DRAFT (18 October 2001) — Do Not Distribute

A LITERATE RAY TRACER

Matt Pharr, Greg Humphreys, and Pat Hanrahan

[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.

— Eric Naggum

Preface

This manuscript contains the documented source code for a reasonably complete ray tracing rendering system. It is written using a programming methodology called *literate programming* that intermingles text describing the system with the source code that implements it. This is a new approach to teaching the concepts embodied in a rendering system, from basic geometric computation to sophisticated light transport algorithms. By showing these ideas in the context of a complete and non-trivial program we also address some issues in the design and implementation of medium-sized rendering software systems.

Overview and Goals

`lrt`, the rendering system that we will describe in this book, 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-end rendering systems; in particular, it means that important practical issues, such as anti-aliasing, be addressed thoroughly. It is often quite difficult to retrofit such functionality to a rendering system after it has been designed, as it can have subtle implications for all components of the system, and even the overall architectural design. In cases where `lrt` lacks a 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.

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 best algorithms we were aware of. Clarity, simplicity, and elegance were our watchwords.

Efficiency was only 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 largely 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. However, exploiting intra-frame parallelism in `lrt` would be straightforward in either a shared-memory or message-passing environment.

In the course of presenting the implementation of `lrt`, we hope to convey some hard-learned lessons gleaned from experience with both commercial and academic rendering research and development. Writing a good renderer is much more complex than stringing together a set of fast algorithms; making the system *robust* is even harder. 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. A renderer is somewhat like an operating system: managing and ordering large amounts of data and computation is a critical task.

RenderMan

Although we won't provide a detailed description here, `lrt` supports the RenderMan interface for specifying scenes. RenderMan is the industry standard high quality rendering API, although the specification does not mandate the actual rendering technique to be used. Scan conversion based implementations of the RenderMan interface exist; the most notable is Pixar's Photorealistic RenderMan, which is used for most special effects we see in movies today. Blue Moon Rendering Tools (BMRT) by Larry Gritz is a raytracing based implementation similar in spirit to `lrt`, but with different goals.

Our ray tracer supports not only the programmatic RenderMan API, but also the Renderman Bytestream Interface (RIB). RIB is a text-file encoding of RenderMan API calls. A good introduction to the RenderMan interface is *The Renderman Companion*(**?**). Blue Moon Rendering Tools is available at `http://www.bmrt.org`. More information about the RIB format and detailed Renderman specifications can be found at Pixar's web site: `http://www.pixar.com`.

Literate Programming

In the course of the development of the TEXtypesetting system, Donald Knuth developed a new programming methodology based on the simple idea that *programs should be written more for people's consumption than for computers' consumption.* He named this methodology *literate programming*. This manuscript (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. LATEXand/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 *literate programming:weaving*), or into source code suitable for compilation (*literate programming: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 an 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 at once, `InitGlobals()` may be a large collection of variable assignments whose meanings 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 the global.

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

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

Here we have defined a *literate progamming:fragment* called ⟨*Function Definitions*⟩. This fragment will be included in a source code file when the literate program is woven for the C++ compiler; which one isn't important to the human reader. The fragment contains the definition of the `InitGlobals` function. The `InitGlobals` function itself includes another fragment, ⟨*Initialize Global Variables*⟩. When we introduce a new global variable `ErrorCount` anywhere in the program, we can now write:

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

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

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

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

```
void InitGlobals() {
  ErrorCount = 0;
  FragmentsProcessed = 0;
}
```

The other main advantage of literate programming is that complex functions can be easily decomposed into logically-distinct parts. We can write a function as a series of fragments:

⟨*Function Definitions*⟩+≡
```
int func(int x, int y, double *data) {
    int errorCount = 0;
    ⟨Check validity of arguments⟩
    if (x < y) {
        ⟨Swap parameter values⟩
    }
    ⟨Do precomputation before loop⟩
    ⟨Loop through data array⟩
    return errorCount;
}
```

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. An 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 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 two 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. The second is that one generally names fragments with more descriptive names than one gives to functions; this improves program readability and understandability.

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.

Coding Conventions

The programming language that we have written `lrt` in is C++. However, we have chosen to use a subset of the entire language, both to make the code easy to understand to the non C++ expert as well as to improve portability, as not all compilers yet support the entire language as specified by ISO/ANSI. In particular, we have avoided multiple inheritance and the use of exceptions. We have used onla a small subset of C++'s extensive standard library for similar reasons. In particular, we use the `iostream` input/output facilities and the `vector` and `list` container classes.

Types, objects, and variables are named to indicated their scope; classes and functions that have global scope all start with capital letters; the system uses no global variables. Short 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 some small pieces of `lrt`'s source code from this document. For example, when there are a number of cases to be considered in some situation, but where the code that handles the different cases is almost identical, we will present one case and note that the code for the remaining cases is omitted (of course, it's not omitted from the final program source code!) Furthermore, when we declare a new class (`Foo`, for example), we won't include the code for the class declaration in this document. Instead of having many fragments like:

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

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

Contents

1.Rendering Overview

This chapter provides a high-level description of the entire rendering system from the top down. We will begin by providing an overview of the task of a rendering system and its inputs. This is followed by an overview of the system presented in ths book. To do this, we will follow the path of a single ray through the entire system. Because this is a top-down exploration of the system, some concepts will be referred to before they are defined. Therefore, the reader would benefit from reading this section more than once. The following chapters will describe components of a rendering system, and will be presented in a bottom-up organization,

1.1 Overview of Rendering

Computer graphics is often divided into three main sub-problems: modeling, rendering, and animation. *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). 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.

1.2 Overview of the Program

`lrt` has three main phases:

1. Defining the scene;
2. Simulating the camera and the process of image formation;
3. Computing the shading of visible points which in turn depends on the illumination.

Defining the Scene.

Scenes are described using the RenderMan Scene Description Format. The `main` function is pretty simple; it parses scene input from files specified on the command line. `main` surrounds the scene parser with the RenderMan calls `RiBegin` and `RiEnd`.

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

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

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

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

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

Parsing of RenderMan input will be described in the appendices of the book.

As the scene file is parsed, different objects are created. For example, the scene will consist of different geometric primitives. These primitives will be created and stored in the `Scene` data structure. Primitives will also have attributes, such as a transformation, color, transparency, and other material properties. Finally, the scene will contain a set of light sources and a camera.

⟨*Global Scene Variable*⟩≡
```
Scene *scene = NULL;
```

Each input file may contain multiple frames, each surrounded by a `RiWorldBegin`/`RiWorldEnd` pair. Because `RiWorldEnd` marks the end of a frame, it begins the rendering process.

⟨*RI Function Definitions*⟩≡
```
RtVoid RiWorldEnd() {
    ⟨Check for valid WorldEnd state⟩
    scene->AddPrimitives(primitives);
    scene->Render();
    ⟨Clean up after world end⟩
    ⟨Print per-frame statistics⟩
}
```

Main Rendering Loop.

The create an image, the `scene`'s `Render` method is invoked.

⟨*Scene Methods*⟩≡
```
void Scene::Render() {
    ⟨Choose an integrator⟩
    ⟨Finalize scene option parameter values⟩
    cerr << "Rendering: ";
    Float sample[5];
    while (sampler->GetNextImageSample(sample)) {
        ⟨Compute eye ray with camera⟩
        ⟨Report rendering progress⟩
        ⟨Evaluate radiance along ray⟩
        ⟨Hand sample to Image⟩
    }
    image->Write();
    cerr << endl;
}
```

First, the results of some computations that will be used frequently while rendering the scene are cached, and then the main rendering loop begins.

Next, any rendering system must provide a camera model that simulates the process of image formation, which in `lrt` is done with ray-casting. A `Sampler` generates 5-dimensional sample values, which are then turned into rays by a `Camera`. The main function of the `Camera` class is to provide a `GenerateRay` method, which takes some information about which point on the image is to be sampled, and generated a ray according to the

⟨*Camera Interface Declarations*⟩≡
```
virtual bool GenerateRay(Float sample[5], Ray &ray) const = 0;
```

The camera may determine that there is no valid ray corresponding to the given sample value; in this case it doesn't set `ray` and returns `false`. Otherwise the ray is initilized and `true`is returned.

⟨*Compute eye ray with camera*⟩≡
```
Ray ray;
if (!camera->GenerateRay(sample, ray)) continue;
```

Given a ray, the `Integrator` traces those rays through the scene and computes the light or radiance that returns along those rays. These sampled radiances are then stored in the `Image`, which is eventually saved in a file or displayed. `lrt` provides a number of different integrators for achieving differing levels of realism or providing different functionality (such as supporting the RenderMan shading language). Here we will present the classic Whitted-style ray-tracing integration method.

⟨*Evaluate radiance along ray*⟩≡
```
HitInfo hitInfo;
Float alpha, hitDist = INFINITY;
Spectrum L = integrator->Integrate(ray, &hitInfo, &hitDist, &alpha);
```

The integrator returns the scene radiance `L`. Scene radiance are represented as a `Spectrum`. One can think of the scene radiance as a function defined in ray space; that is, what is the power carried along a ray. The `Spectrum` may be an RGB triple, or a more general energy distribution for specialized applications. In addition, the integrator also fills in information about the geometric intersection point, the distance from the ray's origin to the intersection, and the opacity of the intersected surface.

⟨*WhittedIntegrator Method Definitions*⟩≡
```
Spectrum WhittedIntegrator::Integrate(const Ray &ray, HitInfo *hitInfo,
        Float *hitDist, Float *alpha) const {
    if (scene->Intersect(ray, 1e-4, hitDist, hitInfo)) {
        Spectrum L(0.);
        *alpha = 1.;
        ⟨Compute emitted and reflected light⟩
        return L;
    }
    else {
        *alpha = 0.;
        return Spectrum(0.);
    }
}
```

For the integrator to determine what point in space is hit by a ray, it calls the `Intersect` method of the `Scene` class. `Intersect` computes the intersection of a ray with the geometry in the scene. However, scenes normally contain a large number of geometric primitives. In order to determine what primitive is hit by a given ray, it is desirable to be able to quickly cull groups of geometry that will not be hit by the ray. Thus, the intersection task is passed onto to an `Accelerator`.

⟨*Scene Methods*⟩+≡
```
bool Scene::Intersect(const Ray &ray, Float mint, Float *maxt,
    HitInfo *hit) const {
    if (!accelerator) return false;
    return accelerator->IntersectClosest(ray, mint, maxt, hit);
}
```

⟨*Accelerator Interface*⟩≡
```
virtual bool IntersectClosest(const Ray &ray, Float mint,
    Float *maxt, HitInfo *hit) = 0;
```

The key method provided by accelerators is the `Accelerator::IntersectClosest` routine. It finds the first intersection of the given ray with any of the primitives in the scene. If a hit is found, `true` is returned, `hit` and `maxt` should be updated. `maxt` stores the ray parameter associated with the intersection, and `hit` returns information about the surface that was hit so that further information may be retrieved during shading calculations.

Above we just show the virtual function for the base class. The primary accelerator used by `lrt` is a uniform grid.

Each geometric primitive in `lrt` is a descendant of the abstract `Primitive` class. `lrt` implements many primitives including triangle meshes, quadric surfaces (spheres, cylinders, cones, paraboloids, hyperboloids), tori, bilinear and bicubic patches, and non-uniform rational b-splines. All `Primitive`s contain a pure virtual function called `IntersectClosest`. `IntersectClosest` returns a boolean indicating whether the primitive was hit. It also fills in information about the intersection if it exists: the distance along the ray, and meta-information about the surface at the hit point.

Once an eye ray has been computed, it is passed into the integrator. In addition to returning the ray's radiance, the integrator returns additional information about the object the ray intersects (if any). First, the `HitInfo` structure is filled in with geometric information about the hit point, what the surface reflectance properties are, etc (see Section 3.2 on page 41.) Second, the distance to the first intersection is stored in the `hitDist` variable. Third, the `alpha` variable is set 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, the returned 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 hand it to the `Image` We first compute the raster-space depth value of the hit point (using the hit distance returned by the `Integrator` 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), setting the radiance and alpha to zero if so. We then call the `AddSample` method of the `Image` which isn't defined until Section 7.3 on page 108; to understand roughly what `AddSample` does before delving into the image sampling and reconstruction chapter, see the definition of `AddSampleBasic` in Section **??**.

⟨*Hand sample to* `Image`⟩≡

```
Float screenz = camera->WorldToScreen(ray(hitDist)).z;
if (screenz > 1.) {
    L = 0.;
    alpha = 0.;
}
Point Praster(sample[0], sample[1], screenz);
image->AddSample(Praster, L, alpha);
```

Shading and Lighting.

The geometric calculations described above provide half of the functionality of `lrt`. The other half lies in the shading system. 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. To compute reflected light, the integrator must have access to material properties of the surface. The total reflected light is computed by integrating the incoming light over a hemisphere above the point being shaded. To compute the incoming light, the integrator must also have access to or compute the lighting environment at that point.

⟨*Compute emitted and reflected light*⟩≡
```
ShadeContext shadeContext(hitInfo, -ray.D);
```
⟨*Compute emitted light if an area light source*⟩
⟨*Evaluate BRDF and create surface shader*⟩
⟨*Compute reflection by integrating over the lights*⟩
⟨*Compute ideal reflection and transmission rays and recursively integrate*⟩
⟨*Clean up from integration*⟩

First, we create a `ShadeContext` object, which holds the information that the surface shader needs to run, including the geometry of the surface at the hit point, the surface material properties, and enough information to perform a hemispherical integral. Note that for the purposes of performing this integral, the reflected ray direction is constant and equal to `-ray.D`.

In order to describe the reflection of light at a point, `lrt` uses a class called `BRDF`, or "Bidirectional Reflection 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, BRDF's use a fraction per-wavelength, so they really return a `Spectrum`). `lrt` provides built-in `BRDF` classes for several standard scattering functions used in computer graphics. Examples of BRDFs include Lambertian reflection, the Phong Model, and the Torrance-Sparrow microfacet model.

The `BRDF` at a surface point provides all information needed to shade that point, but BRDFs may vary across a surface. In fact, in general spatially-varying BRDFs are a function of position as well as incoming and outgoing directions. Surfaces with complex material properties, such as wood or marble, have a different BRDF 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 are described with `Textures`, which in turn may be either described procedurally or stored in texture maps.

⟨*Evaluate BRDF and create surface shader*⟩≡
```
if (!hitInfo->hitPrim->attributes->Surface)
    return L;
BRDF *brdf = hitInfo->hitPrim->attributes->Surface->Shade(shadeContext);
```

Note that the BRDF returned by `Shade` is *partially evaluated*; that is, the surface position and reflected direction are fixed, and so the BRDF is now only a function of the incoming direction.

The next step is to compute the amount of reflected light:

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

```
Point Pw = hitInfo->hitPrim->attributes->ObjectToWorld(hitInfo->Pobj);
const list<Light *> &lights = hitInfo->hitPrim->attributes->Lights;
list<Light *>::const_iterator iter = lights.begin();
while (iter != lights.end()) {
    ⟨Compute reflected light from a light source⟩
    ++iter;
}
```

The class `Light` implements light sources. As with other basic `lrt` objects, there are different types of light sources, including point lights, directional lights, area lights, and ambient lights.

⟨*Compute reflected light from a light source*⟩≡

```
const Light *lt = *iter;
Point Plight;
Spectrum dE = lt->dE(Pw, &Plight);
if (!lt->CastsShadows() || scene->Unoccluded(Pw, Plight)) {
    Vector wi = (Plight - Pw).Hat();
    const Normal &Nw = shadeContext.Ns;
    L += dE * brdf->fr(wi) * Dot(wi, Nw);
}
```

For each light, we compute the light energy, or irradiance, falling on the surface at the point being shaded by calling the light's `dE()` method. This method also returns a point on the light source (for point and directional lights, this has the obvious implementation.). Using the world-space coordinate of the hit point (`Pw`) and the position of the light source (`Plight`), we check if the light is visible and hence the surface point is not in shadow. To evaluate the contribution to the reflection light, we multiple `dE` by the BRDF which is a function of the incoming direction. We also need to include a couple of dot products to make the physics work out right. Finally, we add the contribution from this light source to a running total of reflected radiance, stored in `L`.

In 1979, Turner Whitted developed a new rendering algorithm based onthe fact that light scattered by perfectly specular surfaces (like mirrors or glass objects) could be modeled with the ray-tracing algorithm. When a specularly reflective or transmissive object is hit by a ray, new rays are also traced in the reflected and refracted directions in addition to spawning rays to the light sources. The radiance along these 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 geneated.

Because this process is based on recursive calls to the `WhittedIntegrator`, we keep track of a *ray depth*. After a fixed number of recursive calls, we stop tracing reflected and refracted rays; this can prevent excesively long computation times. By default the depth is five.

`BRDF`s may define one more more specular components (see Section 11.2 on page 144); these represent scattering defined by delta distributions. As such, there is zero probability that a call to the `BRDF`s `fr()` function would happen to hit the specular component. Instead, they must be sampled explicitly.

⟨*Compute ideal reflection and transmission rays and recursively integrate*⟩≡
```
if (++RayDepth < 5) {
    Ray newRay;
    newRay.O = Pw;
    for (int i = 0; i < brdf->SpecularComponents(); ++i) {
        ⟨Sample a specular component⟩
    }
}
--RayDepth;
```

We call the BRDF's `SampleSpecular()` function, which returns us a outgoing direction as a function of the incoming direction, and the fraction of incident light that is scattered along that ray. We when recursively call our `Integrate()` method, which traces the new ray into the scene, computes its scattered radiance, and so forth.

⟨*Sample a specular component*⟩≡
```
Spectrum kernel = brdf->SampleSpecular(i, &newRay.D);
HitInfo hitInfo2;
Float alpha2, maxt = INFINITY;
L += kernel * Integrate(newRay, &hitInfo2, &maxt, &alpha2);
```

That's all there is to it folks!

1.3 Main Include File

The main header file has the usual structure of a header file: it will include some other headers, declare some functions, types, and constants, and define some inline functions. Throughout the rest of the chapters of this book, we will add to the contents of all of these fragments as we go along.

⟨*lrt.h*⟩≡
```
#ifndef LRT_H
#define LRT_H
⟨Global Include Files⟩
⟨Global Type Declarations⟩
⟨Global Forward Declarations⟩
⟨Global Constants⟩
⟨Global Function Declarations⟩
⟨Global Classes⟩
⟨Global Inline Functions⟩
#endif // LRT_H
```

We will 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 egregiously bad stability are avoided, the precision provided by `float` is sufficient in a ray-tracer.

For convenience, we also define shorthand names for unsigned cardinal types: `u_char`, `u_short`, `u_int`, and `u_long`.

⟨*Global Type Declarations*⟩≡
```
typedef float Float;
typedef unsigned char u_char;
typedef unsigned short u_short;
typedef unsigned int u_int;
typedef unsigned long u_long;
```

We will also define a macro that holds `lrt`'s current version number. This is a floating-point value that will be increased as future versions of `lrt` are developed.

⟨*Global Constants*⟩≡
```
#define LRT_VERSION 0.08
```

All files that include `lrt.h` get a number of other include files in the process; this makes it possible for them to just include `lrt.h` and not repeatedly include the others. We try to keep the number of such automatically included files to a minimum; the ones here are necessary for almost all other modules, however.

⟨*Global Include Files*⟩≡
```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

Also, we include files from the standard library to get the `list`, `vector`, and `pair` template classes. The `using` directive brings these container classes into our namespace.

⟨*Global Include Files*⟩+≡
```
#include <list>
using std::list;
#include <vector>
using std::vector;
#ifndef __GNUG__
#include <pair.h>
#endif // !__GNUG__
using std::pair;
```

The only other header file in `lrt` is `ri.h`. It will be defined in Chapter C. This file holds the *external interface* to the renderer—the set of procedure calls that let the user provide the scene description to the renderer. We will reference some of the types and constants defined in `ri.h` in various parts of the implementation to come.

⟨*Global Include Files*⟩+≡
```
#include "ri.h"
```

2. Geometry and Transformations

We now present our representation for the fundamental geometric primitives that `lrt` is built around. 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, and rays. We assume that the reader is familiar with the basics of vector geometry and linear algebra.

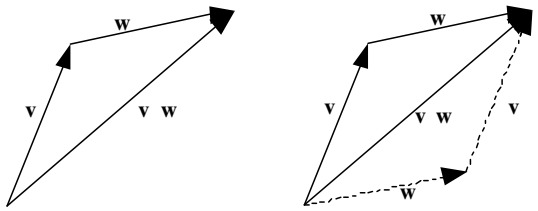
2.1 Vectors

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

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

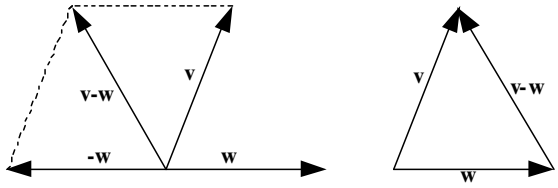
The `Vector` constructor allows values for `x`, `y`, and `z` to be passed in. The default for all these values is 0.0.

⟨*Vector Constructors*⟩≡
```
Vector(Float xx=0.0f, Float yy=0.0f, Float zz=0.0f)
  : x(xx), y(yy), z(zz) {}
```



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

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



(a) (b)

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

Arithmetic.

Adding and subtracting vectors is done component-wise. We assume that the reader is familiar with the normal geometric interpretation of vector addition and subtraction as shown in figures 2.1 and 2.2.

⟨*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:

⟨*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;
}
```

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:

⟨*Vector Methods*⟩+≡
```
Vector operator/ ( Float f ) const {
    Float inv = 1.0/f;
    return Vector(x * inv, y * inv, z * inv);
}

Vector &operator/=( Float f ) {
    Float inv = 1.0/f;
    x *= inv; y *= inv; z *= inv;
    return *this;
}
```

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

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

Normalization.

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

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

Dot and Cross Product.

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

$$x(v)x(w) + y(v)y(w) + z(v)z(w)$$

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

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

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

where θ is the angle between $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$. It follows from this that $(\vec{\mathbf{v}} \cdot \vec{\mathbf{w}})$ is zero if and only if $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$ are perpendicular. In addition, if $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$ are both of unit length, we can replace expensive cosine calculations with their dot product. As the cosine of the angle between two vectors arises often in computer graphics, we will leverage this property frequently.

