *alpha) const = 0;
\end{verbatim}

\subsection{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 \textit{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 \textit{rendering equation}) is built on the assumptions that:
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 \( \mathbf{\omega} \), \( L_o(x, \mathbf{\omega}) \). 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:

\[
L_o(x, \mathbf{\omega}) = L_e(x, \mathbf{\omega}) + \int_{S^2} L_i(x, \mathbf{\hat{\omega}}_i) f(\mathbf{\hat{\omega}}_i, \mathbf{\omega}_o) \cos \theta_i |d\mathbf{\hat{\omega}}_i|.
\]  

(15.1.1)

Because there are no atmospheric effects, radiance is constant along rays through \textit{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 \textit{ray-casting function} \( t(x, \mathbf{w}) \) as returning the first surface point \( x' \) intersected by a ray from \( x \) in the direction \( \mathbf{\omega} \), we can write the incident radiance at \( x \) in terms of outgoing radiance at \( x' \):

\[
L_i(x, \mathbf{\omega}) = L_o(t(x, \mathbf{\omega}), -\mathbf{\omega}).
\]

(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:

\[
L(x, \mathbf{\omega}) = L_e(x, \mathbf{\omega}) + \int_{S^2} L(t(x, \mathbf{\hat{\omega}}_i), -\mathbf{\hat{\omega}}_i) f(\mathbf{\hat{\omega}}_i, \mathbf{\omega}_o) \cos \theta_i |d\mathbf{\hat{\omega}}_i|,
\]  

(15.1.1)
Implications of delta distributions

implicit delta distributions in BxDF and weight values, why they cancel out but also anihilate other stuff...

15.2 Whitted Integrator

\[ \langle \text{whitted.cc} \rangle \equiv \]
\[ \langle \text{Source Code Copyright} \rangle \]
\#include "lrt.h"
\#include "transport.h"
\#include "scene.h"
\langle \text{WhittedIntegrator Declarations} \rangle
\langle \text{WhittedIntegrator Method Definitions} \rangle

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

\[
\begin{align*}
f(x) &= f_1(x) + f_2(x) \\
g(x) &= g_1(x) + g_2(x)
\end{align*}
\]

then

\[
\int f(x)g(x)dx = \int (f_1(x) + f_2(x))(g_1(x) + g_2(x))dx = \int f_1(x)g_1(x)dx + \int f_2(x)g_1(x)dx + \int f_1(x)g_2(x)dx + \int f_2(x)g_2(x)dx.
\]

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

\[
L(x, \vec{\omega}) = L_\omega(x, w) + \int_{S^2} L_\Delta(t(x, \vec{\omega}_i) - \vec{\omega}_o) f_\Delta(\vec{\omega}_i, \vec{\omega}_o) \cos \theta_I \text{d}\vec{\omega}_i + \int_{S^2} L(t(x, \vec{\omega}_i), -\vec{\omega}_o) f_\Delta(\vec{\omega}_i, \vec{\omega}_o) \cos \theta_I \text{d}\vec{\omega}_i
\]

\[
\int_{S^2} L_\Delta(t(x, \vec{\omega}_i), -\vec{\omega}_o) f(\vec{\omega}_i, \vec{\omega}_o) \cos \theta_I \text{d}\vec{\omega}_i + \int_{S^2} L(t(x, \vec{\omega}_i), -\vec{\omega}_o) f(\vec{\omega}_i, \vec{\omega}_o) \cos \theta_I \text{d}\vec{\omega}_i
\]
where terms with a \( \Delta \) subscript have delta components 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.

\[
\int_{S^2} L(t(x, \vec{\omega}_i), -\vec{\omega}_i) f_{\Delta}(\vec{\omega}_i, \vec{\omega}_o) | \cos \theta_i | d\vec{\omega}_i = \sum_{\text{specular}} f(\vec{\omega}, \vec{\omega}_i) L(x, \vec{\omega}_i)
\]

Delta light source integral also a sum:

\[
\int_{S^2} L_{\Delta}(t(x, \vec{\omega}_i), -\vec{\omega}_i) f(\vec{\omega}_i, \vec{\omega}_o) | \cos \theta_i | d\vec{\omega}_i = \sum_{\text{lights}} f(\vec{\omega}, \vec{\omega}_i) | \cos \theta_i | \frac{l_{\text{light}}}{{\| x_{\text{light}} - x \|}^2} V(x_{\text{light}}, x),
\]

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...

\[\langle \text{WhittedIntegrator Declarations} \rangle \equiv \]
\[
\text{class WhittedIntegrator : public SurfaceIntegrator} \{ \\
\text{public:} \{ \\
\text{WhittedIntegrator Methods} \} \\
\text{private:} \{ \\
\text{WhittedIntegrator Private Data} \} \\
\};
\]

\[\langle \text{WhittedIntegrator Private Data} \rangle \equiv \]
\[
\text{int maxDepth;} \\
\text{mutable int rayDepth;}
\]

\[\langle \text{WhittedIntegrator Methods} \rangle + \equiv \]
\[
\text{Sample *WhittedIntegrator::AllocateSample(const Scene *scene) const} \{ \\
\text{vector<int> none;} \\
\text{return new Sample(none, none);} \\
\}
\]

### 15.3 Direct Lighting Integrator

\[\langle \text{directlighting.cc*} \rangle \equiv \]
\[\langle \text{Source Code Copyright} \rangle \]
\[
\text{#include "lrt.h"} \\
\text{#include "transport.h"} \\
\text{#include "scene.h"} \\
\langle \text{DirectLighting Declarations} \rangle \\
\langle \text{DirectLighting Method Definitions} \rangle
\]

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.

\[ \langle \text{DirectLighting Declarations} \rangle \equiv \]
\begin{verbatim}
class DirectLighting : public SurfaceIntegrator {
public:
\langle DirectLighting Methods \rangle
private:
\langle DirectLighting Private Data \rangle
};
\end{verbatim}

\[ \langle \text{DirectLighting Methods} \rangle \equiv \]
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);
}

\[ L(x, \omega) = L_e(x, \omega) + \int_{\Omega} f_r(\omega_i, \omega_j) L_d(x, \omega_i) \cos \theta |d\omega_i \quad (15.3.2) \]

where \( L_d(x, \omega_i) \) 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, \( \langle \text{Compute direct lighting at hit point} \rangle \).

\[ \langle \text{Compute direct lighting for DirectLighting integrator} \rangle \equiv \]
const Point &P = surf.dgShading.P;
const Normal &N = surf.dgShading.Nn;
if (scene->lights.size() > 0) {
  Vector wo = -ray.D.Hat();
  \langle Apply direct lighting strategy \rangle
}

Context for this fragment:

- \( P = \text{surf.dgShading.P} \);
- \( N = \text{surf.dgShading.Nn} \);
- \( \text{bsdf is initialized to BSDF at the hit point} \)

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.
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 $i$th 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/n_{lights} of selecting each particular light. Then, we estimate direct lighting for only that one light, weighting the result by a factor of n_{lights} 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
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...

$$\bar{y} = (1 - \alpha)\bar{y} + \alpha \bar{y}_{i-1}$$

where $\alpha$ 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) $+$

```plaintext
mutable Float *avgY, *avgYsample, *cdf;
mutable Float overallAvgY;
```

Until we find a light source that contributes reflected light, $\text{overallAvgY}$ 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...

(Integrator Utility Functions) $+$

```plaintext
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)
    $L /= \text{lightSampleWeight}$;
  }
  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

```java
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

```java
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;
```

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

```java
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

```java
Float luminance = L.Luminance();
avgY[lightNum] =
    ExponentialAverage(avgY[lightNum], luminance, .99f);
overallAvgY =
    ExponentialAverage(overallAvgY, luminance, .999f);
```

(Global Inline Functions)

```java
inline Float ExponentialAverage(Float avg, Float val, Float alpha) {
    return (1.f - alpha) * val + alpha * avg;
}
```
Estimating the direct lighting integral

Having chosen a particular light to estimate direct lighting from, we need to estimate the value of the integral

$$\int_{\Omega} f_i(\vec{\omega}, \vec{\omega}_i) L_d(x, \vec{\omega}_i) |\cos \theta_i| d\vec{\omega}_i$$

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

$$\int f(x)g(x)dx,$$

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

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}