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

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

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

$$\begin{aligned} x(v \times w) &= (y(v)z(w)) - (z(v)y(w)) \\ y(v \times w) &= (z(v)x(w)) - (x(v)z(w)) \\ z(v \times w) &= (x(v)y(w)) - (y(v)x(w)) \end{aligned}$$

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

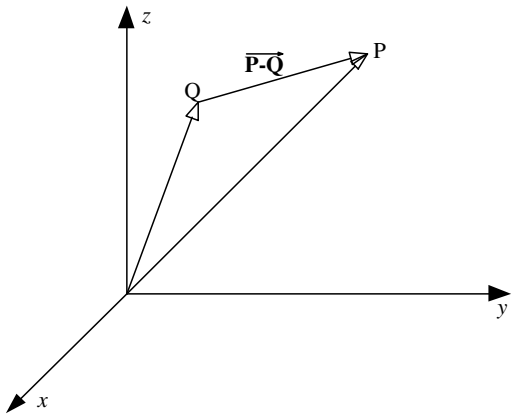


Figure 2.3: Obtaining the vector between two points. The vector $\mathbf{P} \vec{-} \mathbf{Q}$ is the component-wise subtraction of the points **P** and **Q**.

Using the basic properties of the dot product, it can be shown that if $\vec{\mathbf{u}} = \vec{\mathbf{v}} \times \vec{\mathbf{w}}$, then $\|\vec{\mathbf{u}}\| = \|\vec{\mathbf{v}}\| \, \|\vec{\mathbf{w}}\| \sin\theta$, where θ is the angle between $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$. An important implication of this is that the cross product of two perpendicular unit vectors is itself a unit length vector. Note also that the result of the cross product is a degenerate vector if $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$ are parallel.

2.2 Points

A point is a zero-dimensional quantity that represents a location in 3D space. To represent a `Point`, we simply need to know its x, y, and z coordinates. Although it has the same representation as a vector, a `Point` is quite different. As we describe the implementation of the point class, the semantic differences between the two due to their geometric differences should become apparent.

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

⟨*Point Constructors*⟩≡
```
Point(Float xx=0.0, Float yy=0.0, Float zz=0.0 )
  : 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`, obtaining a new `Point`. Alternately, you can subtract one `Point` from another, obtaining a `Vector` between them, as shown in figure 2.3. Adding two `Point`s together or multiplying a point by a scalar have no mathematical meaning and are thus not allowed by the `Point` class.

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

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

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

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

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

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

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

We also provide a `Lerp` function for `Point`s that takes two points and linearly interpolates a position along that line defined by those points. Passing a value of 0 for `t` will return `p1`, and a value of 1 will return `p2`

⟨*Global Function Declarations*⟩≡
```
Point Lerp(Float t, const Point &p1, const Point &p2);
```

⟨*Geometry Functions*⟩≡
```
Point Lerp(Float t, const Point &p1, const Point &p2) {
    return Point(Lerp(t, p1.x, p2.x), Lerp(t, p1.y, p2.y),
        Lerp(t, p1.z, p2.z));
}
```

2.3 Normals

A *surface normal* is a vector that is perpendicular to a two-dimensional surface. One definition of the normal is that it is 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 operations of vectors rather than as an intrinsic quality, they behave differently in subtle ways, especially with respect to transformations. This difference is discussed in section 2.9.

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

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

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

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

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

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

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

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

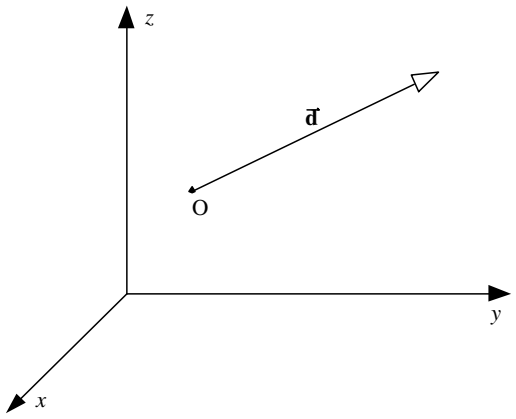


Figure 2.4: A ray is a semi-infinite line defined by its *origin* and *direction*.

2.4 Rays

A *ray* is a semi-infinite line specified by its origin and direction. We represent a `Ray` with a `Point` for the origin, and a `Vector` for the direction. A ray is denoted as $\mathbf{r}$; it has origin $o(\mathbf{r})$ and direction $\vec{d}(\mathbf{r})$, as shown in figure 2.4.

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

$$\mathbf{r}(t) = o(\mathbf{r}) + t\vec{d}(\mathbf{r}) \tag{2.4.1}$$

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

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

Constructing `Ray`s is straightforward. A default constructor is provided, which lets the default constructors of `Point`s and `Vector`s set the origin and direction to $(0,0,0)$. Alternatively, a particular point and direction can be provided.

⟨*Ray Constructor Declarations*⟩≡
```
Ray() {}
Ray( const Point &origin, const Vector &direction ):
    O( origin ), D( direction ) { }
```

Treating a ray as a parametric function, we will overload the function application operator for rays. This will allow us to write code that looks like:

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

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

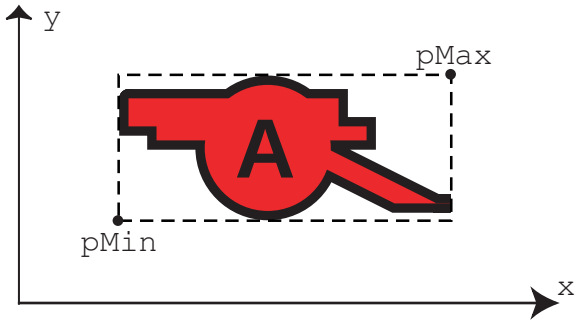


Figure 2.5: An example axis-aligned bounding box. We only store the coordinates of the minimum and maximum points of this box; all other box corners are implicit in this representation.

2.5 Bounding Boxes

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

The 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 quality of 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*. (Another popular choice is a sphere, but a box is often cheaper to process.) A bounding box 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 two of its eight vertices with the minimum and maximum values of each of the x, y, and z vertex positions. A 2D illustration of a bounding box and its reprensetation is shown in figure 2.5.

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

We will not provide publicly-visible default `BBox` constructor; unlike `Point`s and `Vector`s, there isn't a reasonable default value for a bounding box's extent. Instead, we require that the user provide one or two points to define the box. If only one point is given, a degenerate box that holds just that point is created.

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

Since `p1` and `p2` are not necessarily ordered so that `p1.x < p2.x` etc, we need to find their minimum and maximum component-wise values.

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

We will provide a private default constructor, for various `BBox` implementation functions that initialize a new `BBox` and return it. Those methods will initialize `pMin` and `pMax` manually.

⟨*BBox Private Methods*⟩≡
```
BBox() {  }
```

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

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

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

And similarly, we can construct a new bounding box that also encompasses the space encompassed by another bounding box.

⟨*BBox Method Definitions*⟩+≡
```
BBox Union(const BBox &b, const BBox &b2 ) {
    BBox ret;
    ⟨Compute new lower bound⟩
    ⟨Compute new upper bound⟩
    return ret;
}
```

⟨*Compute new lower bound*⟩≡
```
ret.pMin.x = min(b.pMin.x, b2.pMin.x);
ret.pMin.y = min(b.pMin.y, b2.pMin.y);
ret.pMin.z = min(b.pMin.z, b2.pMin.z);
```

⟨*Compute new upper bound*⟩≡
```
ret.pMax.x = max(b.pMax.x, b2.pMax.x);
ret.pMax.y = max(b.pMax.y, b2.pMax.y);
ret.pMax.z = max(b.pMax.z, b2.pMax.z);
```

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

⟨*BBox Method Declarations*⟩+≡
```
friend BBox Intersection(const BBox &b1, const BBox &b2);
```

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

Finally, 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 {
    if (pt.x < pMin.x) return false;
    if (pt.x > pMax.x) return false;
    if (pt.y < pMin.y) return false;
    if (pt.y > pMax.y) return false;
    if (pt.z < pMin.z) return false;
    if (pt.z > pMax.z) return false;
    return true;
}
```

2.6 Affine Spaces

In order to compute numeric coordinates for points and vectors, we need to also have a *coordinate system* that their coordinates are in relation to. An *affine space* is defined by a *frame* given by a point $\mathbf{p}_o$ (the *origin* of the space), and a set of *basis vectors*. In an n-dimensional space, the basis vectors are a set of n linearly independent vectors. All vectors $\vec{\mathbf{v}}$ in the space can be expressed as a linear combination of the basis vectors. Given a vector $\vec{\mathbf{v}}$ and the basis vectors $\vec{\mathbf{v}}_i$, we can compute scalar values s_i such that

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

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

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

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

Using basic properties of linear algebra, it can be shown that (in the three-dimensional case) 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 three different (and incompatible!) ways that a matrix can be interpreted:

1. *Change of coordinates*: given a point expressed in terms of one frame, the matrix could express the location of the same point in a new frame.

2. *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.)

3. *Transformation from one frame to another*: finally, 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.

2.7 Transformations and their Matrix Representation

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

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

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

Linear transformations can be represented by a $4x4$ matrix of scalar values. For the remainder of this chapter, we will assume that the reader is familiar with some basic concepts from linear algebra: how to multiply a matrix by a matrix or a matrix times a column vector, how to compute the transpose of a matrix, etc. A transformation is represented by the elements of the matrix `m[4][4]`. `m` is stored in *row-order* form; to reference the matrix element $m_{i,j}$, where i and j range from zero to three, and where i is the row number and j is the column number, we access element `m[i][j]`.

⟨*Transform Private Data*⟩≡
```
Float m[4][4];
```

Given a frame defined by $(\mathbf{p}, \vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2, \vec{\mathbf{v}}_3)$, there is ambiguity between the representation of a point $(x(p), y(p), z(p))$ and a vector $(x(v), y(v), z(v))$ with equivalent coordinates. However, taking the definition of the representations of points and vectors, we can write the point as $[s_1\, s_2\, s_3\, 1][\vec{\mathbf{v}}_1\, \vec{\mathbf{v}}_2\, \vec{\mathbf{v}}_3\, \mathbf{p}]^T$ and the vector as $[s'_1\, s'_2\, s'_3\, 0][\vec{\mathbf{v}}_1\, \vec{\mathbf{v}}_2\, \vec{\mathbf{v}}_3\, \mathbf{p}]^T$ These four-vectors of s_i and 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$(1, 3, -2)$.

We will not use homogeneous coordinates explicitly in our code; there is no `Homogeneous` class. However, the various transformation routines in the coming sections 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.

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

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

Note that the constructor computes the inverse of the supplied transformation. Transformation inverses are discussed in section 2.9.

⟨*Transform Constructor Declarations*⟩≡
```
Transform(Float t00=1, Float t01=0, Float t02=0, Float t03=0,
          Float t10=0, Float t11=1, Float t12=0, Float t13=0,
          Float t20=0, Float t21=0, Float t22=1, Float t23=0,
          Float t30=0, Float t31=0, Float t32=0, Float t33=1 );
```

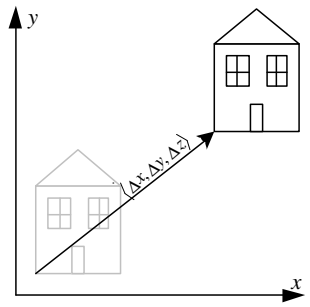


Figure 2.6: Translation in 2D.

⟨*Transform Methods*⟩≡
```
Transform::Transform(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;
    ComputeInverse();
}
```

We also provide constructors that allow initialization from a precomputed 4×4 matrix of `Float`s:

⟨*Transform Constructor Declarations*⟩+≡
```
Transform( Float *m );
Transform( Float m[4][4] );
```

The implementation of these constructors are nearly identical to the one shown above and is omitted.

It will be useful to see if two transforms are equal or not; to test for inequality, we examine all of the elements of the two matrices—if any are unequal, the two `Transform`s are unequal.

⟨*Transform Methods*⟩+≡
```
bool Transform::operator!=(const Transform &t2) const {
    for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 4; ++j)
            if (m[i][j] != t2.m[i][j]) return true;
    return false;
}
```

Translations.

One of the simplest transformations is the translation $\mathbf{T}(\Delta x, \Delta y, \Delta z)$. When applied to a point $\mathbf{p}$, it translates $\mathbf{p}$'s coordinates by Δx, Δy, and Δz, as shown in figure 2.6. The translation has some simple properties:

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

In matrix form, the translation transformation is:

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

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

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

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

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

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

We can now define a routine that creates a new `Transform` matrix that represents a given translation. We define this as a plain global function: rather than operating on a `Transform` that already exists, this returns a new `Transform` with the given translation. Though the possible extra creation of temporary `Transform`s could have a negative performance impact if called frequently, this doesn't have a significant impact on `lrt` since new `Transform`s aren't computed during the main rendering loop after the scene has been specified.

To initialize a `Transform` variable with a translation:

```
Transform t = Translate(0,0,1);
```

⟨*Transform Methods*⟩+≡

```
Transform Translate(const Vector &delta) {
    Transform ret( 1, 0, 0, delta.x,
                   0, 1, 0, delta.y,
                   0, 0, 1, delta.z,
```

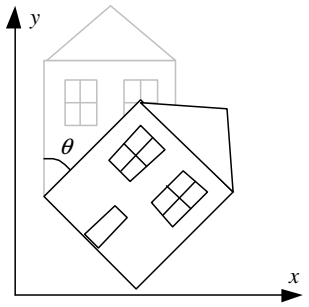


Figure 2.7: Rotation by θ. In this example, the rotation is around the z axis, which is pointing out of the page.

```
                    0, 0, 0,    1     );
    return ret;
}
```

Rotations.

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

Rotations also have some basic properties:

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

where $\mathbf{R}^T$ is the matrix transpose of $\mathbf{R}$. This property, that the inverse of $\mathbf{R}$ is equal to its transpose (a quantity that is much easier to compute than a full matrix inverse!), stems from the fact that we know that $\mathbf{R}$ is an *orthonormal matrix*; its upper $3x3$ components form a basis for three-space and these components each have length 1. (For a good time, verify this from the definition of the rotation matrices below.)

The matrix for rotation around the x axis is

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

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

⟨*Transform Methods*⟩+≡

```
Transform RotateX(Float angle) {
    Float sin_t = sin(Radians(angle));
    Float cos_t = cos(Radians(angle));
    Transform ret( 1,   0,      0,   0,
                   0, cos_t, -sin_t, 0,
                   0, sin_t,  cos_t, 0,
                   0,   0,      0,   1 );
    return ret;
}
```

Similarly, for rotation around y and z, we have

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

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

Finally, we provide rotation around an arbitrary axis. This is commonly done by an approach similar to that used for rotation around an arbitrary point. All rotation matrices can be specified by just their upper left 3×3 submatrix. That is, any 3D rotation matrix is of the form

$$\begin{pmatrix} & & & 0 \\ & R & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Let $\vec{\mathbf{v}} = (x, y, z)$ be the normalized axis of rotation. If we call

$$M = \begin{pmatrix} 0 & -z & y \\ z & 0 & x \\ -y & x & 0 \end{pmatrix}$$

Then the upper left 3×3 matrix R is given by

$$R = vv^{\mathrm{T}} + \cos\theta(I - vv^{\mathrm{T}}) + \sin\theta S$$

⟨*Transform Methods*⟩+≡

```
Transform Rotate(Float angle, const Vector &a) {
    Vector axis = a.Hat();
    Float s = sin(Radians(angle));
    Float c = cos(Radians(angle));
    Float m[4][4];

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

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

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

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

    Transform ret( m );
    return ret;
}
```

Scaling.

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

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

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

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

⟨*Transform Methods*⟩+≡

```
Transform Scale(Float x, Float y, Float z) {
    Transform ret( x, 0, 0, 0,
                   0, y, 0, 0,
                   0, 0, z, 0,
                   0, 0, 0, 1 );
    return ret;
}
```

2.8 Composition of Transformations

Having defined how the matrices representing individual types of transformations are constructed, we can now consider the transformation resulting from a series of individual transformations. It is in this setting that we can see the real value of representing transformations with 4*x*4 matrices. If we were just considering translation, scale, and rotation transformations alone and not in conjunction with each other, then clearly doing the general matrix row-vector multiplication of Section 2.9 is an inefficient way to compute the result of the transformation.

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

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

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

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

⟨*Transform Methods*⟩+≡

```
Transform Transform::operator*(const Transform &t2) const {
    Float retmatrix[4][4];
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
            retmatrix[i][j] = m[i][0] * t2.m[0][j] +
                              m[i][1] * t2.m[1][j] +
                              m[i][2] * t2.m[2][j] +
                              m[i][3] * t2.m[3][j];
        }
    }
    Transform ret( retmatrix );
    return ret;
}
```

Example:Rotation Around an Arbitrary Point.

The rotations described so far all center the rotation around the origin of the active coordinate space. We can use the composition of three transformations in order to rotate around an arbitrary axis that does not pass through the origin. Given an arbitrary axis of rotation defined by a point (x,y,z) and an axis α, and an angle θ to rotate by, the transformation can be constructed in three steps:

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

.

Thus we have

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

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

Efficiency.

The interfaces we have described so far do not lead to the most efficient transformation matrix manipulation routines possible. For example, if we want to compute the matrix that represents the effect of a first translating and then scaling, we write the following code:

```
Transform both = Scale(.5, .5, .5) * Translation(Vector(1,0,0));
```

This requires the construction and initialization of three `Transform` objects, two of which are only needed temporarily, and one full matrix multiplication. In this particular case, we could write specialized routines to try to reduce the number of temporaries and unnecessary operations in computing the product of two matrices, taking advantage of the structure of the matrices. For the purposes of `lrt`, however, these inefficiencies have negligible impact on performance. We will almost always be computing transformations much less often than we will be applying already-computed transformations to `Point`s, `Vector`s, etc, so the greater readability and simplicity of these interfaces is our design choice.

One glaring inefficiency of our approach is the repeated calculation of matrix inverses. Every time a `Transform` is constructed, its inverse is computed whether it is used or not. This is particularly inefficient for temporary transformations computed for simple compositions. A more efficient on-demand computation of inverse transformations would speed up the startup time of certain scenes, and is left as an exercise. In addition, certain types of transformations have very simple inverses which could be supplied explicitly. The reader is referred to Ken Turkowski's Graphics Gem(**?**) for more details on when shortcuts can be taken.

2.9 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.

⟨*Transform Methods*⟩+≡

```
Point Transform::operator()(const Point &pt) const {
    Float x = pt.x, y = pt.y, z = pt.z;

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

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

Vectors.

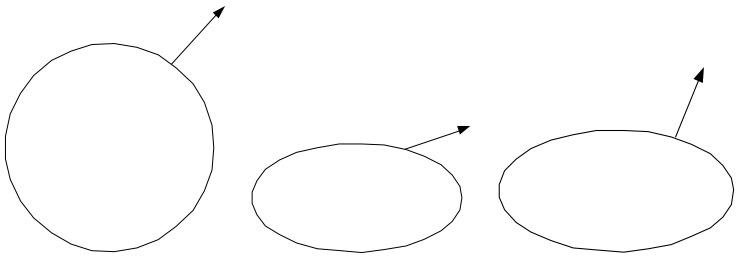
We compute the transformations of vectors in a similar fashion. However, the multiplication of the matrix and the row vector is simplified since the homogeneous w coordinate is zero.

⟨*Transform Methods*⟩+≡

```
Vector Transform::operator()(const Vector &v) const {
    Float x = v.x, y = v.y, z = v.z;

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

    return Vector(xp, yp, zp);
}
```



(a) Original object (b) Scaled object with incorect normal (c) Scaled object with correct normal

Figure 2.8: Transforming surface normals. The circle in (a) is scaled by 50% in the x directon. Note that simply treating the normal as a direction, as shown in (b), will lead to incorrect results.

Normals.

Normals do not transform in the same way that vectors do, as shown in figure 2.8. Although the tangents to the surface transform as expected, normals clearly require special treatment. Mathematically, a surface normal can be thought of as a cross product of two tangent vectors. This definition of surface normal holds regardless of the type of surface; for smooth surfaces, the tangent vectors can be simply the partial derivatives along any two axes.

Because the normal vector N and any tangent vector T are orthogonal by construction, we know that

$$T \cdot N = TN^{\mathrm{T}} = 0.$$

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

$$\begin{aligned} 0 &= T'(N')^{\mathrm{T}} && (2.9.2)\\ &= TM(NS)^{\mathrm{T}} && (2.9.3)\\ &= TMS^{\mathrm{T}}N^{\mathrm{T}} && (2.9.4) \end{aligned}$$

This implies that MS^{T} is the identity matrix, or more precisely that $S = \left(M^{-1}\right)^{\mathrm{T}}$. Therefore, normals are transformed by the inverse transpose of the transformation matrix. This is the main reason why `Transform`s maintain their inverses. In order to compute these inverses, we use the `MatrixInvert` function, defined in appendix A.

⟨*Transform Private Data*⟩+≡
```
Float m_inverse[4][4];
```

⟨*Transform Method Declarations*⟩+≡
```
Float *GetInverse(void) { return &(m_inverse[0][0]); }
```

⟨*Transform Methods*⟩+≡
```
void Transform::ComputeInverse(void) {
    MatrixInvert( m, m_inverse );
}
```

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

⟨*Transform Methods*⟩+≡
```
Normal Transform::operator()(const Normal &n) const {
    Float x = n.x, y = n.y, z = n.z;

    Float xp = m_inverse[0][0] * x +
               m_inverse[1][0] * y +
               m_inverse[2][0] * z;
    Float yp = m_inverse[0][1] * x +
               m_inverse[1][1] * y +
               m_inverse[2][1] * z;
    Float zp = m_inverse[0][2] * x +
               m_inverse[1][2] * y +
               m_inverse[2][2] * z;

    return Normal(xp, yp, zp);
}
```

Rays.

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

⟨*Transform Methods*⟩+≡
```
Ray Transform::operator()(const Ray &r) const {
    Point newO = (*this)(r.O);
    Vector newD = (*this)(r.D);
    return Ray(newO, newD);
}
```

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. (More fun: come up with a bounding box and a transformation that illustrates why the bounding box given by the transformation of the two opposite corner points doesn't necessarily bound the correct bounding box.) 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.

We need to allow `Transform` access to our internal representation so that it can directly access the `BBox`s `pMin` and `pMax` members.

⟨*BBox Private Methods*⟩+≡
```
friend class Transform;
```

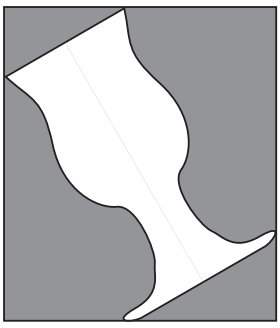
⟨*Transform Methods*⟩+≡
```
BBox Transform::operator()(const BBox &b) const {
    BBox ret((*this)(Point(b.pMin.x, b.pMin.y, b.pMin.z)));
    ret=Union(ret,(*this)(Point(b.pMax.x, b.pMin.y, b.pMin.z)));
    ret=Union(ret,(*this)(Point(b.pMin.x, b.pMax.y, b.pMin.z)));
    ret=Union(ret,(*this)(Point(b.pMin.x,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

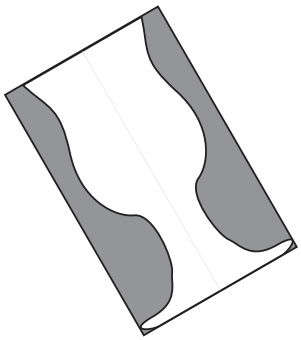
Exercises

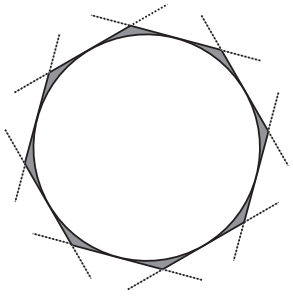
2.1 Our intersection predicate assumes that the bounding box is axis-aligned. Sometimes, it may be useful to have bounding boxes that are allowed to rotate with a moving object. How would you modify the intersection predicate to handle non-axis aligned boxes?

2.2 Implement a more efficient scheme for computing transformation inverses only when needed, and measure the performance impact on several scenes. Add shortcuts for special types of transformations as outlined in Ken Turkowski's Graphics Gem(**?**). Does this have an additional impact on performance? How much harder is the transformation code to understand?

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


Axis-aligned bounding box


Non-axis-aligned bounding box


Arbitrary bounding slabs

3. Geometric Primitives

Every scene is made up of a collection of *geometric primitives*. Each primitive is an object describing some shape or shapes; each supports some basic operations, such as ray-primitive intersection, bounding box computation, and texture coordinate mappings.

We will start this chapter by describing an algorithm for computing intersections between rays and axis-aligned bounding boxes. We then describe the `Primitive` class, which describes the abstract interface for geometric primitives, and then the implementatiion of a number of primitives.

3.1 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 a box, we intersect the ray against each of the 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: three points uniquely define a plane; as does point and two vectors. Here, we will define a plane by a point on the plane $\mathbf{p}$ and the plane normal $\hat{\mathbf{n}}$.

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

$$((\mathbf{p}' - \mathbf{p}) \cdot \hat{\mathbf{n}}) = 0$$

Thus, given a ray $\mathbf{r}$ defined by $\mathbf{r} = o(\mathbf{r}) + t\vec{d}(\mathbf{r})$, we substitute

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

Using basic definitions of the dot product, we have

$$\begin{aligned}((o(\mathbf{r})-\mathbf{p})\cdot\hat{\mathbf{n}})+(t\vec{d}(\mathbf{r})\cdot\hat{\mathbf{n}}) &= 0\\ ((o(\mathbf{r})-\mathbf{p})\cdot\hat{\mathbf{n}})+t(\vec{d}(\mathbf{r})\cdot\hat{\mathbf{n}}) &= 0\\ t(\vec{d}(\mathbf{r})\cdot\hat{\mathbf{n}}) &= -((o(\mathbf{r})-\mathbf{p})\cdot\hat{\mathbf{n}})\\ t &= -\frac{((o(\mathbf{r})-\mathbf{p})\cdot\hat{\mathbf{n}})}{(\vec{d}(\mathbf{r})\cdot\hat{\mathbf{n}})}\end{aligned}$$

As long as $(\vec{d}(\mathbf{r})\cdot\hat{\mathbf{n}})$ is not zero (which would indicate that the ray is parallel to the plane), t is defined. If t is less than zero, the ray faces away from the plane and never intersects it. See Figure 3.1.

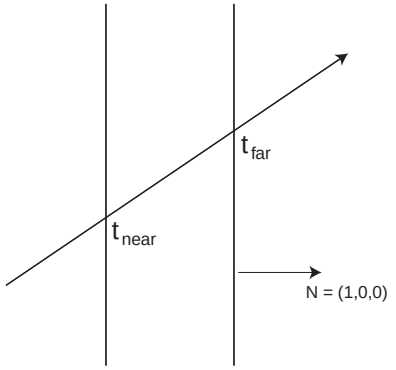


Figure 3.1: Intersecting a ray with a pair of slabs. Each of the slabs shown here is a plane given by $x = c$, for some constant value c. The normal of each slab is $(0,0,1)$.

The basic ray–bounding box 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, giving us values t_{near} and t_{far}. We 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 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 3.2 illustrates this process.

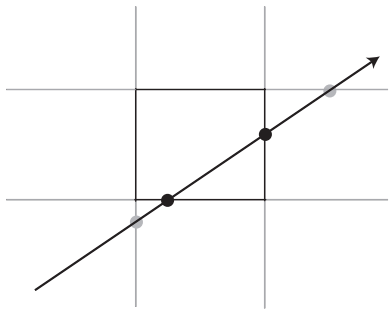


Figure 3.2: Intersecting a ray with an axis-aligned bounding box. We compute intersection points with each pair of slabs, progressively narrowing the pair of intersection points.

The routine to compute the intersection is called `IntersectP` . `IntersectP` is a *predicate* function, meaning that its main purpose is to return a boolean value. If the function returns true, the intersection distance (which is just the minimum value of the interval range from the above calculation) can be returned in the optional argument `t`. Furthermore, if `t` is not `NULL`, then it provides a maximum distance beyond which intersections are not detected.

⟨*BBox Method Definitions*⟩+≡
```
bool BBox::IntersectP( const Ray &ray, Float *t ) const {
    ⟨Initialize parametric interval⟩
    ⟨Check X slab⟩
    ⟨Check Y slab⟩
    ⟨Check Z slab⟩
    if (t != NULL) *t = t0;
    return true;
}
```

We use `t0` and `t1` to hold the current parametric interval of interest. We initialize it to go from zero the user-supplied maximum value, if any, or otherwise to infinity.

⟨*Initialize parametric interval*⟩≡
```
Float t0 = 0., t1;
if (t) t1 = *t;
else   t1 = INFINITY;
```

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 t_1. From the ray–plane test above, we have:

$$t_1 = -\frac{((o(\mathbf{r})-(x_1,0,0))\cdot(1,0,0))}{(\vec{d}(\mathbf{r})\cdot(1,0,0))}$$

Because two of the components of the normal are zero, we can use the definition of the dot product to simplify this substantially:

$$\begin{aligned} t_1 &= -\frac{o(\mathbf{r}_\mathbf{x})-x_1}{\vec{d}(\mathbf{r}_\mathbf{x})} \\ t_1 &= \frac{x_1-o(\mathbf{r}_\mathbf{x})}{\vec{d}(\mathbf{r}_\mathbf{x})} \end{aligned} \tag{3.1.1}$$

The code for the x slab is shown here; the code for the y and z slabs is nearly identical and is omitted. We start by computing the reciprocal of the x component of the ray direction. We will then multiply by this factor when we would otherwise divide by the x direction component; this saves a potentially-expensive divide. We do not need to verify that the x direction component is not zero; if it is, then `invRayDir` will hold an infinite value[1], either $-\infty$ or ∞, and the rest of the algorithm works correctly in this case.

⟨*Check X slab*⟩≡
```
Float invRayDir = 1. / ray.D.x;
Float tNear = (pMin.x - ray.O.x) * invRayDir;
Float tFar  = (pMax.x - ray.O.x) * invRayDir;
⟨Update parametric interval⟩
```

[1] This assumes that the architecture being used supports IEEE floating point arithmetic; this is universal on modern systems.

We then swap the two distances, so that t_{near} holds the closer intersection and t_{far} the farther one. This gives us a parametric range $[t_{\text{near}}, t_{\text{far}}]$. We compute the intersection of this with the current range $[t_0, t_1]$ to compute a new range. If this new range is empty (i.e. $t_0 > t_1$), then we return failure.

⟨*Update parametric interval*⟩≡
```
if (tNear > tFar) swap(tNear, tFar);
t0 = max(tNear, t0);
t1 = min(tFar,  t1);
if (t0 > t1) return false;
```

3.2 Basic Primitive Interface

Each specific primitive is a subclass of the `Primitive` base class, which presents the common interface that will be implemented by each specific primitive. For example, we would like to intersect a ray with a `Primitive` without knowledge about the actual type of primitive (triangle, sphere, etc). This abstraction strategy makes extending the geometric capabilities of the system quite straightforward.

Creation.

All primitives in the scene have `PrimitiveAttributes` and `SurfaceFunction` classes associated with them. These classes are used to help compute shading over the surfaces and are described later, in Sections 3.2 on page 42 and 9.1 on page 117. For now, we will just ensure that we store one of each with each primitive.

⟨*Primitive Interface*⟩≡
```
Primitive(PrimitiveAttributes *a, SurfaceFunction *sf)
    : attributes(a), surfaceFunction(sf) { }
```

⟨*Primitive Public Data*⟩≡
```
PrimitiveAttributes *attributes;
SurfaceFunction *surfaceFunction;
```

When destroying a `Primitive` we make sure to properly free its attribute and surface function data resources. Each primitive has its own unique `SurfaceFunction`, which we just `delete`. However, its `PrimitiveAttributes` may be shared among primitives, so is reference counted; we just call its `Dereference` function to record that one fewer reference to the attributes is being held.

⟨*Primitive Methods*⟩≡
```
Primitive::~Primitive() {
    delete surfaceFunction;
    attributes->Dereference();
}
```

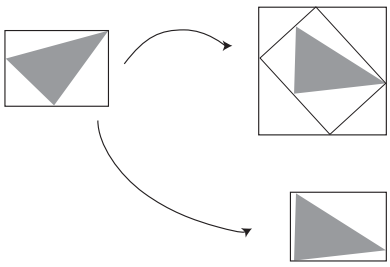


Figure 3.3: If we compute a world-space bounding box of a triangle by transforming its object-space bounding box to world space and then finding the bounding box that encloses the resulting bounding box, a sloppy bound may result (top). However, if we first transform its vertices from object space to world space and then bound those vertices (bottom), we can do much better.

Bounding.

Each `Primitive` subclass must be capable of bounding itself with a bounding box. There are two different bounding methods. The first, `BoundObjectSpace` returns a bounding box in the primitive's object space. The second, `BoundWorldSpace` returns a bounding box in world space. The implementation of the first method obviously must be left up to each individual primitive.

There is a default implementation of the second method, but primitives 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. An example of such a primitive is a triangle; see Figure 3.3.

⟨*Primitive Interface*⟩+≡
```
virtual BBox BoundObjectSpace() const = 0;
```

⟨*Primitive Methods*⟩+≡
```
BBox Primitive::BoundWorldSpace() const {
    return attributes->ObjectToWorld(BoundObjectSpace());
}
```

Refinement.

Not every primitive is capable of directly intersecting a ray with itself. For instance, we might have a special primitive 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 when needed. We can't intersect a ray with such a primitive directly, so its `CanIntersect` routine would return a false boolean value. The default implementation of this function indicates that a primitive can provide an intersection, so only primitives that can not need override this method.

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

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

⟨*Primitive Methods*⟩+≡
```
void Primitive::Refine(vector<Primitive *> *refined) const {
    Severe("Unimplemented Primitive::Refine() method called");
}
```

Intersection.

Finally, we provide two separate intersection routines. The first, `Primitive::IntersectClosest`, returns a single ray-primitive intersection corresponding to the intersection closest to the ray origin. The other, `IntersectAll` returns a complete list of ray-object intersections. This will be useful when we implement constructive solid geometry, where all object hits must be considered simultaneously. A primitive must supply an implementation of either `Primitive::IntersectClosest` *or* `Refine`, but not both. Depending on the value returned by `CanIntersect`, the appropriate one will be called.

Both functions return a boolean value indicating whether or not there were any intersections. In addition to the `Ray` that is passed in, the user must supply `mint` and `maxt` variables. The first, `mint` gives a minimum distance along the ray before which intersections should not be detected. The second, `maxt` serves double duty: it both gives a maximum distance beyond which intersections shouldn't be detected, but is also used to store the distance to the intersection found, if any. The representation of a ray-primitive intersection that is stored in the `HitInfo` structure is described in detail in Section 3.2 on the next page. The `HitInfo` pointer to `Primitive::IntersectClosest` may be `NULL`; if so, this indicates that not only is the caller not interested in detailed intersection information, but also that after *any* intersection before `maxt` is found, the routine may return; it's not necessary to find the closest intersection.

⟨*Primitive Interface*⟩+≡
```
virtual bool IntersectClosest(const Ray &ray, Float mint,
    Float *maxt, HitInfo *hit) const;
```

⟨*Primitive Methods*⟩+≡
```
bool Primitive::IntersectClosest(const Ray &ray, Float mint,
        Float *maxt, HitInfo *hit) const {
    Severe("Unimplemented _IntersectClosest() method called");
    return false;
}
```

The default implementation of `IntersectAll` simply assumes that the primitive will only ever generate one intersection.

⟨*Primitive Methods*⟩+≡
```
bool Primitive::IntersectAll(const Ray &ray, Float mint,
        Float maxt, list<HitInfo *> &hits) const {
    bool anyHit = false;
    HitInfo hitInfo;
    if (IntersectClosest(ray, mint, &maxt, &hitInfo)) {
        anyHit = true;
        hits.push_back(new HitInfo(hitInfo));
    }
    return anyHit;
}
```

Recording Intersections.

We need a self-contained representation that holds relevant information about a particular ray-primitive intersection. This representation should contain all of the information necessary to perform shading and lighting calculations at the hit point–in particular, it needs to abstract away the particular type of geometric primitive that was hit, allowing the rest of the renderer to be implemented generically, not considering different primitive representations. The information that we will store to do this includes the three-dimensional coordinates of the hit point and the surface normal in object space coordinates, the (u,v) parametric coordinates of the point on the surface, and any additional surface parameters.

In `HitInfo` we will store the hit point, *geometric normal*, and parametric coordinates. When we perform shading computations, we may use a user-supplied *shading normal* instead of the geometric normal if one exists.

⟨*HitInfo Data*⟩≡
```
Point Pobj;
Normal NgObj;
Float u, v;
```

We'll also store a pointer to the `Primitive` that was hit, so the shading system can access properties of the surface. Note that this is a pointer to the parent class, so only very generic access to a primitive is possible.

⟨*HitInfo Data*⟩+≡
```
const Primitive *hitPrim;
```

Instead of setting the `HitInfo` members themselves, primitives should call `HitInfo`'s `RecordHit` method to report information about an intersection. The `RecordHit` method just stores the data the primitive is giving us.

⟨*HitInfo Method Definitions*⟩+≡
```
void HitInfo::RecordHit(const Point &Po, const Normal &No,
        Float uu, Float vv, const Primitive *prim) {
    Pobj = Po;
    NgObj = No;
    u = uu, v = vv;
    hitPrim = prim;
}
```

Primitive Attributes.

Each geometric primitive in our system has a set of attributes associated with it. We store these in a class called `PrimitiveAttributes`. This holds information about how the primitive is located in the scene (i.e. its object space to world space and world space to object space transformations), which surface shader is bound to it, etc. All of these attributes are set and managed in the interface layer described in Appendix C.

⟨*PrimitiveAttributes Constructor Declarations*⟩≡
```
PrimitiveAttributes() {
    ⟨PrimitiveAttributes constructor implementation⟩
}
```

Each `PrimitiveAttributes` holds two transformation matrices, one that transforms from object to world space (`ObjectToWorld`), and one that goes back from world space to object space (`WorldToObject`). We just let these be initialized to the default identity transformation, since the RI layer should override them.

⟨*PrimitiveAttributes Public Data*⟩≡
```
Transform ObjectToWorld, WorldToObject;
```

3.3 Triangles

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

- `nt` Number of triangles in this mesh
- `nv` Number of vertices in this mesh
- `vi` Pointer to an array of vertex indices. For the `i`th triangle, its three vertex positions are `P[3*i]`, `P[3*i+1]`, and `P[3*i+2]`.
- `P` Array of `nv` vertex positions.

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

⟨*TriangleMesh Methods*⟩≡
```
TriangleMesh::TriangleMesh(int nt, int nv, int *vi, Point *P,
        PrimitiveAttributes *a, SurfaceFunction *sf)
            : Primitive(a, sf) {
    ntris = nt;
    nverts = nv;
    vertexIndex = new int[3 * ntris];
    memcpy(vertexIndex, vi, 3 * ntris * sizeof(int));
    p = new Point[nverts];
    memcpy(p, P, nverts * sizeof(Point));
}
```

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

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

The object-space bound of a triangle mesh is easily found by computing a bounding box that encompasses all of the vertices of the mesh.

⟨*TriangleMesh Methods*⟩+≡
```
BBox TriangleMesh::BoundObjectSpace() const {
    BBox objectSpaceBounds(p[0], p[1]);
    for (int i = 2; i < nverts ; i++)
        objectSpaceBounds = Union(objectSpaceBounds, p[i]);
    return objectSpaceBounds;
}
```

The `TriangleMesh` primitive is one of the primitives that can usually compute a better world space bound than can be found by transforming its object-space bounding box to world space. We transform each vertex to world space and compute a bounding box of those vertices.

⟨*TriangleMesh Methods*⟩+≡
```
BBox TriangleMesh::BoundWorldSpace() const {
    BBox worldBounds(attributes->ObjectToWorld(p[0]),
        attributes->ObjectToWorld(p[1]));
    for (int i = 2; i < nverts ; i++) {
        worldBounds = Union(worldBounds,
                            attributes->ObjectToWorld(p[i]));
    }
    return worldBounds;
}
```

The `TriangleMesh` primitive does not directly compute intersections. Instead, it splits itself into many separate `Triangle`s, 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 `Primitive` to indicate that `TriangleMesh`s can not be intersected directly.

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

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

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

The `Triangle` doesn't store much data; just a pointer to the parent `TriangleMesh` that it came from and an integer that is the triangle number that this item represents.

⟨*Triangle Interface*⟩≡
```
Triangle(const TriangleMesh *m, int n)
        : Primitive(m->attributes, NULL) {
    attributes->Reference();
    mesh = m;
    triNum = n;
}
```

⟨*Triangle Data*⟩≡
```
const TriangleMesh *mesh;
int triNum;
```

As with `TriangleMesh`es, we can compute better world space bounding boxes for individual triangles by transforming their vertices to world space and then bounding them.

⟨*TriangleMesh Methods*⟩+≡
```
BBox Triangle::BoundObjectSpace() const {
    BBox objectBounds(mesh->p[mesh->vertexIndex[triNum*3]],
                      mesh->p[mesh->vertexIndex[triNum*3 + 1]]);
    return Union(objectBounds,
                 mesh->p[mesh->vertexIndex[triNum*3 + 2]]);
}

BBox Triangle::BoundWorldSpace() const {
    Transform o2w = mesh->attributes->ObjectToWorld;
    BBox objectBounds(o2w(mesh->p[mesh->vertexIndex[triNum*3]]),
                      o2w(mesh->p[mesh->vertexIndex[triNum*3 + 1]]));
    return Union(objectBounds,
                 o2w(mesh->p[mesh->vertexIndex[triNum*3 + 2]]));
}
```

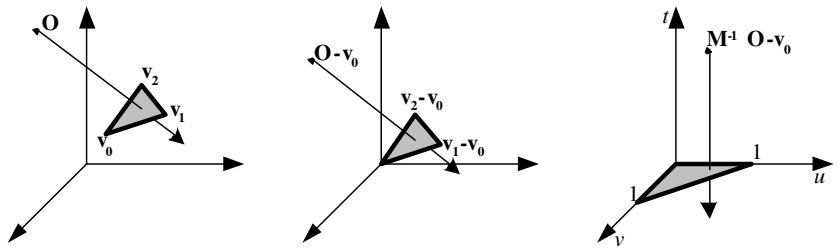


Figure 3.4: Transforming the ray into a more convenient coordinate system for intersection. First, a translation is applied to make a corner of the triangle coincide with the origin. Then, the triangle is rotated and scaled to a unit right-triangle.

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, u and v:

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

The conditions on u and v are that $u \geq 0$, $v \geq 0$, and $u+v \leq 1$. Here we have described the surface in *parametric form*: given u and v surface parameter values, a position on the surface $\mathbf{p}(u,v)$ can be computed. These barycentric coordinates are 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 $(1-u-v)a_0 + ua_1 + va_2$.

To derive an algorithm for intersecting a ray with a triangle, we start with the same technique as for the ray-box intersection: we insert the parametric ray equation intro the triangle equation.

$$o(\mathbf{r}) + t\vec{d}(\mathbf{r}) = (1-u-v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2 \tag{3.3.2}$$

Following the technique described by Möller and Trumbore(**?**), we use the shorthand notation $\vec{\mathbf{e}}_1 = \mathbf{p}_1 - \mathbf{p}_0$, $\vec{\mathbf{e}}_2 = \mathbf{p}_2 - \mathbf{p}_0$, and $\vec{\mathbf{t}} = o(\mathbf{r}) - \mathbf{p}_0$. We can now rearrange terms to obtain the matrix equation:

$$\begin{bmatrix} -\vec{d}(\mathbf{r}) & \vec{\mathbf{e}}_1 & \vec{\mathbf{e}}_2 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \vec{\mathbf{t}}$$

Solving this linear system will give us not only the barycentric coordinates of the intersection point (which can easily be used to compute the 3D intersection point), but also 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 3.4.