```c
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();
}
```

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.
\[\text{Sample BSDF with multiple importance sampling} \]
\begin{verbatim}
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;
}
\end{verbatim}

\section*{15.4 Integral Over Paths}

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 \textit{three-point form} of the light transport equation. The integral over incident directions \(\omega_i\) 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

\[L(x' \rightarrow x'') = L(x', \mathbf{\Omega}),\]

if \(x'\) and \(x''\) are mutually visible and \(\mathbf{\Omega} = \overrightarrow{x''} - \overrightarrow{x'}\). We can also write the BSDF at \(x'\) as

\[f(x \rightarrow x' \rightarrow x'') = f(x', \mathbf{\Omega}_l, \mathbf{\Omega}_o),\]
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 $\bar{\omega}_i = \hat{x} - \hat{x}'$ and $\bar{\omega}_{o} = \hat{x}' - \hat{x}'$. 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

$$L(x' \to x'') = \int_{A} L(x \to x') f(x \to x' \to x'') G(x \leftrightarrow x') dA(x),$$

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:

$$G(x \leftrightarrow x') = V(x,x') \frac{\cos \theta_i \cos \theta'}{\|x - x'\|^2}.$$

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 $x_0 - x$.

$$L(x_0 \to x) = L_o(x_0 \to x) +$$

$$\int_{A} L_o(x_1 \to x_0) f(x_1 \to x_0 \to x) G(x_1 \leftrightarrow x_0) dA(x_1) +$$

$$\int_{A^2} L_o(x_2 \to x_1) f(x_2 \to x_1 \to x_0) G(x_2 \leftrightarrow x_1)$$

$$f(x_1 \to x_0 \to x) G(x_1 \leftrightarrow x') dA(x_2) dA(x_1) + \cdots$$

The pattern becomes clear, and we have

$$L(x' \to x) = L_o(x' \to x) + \sum_{i=1}^{\infty} P_i(\bar{x})$$

(15.4.3)

where $P_i(\bar{x})$ gives the light scattered over paths with $i$ vertices through the scene:

$$P_i(\bar{x}) = \int_{A} \cdots \int_{A} L_o(x_i \to x_{i-1}) G(x_i \leftrightarrow x_{i-1}) \left( \prod_{j=1}^{i-1} f(x_{j-1} \to x_j \to x_{j+1}) G(x_j \leftrightarrow x_{j+1}) \right) dA(x_1) \cdots dA(x_i).$$
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

\( \langle \text{path.cc} \rangle \equiv \)

\( \langle \text{Source Code Copyright} \rangle \)

\#include "lrt.h"
\#include "transport.h"
\#include "scene.h"

\( \langle \text{PathIntegrator Declarations} \rangle \)

\( \langle \text{PathIntegrator Method Definitions} \rangle \)

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

\[
L(x' \rightarrow x) = L_e(x' \rightarrow x) + \sum_{i=1}^{\infty} P_i(\n x)
\]

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 \( P_i(\n x) \) terms

2. Given a particular \( P_i(\n x) \) 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 $i$th 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 $i$th 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 $i$th 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

\[
\begin{align*}
p_1 &= 1 \\
p_i &= p_i^{1} p_{i-1}
\end{align*}
\]

where $p_i^{1}$, the probability that sampling continues after the $i$th term, is $1/4$.

Thus, in pseudo-code, we can estimate the sum by:

```plaintext
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 $i$th 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 $i$th term in terms of the $i-1$st term and only considering the $i$th 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 $p_i^{1}$ 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
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.)

Looking at the form of $P_i(\bar{x})$, the most natural thing to do is to sample according to the differential area of objects in the scene, such that it’s easily 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 $j$th object should be

$$p_j = \frac{A_j}{\sum_k A_k}.$$

Then, given a method to sample a point on the $j$th 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

$$\frac{A_j}{\sum_k A_k A_j}.$$

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 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 \( P_i(\hat{x}) \); 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 objects contributed to most of the 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 contribution 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 \( f(x_{i-1} \to x_i \to x_{i+1}) \) 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):

\[
p_A = p_0 \frac{\| x_i - x_{i+1} \|^2}{\cos \theta}
\]

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:

\[
\text{P (n)} \{ \\
\text{throughput} = 1; \\
\text{wo} = -\text{eyeRay.D.Hat}(); \\
x[0] = \text{trace(eyeRay)}; \\
\text{for (i = 1; i < n; ++i)} \{ \\
\text{wi} = \text{sample_BSDF}(x[i-1], wo, &wi); \\
\text{throughput} *= f(x[i-1], wo, wi) * \\
\text{sampleWeight}(x[i-1], wo, wi) * |\cos(\theta)|; \\
\}
\]

\[
\]
\[ x[i] = \text{trace}(x[i-1], w_i); \]
\[ x[n] = \text{sampleLights}(); \]
\[ \text{return } L_e(x[n], (x[n-1] - x[n])) \times \text{throughput} \times \text{lightWeight}(x[n]); \]

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 \( P_i(\hat{x}) \) term that we evaluate, rather than \( i + 1 \) rays. This introduces correlation among all of the \( P_i(\hat{x}) \) 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 *sample, Float *alpha) const;
    Sample *AllocateSample(const Scene *scene) const;
};

\(<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 &ray, const Sample *sample, Float *alpha) const {
    int pathLength = 0;
    while (1) {
        \(<\text{Find next vertex of path}>\)
        \(<\text{Add emitted light for first segment only}>\)
        \(<\text{Evaluate BSDF at hit point}>\)
        \(<\text{Randomly sample illumination from one light source}>\)
        \(<\text{Randomly sample BSDF to get new path direction}>\)
        \(<\text{Clean up from integration}>\)
        \(<\text{Possibly terminate the path}>\)
        ++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);

XXXXX So here it’s wasteful to sample BSDF twice with same random numbers,
once for direct lighting, once for path tracing...
Randomly sample BSDF to get new path direction

\begin{verbatim}
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);
\end{verbatim}

Possibly terminate the path

\begin{verbatim}
if (pathLength > 3) {
    Float continueProbability = .2f;
    if (RandomFloat() > continueProbability)
        break;
    pathThroughput /= continueProbability;
}\end{verbatim}

15.6 Bidirectional Path Tracing

\begin{verbatim}
#include "lrt.h"
#include "transport.h"
#include "scene.h"
#include "mc.h"
\end{verbatim}

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
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

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 sub-path $x_1, x_2, \ldots, x_i$ 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

$$\bar{x} = x, x_1, x_2, \ldots, x_i, y_1, \ldots, y_j.$$  

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 $x_1, x_2, \ldots, x_i, y_1, \ldots, y_j$, 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 $i \cdot j$ unique paths can be constructed from them, ranging in length from 2 to $i + j$ vertices long. XXX number of paths of length $n = \ldots$ 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 that happened 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.

```cpp
class BidirIntegrator : public SurfaceIntegrator {
  public:
    Spectrum L(const Scene *scene, const RayDifferential &ray, const Sample *sample, Float *alpha) const;
    Sample *AllocateSample(const Scene *scene) const;

  private:
    // BidirIntegrator Private Methods
};
```

```cpp
Bidirectional Method Definitions
Sample *BidirIntegrator::AllocateSample(const Scene *scene) const {
  Assert(1 == 0);
  return NULL;
}
```

```cpp
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 return
}

<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);

<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:

```cpp
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:

```cpp
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...

Bidirectional Method Definitions:

```cpp
Spectrum BidirIntegrator::evalPath(const Scene *scene, BidirVertex *eye, int nEye, 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; // XXXXX
    }
    return L;
}
```

Bidirectional Method Definitions:

```cpp
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

```cpp
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 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 corresponding 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 next section, have been two successful biased methods for light transport.

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 little 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 techniques 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 compute 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 illumination 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
sec. 15.7] Photon Mapping 471

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.

\begin{verbatim}
 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 appearance. 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;
}

Photonmap Method Definitions

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);
}

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() > 10000)) {
    cerr << "No luck shooting photons!!" << endl;
    unsuccessfulShooting = true;
    break;
}

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:

```cpp
int nLights = int(scene->lights.size());
Light *light = scene->lights[RandomInt() % nLights];
Float lightWeight = 1.f / nLights;
```

Generate photonRay from light source:

```cpp
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:

```cpp
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);
    if (hitSpecular && photonBSDF->NumComponents() > photonBSDF->NumSpecular())
        photons.push_back(PhotonData(photonSurf.dgGeom, L,
                                   -photonRay.D.Hat()));
}
```

XXX what to do about shading normals?