Cramer's rule gives a solution to equation 3.3.3:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\left|-\vec{d}(\mathbf{r}) \quad \vec{\mathbf{e}}_1 \quad \vec{\mathbf{e}}_2\right|} \begin{bmatrix} \left| \vec{\mathbf{t}} \quad \vec{\mathbf{e}}_1 \quad \vec{\mathbf{e}}_2 \right| \\ \left|-\vec{d}(\mathbf{r}) \quad \vec{\mathbf{t}} \quad \vec{\mathbf{e}}_2\right| \\ \left|-\vec{d}(\mathbf{r}) \quad \vec{\mathbf{e}}_1 \quad \vec{\mathbf{t}}\right| \end{bmatrix} \tag{3.3.3}$$

This can be rewritten as $\left|\vec{\mathbf{A}} \quad \vec{\mathbf{B}} \quad \vec{\mathbf{C}}\right| = -\left(\vec{\mathbf{A}}\times\vec{\mathbf{C}}\right)\cdot\vec{\mathbf{B}} = -\left(\vec{\mathbf{C}}\times\vec{\mathbf{B}}\right)\cdot\vec{\mathbf{A}}$. We can thus rewrite Equation 3.3.3 as:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\left(\vec{d}(\mathbf{r})\times\vec{\mathbf{e}}_2\right)\cdot\vec{\mathbf{e}}_1} \begin{bmatrix} \left(\vec{\mathbf{t}}\times\vec{\mathbf{e}}_1\right)\cdot\vec{\mathbf{e}}_2 \\ \left(\vec{d}(\mathbf{r})\times\vec{\mathbf{e}}_2\right)\cdot\vec{\mathbf{t}} \\ \left(\vec{\mathbf{t}}\times\vec{\mathbf{e}}_1\right)\cdot\vec{d}(\mathbf{r}) \end{bmatrix} \tag{3.3.4}$$

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

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\vec{\mathbf{s}}_1\cdot\vec{\mathbf{e}}_1} \begin{bmatrix} \vec{\mathbf{s}}_2\cdot\vec{\mathbf{e}}_2 \\ \vec{\mathbf{s}}_1\cdot\vec{\mathbf{t}} \\ \vec{\mathbf{s}}_2\cdot\vec{d}(\mathbf{r}) \end{bmatrix} \tag{3.3.5}$$

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

⟨*TriangleMesh Methods*⟩+≡
```
bool Triangle::IntersectClosest(const Ray &ray, Float mint,
        Float *maxt, HitInfo *hit) const {
    ⟨Initialize triangle intersection statistics⟩
    ⟨Update triangle tests count⟩
    ⟨Compute P · E1⟩
    ⟨Compute first barycentric coordinate⟩
    ⟨Compute second barycentric coordinate⟩
    ⟨Compute t to intersection point⟩
    if (hit != NULL) {
        ⟨Fill in HitInfo from triangle hit⟩
    }
    return true;
}
```

First, we compute the divisor from Equation 3.3.5. We figure out which three mesh vertices are the ones for this particular `Triangle`, and then compute the edge vectors and divisor. Note that if the divisor is zero, this triangle is degenerate and therefore cannot intersect a ray.

⟨*Compute* $P\cdot E_1$⟩≡
```
int *vptr = &(mesh->vertexIndex[3*triNum]);
int vertexIndex0 = *vptr++;
int vertexIndex1 = *vptr++;
int vertexIndex2 = *vptr++;

const Point *p = mesh->p;
Vector E1 = p[vertexIndex1] - p[vertexIndex0];
Vector E2 = p[vertexIndex2] - p[vertexIndex0];
Vector S_1 = Cross( ray.D, E2 );
Float divisor = Dot( S_1, E1 );
if (divisor == 0.)
    return false;
Float invDivisor = 1. / divisor;
```

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

⟨*Compute first barycentric coordinate*⟩≡
```
Vector T = ray.O - mesh->p[vertexIndex0];
Float u = Dot( T, S_1 ) * invDivisor;
if (u < 0. || u > 1.0)
    return false;
```

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

⟨*Compute second barycentric coordinate*⟩≡
```
Vector S_2 = Cross( T, E1 );
Float v = Dot( ray.D, S_2 ) * invDivisor;
if (v < 0 || u + v > 1.0)
    return false;
```

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

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

We now have all the information we need to compute the `HitInfo` structure for this intersection.

⟨*Fill in* `HitInfo` *from triangle hit*⟩≡
```
Normal N = Normal(Cross(E2, E1));
N.Normalize();
hit->RecordHit(ray(t), N, u, v, mesh );
```

3.4 Spheres

Spheres are a special case of a surface called *quadrics*. Quadrics are surfaces described by quadratic polynomials in x, y, and z; they are the simplest type of curved surface that is useful to a ray tracer, and are an interesting introduction to more general ray intersection routines. The sphere is the simplest quadric. The RenderMan specification provides seven quadrics: spheres, cones, disks (a special case of a cone), cylinders, hyperboloids, paraboloids, and (although they are not actually quadrics) tori.

Surfaces like quadrics are be 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 $x^2 + y^2 + z^2 - 1 = 0$.

We can also describe a sphere parametrically; the relevant equations are:

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

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

Construction.

Because the API to `lrt` is based on the RenderMan interface (see Appendix C), our `Sphere` class (as well as all the quadrics) has some non-obvious behaviors. All spheres are centered at the origin in object space; to place them elsewhere in the scene, the user must apply appropriate transformations when specifying them in the input file.

The radius of the sphere can have an arbitrary value, though its 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 section of the sphere with spherical theta values above this theta is also removed.

⟨*Sphere Methods*⟩≡

```
Sphere::Sphere( Float rad, Float z0, Float z1, Float tm,
        PrimitiveAttributes *a, SurfaceFunction *sf)
    : Primitive( a, sf ) {
    radius = rad;
    zmin = Clamp(min(z0, z1), -radius, radius);
    zmax = Clamp(max(z0, z1), -radius, radius);
    thetaMax = Radians( Clamp( tm, 0, 360 ) );
}
```

⟨*Sphere Data*⟩≡

```
Float radius;
Float zmin, zmax;
Float thetaMax;
```

Bounding.

Computing a bounding box for a sphere is straightforward. We will use the values of `zmin` and `zmax` 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 `thetaMax` and see if we can compute a tighter bounding box when that is less than 2π.

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

Intersection.

Because we know that the sphere is centered at the origin, our task for deriving an intersection test is easier than it would be in a more general setting. The object space center point of all spheres in `lrt` is located at the origin. 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.[2] For all primtives, the higher level routine that calls the intersection method will be responsible for passing a ray in the approrpriate coordinate space.

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, **??** into the implicit sphere equation, we have:

$$\Big(x(o(\mathbf{r})) + tx(\vec{d}(\mathbf{r}))\Big)^2 + \Big(y(o(\mathbf{r})) + ty(\vec{d}(\mathbf{r}))\Big)^2 + \Big(z(o(\mathbf{r})) + tz(\vec{d}(\mathbf{r}))\Big)^2 = r^2$$

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

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

where

$$\begin{aligned} A &= x(\vec{d}(\mathbf{r}))^2 + y(\vec{d}(\mathbf{r}))^2 + z(\vec{d}(\mathbf{r}))^2 && (3.4.6)\\ B &= 2(x(\vec{d}(\mathbf{r}))x(o(\mathbf{r})) + y(\vec{d}(\mathbf{r}))y(o(\mathbf{r})) + z(\vec{d}(\mathbf{r}))z(o(\mathbf{r}))) && (3.4.7)\\ C &= x(o(\mathbf{r}))^2 + y(o(\mathbf{r}))^2 + z(o(\mathbf{r}))^2 - r^2 && (3.4.8) \end{aligned}$$

This directly translates to the starting bit of source code.

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

[2] 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.

By the quadradic equation, we know there are two possible solutions to this equation:

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

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

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

If there are two roots, t_0 must be less than t_1, so there is no need to swap the values. Note that we also check t_0 against the user supplied `maxt` and `mint` values, rejecting the hit if it falls outside of the required range.

⟨*Compute quadric* `t` *values*⟩≡
```
Float inv2A = 1. / (2. * A);
Float t0 = (-B - rootDiscrim) * inv2A;
if (t0 > *maxt)
    return false;
Float t1 = (-B + rootDiscrim) * inv2A;
if (t1 < mint)
    return false;
```
⟨*Compute first intersection point*⟩

We now determine which hit is actually first. Although t_0 is guaranteed to be smaller than t_1, t_0 may be less than `mint`. The most common case when this would occur would be when the ray originates inside the sphere.

⟨*Compute first intersection point*⟩≡
```
Float thit = t0;
if (t0 < mint) thit = t1;
if (thit > *maxt) return false;
Phit = ray(thit);
```

In the worst case, this algebraic method incurs a computation cost of 9 additions, 15 multiplies, 1 square root, and 1 divide. Note that the fragments ⟨*Find quadric discriminant*⟩ and ⟨*Compute quadric* `t` *values*⟩ are not specific to spheres, and will be re-used by all of the quadric solvers to follow.

Parameterization.

Now that we have the 3D coordinates of the hit, we need to compute the *inverse mapping* for the hit. This will give us a (u,v) parameterization of the sphere's surface which will be used in shading. The standard spherical parameterization is by latitude and longitude.

To compute these values, we need the unit surface normal at the hit point, which we will call N. Because our sphere is centered at the origin, N is just along the direction from the origin to the hit point. If we wish to normalize N, we know its length already; that's just the sphere radius.

We will also vectors describing the sphere itself. The first is a unit vector pointing to the "north pole" of the sphere, which we will call P. The second is a unit vector pointing anywhere on the equator, which we will call E. The place on the equator pointed to by E will determine where zero longitude lies.

⟨*Compute sphere inverse mapping*⟩≡
  ⟨*Compute latitude*⟩
  ⟨*Compute longitude*⟩

First, we will compute the latitude as the v parameter. The latitude is simply the angle between N and P, scaled to between 0 and 1:

$$\begin{aligned} \phi &= \cos^{-1}(-N \cdot P) \\ v &= \frac{\phi}{\pi} \end{aligned}$$

The negation of N is required so that zero latitude falls at the "south pole" of the sphere. Also, our sphere has $P = (0,0,1)$, which greatly simplifies the dot product to simply the z component of N:

⟨*Compute latitude*⟩≡
```
phi = acos(-Phit.z / radius);
v = phi / M_PI;
```

The longitude is only slightly more complicated. First, if we have exactly hit one of the poles of the sphere, we arbitrarily assign a longitude of zero. Otherwise, we compute:

$$\begin{aligned} x &= \cos^{-1}\left(\frac{E\cdot N}{\sin(\phi)}\right) \\ \theta &= \frac{x}{2\pi} \end{aligned}$$

θ is a value from 0 to .5. To extract the longitude, we must determine which side of E the hit lies on. To do this, we compute the cross product between P and E. The sign (not sine!) of the angle between this cross product and N will give the side of E:

$$u = \begin{cases} \theta & (P\times E)\cdot N > 0 \\ 1-\theta & \text{otherwise} \end{cases}$$

Our sphere uses $E = [1,0,0]$, which again simplifies the dot product, this time to just the x coordinate of the unit normal.

⟨*Compute longitude*⟩≡
```
if (v == 0 || v == 1) {
    u = 0;
} else {
    Float val = Clamp( Phit.x / (radius*sin(phi)), -1, 1 );
    theta = acos(val) / (2*M_PI);
    if (Phit.y > 0)
        u = theta;
    else
        u = 1 - theta;
    u *= (2*M_PI)/thetaMax;
}
```

Final Analysis.

Now that we have a parameterization of the sphere, we must test the hit values against the specified minima and maxima for z and $theta$. If t_0 is rejected, we try again with t_1. Note that the inverse mapping must be re-computed in this case and the new hit tested again. Fortunately, there are only two hits to try.

⟨*Test sphere intersection against clipping parameters*⟩≡
```
if (Phit.z < zmin || Phit.z > zmax || u > 1.) {
    if (thit == t1) return false;
    if (t1 > *maxt) return false;
    thit = t1;
    Phit = ray(thit);
    ⟨Compute sphere inverse mapping⟩
    if (Phit.z < zmin || Phit.z > zmax || u > 1.)
        return false;
}
```

At this point, we are sure that the ray hits the sphere, and we can fill in the `HitInfo` structure. The normal at the hit point is simply the vector from the origin to the hit point. In additon, the computed values of u and v are scaled so that values from 0 to 1 always represent points on the surface.

⟨*Fill in* `HitInfo` *from sphere hit*⟩≡
```
Float vmin = acos(-zmin/radius);
Float vmax = acos(-zmax/radius);
v = (v*M_PI - vmin)/(vmax-vmin);
Normal surfNorm = Normal( Phit - Point(0,0,0) );
surfNorm.Normalize();
hit->RecordHit(Phit, surfNorm, u, v, this );
```

The entire intersection method is shown below. Notice that we need to test `hit` to see if it is non-NULL; we sometimes won't need the details of the intersection, but will just want to know whether or not there is an intersection. In this case, a NULL `HitInfo` can be passed in. Note that we cannot short-circuit the inverse-mapping computation in that case because the resulting information is needed to test against the z and θ clipping parameters.

⟨*Sphere Methods*⟩+≡
```
bool Sphere::IntersectClosest(const Ray &ray, Float mint,
        Float *maxt, HitInfo *hit) const {
    Point Phit;
    Float u, v;
    Float theta, phi;
    ⟨Compute quadratic sphere coefficients⟩
    ⟨Find quadric discriminant⟩
    ⟨Compute quadric t values⟩
    ⟨Compute sphere inverse mapping⟩
    ⟨Test sphere intersection against clipping parameters⟩
    if (hit) {
        ⟨Fill in HitInfo from sphere hit⟩
    }
    *maxt = thit;
    return true;
}
```

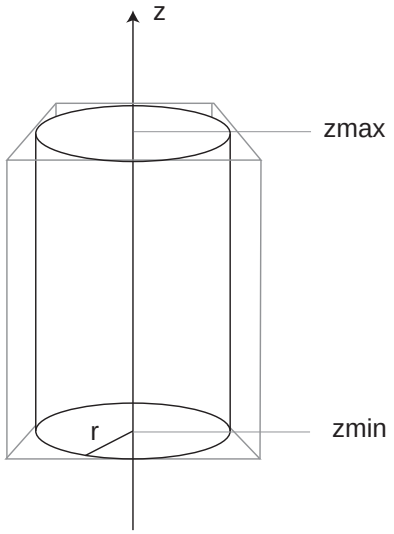


Figure 3.5: Basic setting for the cylinder primitive. It has a radius of r and is covers a range of heights along the z-axis. A partial cylinder may be swept by specifying a maximum θ value.

3.5 Cylinders

`lrt` provides cylinders primitives 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 theta sweep value.

Construction.

Cylinders have a user-specified radius, as well as a minimum and maximum z value. Like the sphere, they also have a maximum θ value that allows partial cylinders to be rendered. In parametric form, a cylinder is described by the equations:

$$\begin{aligned} \theta &= u\,\theta_{\max} \\ x &= r\cos\theta \\ y &= r\sin\theta \\ z &= v\,(z_{\max} - z_{\min}) \end{aligned}$$

Like all primitives, cylinders take a `PrimitiveAttributes` structure and a `SurfaceFunction` structure. In fact, the `Cylinder` constructor is identical to that of the `Sphere`.

⟨*Cylinder Methods*⟩≡
```
Cylinder::Cylinder( Float rad, Float z0, Float z1, Float tm,
        PrimitiveAttributes *a, SurfaceFunction *sf)
    : Primitive( a, sf ) {
    radius = rad;
    zmin = min(z0, z1);
    zmax = max(z0, z1);
    thetaMax = Radians( Clamp( tm, 0, 360 ) );
}
```

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

Bounding.

Like the sphere, we compute a conservative bounding box for the cylinder without taking into account the maximum θ.

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

Intersection.

In a similar manner to the sphere, we can find an 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, **??**, we have:

$$\left(x(o(\mathbf{r})) + tx(\vec{d}(\mathbf{r}))\right)^2 + \left(y(o(\mathbf{r})) + ty(\vec{d}(\mathbf{r}))\right)^2 = r^2$$

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

$$\begin{aligned} A &= x(\vec{d}(\mathbf{r}))^2 + y(\vec{d}(\mathbf{r}))^2 \\ B &= 2(x(\vec{d}(\mathbf{r}))x(o(\mathbf{r})) + y(\vec{d}(\mathbf{r}))y(o(\mathbf{r}))) \\ C &= x(o(\mathbf{r}))^2 + y(o(\mathbf{r}))^2 - r^2 \end{aligned}$$

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

Recall that the solution to the resulting quadratic equation is the same for all quadrics, so the fragments from the `Sphere` solver will be re-used below in the intersection routine.

Parameterization.

Computing a parametric mapping for the cylinder's surface is much easier than the sphere. We can easily invert the parametric description of the cylinder to compute a v value given the z coordinate of hit position. u can be computed by inverting the x and y parametric equations to solve for θ and then solving for u.

⟨*Compute cylinder inverse mapping*⟩≡
```
u = 1. - (atan2(Phit.y, -Phit.x) + M_PI) / thetaMax;
v = (Phit.z - zmin) / (zmax - zmin);
```

Final Analysis.

The analysis to be done on the cylinder is identical to that for the sphere. We 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.

⟨*Test cylinder intersection against clipping parameters*⟩≡
```
if (Phit.z < zmin || Phit.z > zmax || u > 1.) {
    if (thit == t1) return false;
    thit = t1;
    if (t1 > *maxt) return false;
    Phit = ray(thit);
    ⟨Compute cylinder inverse mapping⟩
    if (Phit.z < zmin || Phit.z > zmax || u > 1.)
        return false;
}
```

The normal at the hit point lies in the xy plane, and points from the z axis to the hit point.

⟨*Fill in* `HitInfo` *from cylinder hit*⟩≡
```
Normal surfNorm = Normal(Phit - Point(0,0,Phit.z));
surfNorm.Normalize();
hit->RecordHit(Phit, surfNorm, u, v, this );
```

The entire cylinder intersection routine is shown below. Its structure is identical to the `Sphere` intersection code. In fact, all quadrics will have an identically structured intersection code, so future methods will be omitted.

⟨*Cylinder Methods*⟩+≡
```
bool Cylinder::IntersectClosest(const Ray &ray, Float mint,
        Float *maxt, HitInfo *hit) const {
    Point Phit;
    Float u, v;
    ⟨Compute quadratic cylinder coefficients⟩
    ⟨Find quadric discriminant⟩
    ⟨Compute quadric t values⟩
    ⟨Compute cylinder inverse mapping⟩
    ⟨Test cylinder intersection against clipping parameters⟩
    if (hit) {
        ⟨Fill in HitInfo from cylinder hit⟩
    }
    *maxt = thit;
    return true;
}
```

3.6 Cones

Like cylinders, cones are centered around the z axis. In addition, their base is on the xy plane. For a doubly infinite cone with its apex at the origin, its implicit equation is $x^2 + y^2 - z^2 = 0$. We will compute ray intersections with an algorithm based on this equation and then test to determine if the intersections we find are in fact on the region of the cone of interest.

Cones also have the parametric description

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

Construction.

Cones have a user-specified radius and a height. Like the cylinder, they also have a maximum θ value that allows partial cones to be rendered.

⟨*Cone Methods*⟩≡
```
Cone::Cone( Float ht, Float rad, Float tm,
        PrimitiveAttributes *a, SurfaceFunction *sf)
        : Primitive( a, sf ) {
    radius = rad;
    height = ht;
    thetaMax = Radians( Clamp( tm, 0, 360 ) );
}
```

⟨*Cone Data*⟩≡
```
Float radius, height, thetaMax;
```

Bounding.

Bounding a cone is identical to bounding a cylinder, except we use 0 and `height` instead of `zmin` and `zmax`.

⟨*Cone Methods*⟩+≡
```
BBox Cone::BoundObjectSpace() const {
    Point p1 = Point( -radius, -radius, 0 );
    Point p2 = Point(  radius,  radius, height );
    return BBox( p1, p2 );
}
```

Intersection.

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

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

By substitutuing the ray equation into the implicit cone equation as usual, we get:

$$\begin{aligned} k &= \left(\frac{r}{h}\right)^2 \\ A &= x(\vec{d}(\mathbf{r}))^2 + y(\vec{d}(\mathbf{r}))^2 - kz(\vec{d}(\mathbf{r}))^2 \\ B &= 2(x(\vec{d}(\mathbf{r}))x(o(\mathbf{r})) + y(\vec{d}(\mathbf{r}))y(o(\mathbf{r})) - kz(o(\mathbf{r}))(z(o(\mathbf{r})) - h)) \\ C &= x(o(\mathbf{r}))^2 + y(o(\mathbf{r}))^2 - k(z(o(\mathbf{r})) - h)^2 \end{aligned}$$

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

Parameterization.

The cone's parameterization is the same as the cylinder, except that `v` is computed differently (again, by inverting the parametric equation).

⟨*Compute cone inverse mapping*⟩≡
```
u = 1. - (atan2(Phit.y, -Phit.x) + M_PI) / thetaMax;
v = Phit.z / height;
```

Final Analysis.

Testing the hit for validity is almost the same as the cylinder, except we need to use 0 and `height` instead of `zmin` and `zmax`.

⟨*Test cone intersection against clipping parameters*⟩≡
```
if (Phit.z < 0 || Phit.z > height || u > 1.) {
    if (thit == t1) return false;
    thit = t1;
    if (t1 > *maxt) return false;
    Phit = ray(thit);
    ⟨Compute cone inverse mapping⟩
    if (Phit.z < 0 || Phit.z > height || u > 1.)
        return false;
}
```

The cone is the first primitive so far where how to compute the surface normal at a hit position isn't immediately obvious. We can use the parametric description of the cone to solve this reasonably easily, however.

A basic fact from differential geometry is that the normal of a parametric surface is defined by the cross product of its partial derivatives in u and v.

$$N = \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v}$$

Recall that the parametric description of the cone is

$$P(u,v) = (r(1-v)\cos(u\theta_{\max}), r(1-v)\sin(u\theta_{\max}), v\,\text{height})$$

The partial derivative in u is

$$(r(1-v) - \sin(u\theta_{\max})\theta_{\max}, r(1-v)\cos(u\theta_{\max})\theta_{\max}, 0)$$

And in v

$$(-r\cos(u\theta_{\max}), -r\sin(u\theta_{\max}), \text{height})$$

We take the cross product of the partial derivatives and remove all of the constant values that are in each term, such as like r, $1-v$, $\theta_{\max}$. (We can do this since they just scale the resulting normal by a constant amount; since we'll normalize the normal when we're all done anyway, they don't affect the final result.)

This gives us:

$$N(u,v) = (\cos(u\theta_{\max})\text{height}, \sin(u\theta_{\max})\text{height}, r(\sin(u\theta_{\max})\sin(u\theta_{\max})) - \cos(u\theta_{\max})\cos(u\theta_{\max})) \quad (3.6.9)$$

The z component of the normal can be simplified further, using the trigonometric identity $\cos^2 x + \sin^2 x = 1$, giving the formula implemented in the code below.

⟨*Fill in* `HitInfo` *from cone hit*⟩≡

```
Float nTheta = u*thetaMax;
Normal surfNormal(cos(nTheta) * height, sin(nTheta) * height, radius );
surfNormal.Normalize();
hit->RecordHit(Phit, surfNormal, u, v, this );
```

3.7 Paraboloids

Paraboloids are also centered on the z axis. Unlike cones, the apex of the paraboloid is located at the origin. The implicit equation for a paraboloid is $x^2 + y^2 - z = 0$ and its parametric form is

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

Construction.

Paraboloids have a minimum and maximum z value. In addition, the radius of the paraboloid at the maximum z value is specified. Like the other quadrics, a maximum angle ϕ can also be provided. The constructor is therefore the same as for a `Sphere` and `Cylinder`.

⟨*Paraboloid Methods*⟩≡
```
Paraboloid::Paraboloid( Float rad, Float z0, Float z1, Float tm,
        PrimitiveAttributes *a, SurfaceFunction *sf)
    : Primitive( a, sf ) {
    radius = rad;
    zmin = z0;
    zmax = z1;
    thetaMax = Radians( Clamp( tm, 0, 360 ) );
}
```

⟨*Paraboloid Data*⟩≡
```
Float radius;
Float zmin, zmax;
Float thetaMax;
```

Bounding.

Bounding paraboloids is the same as all the quadrics we've encountered so far:

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

Intersection.

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

$$\frac{hx^2}{r^2} + \frac{hy^2}{r^2} - z = 0 \tag{3.7.10}$$

By substitutuing the ray equation into the implicit paraboloid equation as usual, we get:

$$k = \left(\frac{h}{r^2}\right) \tag{3.7.11}$$

$$A = k(x(\vec{d}(\mathbf{r}))^2 + y(\vec{d}(\mathbf{r}))^2) \tag{3.7.12}$$

$$B = 2k(x(\vec{d}(\mathbf{r}))x(o(\mathbf{r})) + y(\vec{d}(\mathbf{r}))y(o(\mathbf{r}))) - z(\vec{d}(\mathbf{r})) \tag{3.7.13}$$

$$C = k(x(o(\mathbf{r}))^2 + y(o(\mathbf{r}))^2) - z(o(\mathbf{r})) \tag{3.7.14}$$

⟨*Compute quadratic paraboloid coefficients*⟩≡

```
Float k = zmax/(radius*radius);
Float A =   k*(ray.D.x * ray.D.x + ray.D.y * ray.D.y );
Float B = 2*k*(ray.D.x * ray.O.x + ray.D.y * ray.O.y ) - ray.D.z;
Float C =   k*(ray.O.x * ray.O.x + ray.O.y * ray.O.y ) - ray.O.z;
```

Parameterization.

The paraboloid's parameterization is the same as the cylinder; inverting the parametric form gives us the following:

⟨*Compute paraboloid inverse mapping*⟩≡

```
u = 1. - (atan2(Phit.y, -Phit.x) + M_PI) / (2. * M_PI);
v = (Phit.z-zmin) / (zmax-zmin);
```

Final Analysis.

Testing the hit for validity is also the same as the cylinder.

⟨*Test paraboloid intersection against clipping parameters*⟩≡

```
if (Phit.z < zmin || Phit.z > zmax || u * 2 * M_PI > thetaMax) {
    if (thit == t1) return false;
    thit = t1;
    if (t1 > *maxt) return false;
    Phit = ray(thit);
    ⟨Compute paraboloid inverse mapping⟩
    if (Phit.z < zmin || Phit.z > zmax || u * 2 * M_PI > thetaMax)
        return false;
}
```

Computing the normal to a paraboloid takes can be done the same cross product of partial derivatives machinery that we use above for cones.

⟨*Fill in* `HitInfo` *from paraboloid hit*⟩≡
```
u *= (2*M_PI)/thetaMax;

Float crossu = atan2(Phit.y, -Phit.x) + M_PI;
Float crossv = Phit.z / zmax;
Vector dPdv( (radius*cos(crossu))/(2*sqrt(crossv)*sqrt(zmax) ),
       (radius*sin(crossu))/(2*sqrt(crossv)*sqrt(zmax) ),
       1);
Vector dPdu( (-radius*sqrt(crossv)*sin(crossu))/sqrt(zmax),
       (radius*sqrt(crossv)*cos(crossu))/sqrt(zmax),
       0);
Normal surfNorm = Normal(Cross(dPdu, dPdv));
surfNorm.Normalize();
hit->RecordHit(Phit, surfNorm, u, v, this );
```

3.8 Hyperboloids

Hyperboloids are defined differently from the quadrics we have encountered so far. A hyperboloid is defined as the surface resulting from revolving a given line around the z axis. Interestingly, `Cone`s, `Cylinder`s, and `Disk`s can be defined this way as well. It would be possible to make those primitives special cases of `Hyperboloid`s.

The parametric form of a hyperboloid is

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

Construction.

Hyperboloids have two points that define the endpoints of a line segment to be revolved around the z axis. A maximum angle θ can also be provided. The constructor computes the maximum radius of the hyperboloid for use in constructing bounding boxes.

⟨*Hyperboloid Methods*⟩≡
```
Hyperboloid::Hyperboloid( const Point &point1, const Point &point2, Float tm,
        PrimitiveAttributes *a, SurfaceFunction *sf)
    : Primitive( a, sf ) {
    p1 = point1;
    p2 = point2;
    thetaMax = Radians( Clamp( tm, 0, 360 ) );
    Float rad1 = sqrt( p1.x*p1.x + p1.y*p1.y );
    Float rad2 = sqrt( p2.x*p2.x + p2.y*p2.y );
    rmax = max(rad1,rad2);
    zmin = min(p1.z,p2.z);
    zmax = max(p1.z,p2.z);
}
```

⟨*Hyperboloid Data*⟩≡
```
Point p1, p2;
Float zmin, zmax;
Float thetaMax;
Float rmax;
```

Bounding.

Because we have computed the maximum radius, we simply use this in constructing the bounding box.

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

Intersection.

⟨*Compute quadratic hyperboloid coefficients*⟩≡
```
Float t, a2, c2;
Float a, c;
Point pr;

if(p1.z == p2.z)
return false;
t = -p2.z/(p1.z-p2.z);
pr = Point(t*p1.x + (1-t)*p2.x,
     t*p1.y + (1-t)*p2.y,
     t*p1.z + (1-t)*p2.z);
a2 = pr.x*pr.x + pr.y*pr.y;
if(p1.x*p1.x + p1.y*p1.y == a2)
return false;
c2 = (a2*p1.z*p1.z)/(p1.x*p1.x + p1.y*p1.y - a2);

Float A = ray.D.x*ray.D.x/a2 +
          ray.D.y*ray.D.y/a2 -
          ray.D.z*ray.D.z/c2;
Float B = 2 * (ray.D.x*ray.O.x/a2 +
               ray.D.y*ray.O.y/a2 -
               ray.D.z*ray.O.z/c2);
Float C = ray.O.x*ray.O.x/a2 +
          ray.O.y*ray.O.y/a2 -
          ray.O.z*ray.O.z/c2 - 1;
```

Parameterization.

⟨*Compute hyperboloid inverse mapping*⟩≡
```
c = sqrt(c2);
a = sqrt(a2);

v = (Phit.z - p1.z)/(p2.z - p1.z);

pr = Point((1-v)*p1.x + v*p2.x,
     (1-v)*p1.y + v*p2.y,
     (1-v)*p1.z + v*p2.z);

u = atan2(Phit.y, Phit.x) - atan2(pr.y, pr.x);
if(u < 0)
u += 2*M_PI;
u /= 2*M_PI;
```

Final Analysis.

⟨*Test hyperboloid intersection against clipping parameters*⟩≡
```
if (Phit.z < zmin || Phit.z > zmax || u * 2 * M_PI > thetaMax) {
    if (thit == t1) return false;
    thit = t1;
    if (t1 > *maxt) return false;
    Phit = ray(thit);
    ⟨Compute hyperboloid inverse mapping⟩
    if (Phit.z < zmin || Phit.z > zmax || u * 2 * M_PI > thetaMax)
        return false;
}
```

⟨*Fill in* `HitInfo` *from hyperboloid hit*⟩≡
```
Float crossu = atan2(Phit.y, Phit.x) + M_PI;
Float crossv = Phit.z/c;
Vector dPdu(-a*cosh(crossv)*sin(crossu),
    a*cosh(crossv)*cos(crossu), 0);
Vector dPdv(a*sinh(crossv)*cos(crossu), a*sinh(crossv)*sin(crossu),
    c*cosh(crossv));
Normal surfNorm = Normal(Cross(dPdu, dPdv));
surfNorm.Normalize();
hit->RecordHit(Phit, surfNorm, u, v, this);
```

3.9 Disks

Although a disk is really just a special case of a cone, we provide a special implementation of disks which can be intersected with more efficiently. A `Disk` is a circular disk of user-supplied radius at some height along the z axis. In order to make partial disks, the user may specify a maximum theta value beyond which the disk is cut off (see Figure 3.6).

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

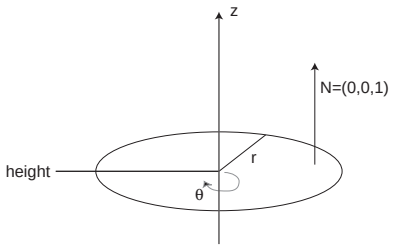


Figure 3.6: Basic setting for the disk primitive. The disk has radius of r and is located at some height along the z-axis. A partial disk may be swept by specifying a maximum θ value.

Construction.

The `Disk` constructor takes the height, radius, and maximum theta values supplied by the user as well as `GeomAttribuetes` and `SurfaceFunction` for it.

⟨*Disk Methods*⟩≡
```
Disk::Disk(Float ht, Float r, Float tmax, PrimitiveAttributes *attr,
        SurfaceFunction *sf)
        : Primitive(attr, sf) {
    Height = ht;
    Radius = r;
    ThetaMax = Radians( Clamp( tmax, 0, 360 ) );
}
```

⟨*Disk Private Data*⟩≡
```
Float Height, Radius, ThetaMax;
```

Bounding.

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

⟨*Disk Methods*⟩+≡
```
BBox Disk::BoundObjectSpace() const {
    return BBox(Point(-Radius,-Radius,Height),
                Point( Radius, Radius,Height) );
}
```

Intersection.

Intersecting a ray with a disk is also quite easy. We intersect the ray with the disk's plane, and then see if the intersection point lies inside the disk. The intersection is made even easier by the fact that the disk always lies parallel to the *xy* plane.

⟨*Disk Methods*⟩+≡
```
bool Disk::IntersectClosest(const Ray &ray, Float mint, Float *maxt,
        HitInfo *hit) const {
    ⟨Compute plane intersection for disk⟩
    ⟨See if hit point is inside disk radius and thetamax⟩
    if (hit) {
        ⟨Fill in HitInfo from disk hit⟩
    }
    *maxt = hitt;
    return true;
}
```

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

$$h = z(o(\mathbf{r})) + t * z(\vec{d}(\mathbf{r}))$$

So t is

$$t = \frac{h - z(o(\mathbf{r}))}{z(\vec{d}(\mathbf{r}))}$$

We take this value and see if it is inside the legal range of t values, namely $[\texttt{mint}, \texttt{maxt}]$. If not, we can return false.

⟨*Compute plane intersection for disk*⟩≡
```
Float hitt = (Height - ray.O.z) / ray.D.z;
if (hitt < mint || hitt > *maxt)
    return false;
```

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

⟨*See if hit point is inside disk radius and thetamax*⟩≡
```
Point Phit = ray(hitt);
Float Dist2 = Phit.x * Phit.x + Phit.y * Phit.y;
if (Dist2 > Radius * Radius)
    return false;
⟨Compute disk inverse mapping⟩
if (u * 2. * M_PI > ThetaMax)
    return false;
```

Parameterization.

If this test passes, we perform the final test, making sure that the theta value of the hit point is between zero and `ThetaMax` specified by the user. The disk is parametrized in (u,v) as $(r(1-v)\cos(u\text{ThetaMax}), r(1-v)\cos(u\text{ThetaMax}), \text{Height})$. We then go back to θ from `u` and make sure that we're not beyond `ThetaMax`, returning false if so.

⟨*Compute disk inverse mapping*⟩≡
```
Float u = 1. - (atan2(Phit.y, -Phit.x) + M_PI) / (2. * M_PI);
Float v = 1. - (sqrt(Phit.x * Phit.x + Phit.y * Phit.y) / Radius);
```

If we've gotten this far, we know that there is an intersection with the disk. If a non-`NULL` `HitInfo` pointer was passed in, we fill that in with information about the intersection. The paramter `u` is scaled to reflect the partial disk specified by `ThetaMax`.

⟨*Fill in* `HitInfo` *from disk hit*⟩≡
```
u *= (2*M_PI)/ThetaMax;
hit->RecordHit(Phit, Normal(0, 0, 1), u, v, this );
```

3.10 Heightfield

We will provide a stub implementation of a *heightfield* primitive. Usually used to represent terrains, heightfieldss are defined by a regular grid of z altitudes over the range $(0,0) \rightarrow (1,1)$ in x and y. We will use `nx` to denote the number of altitude samples in the x direction and `ny` to denote the nubmer in y (see Figure 3.7).

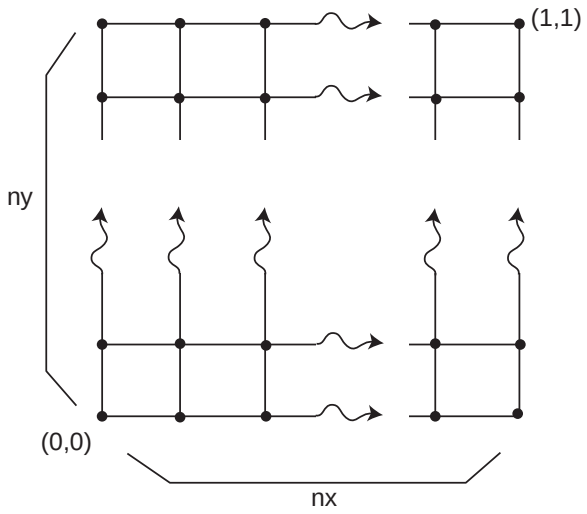


Figure 3.7: Setting for the heightfield primitive: altitudes are defined over a regular grid of `nx` by `ny` samples.

When a new heightfield is created, we mostly just need to copy the z altitude values from the user.

⟨*Heightfield Methods*⟩≡
```
Heightfield::Heightfield(int x, int y, float *zs, PrimitiveAttributes *a,
        SurfaceFunction *sf)
    : Primitive(a, sf) {
    nx = x;
    ny = y;
    z = new float[nx*ny];
    memcpy(z, zs, nx*ny*sizeof(float));
}
```

⟨*Heightfield Data*⟩≡
```
float *z;
int nx, ny;
```

⟨*Heightfield Methods*⟩+≡
```
Heightfield::~Heightfield() {
    delete[] z;
}
```

Bounding a heightfield is straightforward. We just need to compute the range of z values, since it is defined to cover $(0,0)$ to $(1,1)$ in xy.

⟨*Heightfield Methods*⟩+≡
```
BBox Heightfield::BoundObjectSpace() const {
    float minz = z[0], maxz = z[0];
    for (int i = 1; i < nx*ny; ++i) {
        if (z[i] < minz) minz = z[i];
        if (z[i] > maxz) maxz = z[i];
    }
    return BBox(Point(0,0,minz), Point(1,1,maxz));
}
```

We will provide a stub implementation of the primitive that refines into a `TriangleMesh`. This can be used for testing your smarter implementation.

⟨*Heightfield Methods*⟩+≡
```
bool Heightfield::CanIntersect() const {
    return false;
}
```

When we are refined, we need to triangulate the heightfield, compute true (x,y,z) positions for the mesh vertices, and make a new `TriangleMesh`.

Each quadrilateral in the heightfield will turn into two triangles, so the total number of triangles in the resulting mesh is `2*(nx-1)*(ny-1)`. For each triangle, we need three integer vertex offsets to record which `P` values that triangle uses.

⟨*Heightfield Methods*⟩+≡
```
void Heightfield::Refine(vector<Primitive *> *refined) const {
    int ntris = 2*(nx-1)*(ny-1);
    int *verts = new int[3*ntris];
    Point *P = new Point[nx*ny];
    int x, y;
    ⟨Compute heightfield vertex positions⟩
    ⟨Fill in heightfield vertex offset array⟩
    refined->push_back(new TriangleMesh(ntris, nx*ny, verts, P,
        attributes, new SurfaceFunction(*surfaceFunction)));
    delete[] P;
    delete[] verts;
}
```

Finding the (x,y,z) position for each vertex follows directly from the definition of the heightfield primitive.

⟨*Compute heightfield vertex positions*⟩≡
```
P = new Point[nx*ny];
int pos = 0;
for (y = 0; y < ny; ++y) {
    for (x = 0; x < nx; ++x) {
        P[pos].x = (float)x / (float)(nx-1);
        P[pos].y = (float)y / (float)(ny-1);
        P[pos].z = z[pos];
        ++pos;
    }
}
```

And now, for each of the triangles in the mesh, we need to compute the offsets of its vertices in the `P` array we filled in above. Note that given how we stepped through `P` above, the offset to the (x,y)th vertex is $x+y*nx$.

We loop over all of the quads, creating two triangles for each one. The `VERT` macro holds the logic for getting the (x,y)th vertex offset.

⟨*Fill in heightfield vertex offset array*⟩≡

```
int *vp = verts;
for (y = 0; y < ny-1; ++y) {
    for (x = 0; x < nx-1; ++x) {
#define VERT(x,y) ((x)+(y)*nx)
        *vp++ = VERT(x, y);
        *vp++ = VERT(x+1, y);
        *vp++ = VERT(x+1, y+1);

        *vp++ = VERT(x, y);
        *vp++ = VERT(x+1, y+1);
        *vp++ = VERT(x, y+1);
    }
#undef VERT
}
```

4. Intersection Acceleration

In this chapter we will describe some classes that manage all of the geometric primitives that are in a scene. First we discuss the need for reducing the number of ray–primitive intersection tests as well as general strategies for doing so. We then provide the implementation of a simple ray–intersection accelerator, a regular grid. We wrap up the chapter by describing the `Scene` class, which has overall responsibility for managing the primitives and choosing appropriate intersection acceleration data structures; given that there is currently only the regular grid, it's a pretty simple class.

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. 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. In many early ray tracers, the vast majority of execution time was spent on computing intersections with geometry. As time has gone by, however, innovation in algorithms to reduce the number of unnecessary ray intersection tests has greatly reduced the time spent on intersections. Another important trend has been that shading models have become much more sophisticated and computationally expensive; this has also shifted the balance away from intersection time as the bottleneck in ray tracing.

The `BBox` class that we introduced previously is one step toward reducing intesection tests. We can check the ray against the bounding box of the geometry first, and only try to find an intersection with the geometry if the ray hits the box. As long as the bounding box is a good fit for the geometry and computing a real intersection with the geoemtry is significantly slower than testing the ray against the box, we can save a lot of time in this manner.

The use of bounding boxes can be generalized into even better data structures. Clearly, if we can only test the primitives that the ray approaches and can quickly reject the other primitives, we can save a lot of unnecesary intersection tests. A number of techniques have been proposed that take the scene geometry and store it in a spatial

data structure of some sort. The ray can then travel through the spatial data structure, only attempting to find intersections with objects that it can potentially intersect.

Regular Grid.

The *regular grid* is a rectangular region of space subdivided into *voxels* (see Figure 4.1). Each voxel is exactly the same size. In each voxel, we keep a list of all of the primitives that overlap the voxel. Then when we have a ray to trace, we step it through each of the voxels that it passes through in turn. We check for intersection only with the primitives in the voxel that the ray is in, and stop stepping once we have found the closest possible intersection.

The regular grid usually performs reasonably well. It was designed largely for the efficiency of being able to step a ray through it to the next voxel quickly. However, 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. The basic problem is that the data structure doesn't adapt well to the distribution of the data.

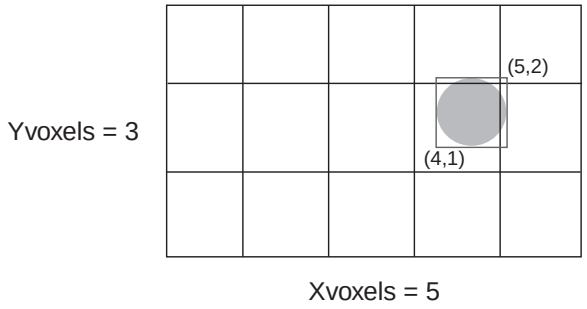


Figure 4.1: Primitives in the scene (such as the sphere shown here) are stored in all of the voxels that they overlap in the grid. Typically, the primitive's bounding box is used to determine which voxels it overlaps. In this case, the sphere is incorrectly stored in the upper-right voxel since its bounding box overlaps the voxel even though the primitive does not.

Hierarchical Bounding Volume.

Another approach is the *hierarchical bounding volume* (HBV). Given some method of bounding primitives (e.g. axis aligned bounding boxes), a hierarchy of the bounding primitives is constructed. The top node of the hierarchy encompasses all of the primitives in the scene (see Figure 4.2. This 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.

A 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.

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 betwen the sets as well as the size of the bounding volumes that result. A final disadvantage is that once some ray intersection is found, traversal can't immediately stop. Instead, the walk throuh the

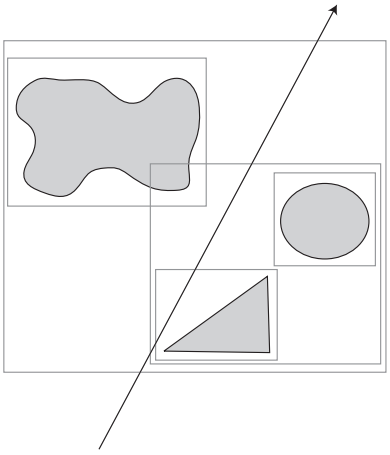


Figure 4.2: A set of primitives are stored in a bounding volume hierarchy. When a ray is being traced, we first see if it intersects the top-level bounding volume. If so, we recursively process the children bounding volumes, continuing on with those that are intersected, until we reach the geometric primitives.

tree needs to continue until all bounding volumes have been checked, since a closer intersection may be present in an as-yet unexplored part of the tree.

Sitting somewhere between HBVs and grids are *octrees*. These both adaptively subdivide space, but in a more regular fashion. An octree 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 into eight smaller boxes, each one one-half the dimensions of the parent in each direction. Primitives are then redistributed to the boxes that they overlap, and this process continues recursively until either a small enough number of primitives is in each box or a maximum depth is reached.

Octree.

Octrees are also effective due to their being adaptive based on the distribution of primitives in space. However, because the octree adaptively divides up space, it takes longer to compute the next octree node that a ray enters after leaving another node than it does to step a ray to the next grid cell it enters. Holy wars have been fought over whether fast stepping in grids is more important than adaptive refinement in octrees.