Handle non-specular photon hit:

```cpp
if (hitSpecular && photonBSDF->NumComponents() > photonBSDF->NumSpecular())
    photons.push_back(PhotonData(photonSurf.dgGeom, L,
                                -photonRay.D.Hat()));
```

Photonmap Local Declarations:

```cpp
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;
};
```
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;
}

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
}
\[ \textbf{Photonmap Local Declarations} \]

```cpp
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;
};
```

\[ \textbf{Photonmap Local Declarations} \]

```cpp
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;
};
```

\[ \textbf{Photonmap Method Definitions} \]

```cpp
PhotonProcess::PhotonProcess(u_int mp, const Point &p, const Normal &n) :
    P(p), N(n) {
    maxPhotons = mp;
}
```

\[ \textbf{Photonmap Method Definitions} \]

```cpp
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) {
        \{ \textit{Add photon to unordered array of photons} \}
    } else {
        \{ \textit{Remove most distant photon from heap and add new photon} \}
    }
}
```
Add photon to unordered array of photons

\[
\text{photons}.\text{push\_back}(\text{ClosePhoton}(\text{\&photon, d2}));
\]
if (photons.size() == maxPhotons) {
    \text{std::make\_heap(photons.begin(), photons.end());}
    maxDist = \sqrt{\text{photons[0].maxDist2}};
}

Remove most distant photon from heap and add new photon

\[
\text{std::pop\_heap(photons.begin(), photons.end());}
\text{photons[maxPhotons-1] = ClosePhoton(\text{\&photon, d2});}
\text{std::push\_heap(photons.begin(), photons.end());}
maxDist = \sqrt{\text{photons[0].maxDist2}};
\]

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

\[
\text{vector<ClosePhoton> \&photons = proc.photons;}
\]
if (photons.size() > 0) {
    \text{(Compute photon scale factor with density estimation)}
    \text{for (\text{u\_int i = 0; i < photons.size(); ++i})}
    \text{L += scale * bsdf->f(wo, photons[i].data->wi) *}
    \text{fabsf(Dot(photons[i].data->wi, N)) *}
    \text{photons[i].data->dE;}
}

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

\[
\text{Float scale = photons.size() / (nLookup * M\_PI * md * md);}
\]

15.8 Irradiance Caching
then uses them to compute indirect lighting. Recall that irradiance is
\[
E(x) = \int_{H^2} L(x, \mathbf{\omega}) |\cos \theta| d\mathbf{\omega}.
\]

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.

\textit{IrradianceCache Declarations}\\
\begin{verbatim}
class IrradianceCache : public SurfaceIntegrator {
  public:
    IrradianceCache(int md, Float maxerr, int nsamples,
                     bool ss);
    Spectrum L(const Scene *scene, const RayDifferential &ray,
                const Sample *sample, Float *alpha) const;
    Sample *AllocateSample(const Scene *scene) const;
  private:
    IrradianceCache Private Data
    IrradianceCache Private Methods
};
\end{verbatim}

\textit{IrradianceCache Method Definitions}\\
\begin{verbatim}
IrradianceCache::IrradianceCache(int md, Float maxerr, int ns,
                                  bool ss) {
  maxDepth = md;
  maxError = maxerr;
  nSamples = ns;
  showSamples = ss;
  rayDepth = 0;
  indirectDepth = 0;
}
\end{verbatim}

\textit{IrradianceCache Private Data}\\
Float maxError;
bool showSamples;
int nSamples;
mutable int rayDepth, indirectDepth;
int maxDepth;

\textit{IrradianceCachee Method Definitions}\\
\begin{verbatim}
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);
}
\end{verbatim}
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);

Recall the scattering equation, 5.4.8. Take the reflection-only part of it and you have
\[
L_o(x, \bar{\omega}_o) = \int_{S^2} L_i(x, \bar{\omega}_i) f(\bar{\omega}_i, \bar{\omega}_o) \cos \theta_i |d\bar{\omega}_i|
\]

Here we will make the approximation
\[
L_o(x, \bar{\omega}_o) \approx \left( \int_{S^2} L_i(x, \bar{\omega}_i) |\cos \theta_i| |d\bar{\omega}_i| \right) \left( \int_{S^2} f(\bar{\omega}_i, \bar{\omega}_o) |d\bar{\omega}_i| \right) = \frac{E(x) \rho_{dh}(\bar{\omega}_o)}{4\pi}
\]

Where \(E\) denotes irradiance, as defined in Equation 5.2.5 and \(\rho_{dh}\) 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

\[ E = \int_{S^2} L_i(x, \vec{\theta}_i) |\cos \theta_i| d\vec{\theta}_i. \] (15.8.4)

Determine how many samples to take for irradiance estimate

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 \( \theta \) is the ray’s angle with the surface normal. However, because 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 \( \pi \).

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.

Update irradiance statistics for rays traced

\[ E += M_{\pi} \cdot \text{scene->L(r)}; \]
\[ \text{invSumDists} += 1.0 / (r.maxt \cdot \text{r.D.Length()}); \]
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 computed");
++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;
};
<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 Private Data> +≡

mutable Octree<IrradSample, IrradProcess> *octree;
XXX need to delete it...

<IrradianceCache Constructor Implementation> ≡

octree = NULL;

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));

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, radi-
ance 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:

\[
S(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \int_{\mathbb{S}^2} p(x, \vec{\omega} \rightarrow \vec{\omega}') L_i(x, \vec{\omega}') d\vec{\omega}'
\]

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; ?):

\[
\frac{\partial}{\partial t} L(x') = -\sigma_t L(x') + S(x')
\] (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

\[
L(x, \vec{\omega}) = \int_0^\infty T_r(x \rightarrow x + t \vec{\omega}) S(x + t \vec{\omega}, \vec{\omega}) dt.
\] (15.9.6)

More generally, if there are reflecting and/or emitting surfaces in the scene, we have:

\[
L(x, \vec{\omega}) = T_r(x \rightarrow x')(L_e(x_0, -\vec{\omega}) + L_o(x_0, -\vec{\omega})) + \int_0^{t'} T_r(x \rightarrow x + t \vec{\omega}) S(x + t \vec{\omega}, \vec{\omega}) dt
\] (15.9.7)

where \( t' \) is the distance along the ray to the first surface, \( x' \) is the point on the surface, \( x' = x + t' \vec{\omega} \), \( L_e \) is the emitted radiance from the surface, and \( L_o \) 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

\[\langle \text{Volume Scattering Declarations} \rangle + \Xi\]

```cpp
class VolumeIntegrator {
   public:
      \langle VolumeIntegrator Methods \rangle
   };

\langle VolumeIntegrator Methods \rangle + \Xi
   virtual Spectrum Transmittance(const Scene *, const Ray &ray, const Sample *sample, Float *alpha) const = 0;
```
Null Integrator

```cpp
#include "volume.h"

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.);
    }
};
```

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...

```cpp
#include "scene.h"

class EmissionIntegrator : public VolumeIntegrator {
public:
    EmissionIntegrator(int ns) { nSamples = ns; }

private:
    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
        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));
    }
    return Lv;
}

Do emission-only volume integration in vr

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
        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));
    }
    return Lv;
}

Do emission-only volume integration in vr

Single Scattering Integrator

Will re-use some fragments from emission-only integrator...

Source Code Copyright

#include "volume.h"
#include "scene.h"

SingleScattering Declarations

SingleScattering Function Definitions
class SingleScattering : public VolumeIntegrator {
    
    SingleScattering(int ns) { nSamples = ns; }

private:
    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
        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));
    }
    return Lv;
}

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

```cpp
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

```cpp
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);
```

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)
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 energy 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 differen-
tial 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 direc-
tions. Modify these interfaces so that the BSDF’s value is computed before
the light traces the visibility ray. How much does this speed up $\text{lrt}$?

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. Ten-
tatively 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 impor-
tant? 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 volumes

15.12 bidir for lighting in volumes, cite path integral generalization to volumes by the volume metropolis guys
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 Stroustrup 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
We will now define some of the assorted utility routines that were used throughout the system. These routines, though key to \texttt{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 \texttt{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 \texttt{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.)
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 template class, a vector of ints (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:

```cpp
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 vectors 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 used in conjunction with the [] operator to access items in the vector directly:

```cpp
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 can 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:

```cpp
int i = 0, Foo *foop = NULL;
pair<Foo *, int> p = make_pair(foop, i);
p.first = new Foo;
```

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,

```cpp
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:

```cpp
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.

```cpp
// Setup printf format
#else __GNUG__
#define PRINTF_FORMAT __attribute__ ((__format__ (__printf__, 1, 2)))
#endif
```

```cpp
#define PRINTF_FORMAT
#endif // __GNUG__
```
Now we can declare the four error reporting functions, using PRINTF_FORMAT if available.