Octrees can be generalized a bit into *k-d trees* and *binary space partitioning trees* (BSP trees). A k-d tree removes the restriction that the space is divided into equal-sized pieces. Instead, space is split into half at each level of the tree at a point along one of the coordinate axes in an effort to ensure a more even distribution of primitives. BSP trees generalize this further, removing the restriction that splits be perpindicular to coordinate axes.

Meta-Hierarchies.

The idea of using spatial data structures can be generalized to include spatial data structures that themselves hold other spatial data structures. 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.

Other Refinements.

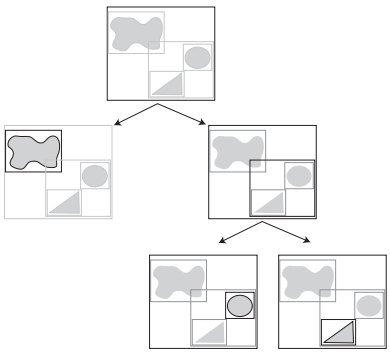


Figure 4.3: Structure of a bounding volume hierarchy. The top node of the tree holds the bounding box of the entire scene and then pointers to children nodes that hold subsets of the scene. This continues recursively until the leaf nodes, which hold pointers to geometric primitives in the scene.

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 eye 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. In `lrt`, the `Primitive` intersection routines and the acceleration structures can be passed a `NULL` pointer to a `HitInfo` in this case, it treats the ray as a shadow ray, returning as soon as possible.

Another technique that takes advantage of this property of shadow rays is the *shadow cache*; for each light source in the scene, we keep a pointer to the last primitive that occluded light from the emitter. Subsequent shadow rays are fist checked against this blocker–since the blocking object will often block a number of shadow rays in a row, this can make it much faster to find the blocker.

For non-shadow rays, after we have found an intersection we keep track of the parametric distance to that hit. We have effectively turned our semi-infinite ray into a line segment, and we can cull from testing any primitives that are further along the ray than the hit point. We use this optimization in `lrt`; in Section 4.2 we will describe how this is used to reduce work in the grid accelerator.

A last technique has been dubbed *mailboxes*. Because a primitive may overlap multiple cells in grid or octree-type accelerators, we can keep track of which primitives have already been tested against the ray and void testing them multiple times as the ray goes through multiple cells that they overlap.

4.2 Regular Grid Accelerator

Creation.

Here we will define an accelerator based on a regular grid. We are handed a set of primitives to manage and try to construct a good grid that holds them.

⟨*GridAccelerator Constructors*⟩≡

```
GridAccelerator(const vector<Primitive *> &primitives);
```

⟨*GridAccelerator Method Definitions*⟩≡
```
GridAccelerator::GridAccelerator(
        const vector<Primitive *> &primitives) {
    ⟨Compute overall grid bounds⟩
    ⟨Expand primitives that need refinement⟩
    ⟨Choose grid resolution⟩
    ⟨Compute voxel widths and allocate voxels⟩
    addPrimitivesToVoxels(expandedPrimitives);
}
```

We start by looping through all of the primitives and computing a bounding box that bounds all of them.

⟨*Compute overall grid bounds*⟩≡
```
Assert(primitives.size() > 0);
bbox = primitives[0]->BoundWorldSpace();
for (u_int i = 1; i < primitives.size(); ++i)
    bbox = Union(bbox, primitives[i]->BoundWorldSpace());
```

⟨*GridAccelerator Private Data*⟩≡
```
BBox bbox;
```

Note that we need to get at the internals of a bounding box; we need to open up that data to this class.

⟨*BBox Method Declarations*⟩+≡
```
friend class GridAccelerator;
```

Our grid accelerator is less efficient than it might be for `Primitive`s that are not immediately directly intersectable. Rather than lazily refining them into intersectable primitives as needed, we instead refine everything all the way as we're constructing the grid. We do this because we need to decide the resolution of the grid now; because refinement might turn a single primitive into a million primitives, we don't want to decide to make a grid that doesn't have a sufficiently high resolution for the actual number of primitives.

One advantage of other acceleration structures like hierarchial bounding volumes is that can handle refinement of primitives on the fly more robustly.

We keep two `vector`s of primitives. All primitives start out in `primitivesToProcess`. We remove each primitive from that vector in turn and consider it: if it doesn't need to be refined, we just add it to the `expandedPrimitivesVector`. Otherwise, we refine it and add the resulting primitives to `primitivesToProcess`, in case any of them need to be refined. This continues until `primitivesToProcess` is empty.

⟨*Expand primitives that need refinement*⟩≡
```
vector<Primitive *> expandedPrimitives;
vector<Primitive *> primitivesToProcess = primitives;
while (primitivesToProcess.size()) {
    int lastPrim = primitivesToProcess.size() - 1;
    Primitive *prim = primitivesToProcess[lastPrim];
    primitivesToProcess.pop_back();

    if (prim->CanIntersect())
        expandedPrimitives.push_back(prim);
    else {
        vector<Primitive *> refined;
        prim->Refine(&refined);
        for (u_int j = 0; j < refined.size(); ++j)
            primitivesToProcess.push_back(refined[j]);
    }
}
```

Now we have our set of primitives to bound, so we can pick 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 other directions are set such that they are proportional to the sizes in the maximum dimension, in an effort to create voxels that are as square as possible.

⟨*Choose grid resolution*⟩≡
```
int cubeRoot = (int)pow(expandedPrimitives.size(), .333333333);
Float dx = 1.00001 * (bbox.pMax.x - bbox.pMin.x);
Float dy = 1.00001 * (bbox.pMax.y - bbox.pMin.y);
Float dz = 1.00001 * (bbox.pMax.z - bbox.pMin.z);
Float invmaxWidth = 1.0/max(dx, max(dy, dz));
Assert(invmaxWidth > 0.);
XVoxels = min( 100, max(1, 3 * Round(cubeRoot * dx * invmaxWidth)) );
YVoxels = min( 100, max(1, 3 * Round(cubeRoot * dy * invmaxWidth)) );
ZVoxels = min( 100, max(1, 3 * Round(cubeRoot * dz * invmaxWidth)) );
```

⟨*GridAccelerator Private Data*⟩+≡
```
int XVoxels, YVoxels, ZVoxels;
```

We now use this 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 = dx / XVoxels;
YWidth = dy / YVoxels;
ZWidth = dz / ZVoxels;
InvXWidth = (XWidth == 0.) ? 0. : 1. / XWidth;
InvYWidth = (YWidth == 0.) ? 0. : 1. / YWidth;
InvZWidth = (ZWidth == 0.) ? 0. : 1. / ZWidth;
cells = new list<Primitive *>[XVoxels * YVoxels * ZVoxels];
```

⟨*GridAccelerator Private Data*⟩+≡
```
Float XWidth, YWidth, ZWidth;
Float InvXWidth, InvYWidth, InvZWidth;
```

To add primitives to the grid, we loop through the primitives in turn, adding each one to the `list`s in the cells that its bounding box overlaps.

⟨*GridAccelerator Private Methods*⟩≡
```
void addPrimitivesToVoxels(const vector<Primitive *> &primtives);
```

⟨*GridAccelerator Method Definitions*⟩+≡
```
void GridAccelerator::addPrimitivesToVoxels(
        const vector<Primitive *> &primitives) {
    for (u_int i = 0; i < primitives.size(); ++i) {
        ⟨Find cell extent of primitive⟩
        ⟨Add primitive to overlapping cells⟩
    }
}
```

⟨*GridAccelerator Private Data*⟩+≡
```
list<Primitive *> *cells;
```

We find the world space bounds of the primitive and compute the integer set of voxels that it overlaps. We use the utility functions `x2v` et al, which turn a world space x, y, or z coordinate into voxel numbers. These values are then clamped to the range of valid voxel addresses.

⟨*Find cell extent of primitive*⟩≡
```
BBox primBounds = primitives[i]->BoundWorldSpace();
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 coordiantes of their lower-left corners.

⟨*GridAccelerator Method Declarations*⟩≡
```
int x2v( Float x ) const { return int(( x - bbox.pMin.x ) *
                    InvXWidth); }
int y2v( Float y ) const { return int(( y - bbox.pMin.y ) *
                    InvYWidth); }
int z2v( Float z ) const { return int(( z - bbox.pMin.z ) *
                    InvZWidth); }
Float v2x( int x ) const { return x * XWidth + bbox.pMin.x; }
Float v2y( int y ) const { return y * YWidth + bbox.pMin.y; }
Float v2z( int z ) const { return z * ZWidth + bbox.pMin.z; }
```

We just loop over the voxel addresses that the primitive covers, compute the offset into the array of voxel `lists`, and drop the pointer on the back.

⟨*Add primitive to overlapping cells*⟩≡
```
for (int x = x0; x <= x1; ++x)
    for (int y = y0; y <= y1; ++y)
        for (int z = z0; z <= z1; ++z) {
            int offset = z*XVoxels*YVoxels + y*XVoxels + x;
            cells[offset].push_back(primitives[i]);
        }
```

The destructor just has to free up the array of primitive `listss` that we made. Deleting the primitives themselves is the responsibility of the code that created the accelerator in the first place.

⟨*GridAccelerator Method Definitions*⟩+≡
```
GridAccelerator::~GridAccelerator() {
    delete[] cells;
}
```

Traversal.

We now come to the most interesting part of the grid, where we have a ray to compute primitive intersections with. We need to step through all of the cells that the ray passes through in order, first to last, and bail out as soon as we have found an intersection and can guarantee that there is no closer intersection (or, for shadow rays, any intersection will do.)

⟨*GridAccelerator Method Declarations*⟩+≡
```
bool IntersectClosest(const Ray &ray, Float mint, Float *maxt,
    HitInfo *hit);
```

⟨*GridAccelerator Method Definitions*⟩+≡
```
bool GridAccelerator::IntersectClosest(const Ray &ray, Float mint,
        Float *maxt, HitInfo *hitInfo) {
    ⟨Check ray against overall grid bounds⟩
    ⟨Set up 3DDDA for this ray⟩
    ⟨Walk grid⟩
}
```

We first check to see at what point the ray enters the grid. We first check the ray's origin with the grid's bounding box: if it's inside, then that's our starting point. Otherwise we try to intersect the ray with the grid's bounding box; if it hits, the parametric hit distance along the ray is our starting point. Otherwise, there can be no intersection with any of the geometry in the grid, so we return immediately.

⟨*Check ray against overall grid bounds*⟩≡
```
Float rayT = *maxt;
if (bbox.Inside(ray(mint)))
    rayT = mint;
else if (!bbox.IntersectP(ray, &rayT))
    return false;
Point gridIntersect = ray(rayT);
```

Next, we set up the initial (x,y,z) 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 **??**):

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.

⟨*Set up 3DDDA for this ray*⟩≡
⟨*Set up X stepping*⟩
⟨*Set up Y stepping*⟩
⟨*Set up Z stepping*⟩

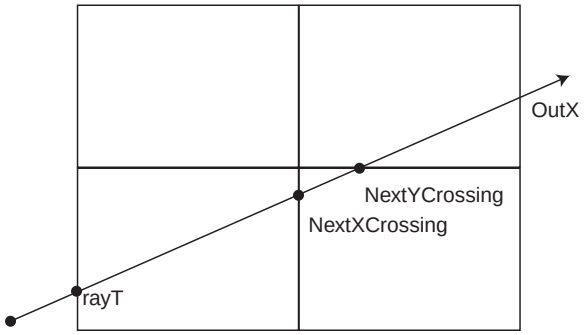


Figure 4.4: Stepping a ray through a voxel grid. We first compute `rayT`, the distance along the ray to the first intersection with the grid. We then compute distances along the ray to the next time we cross into the next voxel in the x direction, `NextXCrosing`, and in the y and z (not shown) directions. When we cross into the next x voxel, for example, we can immediately update the value of `NextXCrossing` by adding a fixed value, the voxel width in x divided by the ray's x direction, to it.

⟨*Set up X stepping*⟩≡
```
⟨Compute current x voxel⟩
Float NextXCrossing, DeltaX;
int StepX, OutX;
if (fabs(ray.D.x) < RI_EPSILON) {
    ⟨Handle ray perpindicular 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 `gridIntersect` is actually slightly outside of the grid).

⟨*Compute current x voxel*⟩≡
```
int x = x2v(gridIntersect.x);
if (x == XVoxels) x--;
Assert( x >= 0 && x < XVoxels );
```

Now for each of x, y, and z, we compute crossing distances, changes in crossing distances when we step in that direction, and the exiting voxel numbers. If the ray's x component is nearly zero, then we'll *never* step in the x direction. We set the x crossing distance to infinity, so that we always decide that one of the other directions has the shortest parametric distance to the next voxel. As such, the values of `DeltaX` and `OutX` won't be used, but we'll set them to silence over-aggressive compiler warnings about uninitialized variables.

⟨*Handle ray perpindicular to x*⟩≡
```
NextXCrossing = INFINITY;
DeltaX = 0;
OutX = -1;
```

Things are more interesting in the common case. For a ray with a positive x direction component, the parametric value along the ray where we cross into the next voxel in x, `NextXCrossing` is our parametric starting point, `rayT` plus the x distance to the next voxel, divided by the x direction component. Similarly, dividing the width of a voxel in x by the ray's direction component gives us the parametric distance along the ray that we have to travel to get from one side of a voxel to the other, in the x direction.

`StepX` just tells us that when we leave a voxel in the x direction, we move 1 voxels. `OutX` says that when we reach a voxel with component `XVoxels`, we've left the grid and are done.

⟨*Handle ray with positive x direction*⟩≡
```
NextXCrossing = rayT + ( v2x( x+1 ) - gridIntersect.x )/ray.D.x;
DeltaX = XWidth / ray.D.x;
StepX = 1;
OutX = XVoxels;
```

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

⟨*Handle ray with negative x direction*⟩≡
```
NextXCrossing = rayT + ( v2x( x ) - gridIntersect.x )/ray.D.x;
DeltaX = - XWidth / ray.D.x;
StepX = -1;
OutX = -1;
```

This leads us to the code that walks through the grid. Starting with the first voxel, we check for intersection with the primitives inside that voxel. If we find a hit, `hitSomething` is set to true. Since we may have found a hit that is outside of the current voxel, however, we don't immediately return when through processing a voxel with an intersection. Instead, since the primitive's intersection routine will update the `maxt` variable, setting it to the parametric hit distance, we'll leave the grid stepping code to detect when we've walked into a voxel that's past an already-found hit.

If no hit is found after all, we step forward to the next voxel that the ray enters.

⟨*Walk grid*⟩≡
```
bool hitSomething = false;
for (;;) {
    int offset = z*XVoxels*YVoxels + y*XVoxels + x;
    list<Primitive *> *primitiveList = cells + offset;
    if (primitiveList->size() > 0) {
        ⟨Check single voxel⟩
    }
    ⟨Advance to next voxel⟩
}
return hitSomething;
```

To check the primitives in a voxel, we just loop through them, calling their intersection routines. We first need to transform the ray given to us from world space into the primitive's object space; the fragment ⟨*Transform to ray to object space*⟩ does this. Imagine that. If we do find a hit and if we're tracing a shadow ray, `hitInfo` will be `NULL`; in this case we can break from the infinite `for` loop above immediately and return success.

⟨*Check single voxel*⟩≡
```
const Transform *lastTransform = NULL;
Ray rayObj;
list<Primitive *>::iterator iter = primitiveList->begin();
while (iter != primitiveList->end()) {
    Primitive *prim = *iter;
    ⟨Transform ray to object space⟩
    ⟨Check for ray-primitive intersection⟩
    ++iter;
}
```

In order to try to reduce the time spent transforming rays into the object space of each primitive, we keep a little cache of the previous transformed ray. We compare the pointer to the `WorldToObject` transform of subsequent primitives to a cached `WorldToObject` transform. If they match, we know that the world-space ray transforms to the same object space ray and we reuse that. Otherwise, we do the work of the transformation and squirrel it away.

We could do even fewer transformations by comparing the acutal transformations for equality, rather than just their pointers (all transforms at the same memory location must be the same, but not all transforms that are numerically equal must be at the same memory location). However, since this is 16 comparisons rather than one, we just check pointers and sometimes do extra work.

⟨*Transform ray to object space*⟩≡
```
const Transform *primW2O = &(prim->attributes->WorldToObject);
if (primW2O != lastTransform) {
    rayObj = (*primW2O)(ray);
    lastTransform = primW2O;
}
```

If we find an intersection and `hitInfo` is null, we can exit the loop immediately. Otherwise we keep checking for intersections until we find what is clearly the closest one (this is detected below).

⟨*Check for ray-primitive intersection*⟩≡
```
if (prim->IntersectClosest(rayObj, mint, maxt, hitInfo)) {
    hitSomething = true;
    if (!hitInfo)
        break;
}
```

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 traveral loop.

⟨*Advance to next voxel*⟩≡
```
if (NextXCrossing < NextYCrossing && NextXCrossing < NextZCrossing) {
    ⟨Step in X⟩
}
else if (NextZCrossing < NextYCrossing) {
    ⟨Step in Z⟩
}
else {
    ⟨Step in Y⟩
}
```

We first see if an intersection has been found that is inside the current voxel. If so, we're done and can exit. This is the case if `maxt` is less than the parametric distance at which we enter the next x voxel, `NextXCrossing` Otherwise we update the variable that holds the current voxel address by adding `StepX` (which is either -1 or 1) to it. If we have left the grid (`x == OutX`), then we also break. Otherwise we update the value of `NextXCrossing` to the above computed `DeltaX` value.

⟨*Step in X*⟩≡
```
if (*maxt < NextXCrossing)
    break;
x += StepX;
if (x == OutX)
    break;
NextXCrossing += DeltaX;
```

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

4.3 Scene Representation

We will also provide a class, `Scene` that manges the scene geometry; this lets the rest of the system call a single simple function that returns the first intersection of the geometry in the scene with a given ray. This gives us a layer in which to abstract away which intersection acceleration method is used and how the geometric primitives are stored in memory and managed.

⟨*Scene Constructor Declarations*⟩≡
```
Scene();
```

We'll just build a basic grid if there are any primitives to be rendered.

⟨*Scene Methods*⟩+≡
```
Scene::Scene() {
    ⟨Initialize Default Scene Options⟩
    accelerator = NULL;
    camera = new PinholeCamera;
    image = new Image;
    sampler = new JitterSampler;
}
```

⟨*Scene Public Data*⟩≡
```
Camera *camera;
Image *image;
JitterSampler *sampler;
```

⟨*Scene Private Data*⟩≡
```
Accelerator *accelerator;
vector<Primitive *> primitives;
```

⟨*Scene Method Declarations*⟩+≡
```
void AddPrimitives(const vector<Primitive *> &prim);
```

⟨*Scene Methods*⟩+≡
```
void Scene::AddPrimitives(const vector<Primitive *> &prim) {
    primitives = prim;
    if (primitives.size() > 0)
        accelerator = new GridAccelerator(primitives);
}
```

⟨*Finalize scene option parameter values*⟩≡
```
camera->FinalizeOptions();
sampler->FinalizeValues();
image->FinalizeValues();
```

We print out a row of plus signs, one per ten thousand eye rays traced, to convince the user that the renderer hasn't gone off into hyper-space.

⟨*Report rendering progress*⟩≡
```
static int eyeRaysTraced = 0;
if (eyeRaysTraced == 0)
    StatsRegisterCounter(STATS_BASIC, "Camera", "Eye Rays Traced",
        &eyeRaysTraced);
++eyeRaysTraced;
if (eyeRaysTraced % 10000 == 0) cerr << '+';
```

⟨*Scene Methods*⟩+≡
```
Scene::~Scene() {
    delete accelerator;
    delete camera;
    delete sampler;
    delete image;
//  for (u_int i = 0; i < primitives.size(); ++i)
//      delete primitives[i];
}
```

5. Camera

In addition to describing the objects that make up the scene, we also need to describe how the scene is viewed and how its three-dimensional representation is *imaged* into a two-dimensional image. We will start by presenting the `Camera` class, which generates of *eye rays* that sample the scene. In conjunction with the *imaging pipeline*, described in the next chapter, the camera is a key part of the process that takes the scene sample values from eye rays and generates the final output image.

5.1 Camera Model

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 cameras implement is `GenerateRay`, which was previously defined in Section 1.2.

⟨*Camera Method Implementations*⟩≡
```
Camera::Camera() {
    ⟨Camera Options Initialization⟩
}
```

A few user-settable options will be useful to define the camera's imaging process; these will be set by code in the RenderMan interface, defined in Appendix C.

After the image has been projected onto the viewing screen, a *screen window* describes which part of the viewing screen is matched to the output image. Normally, this is centered at the origin of screen space and is symmetric in x and y.

⟨*Camera Options*⟩+≡
```
Float ScreenLeft, ScreenRight, ScreenBottom, ScreenTop;
```

⟨*Camera Options Initialization*⟩+≡
```
ScreenLeft = -4. / 3.;
ScreenBottom = -1;
ScreenRight = 4. / 3.;
ScreenTop = 1.;
```

The user can also specify near and far *clipping planes*; these 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. For general overall accuracy and efficiency, they should encompass the scene as tightly as possible.

⟨*Camera Options*⟩+≡
```
Float ClipHither, ClipYon;
```

⟨*Camera Options Initialization*⟩+≡
```
ClipHither = 1e-2;
ClipYon = RI_INFINITY;
```

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.

- *Object space*: This is the coordinate system in which geometric primitives are defined. For example, spheres in `lrt` are defined to be centered at the origin of their object space.
- *World space*: While each primitive may have its own object space, there is a single world space that the objects in the scene are placed in relation to. Each primitive has an object to world transformation that determines how it is located in world space. World space is the standard frame that all spaces are defined in terms of.
- Camera space: A virtual camera is placed in the scene at some point with a particular viewing direction and "up" vector. This defines new coordinate space around that point with the origin at the camera's location, the z axis is mapped to the viewing direction and the y axis mapped to the up direction.
- Screen space: Screen space is defined on the image plane. A projective transformation projects objects in camera space onto the image plane; the parts inside the *screen window* are visible in the image that is generated. Depth z values in screen space range from zero to one, corresponding to points at the near and far clipping planes, respectively.
- Raster space: Raster space is the coordinate system for the actual image being rendered–in x and y, it ranges from $(0,0)$ to $(\text{xResolution}, \text{yResolution})$, 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.

We store two transformations that let us move around these spaces in the camera options. The first is the world space to camera space transformation; this can be used transform primitives in the scene into camera space. The origin of camera space is the camera's position, and the camera looks down the camera space z axis. Next is the camera space to screen space transformation. This describes the projection of the three dimensional scene in camera space onto the two dimensional camera plane in screen space. We also store a boolean value that records if the camera to screen projection is orthographic; this is handy when we generate eye rays.

⟨*Camera Options*⟩+≡
```
Transform WorldToCamera[2];
Transform CameraToScreen;
enum { Orthographic, Perspective } ProjectionType;
```

⟨*Camera Options Initialization*⟩+≡
```
ProjectionType = Perspective;
```

After the user has finished setting all of the camera options, we do some additional computation. In addition to precomputing one divided by the near clip plane distance, which will also be useful later, we also pre-compute a variety of transformations and the inverses that will be useful.

⟨*Camera Options Finalization*⟩≡
```
invClipHither = 1. / ClipHither;
```
⟨*Compute various useful camera transformations*⟩

⟨*Camera Options*⟩+≡
```
Float invClipHither;
```

The only non-trivial one of the precomput4ed transformations is `ScreenToRaster` note the composition of transformations where (reading backwards), we start with a point in screen space, translate so that the upper left corner of the screen is at the origin, and then scale by one over the screen width and height, giving us a point with x and y coordinates between negative one and zero. Then we scale this by $(1,-1,1)$, flipping it in y. 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.

We also store two `CameraToWorld` transformations; the second one will be useful when implementing a moving camera for motion blur (see Section 5.3).

⟨*Compute various useful camera transformations*⟩≡
```
CameraToWorld[0] = Transform(WorldToCamera[0].GetInverse());
CameraToWorld[1] = Transform(WorldToCamera[1].GetInverse());
ScreenToCamera = Transform(CameraToScreen.GetInverse());

ScreenToRaster =
    Scale(scene->image->XResolution, scene->image->YResolution, 1.) *
    Scale(1, -1, 1) *
    Scale(1. / (ScreenRight - ScreenLeft),
        1. / (ScreenTop - ScreenBottom), 1.) *
    Translate(Vector(-ScreenLeft, -ScreenTop, 0.));
RasterToScreen = Transform(ScreenToRaster.GetInverse());

WorldToScreen = CameraToScreen * WorldToCamera[0];
RasterToCamera = ScreenToCamera * RasterToScreen;
```

⟨*Camera Options*⟩+≡
```
Transform CameraToWorld[2];
Transform ScreenToCamera;
Transform ScreenToRaster, RasterToScreen;
Transform RasterToCamera;
Transform WorldToScreen;
```

5.2 Pinhole Camera

We can now define a simple pinhole camera model. This camera doesn't handle moving cameras and doesn't try to simulate depth of field, but shows a simple `GenerateRay` function.

Given a sample value, we just make sure that the corresponding location on the image is inside any user-specified crop window, and then go ahead and generate a ray.

⟨*PinholeCamera Method Implementations*⟩≡
```
bool PinholeCamera::GenerateRay(Float sample[5], Ray &ray) const {
    ⟨Check sample against CropWindow⟩
    ⟨Convert sample to camera space and generate eye ray⟩
}
```

The sampler generates a sequence of samples in raster space. The first thing we do is make sure that the current sample is relevant, based on the crop window that the user may have specified. (This process could be made more efficient if the sampler paid attention to the crop window and didn't generate any samples outside of it, though because sample generation is generally quite fast and there's usually no crop window, anyway, we just discard any unneeded ones.)

⟨*Check sample against CropWindow*⟩≡

```
if (sample[0] < scene->image->SampleCropLeft ||
    sample[0] > scene->image->SampleCropRight ||
    sample[1] < scene->image->SampleCropBottom ||
    sample[1] > scene->image->SampleCropTop)
    return false;
```

We can now convert the image sample into an to eye ray. An eye ray is generated for each sample that passed the crop window test by the following process: first, we transform the raster-space samples into points in camera space; this gives us a point located on the near clipping plane. Then we can generate the eye ray. There are two cases to handle, depending on whether the projection being used is a perspective projection or an orthographic projection. In both cases, we start by generating a ray in camera space.

For a perspective projection, all rays original from the origin, $(0,0,0)$ in camera space. Thus, 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 direcion is component-wise equal to its point position. Rather than doing a useless subtraction to convert the point to a direction, we just component-wise initialize the vector `ray.D` from the point `Pcamera`

For an orthographic projection, all we do is take the camera-space viewing direction down the z axis, $(0,0,1)$, and construct a ray from the camera space sample position in that direction.

In both cases, we then transform the ray into world space using the camera to world transformation. This maps both the camera-space origin and direction into world space, giving a ray that can be traced into the scene.

This overall process is slightly inefficient in two ways: first, we are re-computing the world-space position of the origin or the direction $(0,0,1)$ in camera-space for each ray (depending on the type of projection), even though these values are the same each time around. Second, we could compute a single raster to world transformation, and directly transform the raster-space sample into world space; the world-space direction is then equal to the difference between that point and the world-space origin. As these inefficiencies don't significantly contribute to overall performance, however, we leave them here in order to improve clarity of exposition.

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.

⟨*Convert sample to camera space and generate eye ray*⟩≡

```
Point Pcamera;
switch (ProjectionType) {
    case Camera::Orthographic:
        Pcamera = RasterToCamera(Point(sample[0], sample[1], 0));
        ray = Ray(Pcamera, Vector(0,0,1));
        break;
    case Camera::Perspective:
        Pcamera = RasterToCamera(Point(sample[0], sample[1], 0));
        ray = Ray(Point(0, 0, 0), Vector(Pcamera.x, Pcamera.y, Pcamera.z));
        ray.D *= invClipHither;
        break;
}
ray = CameraToWorld[0](ray);
return true;
```

5.3 Moving Camera

If we are computing an image with moving objects and want to include motion blur, the user needs to specify times at which the camera shutter opens and closes; only the motion of objects between these two times will be recorded in the image.

⟨*Camera Options*⟩+≡
```
Float ShutterStart, ShutterEnd;
```

⟨*Camera Options Initialization*⟩+≡
```
ShutterStart = 0.;
ShutterEnd = 1.;
```

5.4 Depth of Field

For depth of field, the user needs to specify some parameters that describe where the camera is focused and how large its aperture is; The default values here disable depth of field.

⟨*Camera Options*⟩+≡
```
Float FStop, FocalLength, FocalDistance;
```

⟨*Camera Options Initialization*⟩+≡
```
FStop = RI_INFINITY;
FocalLength = FocalDistance = RI_INFINITY;
```

5.5 Projective Transformations

This text moved here from the geometry and transformations chapter

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

⟨*Transform Methods*⟩+≡
```
Transform Orthographic(Float n, Float f) {
    Transform scale = Scale(1., 1., 1. / (f-n));
    Transform translate = Translate(Vector(0., 0., n));
    Transform ret = scale * translate;
    return Transform( ret.GetInverse() );
}
```

perspective:

⟨*Transform Methods*⟩+≡
```
Transform Frustum(Float left, Float right,
                  Float bottom, Float top,
                  Float near, Float far) {
    Float m[4][4];

    m[0][0] = 2. * near / (right - left);
    m[0][1] = 0;
    m[0][2] = (right + left) / (right - left);
    m[0][3] = 0;

    m[1][0] = 0;
    m[1][1] = 2. * near / (top - bottom);
    m[1][2] = (top + bottom) / (top - bottom);
    m[1][3] = 0;

    m[2][0] = 0;
    m[2][1] = 0;
    m[2][2] = -(far + near) / (far - near);
    m[2][3] = 2. * far * near / (far - near);

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

    Transform ret(m);
    return ret;
}
```

⟨*Transform Methods*⟩+≡
```
Transform Perspective(Float fovy, Float aspect,
                      Float znear, Float zfar) {
    Float ymax = znear * tan(Radians(fovy) / 2.);
    Float ymin = -ymax;
    Float xmin = ymin * aspect;
    Float xmax = ymax * aspect;
    return Frustum(xmin, xmax, ymin, ymax, znear, zfar);
}
```

6. Imaging Model

We store the image that's being computed in the `Image` class. This stores the final pixels computed from the image sample values until all of the samples have arrived and then applies a set of *imaging* operations which improve the final image before it is written out. In general, an image stores a set of *channels*; numeric values at a regular grid of pixel sample locations. Each channel has a particular semantic meaning. Many image formats just store RGB channels, representing weighted red, green, and blue color contributions. More generally, we can store some combination of a color representation, the depth to the first visible object, an alpha value, etc.

In the image constructor, we just initialize default values for the image's various options. Only after all of those options have been set by the user and we know the final image resolution, etc, will we go ahead and allocate space for the image.

⟨*Image Method Definitions*⟩≡
```
Image::Image() {
    Pixels = NULL;
    Alphas = NULL;
    Depths = NULL;
    WeightSums = NULL;
    ⟨Image Options Initialization⟩
}
```

One of the most important image parameters is the overall image resolution–`XResolution` and `YResolution` hold the total number of pixels in the x and y directions.

⟨*Image Options*⟩≡
```
int XResolution, YResolution;
```

These have reasonable values as defaults.

⟨*Image Options Initialization*⟩≡
```
XResolution = 640;
YResolution = 480;
```

Once the resolutions hold their final values (and sundry other imaging options have been set, the Image's `FinalizeValues` method is called. This computes some additional parameter values based on the ones the user set, and allocates arrays needed to store the image channels as well as an accumulated weight for each pixel in `WeightSums`. For now, just consider `WeightSums` as 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.

⟨*Image Method Definitions*⟩+≡
```
void Image::FinalizeValues() {
    ⟨Compute sample crop window⟩
    ⟨Initialize base and differences⟩
    ⟨Allocate image storage⟩
}
```

This *crop window* 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 given as floating-point offsets into the image, with each coordinate ranging from zero to one and where $(0,0)$ is the upper left hand corner of the image. Perversely, this means that the "bottom" crop coordinate, though less than the "top", specifies a line closer to the top of the image than "top". Here are the new fields in the `Image` that hold the window's extent:

⟨*Image Options*⟩+≡
```
Float CropLeft, CropRight, CropBottom, CropTop;
```

⟨*Image Options Initialization*⟩+≡
```
CropLeft = CropBottom = 0.;
CropRight = CropTop = 1.;
```

In conjunction with the overall image resolution, the crop window extent gives us the extent of integer pixel locations that we'll be writing out. `XBase` and `YBase` store the corner of the crop window, and `XWidth` and `YWidth` give the pixel widths in each direction. Given a pixel (x,y) (which must be inside the pixel crop window), the pixel arrays are indexed as `(y - YBase) * XWidth + (x - XBase)`.

⟨*Initialize base and differences*⟩≡
```
XBase  = (int)Clamp(ceil(XResolution * CropLeft),   0, XResolution);
XWidth = (int)Clamp(ceil(XResolution * CropRight),  0, XResolution) - XBase;
YBase  = (int)Clamp(ceil(YResolution * CropBottom), 0, YResolution);
YWidth = (int)Clamp(ceil(YResolution * CropTop),    0, YResolution) - YBase;
```

⟨*Image Private Data*⟩≡
```
int XBase, YBase, XWidth, YWidth;
```

6.1 Image Storage

We add a field to our options so that we can record which channels of the final image to store (e.g. just red, green, and blue, or just z depth, etc.); this is called `DisplayMode`

⟨*Image Options*⟩+≡
```
RtToken DisplayMode;
```

By default, we compute images that store RGB color channels and an alpha channel.

⟨*Image Options Initialization*⟩+≡
```
DisplayMode = RI_RGBA;
```

Now that we know the pixel resolution of the image, we allocate arrays to store pixel values as samples come in.

⟨*Allocate image storage*⟩≡
```
⟨Allocate Pixels if needed⟩
⟨Allocate Alphas if needed⟩
⟨Allocate Depths if needed⟩
WeightSums = new Float[XWidth * YWidth];
for (int i = 0; i < XWidth * YWidth; ++i)
    WeightSums[i] = 0.;
```

⟨*Image Private Data*⟩+≡
```
Spectrum *Pixels;
Float *Alphas, *Depths;
Float *WeightSums;
```

⟨*Allocate* Pixels *if needed*⟩≡
```
if (DisplayMode == RI_RGB  || DisplayMode == RI_RGBA ||
    DisplayMode == RI_RGBZ || DisplayMode == RI_RGBAZ) {
    Pixels = new Spectrum[XWidth * YWidth];
}
else Pixels = NULL;
```

⟨*Allocate* Alphas *if needed*⟩≡
```
if (DisplayMode == RI_RGBA || DisplayMode == RI_RGBAZ ||
    DisplayMode == RI_A    || DisplayMode == RI_AZ) {
    Alphas = new Float[XWidth * YWidth];
    for (int i = 0; i < XWidth * YWidth; ++i)
        Alphas[i] = 0.;
}
else Alphas = NULL;
```

⟨*Allocate* Depths *if needed*⟩≡
```
if (DisplayMode == RI_RGBAZ || DisplayMode == RI_RGBZ ||
    DisplayMode == RI_AZ    || DisplayMode == RI_Z) {
    Depths = new Float[XWidth * YWidth];
    for (int i = 0; i < XWidth * YWidth; ++i)
        Depths[i] = 0.;
}
else Depths = NULL;
```

The only thing to be done in the destructor is to delete the memory allocated in the constructor.

⟨*Image Method Declarations*⟩+≡
```
~Image();
```

⟨*Image Method Definitions*⟩+≡
```
Image::~Image() {
    delete[] Pixels;
    delete[] Alphas;
    delete[] Depths;
    delete[] WeightSums;
}
```

`AddSample` is the `Image` method called by the camera after it has computed the radiance along a ray. It passes the raster-space position of the image sample as well as its the radiance value. Because some of the details of `AddSample` only make sense after we describe sampling theory in Chapter 7, we will here present a simplified version.

⟨*Image Method Declarations*⟩+≡
```
void AddSampleBasic(const Point &PRaster, const Spectrum &radiance, Float alpha);
```

Here, we decide which image pixel the sample contributes to by taking its integer component; as long as this is in fact inside the image bounds, it adds its contribution to the appropriate pixel. (It may legitimately be outside the image bounds; this is covered in Chapter 7. Here, we will just discard these samples.)

⟨*Image Method Definitions*⟩+≡
```
void Image::AddSampleBasic(const Point &Praster, const Spectrum &radiance,
            Float alpha) {
    ⟨Check for out-of-bounds sample⟩
    ⟨Store sample data in arrays⟩
}
```

⟨*Check for out-of-bounds sample*⟩≡
```
if (Praster.x < XBase || Praster.x >= XBase + XWidth ||
    Praster.y < YBase || Praster.y >= YBase + YWidth)
    return;
```

So long as the (x,y) position is ok, we can compute the offset into the relevant arrays and add the ray's contribution. In the end, we will divide the accumulated value in each channel by its `WeightSums` value, averaging multiple contributing samples.

⟨*Store sample data in arrays*⟩≡
```
int x = int(Praster.x), y = int(Praster.y);
int offset = (y - YBase) * XWidth + (x - XBase);
if (Pixels) Pixels[offset] += radiance;
if (Alphas) Alphas[offset] += alpha;
if (Depths) Depths[offset] += Praster.z;
WeightSums[offset] += 1;
```

6.2 Imaging Pipeline

Once the camera has computed all of the image samples, the pixels in the image passes through an *imaging pipeline*. This is a set of transformations that are applied to the final image that help compensate for the discrete nature of output formats (such as bitmap files or displays) in order to create the highest-quality result possible. The `Write()` method implements this pipeline and writes the final pixel values out to disk.

⟨*Image Method Declarations*⟩+≡
```
void Write() const;
```

We subdivide the write function into three main parts: first we take the accumulated values stored in `Pixels` `Alphas` and `Depths` and compute a final floating-point RGB, alpha, and depth values, as appropriate, for each output pixel. We then apply a set of imaging transformations that make last-minute changes to the pixel values, in order to improve the quality of the final image or compensate for the display device. Finally, the image is saved to disk and we clean up memory allocated in this function, etc.

⟨*Image Method Definitions*⟩+≡
```
void Image::Write() const {
    ⟨Compute floating-point RGB pixels⟩
    ⟨Apply exposure to RGB pixels and dither⟩
    ⟨Save image and cleanup⟩
}
```

The first stage of this process is also divided into three parts.

⟨*Compute floating-point RGB pixels*⟩≡
```
⟨Allocate working imaging memory and convert to RGB⟩
⟨Apply filter weights⟩
⟨Compute premultiplied alpha color values⟩
```

We allocate temporary space to hold copies of whichever of the the spectrum, alpha, and depth pixel values are being recorded. This lets us work on a temporary copy of the data without disturbing the `Pixels` `Alphas` or `Depths` arrays. This makes it possible to call `Image`'s `Write` method multiple times, or to render part of the image, call it, and then continue rendering into the same `Image` While we're doing this, we'll also use `Spectrum`'s `ConvertToRGB` method to convert the spectral representation to RGB in case it isn't RGB internally.

⟨*Allocate working imaging memory and convert to RGB*⟩≡
```
Float *RGBOut = NULL, *AlphaOut = NULL, *DepthOut = NULL;
if (Pixels) {
    RGBOut = new Float[3 * XWidth * YWidth];
    for (int offset = 0; offset < XWidth * YWidth; ++offset)
        Pixels[offset].ConvertToRGB(&RGBOut[3 * offset]);
}
if (Alphas) {
    AlphaOut = new Float[XWidth * YWidth];
    memcpy(AlphaOut, Alphas, XWidth * YWidth * sizeof(Float));
}
if (Depths) {
    DepthOut = new Float[XWidth * YWidth];
    memcpy(DepthOut, Depths, XWidth * YWidth * sizeof(Float));
}
```

Next, we divide each pixel sample value by the value of `WeightSums` for that pixel; again, see Section 7.3 for details. For efficiency, we compute one over the weight value once and then multiply by that instead of dividing by the weight value each time.

⟨*Apply filter weights*⟩≡
```
for (int offset = 0; offset < XWidth * YWidth; ++offset) {
    if (WeightSums[offset] == 0.)
        continue;

    Float invWt = 1. / WeightSums[offset];
    if (RGBOut) {
        for (int c = 0; c < 3; ++c) {
            RGBOut[3*offset + c]     *= invWt;
            if (RGBOut[3*offset + c] < 0.) RGBOut[3*offset + c] = 0.;
        }
    }
    if (AlphaOut) AlphaOut[offset] *= invWt;
    if (DepthOut) DepthOut[offset] *= invWt;
}
```

If we're storing color and alpha values together, we'll compute *premultiplied alpha* (also known as *associated alpha*) color. This just means that we multiply each color by its alpha value; this has 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*⟩≡
```
if (RGBOut && AlphaOut) {
    for (int i = 0; i < XWidth * YWidth; ++i) {
        for (int j = 0; j < 3; ++j)
            RGBOut[3*i+j] *= AlphaOut[i];
    }
}
```

Now we can move onto the second stage of the imaging and output pipeline. We apply a set of transformations to the pixels, taking care of things like converting them to integer values, if the output format doesn't support floating-point, etc.

⟨*Apply exposure to RGB pixels and dither*⟩≡
```
⟨Apply exposure and gamma correction⟩
⟨Apply imager or tone reproduction⟩
⟨Scale to output range and dither⟩
```

First, we apply *gain* and *gamma correction* to the pixel values. Gain is a scalar value that all pixels are multiplied by–this can brighten or darken the image as desired by the user–and gamma correction is a way of compensating for nonlinearities in the display. For example, CRT displays have a non-linear response to the voltage applied to them. Thus, if `lrt` computes one pixel with an intensity of 100 and another with an intensity of 50, in general, the first pixel won't be twice as bright as the second on your monitor. Gamma correction corrects for this by modeling the monitor's response with an exponential curve. (A better solution is for computers' display subsystems to compensate for display non-linearity themselves. Unfortunately, this is not usually done.) The gamma correction step is thus not applied to depth values.

In short, here we compute:

$$\text{color}' = (\text{color} \times \text{gain})^{1/\text{gamma}}$$

If `Gain` and `Gamma` are both one (their default values), this is a no-op, in which case we skip the unnecessary and expensive `pow()` function calls.

We'll start out by adding fields in the our options to hold these values, as set by the user.

⟨*Image Options*⟩+≡
```
Float Gain, Gamma;
```

⟨*Image Options Initialization*⟩+≡
```
Gain = Gamma = 1.;
```

And now it's a straightforward application of the above formula to the channels that we have to work with.

⟨*Apply exposure and gamma correction*⟩≡
```
if ((RGBOut || AlphaOut) && (Gain != 1. || Gamma != 1.)) {
    Float invGamma = 1. / Gamma;
    if (RGBOut)
        for (int offset = 0; offset < 3 * XWidth * YWidth; ++offset)
            RGBOut[offset] = pow(RGBOut[offset] * Gain, invGamma);
    if (AlphaOut)
        for (int offset = 0; offset < XWidth * YWidth; ++offset)
            AlphaOut[offset] = pow(AlphaOut[offset] * Gain, invGamma);