(Global Function Declarations) +

Set up 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
```

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 if the error was a big one.
\[\text{Report error}\]  
\[
\text{switch (disposition) \{} \\
\text{case LRT_ERROR_IGNORE:} \\
\text{\hspace{1em} return;} \\
\text{case LRT_ERROR_CONTINUE:} \\
\text{\hspace{1em} fprintf(stderr, "\%s: %s\n", message, errorBuf);} \\
\text{\hspace{1em} \langle Print scene file and line number, if appropriate \rangle} \\
\text{\hspace{1em} break;} \\
\text{case LRT_ERROR_ABORT:} \\
\text{\hspace{1em} fprintf(stderr, "\%s: %s\n", message, errorBuf);} \\
\text{\hspace{1em} \langle Print scene file and line number, if appropriate \rangle} \\
\text{\hspace{1em} abort();} \\
\text{\} \]

\[\text{extern int line_num;} \]
\[
\text{if (line_num != 0) \{} \\
\text{\hspace{1em} extern string current_file;} \\
\text{\hspace{1em} fprintf(stderr, "\%d, file %s\n", line_num,} \\
\text{\hspace{1em} \hspace{1em} current_file.c_str());} \\
\text{\} } \\
\]

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 \text{va_start} is called, the \text{args} variable encapsulates information about the remaining arguments to the function. However, rather than calling the \text{va_arg} function to examine the subsequent arguments, we just pass the \text{args} variable into \text{processError}. It then passes it in to \text{vsprintf}, which handles unpacking the arguments.

\[\text{\langle Error Reporting Functions\rangle} + \]
\[
\text{void Info(const char *format, ...) \{} \\
\text{\hspace{1em} va_list args;} \\
\text{\hspace{1em} va_start(args, format);} \\
\text{\hspace{1em} processError(format, args, "Notice", LRT_ERROR_CONTINUE);} \\
\text{\hspace{1em} va_end(args);} \\
\text{\} } \\
\]

\[\text{\langle Error Reporting Functions\rangle} + \]
\[
\text{void Warning(const char *format, ...) \{} \\
\text{\hspace{1em} va_list args;} \\
\text{\hspace{1em} va_start(args, format);} \\
\text{\hspace{1em} processError(format, args, "Warning", LRT_ERROR_CONTINUE);} \\
\text{\hspace{1em} va_end(args);} \\
\text{\} } \\
\]
void Error(const char *format, ...) {
    va_list args;
    va_start(args, format);
    processError(format, args, "Error", LRT_ERROR_CONTINUE);
    va_end(args);
}

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.

#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

Reporting Progress

struct ProgressReporter {
    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;
    }

#include <sys/time.h>
\langle \text{ProgressReporter Data} \rangle \equiv 
\begin{align*}
&\text{const int width, frequency, total;} \\
&\text{mutable int count, nPlusses;} \\
&\text{struct timeval start;} \\
&\text{string title;}
\end{align*}

\langle \text{ProgressReporter Methods} \rangle + \equiv 
\begin{align*}
&\text{void operator() (FILE *file) const} \\
&\quad\{ \\
&\quad\quad\text{if (count-- == 0) } \\
&\quad\quad\quad\text{count = frequency;} \\
&\quad\quad\quad\langle \text{Update progress plus signs} \rangle \\
&\quad\quad\quad\langle \text{Update elapsed time and estimated time to completion} \rangle \\
&\quad\}
\end{align*}

\langle \text{Update progress plus signs} \rangle \equiv
\begin{align*}
&\text{fprintf(file, ”\r%s: [”, title.c_str());} \\
&\quad\text{+nPlusses;} \\
&\quad\text{for (int i = 0; i < nPlusses; ++i)} \\
&\quad\quad\text{fprintf(file, ”+”);} \\
&\quad\text{for (int i = 0; i < width - nPlusses; ++i)} \\
&\quad\quad\text{fprintf(file, ” “);} \\
&\quad\text{fprintf(file, ”]”);} \\
\end{align*}

\langle \text{Update elapsed time and estimated time to completion} \rangle \equiv
\begin{align*}
&\text{struct timeval now;} \\
&\text{gettimeofday(&now, NULL);} \\
&\text{Float percentDone = (Float)nPlusses / (Float)width;} \\
&\text{Float seconds = now.tv_sec - start.tv_sec +} \\
&\quad\text{(now.tv_usec - start.tv_usec) / 1e6f;} \\
&\text{Float estRemaining = seconds / percentDone - seconds;} \\
&\text{fprintf(file, ” (%.2fs|%.2fs) “, seconds, max(0.f, estRemaining));}
\end{align*}

\textbf{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:

- \textit{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.

- \textit{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.

\[ \text{Global Type Declarations} \]
typedef double StatsCounterType;

\[ \text{Statistics Definitions} \]
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.

\[ \text{Statistics Functions} \]
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 StatTrackers are stored in a static vector.

\[ \text{Statistics Definitions} \]
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 StatTrackers 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

```c++
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);
}
```

Global Classes

```c++
class StatsCounter {
public:
    StatsCounter(const string &category, const string &name);
private:
    StatsCounterType num;
};
```

Statistics Functions

```c++
StatsCounter::StatsCounter(const string &category, const string &name) {
    num = 0;
    addTracker(new StatTracker(category, name, &num));
}
```

```c++
void operator++() { ++num; }
void operator++(int) { ++num; }
```

```c++
void stat_max(StatsCounterType val) { num = max(val, num); }
void stat_min(StatsCounterType val) { num = min(val, num); }
operator int() { return (int)num; }
```

Global Classes

```c++
class StatsRatio {
public:
    StatsRatio(const string &category, const string &name, bool percent = true);
private:
    StatsRatioType num;
};
```
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);
    }
};

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");

(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.

```c
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.

```c
(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;
}
```

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.

### A.4 Memory Management

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.

```c
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.

```c
(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;
}
```

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...

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 \texttt{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.

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

```c
void *AllocL1CacheAligned(size_t size) {
    #ifndef L1_CACHE_LINE_SIZE
    #define L1_CACHE_LINE_SIZE
    #define L1_CACHE_LINE_SIZE 32
    #endif
    return memalign(L1_CACHE_LINE_SIZE, size);
}
```
void *AllocL2CacheAligned(size_t size) {
    #ifndef L2_CACHE_LINE_SIZE
    #define L2_CACHE_LINE_SIZE 256
    #endif
    return memalign(L2_CACHE_LINE_SIZE, size);
}

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.

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: (unions, 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)++

```cpp
template <class T> class Reference {
public:
  // Reference constructors
  Reference(T *p = NULL) {
    ptr = p;
    if (ptr) ++ptr->nReferences;
  }
  // Reference assignment operators
  Reference &operator=(const Reference<T> &r) {
    ptr = r.ptr;
    if (ptr) ++ptr->nReferences;
    return *this;
  }
  Reference &operator=(T *p) {
    if (p) p->nReferences++;
    if (ptr && --ptr->nReferences == 0) delete ptr;
    ptr = p;
    return *this;
  }
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)++

```cpp
Reference(T *p = NULL) {
  ptr = p;
  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 referring to if it only has one reference.

(Reference assignment operators)++

```cpp
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)+

```cpp
Reference &operator=(T *p) {
  if (p) p->nReferences++;
  if (ptr && --ptr->nReferences == 0) delete ptr;
  ptr = p;
  return *this;
}
```
\( \text{Reference destructor} \equiv \)

```cpp
~Reference() {
    if (ptr && --ptr->nReferences == 0)
        delete ptr;
}
```

Finally, a bit of C++ trickery so that we can use \( \rightarrow \) to call methods of objects we hold references to, etc. The \texttt{operator bool} allows us to check to see if a reference points to a \texttt{NULL} object with code like \( [[\text{if } (!r) ...]] \).

\( \text{Reference operators} \equiv \)

```cpp
T *operator->() { return ptr; }
const T *operator->() const { return ptr; }
operator bool() const { return ptr != NULL; }
```

A.5 Matrix Routines

2x2 Linear Systems

Solve \( Ax = B \).

\( \text{Matrix Method Definitions} \equiv \)

```cpp
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 \texttt{Matrix4x4} structure provides a low-level representation of 4 by 4 matrices. It is an integral part of the \texttt{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 \texttt{Matrix4x4}s, so that the Transform class only needs to hold \texttt{Matrix4x4} references, rather than holding the much larger complete matrices.

\( \text{Global Classes} \equiv \)

```cpp
struct Matrix4x4 : public ReferenceCounted<Matrix4x4> {
    \textit{Matrix4x4 Methods}
    Float m[4][4];
};
```

The default constructor sets the matrix to the identity matrix.
Matrix 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(Float mat[4][4]) {
    memcpy(m, mat, 16*sizeof(Float));
}

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;
}

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 $i$th row of $M_1$ with the $j$th column of $M_2$. 
\textbf{Matrix4x4 Methods}\n\begin{verbatim}
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);
}
\end{verbatim}

Finally, Inverse() returns the inverse of the matrix. Our implementation, uses a numerically stable Gauss–Jordan elimination routine to compute the inverse.

\textbf{Matrix4x4 Methods}\n\begin{verbatim}
Reference<Matrix4x4> Inverse() const;
\end{verbatim}

\section*{A.6 Mathematical Utility Functions}

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 \texttt{pos} parameter. When \texttt{pos} is zero, the result is start; when \texttt{pos} is one, the result is end, etc.

Lerp is written as
\[
\text{lerp}(t, v_1, v_2) = (1 - t)v_1 + tv_2
\]
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 \texttt{Lerp} returns \textit{exactly} the values start and end when \texttt{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.

\textbf{Global Inline Functions}\n\begin{verbatim}
inline Float Lerp(Float pos, Float start, Float end) {
    return (1.f - pos) * start + pos * end;
}
\end{verbatim}

\texttt{Clamp} clamps a value \texttt{val} to be between the values \texttt{low} and \texttt{high}. If \texttt{val} is out of that range, \texttt{low} or \texttt{high} is returned as appropriate.

\textbf{Global Inline Functions}\n\begin{verbatim}
inline Float Clamp(Float val, Float low, Float high) {
    if (val < low) return low;
    else if (val > high) return high;
    else return val;
}
\end{verbatim}
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.

```c
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.

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.

```c
inline int Mod(int a, int b) {
    int n = int(a/b);
    a -= n*b;
    if (a < 0)
        a += b;
    return a;
}
```

Unfortunately, not all system `math.h` files store the value of $\pi$ in `M_PI`. If it is not defined, we do it ourself.

```c
#ifndef M_PI
#define M_PI 3.14159265358979323846f
#endif
```

One over 255 and one over $\pi$.

```c
#define INV_255 .00392156862745098039f
#define INV_PI 0.31830988618379067154f
```
We define a generally-useful \texttt{INFINITY} value using \texttt{FLT_MAX} from the standard math library, which is the largest representable floating point number.

\begin{verbatim}
(Global Constants) +≡
#ifndef INFINITY
#define INFINITY FLT_MAX
#endif
\end{verbatim}

Two simple functions convert from angles expressed in degrees to radians, and vice versa.

\begin{verbatim}
(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; }
\end{verbatim}

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:

\begin{verbatim}
Float a = ..., b = ...;
int i = (int)(a * b);
\end{verbatim}

may take 80 times longer than the multiplication \(a \times b\). 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.

\texttt{lrt} needs to convert \texttt{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 \texttt{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 \texttt{Float} value and returning an integer:

1. Float2Int\(f\): This is the same as the basic cast \((\text{int})f\).
2. Round2Int\(f\): This rounds the floating point value \(f\) to the nearest integer, returning the result as an \texttt{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.

\begin{verbatim}
(Global Inline Functions) +≡
inline int Log2Int(Float v) {
    return (*((int *) &v) >> 23) - 127;
}
\end{verbatim}
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 generator 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 $2^{32} - 1$.

\[
\text{Global Inline Functions} \equiv
\begin{align*}
\text{inline Float RandomFloat(Float min = 0.f, Float max = 1.f) { } } \quad &\text{Lerp} \\
&\text{return Lerp(genrand_real1(), min, max); } \\
\text{inline unsigned long RandomInt() { } } \quad &\text{genrand_int32} \\
&\text{return genrand_int32(); }
\end{align*}
\]

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.

\[
\text{Global Classes} \equiv
\begin{align*}
\text{class StringHashTable { } } \quad &\text{StringHashTable Methods} \\
\text{public: } \quad &\text{StringHashTable Private Data} \\
\text{private: } \quad &\text{} \\
\end{align*}
\]

\[
\text{StringHashTable Private Data} \equiv
\begin{align*}
\text{static const int NUM_BUCKETS = 1047; } \quad &\text{ } \\
\text{typedef vector<pair<string, void *>> ItemType; } \quad &\text{} \\
\text{ItemType buckets[NUM_BUCKETS]; } \quad &\text{ }
\end{align*}
\]
### StringHashTable Method Definitions

```cpp
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;
}
```

```cpp
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;
}
```

```cpp
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));
}
```

### Octrees

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-eighth 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...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.

\[
\text{\texttt{Octree Declarations}} \equiv \\
\text{template <class NodeData> struct OctNode {}
\]

\[
\text{\hspace{1em} OctNode *children[8];}
\]

\[
\text{\hspace{1em} vector<NodeData> data;}
\]

\]

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.
The constructor just stores the overall bound of the tree. Items are added individually with the add() method, below.

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.

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.

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.
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 a \textit{quadtree}).

\[\text{Possibly add data item to current octree node} \equiv\]
\[
\text{if } (\text{DistanceSquared}(\text{nodeBound.pMin, nodeBound.pMax}) < \text{diag2}) \{
\text{node}->\text{data.push_back(dataItem);} \\
\text{return;}
\}
\]

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 \textit{x} range of the object’s bounding box is entirely on the left side of the plane that splits the tree node in the \textit{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 \textit{pMid}, the position of the center of the current node. The fragment \textit{(Determine which children the item overlaps)} then efficiently sets an array of boolean values, \texttt{over}, such that the \texttt{i}th element is true only if the bounds of the data item being added overlap the \texttt{i}th child of the current node. We can then loop over the eight children and recursively call \texttt{add()} for the ones that the object overlaps.

\[\text{Otherwise add data item to octree children} \equiv\]
\[
\text{Point pMid = .5 * nodeBound.pMin + .5 * nodeBound.pMax;}
\]
\[
\text{\textit{Determine which children the item overlaps} \equiv for (int child = 0; child < 8; ++child) \{} \\
\text{\quad if (!over[child]) continue;}
\]
\[
\text{\quad \textit{Hand data item down to child number child} \}}
\]

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 \textit{z} component is on the low side of the \textit{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 \textit{y} plane, and the third bit is set based on it’s position with respect to the \textit{x} plane.

Thus, given boolean variables that classify a child node with respect to the splitting planes (\texttt{true} if it is above the plane, the child number of a given node is equal to:

\[4 \times (x\text{High ? 1 : 0}) + 2 \times (y\text{High ? 1 : 0}) + 1 \times (z\text{High ? 1 : 0})\]

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 \textit{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.

\[
\text{Determine which children the item overlaps} \equiv \\
\text{bool } \text{over}[8] = \\
\text{over}[0] = \text{over}[1] = \text{over}[2] = \text{over}[3] = (\text{dataBound.pMin.x} \leq \text{pMid.x}) ; \\
\text{over}[4] = \text{over}[5] = \text{over}[6] = \text{over}[7] = (\text{dataBound.pMax.x} > \text{pMid.x}) ; \\
\text{over}[0] = \text{over}[1] = \text{over}[4] = \text{over}[5] = (\text{dataBound.pMin.y} \leq \text{pMid.y}) ; \\
\text{over}[2] = \text{over}[3] = \text{over}[6] = \text{over}[7] = (\text{dataBound.pMax.y} > \text{pMid.y}) ; \\
\text{over}[0] = \text{over}[2] = \text{over}[4] = \text{over}[6] = (\text{dataBound.pMin.z} \leq \text{pMid.z}) ; \\
\text{over}[1] = \text{over}[3] = \text{over}[5] = \text{over}[7] = (\text{dataBound.pMax.z} > \text{pMid.z}) ;
\]

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.

\[
\text{Hand data item down to child number } \text{child} \equiv \\
\text{if (!node->children[child])} \\
\quad \text{node->children[child] = new OctNode<NodeData>;} \\
\text{Compute childBound for child} \\
\quad \text{add(node->children[child], childBound, dataItem, dataBound, diag2);} \\
\text{Here again we take advantage of the child node numbering scheme to quickly determine which values give the bounding box of the node.}
\]

\[
\text{Compute childBound for child} \equiv \\
\text{BBox childBound;} \\
\quad \text{childBound.pMin.x} = (\text{child} & 4) ? \text{pMid.x} : \text{nodeBound.pMin.x} ; \\
\quad \text{childBound.pMax.x} = (\text{child} & 4) ? \text{nodeBound.pMax.x} : \text{pMid.x} ; \\
\quad \text{childBound.pMin.y} = (\text{child} & 2) ? \text{pMid.y} : \text{nodeBound.pMin.y} ; \\
\quad \text{childBound.pMax.y} = (\text{child} & 2) ? \text{nodeBound.pMax.y} : \text{pMid.y} ; \\
\quad \text{childBound.pMin.z} = (\text{child} & 1) ? \text{pMid.z} : \text{nodeBound.pMin.z} ; \\
\quad \text{childBound.pMax.z} = (\text{child} & 1) ? \text{nodeBound.pMax.z} : \text{pMid.z} ;
\]

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 \text{Lookup()} function walks down the tree, processing the nodes that the given point overlaps. The user-supplied callback, \text{process} is called for each \text{NodeData} item that overlaps the given point.

As with the \text{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.

\[
\text{Octree Method Declarations} \equiv \\
\text{void Lookup(const Point &P, const LookupProc &process) {} \\
\text{if (!bound.Inside(P)) return; \\
\text{lookup(&root, bound, P, process);} 
\}}
\]

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
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.

\( \text{Octree Method Definitions} \) +

\[
\begin{align*}
\text{template <class NodeData, class LookupProc>}
\text{void Octree<NodeData, LookupProc>::lookup(OctNode<NodeData> *node,}
\text{ const BBox &nodeBound, const Point &P,}
\text{ const LookupProc &process)}
\text{ (Determine which octree child node \( P \) is inside)}
\text{ for (u_int i = 0; i < node->data.size(); ++i)}
\text{ process(P, node->data[i]);}
\text{ if (node->children[child])}
\text{ Compute childBound for child}
\text{ lookup(node->children[child], childBound, P, process);}\)
\end{align*}
\]

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 direction.

\( \text{Kd Tree Method Definitions} \) +

\[
\begin{align*}
\text{Point pMid = .5 * nodeBound.pMin + .5 * nodeBound.pMax;}
\text{int child = (P.x > pMid.x ? 4 : 0) +}
\text{(P.y > pMid.y ? 2 : 0) + (P.z > pMid.z ? 1 : 0);}\)
\end{align*}
\]

A.10 Kd Trees

\( \text{Source Code Copyright} \)

\[
\begin{align*}
\text{ifndef KDTREE_H}
\text{define KDTREE_H}
\text{include "lrt.h"}
\text{include "geometry.h"}
\text{Kd Tree Declarations}
\text{Kd Tree Method Definitions}
\text{endif // KDTREE_H}
\end{align*}
\]

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.

```cpp
enum SplitAxis { SPLIT_X, SPLIT_Y, SPLIT_Z };

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]; }
};
```

The KdTree itself only needs to hold a pointer to the root node of the tree.

```cpp
template <class NodeData, class LookupProc> class KdTree {
public:

    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
```
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 [start, end) that need to be processed.

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 building the kd tree.

⟨Kd Tree Method Definitions⟩

```cpp
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 midth 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>());
}

(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.
\( \text{Kd Tree Declarations} \)\(+\equiv\)

```cpp
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.

\( \text{Allocate Kd tree node and continue recursively} \)\(\equiv\)

```cpp
*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.

As usual, we immediately call to an internal lookup procedure, passing in a pointer to the current node to be processed.

\( \text{Kd Tree Method Definitions} \)\(+\equiv\)

```cpp
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.

```
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 \( \text{maxDist} \) 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.

\[\textbf{Process Kd node's children based on x split} \equiv \]

\[
\begin{cases} 
\text{if } (P.x \leq \text{node->data.P.x}) \{ \\
\text{recursiveLookup(node->children[0], P, process, maxDist);} \\
\text{if } (P.x + \text{maxDist} \geq \text{node->data.P.x}) \\
\text{recursiveLookup(node->children[1], P, process, maxDist);} \\
\} \\
\text{else } \{ \\
\text{recursiveLookup(node->children[1], P, process, maxDist);} \\
\text{if } (P.x - \text{maxDist} \leq \text{node->data.P.x}) \\
\text{recursiveLookup(node->children[0], P, process, maxDist);} \\
\} 
\end{cases}
\]

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.

\[\textbf{Hand Kd tree node to processing function} \equiv \]

\[
\text{if } (\text{DistanceSquared(node->data.P, P)} < \text{maxDist*maxDist}) \\
\text{process(node->data, maxDist);} 
\]

### 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.

\[\textbf{lrt.h} \equiv \]

\[
\begin{cases} 
\text{\textit{Source Code Copyright}} \\
\text{`#ifndef LRT_H} \\
\text{`#define LRT_H} \\
\text{\textit{Global Include Files}} \\
\text{\textit{Platform-specific definitions}} \\
\text{\textit{Global Type Declarations}} \\
\text{\textit{Global Forward Declarations}} \\
\text{\textit{Global Constants}} \\
\text{\textit{Global Function Declarations}} \\
\text{\textit{Global Classes}} \\
\text{\textit{Global Inline Functions}} \\
\text{`#endif // LRT_H} \\
\end{cases}
\]

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.

```
#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.

```
#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`.

```
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.

```
#define LRT_VERSION 1.0
```

SGI’s old C++ compiler...
\textit{Platform-specific definitions}:
\begin{verbatim}
#define for if (0) ; else for
\end{verbatim}

\section*{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 \url{http://www.math.keio.ac.jp/~matumoto/emt.html}.
Float to int stuff at \url{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 geometry (dBvKOS00)
B. TIFF Input and Output

 ifndef TIFFIO_CC
  #define TIFFIO_CC "tiffio.cc"
 endif

 // 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)

```c
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
}
```

(Try to open TIFF file)

```c
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)

```c
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 \(\text{Clean up after TIFF reading error}\), 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 RGBRBRGB 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.

\(\text{Make sure this is a TIFF we can read}\)\(\equiv\)

\[
\text{short int bitsPerSample, sampleFormat = SAMPLEFORMAT_UINT;}
\]
\[
\text{if (!TIFFGetField(tiff, TIFFTAG_BITSPERSAMPLE, &bitsPerSample)) {}
\]
\[
\text{Error("TIFFRead: bits per sample not set in TIFF");}
\]
\[
\text{\(\text{Clean up after TIFF reading error}\)}
\]
\[
\}
\]
\[
\text{if (!TIFFGetField(tiff, TIFFTAG_SAMPLEFORMAT, &sampleFormat)) {}
\]
\[
\text{if (bitsPerSample == 32)}
\]
\[
\text{sampleFormat = SAMPLEFORMAT_IEEEFP;}
\]
\[
\text{else}
\]
\[
\text{sampleFormat = SAMPLEFORMAT_UINT;}
\]
\[
\}
\]
\[
\text{if (bitsPerSample == 32) {}
\]
\[
\text{if (sampleFormat != SAMPLEFORMAT_IEEEFP) {}
\]
\[
\text{Error("TIFFRead: 32 bit TIFF not stored in floating point format");}
\]
\[
\text{\(\text{Clean up after TIFF reading error}\)}
\]
\[
\}
\]
\[
\text{else { }
\]
\[
\text{if (bitsPerSample != 8 && bitsPerSample != 32) {}
\]
\[
\text{Error("TIFFRead: only 8 and 32 bits per sample supported");}
\]
\[
\text{\(\text{Clean up after TIFF reading error}\)}
\]
\[
\}
\]
\[
\text{if (sampleFormat != SAMPLEFORMAT_UINT) {}
\]
\[
\text{Error("TIFFRead: 8 bit TIFFs must be stored as unsigned ints");}
\]
\[
\text{\(\text{Clean up after TIFF reading error}\)}
\]
\[
\}
\]
\[
\text{int bytesPerSample = bitsPerSample / 8;}
\]
\[
\text{if (nSamples * xSize * bytesPerSample != TIFFScanlineSize(tiff)) {}
\]
\[
\text{Error("TIFFRead: RGB not interleaved in TIFF %s", name.c_str());}
\]
\[
\text{\(\text{Clean up after TIFF reading error}\)}
\]
\]

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 \text{PHOTOMETRIC} field doesn’t indicate that there is a palette (aka colo-
ormap) 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.

\textit{(Read TIFF colormap if present)}
\begin{verbatim}
  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");
      \textit{(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.

\textit{(Allocate space for pixels and buffers)}
\begin{verbatim}
  pixels = new Spectrum[*xSize * *ySize];
  Spectrum *pixelp = pixels;
  if (bitsPerSample == 32) fbuf = new float[nSamples * *xSize];
  else ubuf = new u_char[nSamples * *xSize];
\end{verbatim}
\end{verbatim}

\textit{(Read a TIFF scanline)}
\begin{verbatim}
  if (fbuf) {
    \textit{(Read floating point TIFF scanline)}
  }
  else {
    \textit{(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.

\textit{(Read floating point TIFF scanline)}
\begin{verbatim}
  float *fbufp = fbuf;
  if (TIFFReadScanline(tiff, fbuf, y, 1) == -1) {
    \textit{(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;
  }
\end{verbatim}
\end{verbatim}
Similarly, we can do similar tricks with ubuf and ubufp when reading eight-bit TIFFs.

\[ \text{Read 8-bit TIFF scanline} \equiv \]
\[
\text{u_char } \ast \text{ubufp} = \text{ubuf};
\]
\[
\text{if (TIFFReadScanline(tiff, ubuf, y, 1) == -1) } \{ \\
\text{\{Clean up after TIFF reading error\}}
\}
\]
\[
\text{for (int } x = 0; x < \ast \text{xSize}; ++x) \{ \\
\text{if (nSamples == 1) } \{ \\
\text{\{Decode TIFF colormap entry\}}
\}
\text{else} \{ \\
\text{\{Convert standard 8-bit TIFF pixel\}}
\}
\text{++pixelp;}
\text{ubufp += nSamples;}
\}
\]

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.

\[ \text{Decode TIFF colormap entry} \equiv \]
\[
\text{int mapOffset} = \ast \text{ubufp;}
\text{Assert(COLOR_SAMPLES == 3);}
\text{Float cs[3] = \{ mapR[mapOffset] * INV_255 * INV_255,}
\text{mapG[mapOffset] * INV_255 * INV_255,}
\text{mapB[mapOffset] * INV_255 * INV_255 \};}
\text{*pixelp = Spectrum(cs);}
\]

And reading a normal pixel is easy; we just need to scale by \(1/255\).

\[ \text{Convert standard 8-bit TIFF pixel} \equiv \]
\[
\text{Float cs[COLOR_SAMPLES];}
\text{for (int } i = 0; i < \text{COLOR_SAMPLES}; ++i)
\text{cs[i] = ubufp[i] * INV_255;}
\text{*pixelp = Spectrum(cs);}
\]

\[ \text{Close TIFF and return} \equiv \]
\[
\text{delete[] ubuf;}
\text{delete[] fbuf;}
\text{TIFFClose(tiff);}
\text{return pixels;}
\]

\[ \text{Clean up after TIFF reading error} \equiv \]
\[
\text{delete[] pixels;}
\text{delete[] ubuf;}
\text{delete[] fbuf;}
\text{TIFFClose(tiff);}
\text{return NULL;}
\]
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.

\begin{quote}
\textbf{TIFF Function Definitions} \texttt{+}\texttt{=} \texttt{≡}
\begin{verbatim}
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
  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);
  \texttt{≡}
  \textbf{Close 8-bit TIFF file}
  Write 8-bit scanlines
  for (int y = 0; y < YRes; y++) {
    TIFFWriteScanline8Bit(tiff, pixels + y * nChannels,
                        alpha + y * nChannels, XRes);
  }
  \textbf{Open 8-bit TIFF file for writing}
\end{verbatim}
\end{quote}

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.

\begin{quote}
\textbf{TIFF Function Definitions} \texttt{+}\texttt{=} \texttt{≡}
\begin{verbatim}
void TIFFWrite32Bit(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
  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);
  \texttt{≡}
  \textbf{Close 8-bit TIFF file}
  Write 32-bit scanlines
  for (int y = 0; y < YRes; y++) {
    TIFFWriteScanline32Bit(tiff, pixels + y * nChannels,
                           alpha + y * nChannels, XRes);
  }
  \textbf{Open 8-bit TIFF file for writing}
\end{verbatim}
\end{quote}
Output 537

Compute and set up samples per pixel

```c
int sampleCount = 0;
if (pixels) sampleCount += nChannels;
if (alpha) ++sampleCount;
TIFFSetField(tiff, TIFFTAG_SAMPNPERSPIXEL, 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

```c
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);
```

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

```c
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

```c
for (int s = 0; s < nChannels; ++s)
    *bufp++ = (u_char)(Clamp(*pixelp++, 0.f, 255.f));
```

Pack 8-bit alpha samples into buf

```c
if (alphap)
    *bufp++ = (u_char)(Clamp(*alphap++, 0.f, 255.f));
```
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**

```c
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)
    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)

    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 type-def’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.
```
\(\text{Write Float scanlines}\)\(\equiv\)

```c
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;
```

\(\text{Close Float TIFF file}\)\(\equiv\)

```c
TIFFClose(tiff);
```
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.

```c
#include "lrt.h"

// Dynamic Loading Declarations
#endif // DYNLOAD_H
```
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.

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.
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.

\[\textbf{ParamSet Declarations}\]

```cpp
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;
};
```

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

```c
#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

```c
#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.

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 specified at each vertex of the mesh.

```c
#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

```c
ParamSet(int nv = 0) { nVertex = nv; }
```

### ParamSet Public Data

```c
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

```c
int nAlloc = arraySize;
if (type & PARAM_TYPE_VARYING) nAlloc *= 4;
if (type & PARAM_TYPE_VERTEX) nAlloc *= 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.

```
    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.

```
    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.
\textbf{ParamSet Methods}:

\begin{verbatim}
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;
}
\end{verbatim}

As above, here are the declarations for the rest of the analogous methods.

\textbf{ParamSet Interface}:

\begin{verbatim}
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;
\end{verbatim}

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.

\textbf{ParamSet Methods}:

\begin{verbatim}
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;
}
\end{verbatim}

These are the rest of the analogous lookup functions.
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;

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.

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::ParamSet() {
    clear();
}
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

```cpp
Reference<Shape> CreateShape(const Transform &o2w, const ParamSet &params);
```

Because all Shapes 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 calls the constructor, returning a newly-allocated sphere.

Sphere Methods

```cpp
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}+≡

```cpp
class DSO {
public:
  {DSO Methods}
private:
#if defined(WIN32)
   HMODULE hinstLib;
#else
   void *hinstLib;
#endif
};
```

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}≡

```cpp
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

```cpp
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.

DRS Method Definitions

```cpp
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;
}
```

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

```cpp
class ShapeDSO : public DSO {
    public:
        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;
}
```

### 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;
```
Dynamic Object Creation

{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 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)

creation stuff
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;
        }
    }
}
\textbf{Material Method Definitions}\[\text{\textasciicircum}^\equiv\]

```
Texture<Float> *Material::MakeFloatTex(const ParamSet &pGeom, const ParamSet &pShader,
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;
        }
    }
}
```

\textbf{HomogeneousRegion Definitions}\[\text{\textasciicircum}^\equiv\]

```
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);
}
```
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);
}
D. Rendering Interface

```c
#define RI_H
#include "lrt.h"
#ifdef __cplusplus
extern "C" {
#endif /* C++ */

RI Function Declareations

#endif /* RI_H */
```
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.

\textit{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;
};
```

\textit{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.

\textit{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.

\textit{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.

\[\text{Search for token in token table}\] =

Token *token = (Token *)TokenTable.Search(name);
if (token) {
  \[\text{Complain if token was redeclared to a different type}\]
  return token->token.c_str();
}

\[\text{Complain if token was redeclared to a different type}\] =

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.

\[\text{Add token to table if not found}\] =

Token *newToken = new Token(name, tokenType, narray);
TokenTable.Add(name, newToken);
return newToken->token.c_str();

\[\text{System-wide Initialization}\] =

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 stringToDate and typeToString functions here; both are just a matter of parsing and string management. The first function, stringToDate, takes strings of the form

\[\text{uniform point[2]},\]

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

\[\text{PARAM_TYPE_UNIFORM | PARAM_TYPE_POINT}.\]
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);
}  

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.

\[\text{RI Function Definitions}\]

```c
void RiBegin() {
    \textit{(System-wide Initialization)}
}

Similarly most of RiEnd is filled in elsewhere as well.

\[\text{RI Function Definitions}\]

```c
void RiEnd() {
    \textit{(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().

\[\text{API Static Data}\]

```c
#define STATE_UNINITIALIZED 0
#define STATE_BEGIN 1
#define STATE_WORLD_BEGIN 2
static int currentApiState = STATE_UNINITIALIZED;
```

\[\text{System-wide Initialization}\]

```c
if (currentApiState != STATE_UNINITIALIZED)
    Severe("RiBegin() has already been called.");
    currentApiState = STATE_BEGIN;
```

\[\text{System-wide cleanup}\]

```c
    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**

```c
#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**

```c
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.

**API Local Classes**

```c
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**

```c
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**

```c
curGfxOptions = new GfxOptions;
```

**System-wide cleanup**

```c
delete curGfxOptions;
curGfxOptions = NULL;
```

**GfxOptions Method Declarations**

```c
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.

\[ \text{RI Function Definitions} \] 

\begin{verbatim}
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));
}
\end{verbatim}

And appropriate fields are added to GfxOptions.

\[ \text{Graphics Options} \] 

\begin{verbatim}
int PixelSamples[2];
\end{verbatim}

\[ \text{GfxOptions Constructor Implementation} \] 

\begin{verbatim}
PixelSamples[0] = PixelSamples[1] = 2;
\end{verbatim}

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.

\[ \text{RI Function Declarations} \] 

\begin{verbatim}
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);
\end{verbatim}

\[ \text{Graphics Options} \] 

\begin{verbatim}
Float PixelAspectRatio, FrameAspectRatio;
int XResolution, YResolution;
\end{verbatim}

\[ \text{GfxOptions Constructor Implementation} \] 

\begin{verbatim}
PixelAspectRatio = 1.f;
FrameAspectRatio = 4.f / 3.f;
XResolution = 640;
YResolution = 480;
\end{verbatim}
Update screen window from format values

```c
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

```c
Extent2D ScreenExtent;
bool ScreenExtentSet;
```

Graphics Options Constructor Implementation

```c
Extent2D 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.

RI Function Definitions

```c
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

```c
Extent2D Crop;
```

Graphics Options Constructor Implementation

```c
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

```c
Float ClipHither, ClipYon;
```

Graphics Options Constructor Implementation

```c
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.
By default, the shutter is only open for an instant, so no motion blur is seen.

There is also a function to set depth of field parameters for Cameras that support this effect.

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.

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.
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.;

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;
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 nArgs,
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;

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;
acceleratorName = "grid";

XXX where should this go?? XXXX

Scene *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 || !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;
}

Filter *filter = CreateFilter(filterName, SearchPath, FilterParams,
                       FilterXWidth, FilterYWidth);

Sampler *sampler = CreateSampler(samplerName, SearchPath, samplerParams,
                           XResolution, YResolution, PixelSamples[0],
                           PixelSamples[1], Crop, filter);

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);
\{\textit{Initialize displayInfo from API settings}\}\equiv
\begin{verbatim}
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;
\end{verbatim}
\{\textit{Initialize surfaceIntegrator from API settings}\}\equiv
\begin{verbatim}
SurfaceIntegrator *surfaceIntegrator = CreateSurfaceIntegrator(surfIntegratorName,
    curGfxOptions->SearchPath, surfIntegratorParams);
\end{verbatim}
\{\textit{Initialize volumeIntegrator from API settings}\}\equiv
\begin{verbatim}
VolumeIntegrator *volumeIntegrator = CreateVolumeIntegrator(volIntegratorName,
    curGfxOptions->SearchPath, volIntegratorParams);
\end{verbatim}
\{\textit{Initialize accelerator from API settings}\}\equiv
\begin{verbatim}
Primitive *accelerator = CreateAccelerator(acceleratorName,
    SearchPath, primitives, acceleratorParams);
if (!accelerator) {
    ParamSet ps;
    accelerator = CreateAccelerator("grid", SearchPath,
        primitives, ps);
}
\end{verbatim}

\textbf{RI Function Declarations}+≡
\begin{verbatim}
extern void RiSearchPath(const char *path);
\end{verbatim}

\textbf{Graphics Options}+≡
\begin{verbatim}
string SearchPath;
\end{verbatim}

\textbf{GfxOptions Constructor Implementation}+≡
\begin{verbatim}
#ifdef LRT_BUILDDIR
    SearchPath = ".:." LRT_BUILDDIR ".:/texture";
#else
    SearchPath = ".:.";
#endif
\end{verbatim}

\textbf{RI Function Declarations}+≡
\begin{verbatim}
extern void RiToneMap(const char *name, int n, const char *tokens[],
    void *params[]);
\end{verbatim}

\textbf{Graphics Options}+≡
\begin{verbatim}
string ToneMapping;
ParamSet ToneMapParams;
\end{verbatim}
void RiSurfaceIntegratorV(const char *name, int n, const char *tokens[],
void * params[]) {
    curGfxOptions->surfIntegratorName = name;
    curGfxOptions->surfIntegratorParams.init(n, tokens, params);
}

void RiVolumeIntegratorV(const char *name, int n, const char *tokens[],
void * params[]) {
    curGfxOptions->volIntegratorName = name;
    curGfxOptions->volIntegratorParams.init(n, tokens, params);
}

string surfIntegratorName, volIntegratorName;
ParamSet surfIntegratorParams, volIntegratorParams;

surfIntegratorName = "whitted";
volIntegratorName = "null";

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

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

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);

PRINT per-frame statistics
statsPrint(stderr);
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:

```
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 Transforms. 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;
```
void RiTransformBegin() {
    for (int i = 0; i < MOTION_LEVELS; ++i)
        transformStack[i].push_back(curTransform[i]);
    hierarchicalState.push_back('t');
}

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.

struct GfxState {
    GfxState();
    (Graphics State)
    (Graphics State Methods)
};

GfxState::GfxState() {
    (GfxState Constructor Implementation)
}

When RiWorldBegin is called, we initialize the current graphics state to hold default values.

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.

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.
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.

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.

```cpp
void RiSurfaceV(const char * name, int n, const char * tokens[],
void * params[]) {
    curGfxState.surfaceParams.init(n, tokens, params);
    curGfxState.surface = name;
}
```
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**

```c
#define MOTION_LEVELS 2
```

**API Static Data**

```c
static int motionLevel = 0;
static bool inMotionBlock = false;
static Transform curTransform[MOTION_LEVELS];
```

**System-wide Initialization**

```c
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.

\texttt{RI Function Definitions}+\equiv
\begin{verbatim}
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.");
}
\end{verbatim}

\texttt{RI Function Definitions}+\equiv
\begin{verbatim}
void RiMotionEnd() {
    if (!inMotionBlock)
        Error("Unmatched MotionEnd statement");
    inMotionBlock = false;
}
\end{verbatim}

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.

\texttt{RI Function Definitions}+\equiv
\begin{verbatim}
void RiIdentity() {
    \texttt{Prepare for motion transform}
    curTransform[xform] = Transform();
    \texttt{Update transform for motion block}
}
\end{verbatim}

If there is no current motion block, then we just update \texttt{curTransform}[0]. Otherwise, we update the appropriate one depending on how many transforms have been given in this block so far.

\texttt{Prepare for motion transform}+\equiv
\begin{verbatim}
int xform = 0;
if (inMotionBlock)
    xform = motionLevel;
if (motionLevel > MOTION_LEVELS) {
    Warning("Only \%d motion levels are supported. Ignoring.",
            motionLevel);
    return;
}
\end{verbatim}

\texttt{Update transform for motion block}+\equiv
\begin{verbatim}
if (inMotionBlock)
    ++motionLevel;
\end{verbatim}
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.

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 polygon is convex. Here, we will tessellate the given polygon and create a TriangleMesh.

```c
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.

```c
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

```c
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

```c
void GfxState::AddShape(const Reference<Shape> &shape,
               ParamSet &geomParams) {
    if (!shape) return;
    AreaLight *area = NULL;
    AreaLight *area; // 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

```c
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 Primitives and Lights 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.
mutable vector<Reference<Primitive> > primitives;
mutable vector<Light *> lights;
mutable vector<VolumeRegion *> volumeRegions;

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;
}

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.

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
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.

\[
\begin{align*}
\text{AddFloat} & \quad 545 \\
\text{AddShape} & \quad 578 \\
\text{curGfxOptions} & \quad 563 \\
\text{curGfxState} & \quad 573 \\
\text{curTransform} & \quad 575 \\
\text{motionLevel} & \quad 575 \\
\text{ParamSet} & \quad 542
\end{align*}
\]

\[
\{\text{RI Function Definitions}\} + \equiv
\]

\[
\begin{align*}
\text{void RiSphereV}(\text{Float radius, Float zmin, Float zmax,} \\
\text{Float thetaMax, int n, const char * tokens[],} \\
\text{void * params[]}) 
\text{\{} \\
\text{ParamSet geomParams(n, tokens, params);} \\
\text{geomParams.AddFloat("_radius", &radius);} \\
\text{geomParams.AddFloat("_zmin", &zmin);} \\
\text{geomParams.AddFloat("_zmax", &zmax);} \\
\text{geomParams.AddFloat("_thetamax", &thetaMax);} \\
\text{curGfxState.AddShape(} \\
\text{\quad CreateShape("sphere", curGfxOptions->SearchPath,} \\
\text{\quad curTransform[motionLevel], geomParams),} \\
\text{\quad geomParams);} \\
\text{\})
\end{align*}
\]
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

```c
void *RiLightSourceV(const char * name, int nArgs,
                   const char *tokens[], void * params[]) {
    ParamSet paramSet(nArgs, tokens, params);
    bool shadows = paramSet.FindElement("shadows", 1) != 0;
    Light *lt = CreateLight(name, curGfxOptions->SearchPath,
                            shadows, curTransform[motionLevel],
                            paramSet);
    Add new light to graphics state
    return lt;
}
```

```c
if (lt == NULL) {
    Error("RiLightSource: light type ‘%s’ unknown.", name);
} else {
    curGfxOptions->lights.push_back(lt);
}
```

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

```c
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

```c
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

```cpp
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

```cpp
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);
}
```

Further Reading

- RenderMan companion(Ups89)
- RI Spec(Pix89)
- Advanced RMan book
- OpenGL stuff
Bibliography


Chris Fraser and David Hanson, *A retargetable C compiler: Design and implementation*, Addison-Wesley, 1995.


Johann Heinrich Lambert, *Photometry, or, on the measure and gradations of light, colors, and shade*, The Illuminating Engineering Society of North America, 2001, Translated by David L. DiLaura.


Gregory Ward Larson, Holly Rushmeier, and Christine Piatko, *A visibility matching tone reproduction operator for high dynamic


Eric Veach and Leonidas Guibas, Bidirectional estimators for light transport, Fifth Eurographics Workshop on Rendering (Darmstadt, Germany), June 1994, pp. 147–162.


Index of Identifiers