}
```

Next the imager shader (if any) is applied. In general, this is a place where the user can specify general operations to be applied to all of the pixels. We use this as a chance to apply tone reproduction to the image. This also needs a field in our options.

⟨*Image Options*⟩+≡
```
RtToken Imager;
```

⟨*Image Options Initialization*⟩+≡
```
Imager = RI_NULL;
```

⟨*Apply imager or tone reproduction*⟩≡
```
if (Imager != RI_NULL)
    ApplyImager(Imager, RGBOut, AlphaOut, DepthOut, XWidth * YWidth);
```

We now have a set of pixel values that are almost ready to be written out. The final step is to apply *quantization* and dithering. Quantization maps the continuous floating-point pixel and depth values that we have computed to a set of discrete integer values. Given a user-specified number that pixels with value 1 should map to, stored in `ColorQuantOne` (or `DepthQuantOne` for depth values), we scale the pixels by this amount. The result is rounded to the nearest integer and clamped to lie between the minimum and maximum values that the display supports (`ColorQuantMin` and `ColorQuantMax` with respective values for depth as well.)

To reduce visual artifacts due to the quantization step, *dithering* can be applied. A small random number is added to each channel of each pixel; this can make the transition between two integer quantized values across the image less visually noticeable.

⟨*Image Options*⟩+≡
```
int ColorQuantOne, ColorQuantMin, ColorQuantMax;
Float ColorQuantDither;
```

⟨*Image Options Initialization*⟩+≡
```
ColorQuantOne = 255;
ColorQuantMin = 0;
ColorQuantMax = 255;
ColorQuantDither = 0.5;
```

And for depth value quantization...

⟨*Image Options*⟩+≡
```
int DepthQuantOne, DepthQuantMin, DepthQuantMax;
Float DepthQuantDither;
```

⟨*Image Options Initialization*⟩+≡
```
DepthQuantOne = 0;
DepthQuantMin = 0;
DepthQuantMax = 0;
DepthQuantDither = 0.;
```

We skip the appropriate quantization and dithering steps for color and depth if the user has set the value of `ColorQuantOne` or `DepthQuantOne` to zero, respectively; this indicates that the output format supports continuous floating point values directly and that dithering is unnecessary.

⟨*Scale to output range and dither*⟩≡
```
if (ColorQuantOne != 0 && RGBOut)
    scaleAndDither(ColorQuantOne, ColorQuantMin,
        ColorQuantMax, ColorQuantDither, RGBOut,
        3 * XWidth * YWidth);
if (ColorQuantOne != 0 && AlphaOut)
    scaleAndDither(ColorQuantOne, ColorQuantMin,
        ColorQuantMax, ColorQuantDither, AlphaOut,
        XWidth * YWidth);
if (DepthQuantOne != 0 && DepthOut)
    scaleAndDither(DepthQuantOne, DepthQuantMin,
        DepthQuantMax, DepthQuantDither, DepthOut,
        XWidth * YWidth);
```

Since quantization and dithering happens in the same way for all of the different types of image channels, we gather the code in a single place. The `pixels` array is a pointer to the floating-point data to be processed, and `nSamples` is a count of the total number of samples in the array to process.

⟨*Image Private Functions*⟩≡
```
static void scaleAndDither(Float one, Float min, Float max, Float dither,
    Float *pixels, int nPixels);
```

⟨*Image Method Definitions*⟩+≡
```
void Image::scaleAndDither(Float one, Float min, Float max, Float ditheramp,
    Float *pixels, int nSamples) {
    for (int i = 0; i < nSamples; ++i) {
        pixels[i] = Round(pixels[i] * one + RandomFloat(-ditheramp, ditheramp));
        pixels[i] = Clamp(pixels[i], min, max);
    }
}
```

6.3 Output

And now we can finally write out the image that we've generated and cleanup. This is the easiest part!

⟨*Save image and cleanup*⟩≡
⟨*Write TIFF*⟩
⟨*Pop up image viewer for debugging*⟩
⟨*Release allocated image writing memory and cleanup*⟩

As it turns out, the TIFF writing routines (see Appendix B) have interfaces such that the representation we chose for pixels above maps to them perfectly. We just choose the appropriate one, depending on whether the user wants a floating-point TIFF or the more common 8-bit per channel TIFF.

A few more fields in our options are needed to tell us what to do with the final image (which filename to write it out to, etc). By default, they selected components from `DisplayMode` are saved in a file named `out.tiff`.

⟨*Image Options*⟩+≡
```
RtToken DisplayType, DisplayName;
```

⟨*Image Options Initialization*⟩+≡
```
DisplayType = RI_FILE;
DisplayName = "out.tiff";
```

⟨*Write TIFF*⟩≡
```
if (ColorQuantOne != 0.)
    TIFFWrite8Bit(DisplayName, RGBOut, AlphaOut, DepthOut, XWidth, YWidth);
else
    TIFFWriteFloat(DisplayName, RGBOut, AlphaOut, DepthOut, XWidth, YWidth);
```

For the user's convenience, we optionally run an image viewing program after the image has been written out; if this is desired, the `ImageViewer` field in `Image` will be non-`NULL`.

⟨*Image Options*⟩+≡
```
char *ImageViewer;
```

⟨*Image Options Initialization*⟩+≡
```
ImageViewer = NULL;
```

⟨*Pop up image viewer for debugging*⟩≡
```
if (ImageViewer != NULL && !fork())
    if (execlp(ImageViewer, ImageViewer, DisplayName,
            NULL) == -1) {
        Error("Unable to run image viewing program %s: %s", ImageViewer,
            strerror(errno));
        exit(1);
}
```

⟨*Release allocated image writing memory and cleanup*⟩≡
```
delete[] RGBOut;
delete[] AlphaOut;
delete[] DepthOut;
```

7. Sampling and Reconstruction

We'll can now fill in some blanks about 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 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 a scene with rays and reconstructing a set of 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*. 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. 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.

7.1 Basic Sampling Theory

The Fourier transform is the basis of most of sampling theory; we will not explain the transform or its application, however–some good references are given in the further reading section. Instead, we will draw upon results derived with the Fourier transform, stating them without proof.

Intuitively, the more smooth that a signal 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 higher frequencies are added to it and its *frequency content* increases), 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*

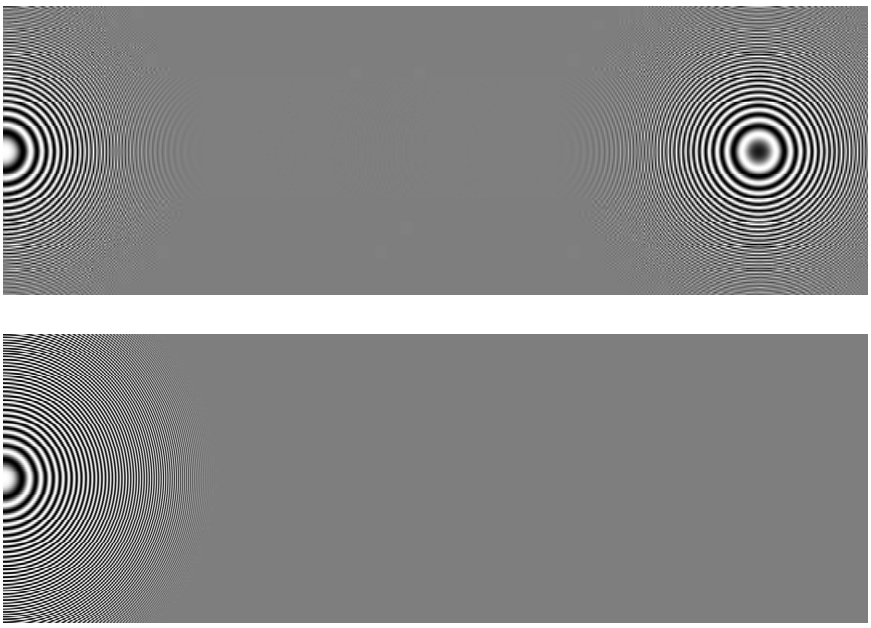


Figure 7.1: Aliasing from point sampling the function $\cos(x^2+y^2)$; at the left side of the image, the function has a low frequency–tens of pixels per cycle–so it is represented accurately. Moving to the right, however, aliasing artifacts appear in the top image since the sampling rate doesn't keep up with the function's highest frequency. If high frequency elements of the signal are filtered 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.)

frequency): this is the separation in Δx between adjacent samples of the signal. If the sampling rate is uniform, the spacing between all of the samples is constant.

The *sampling theorem* makes the relationship between sampling rate and the accuracy of the reconstruction precise: it says that so long as the frequency of sample points (ω_s) is greater than twice the maximum frequency present in the signal (ω_m), it is possible to reconstruct the original signal *perfectly* from the samples. This minimum sampling frequency is called the *Nyquist frequency*.

In order to perform this perfect reconstruction, a particular function must be used to generate the new signal. This *ideal reconstruction filter* is the *sinc function*:

$$r(x) = \text{sinc}\left(\frac{\omega_m}{2\pi}x\right)$$

where $\text{sinc}(x) = (\sin x)/x$.

If we don't sample a signal 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. Figure 7.1 shows the effect of sampling the function $f(x,y) = \cos(x^2+y^2)$; the origin $(0,0)$ is at the center of the left side 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 the image are 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 10.1, for instance. For a given sampling rate, the best way to avoid aliasing is to *pre-filter* the signal and remove any frequencies higher than the Nyquist limit.

One disadvantage of the sinc reconstruction filter is that it has *infinite support*; the filter is non-zero over all values from $-\infty$ to ∞, so all of the samples taken must be filtered in order to reconstruct the signal at any point. In the interests of efficiency, other filters with *finite support* are usually used instead. Although these other filters do not have the same theoretical ideal reconstruction properties as the sinc, in practice they can perform nearly as well; a number of them will be described in Section 7.3.

The extension of these ideas to the two-dimensional case of sampling and reconstructing images is straightforward; we have an image, which (following Mitchell), we can think of as a function of image locations to radiance values

$$f(x,y) \rightarrow L$$

where $x \in [0, \text{xResolution})$ and $y \in [0, \text{yResolution})$. The good news is that, with our ray-tracer, we can evaluate this at any (x,y) we choose. The bad news is that we can only point sample f and that in general we can't pre-filter it to remove high-frequencies.

7.2 Image Sampling

We can now describe the operation of a few classes that generate good image sampling patterns. Both inherit from an abstract `Sampler` class that defines their interface. The `Scene` calls their `GetNextImageSample` method until it returns `false`; as long as it keeps returning `true`, it will fill in the `sample` array with a five-dimensional sample point. The first two entries in the `sample` array 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. Finally, the last two samples give a (u,v) lens position to sample for depth of field; these also vary from zero to one.

⟨*Sampler Interface*⟩≡
```
virtual bool GetNextImageSample(Float sample[5]) = 0;
```

Jittered Sampling.

The first sample generator that we will introduce is a simple jittered sampler. This starts with a uniform sample pattern aligned with the pixel grid and optionally jitters each sample position. If jittering is not enabled, then sampling is just over a uniform grid.

Figure 7.2 shows a comparison of a few basic sampling patterns. On the right 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 a uniform pattern; as described above, uniform sampling can have result in aliasing artifacts that can be avoided by jittering. On the right, we have jittered the uniform pattern, adding a random offset to each sample's location. 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.

⟨*JitterSampler Method Definitions*⟩≡
```
JitterSampler::JitterSampler() {
    ⟨Initialize current raster position⟩
}
```

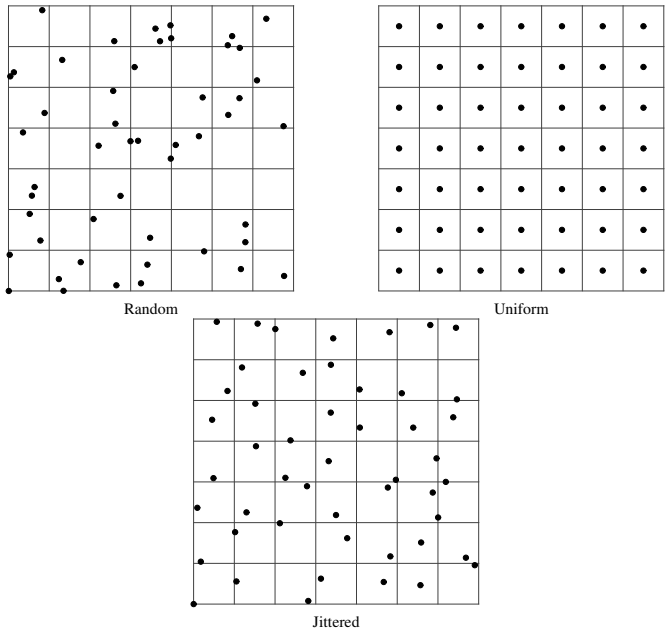


Figure 7.2: Three sampling patterns. The random pattern on the left is a poor pattern, with many clumps of samples that leave large sections of the image poorly sampled. In the middle is a uniform pattern which is better distributed but that can exacerbate aliasing artifacts. On the right is a jittered pattern, which turns aliasing from the uniform pattern into high-frequency noise.

We divide the image plane into a set of *strata*; consider a grid 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. Rather than always having just one stratum per pixel, the user can also ask `lrt` to vary the average number of image samples that are computed per output pixel area. The default, four samples (two in the x direction and two in y), gives reasonably good results on many images.

⟨*Sampler Options*⟩≡
```
Float PixelSamples[2];
```

⟨*Sampler Options Initialization*⟩≡
```
PixelSamples[0] = PixelSamples[1] = 2.;
```

The `JitterSamples` field in our options selects between uniform and jittered sampling patterns.

⟨*Sampler Options*⟩+≡
```
bool JitterSamples;
```

⟨*Sampler Options Initialization*⟩+≡
```
JitterSamples = true;
```

For convenience, we consider each cell to be one unit wide; in order to convert from a sample grid position to a raster-space pixel position, we divide by the number of pixel samples; these correction factors are precomputed and stored in `DeltaX` and `DeltaY`, after we're sure that the final values of `PixelSamples` have been set. For convenience, we also cache the total number of samples in the x and y directions.

⟨*JitterSampler Method Definitions*⟩+≡
```
void JitterSampler::FinalizeValues() {
    ⟨Grab options and pre-compute for sampling⟩
    Sampler::FinalizeValues();
}
```

⟨*Grab options and pre-compute for sampling*⟩≡
```
XMax = (int)ceil(scene->image->XResolution * PixelSamples[0]);
YMax = (int)ceil(scene->image->YResolution * PixelSamples[1]);
DeltaX = 1. / PixelSamples[0];
DeltaY = 1. / PixelSamples[1];
```

⟨*JitterSampler Private Data*⟩≡
```
int XMax, YMax;
Float DeltaX, DeltaY;
```

We keep track of which grid cell we've last provided a sample for in `XPos` and `YPos`. We initialize `XPos` to an illegal value here, since the first thing we'll do when `GetNextImageSample` is called is to increment `XPos` and see if we're at a valid sample location.

⟨*Initialize current raster position*⟩≡
```
XPos = -1;
YPos = 0;
```

⟨*JitterSampler Private Data*⟩+≡
```
int XPos, YPos;
```

We can now write the `GetNextImageSample` function.

⟨*JitterSampler Method Definitions*⟩+≡
```
bool JitterSampler::GetNextImageSample(Float sample[5]) {
    ⟨Try to advance to next jittered sample⟩
    ⟨Compute sample point⟩
}
```

We first advance to the next cell in the sampling grid (which goes from $(0,0)$ to $(\mathrm{XMax}, \mathrm{YMax})$, 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 zero and try to advance the y position. If that takes us off the image as well, then we've provided all of the image samples and we return `false`.

⟨*Try to advance to next jittered sample*⟩≡
```
if (++XPos >= XMax) {
    XPos = 0;
    if (++YPos >= YMax)
        return false;
}
```

Now we need to generate a sample in the `XPos YPos` cell. If jittereing is enabled, we add a random offset between zero and one to each position and then convert it to raster-space by dividing by the number of samples per pixel. Time and lens positions are just sampled randomly.

⟨*Compute sample point*⟩≡
```
if (JitterSamples) {
    sample[0] = (XPos + RandomFloat()) * DeltaX;
    sample[1] = (YPos + RandomFloat()) * DeltaY;
    sample[2] = RandomFloat();
    sample[3] = RandomFloat();
    sample[4] = RandomFloat();
}
else {
    sample[0] = (XPos + 0.5) * DeltaX;
    sample[1] = (YPos + 0.5) * DeltaY;
    sample[2] = 0.5;
    sample[3] = 0.5;
    sample[4] = 0.5;
}
return true;
```

7.3 Image Reconstruction

Once we have a set of image samples, we want to *resample* them and compute a final set of samples at the output pixel locations. This is a three step process: first we want to reconstruct a new signal from the samples; next, we need to prefilter that signal, removing frequencies higher than the pixel sampling frequency; finally, we need to resample that signal at the pixel locations.

It turns out that these three steps can be rolled into one; because we know the pixel locations ahead of time, we can just reconstruct and filter incoming samples from the camera as they're computed. .

Box Filter.

One of the most commonly used filters in graphics is the *box filter* (and in fact, when filtering and reconstruction isn't addressed explicitly, the box filter is the *de facto* result–that is effectively what the `Image::AddSampleBasic` method in Section 6.1 on page 96 does.) The box filter equally weights all samples within a square region of the image. Though computationally efficient, it's just about the worst filter possible.

⟨*Sampling Declarations*⟩+≡
```
extern Float BoxFilter(Float x, Float y, Float xwidth, Float ywidth);
```

All of the following filter functions take four parameters: `x` and `y`, the position of the sample being filtered in relationship to the output pixel, and `xwidth` and `ywidth`, the filters' total extent in the x and y directions.

⟨*Filter Function Definitions*⟩≡
```
Float BoxFilter(Float x, Float y, Float xwidth, Float ywidth) {
    if (x >= -xwidth * 0.5 && x <= xwidth * 0.5 &&
        y >= -ywidth * 0.5 && y <= ywidth * 0.5)
        return 1.;
    else return 0.;
}
```

Triangle Filter.

The triangle filter is slightly better 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.

⟨*Sampling Declarations*⟩+≡
```
extern Float TriangleFilter(Float x, Float y, Float xwidth, Float ywidth);
```

⟨*Filter Function Definitions*⟩+≡
```
Float TriangleFilter(Float x, Float y, Float xwidth, Float ywidth) {
    if (x >= -xwidth * 0.5 && x <= xwidth * 0.5 &&
        y >= -ywidth * 0.5 && y <= ywidth * 0.5)
        return (xwidth * 0.5 - fabs(x)) * (ywidth * 0.5 - fabs(y));
    else return 0.;
}
```

Gaussian Filter.

The Gaussian is the first filter that's not unreasonable. 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.

⟨*Sampling Declarations*⟩+≡
```
extern Float GaussianFilter(Float x, Float y, Float xwidth, Float ywidth);
```

⟨*Filter Function Definitions*⟩+≡
```
Float GaussianFilter(Float x, Float y, Float xwidth, Float ywidth) {
    Float d2 = (x*x+y*y);
    Float d = sqrt( d2 );
    Float w2 = 0.5*(xwidth*xwidth + ywidth*ywidth);
    Float w = sqrt( w2 );
    if (d > w)
        return 0.0;
    else
        return exp(-d2) - exp(-w2);
}
```

Mitchell Filter.

Filter design is a difficult craft, mixing mathematical analysis and perceptual experiments: like sampling patterns, best filter mathematically doesn't necessarily look the best to human observers. Mitchell and Netravali have developed a family of parametrized filter functions. 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 *ringing* and *blurring*, two common artifacts from poor reconstruction filters.

⟨*Sampling Declarations*⟩+≡
```
extern Float MitchellFilter(Float x, Float y, Float xwidth, Float ywidth);
```

⟨*Filter Function Definitions*⟩+≡
```
Float MitchellFilter(Float x, Float y, Float xwidth, Float ywidth) {
    x /= xwidth;
    y /= ywidth;
    Float r = sqrt(x*x + y*y);
    if (r > 2.) return 0.;
#define B .33333333333333333
#define C .33333333333333333
#define ONE_SIXTH .16666666666666666
    else 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;
    }
#undef B
#undef C
#undef ONE_SIXTH
}
```

Processing camera samples.

We can now put all this together and write the function that takes image samples and filters them to compute pixel values. First, we store a function pointer to the particular filter function that the user wants to be used.

⟨*Sampler Options*⟩+≡
```
RtFilterFunc Filter;
```

⟨*Sampler Options Initialization*⟩+≡
```
Filter = RiGaussianFilter;
```

We also need the width of the filter in each direction, which can be set by the user.

⟨*Sampler Options*⟩+≡
```
Float FilterXWidth, FilterYWidth;
```

⟨*Sampler Options Initialization*⟩+≡
```
FilterXWidth = FilterYWidth = 2.0;
```

If the user did specify a crop window, we'll compute a sample crop window; due to the non-zero sample filter widths, some samples outside of the crop window must be computed in order to correctly compute the pixels inside the crop window. The sample crop window gives a raster space bound on the sample coordinates that need to be computed.

⟨*Image Options*⟩+≡
```
Float SampleCropLeft, SampleCropRight;
Float SampleCropTop, SampleCropBottom;
```

Since values can only be computed after both the overall image resolution and the sample filter widths have been set, we defer their computation until we're sure that those values have been finalized.

⟨*Compute sample crop window*⟩≡
```
int RasterCropLeft = (int)Clamp(ceil(XResolution * CropLeft),
    0, XResolution);
int RasterCropRight =  (int)Clamp(ceil(XResolution * CropRight),
    0, XResolution);
int RasterCropBottom = (int)Clamp(ceil(YResolution * CropBottom),
    0, YResolution);
int RasterCropTop =    (int)Clamp(ceil(YResolution * CropTop),
    0, YResolution);
SampleCropLeft =   Clamp(RasterCropLeft   - scene->sampler->FilterXWidth, 0.,
    XResolution);
SampleCropRight =  Clamp(RasterCropRight  + scene->sampler->FilterXWidth, 0.,
    XResolution);
SampleCropBottom = Clamp(RasterCropBottom - scene->sampler->FilterYWidth, 0.,
    YResolution);
SampleCropTop =    Clamp(RasterCropTop    + scene->sampler->FilterYWidth, 0.,
    YResolution);
```

And now we can define the real `AddSample` method that processes camera samples in the `Image` class (see Section **??** on page ??.)

⟨*Image Method Definitions*⟩+≡
```
void Image::AddSample(const Point &Praster, const Spectrum &radiance,
            Float alpha) {
    ⟨Compute filter's raster extent⟩
    ⟨Loop over filter support and add sample to pixel arrays⟩
}
```

The first thing that we do is compute the bounds in raster-space of the pixels that will be affected by the sample. This is just half of the overall filter width in each direction from the sample locations, rounded up or down appropriately.

⟨*Compute filter's raster extent*⟩≡
```
Float HalfXWidth = scene->sampler->FilterXWidth / 2.;
Float HalfYWidth = scene->sampler->FilterYWidth / 2.;
int x0 = max((int)ceil (Praster.x - HalfXWidth), XBase);
int x1 = min((int)floor(Praster.x + HalfXWidth), XBase + XWidth);
int y0 = max((int)ceil (Praster.y - HalfYWidth), YBase);
int y1 = min((int)floor(Praster.y + HalfYWidth), YBase + YWidth);
```

Now, given the extent of pixels that are affected by this sample (`x0,y0` to `x1,y1`, inclusive), we loop over all of those pixels and then filter the sample value appropriately.

⟨*Loop over filter support and add sample to pixel arrays*⟩≡
```
for (int y = y0; y < y1; ++y)
    for (int x = x0; x < x1; ++x) {
        ⟨Compute filter deltas⟩
        ⟨Compute filter value and update arrays⟩
    }
```

⟨*Compute filter deltas*⟩≡
```
Float fx = x - Praster.x;
Float fy = y - Praster.y;
```

Now we go ahead can compute the filter value and apply the weighting to the pixel.

⟨*Compute filter value and update arrays*⟩≡

```
Float filterWt = scene->sampler->Filter(fx, fy, XWidth, YWidth);
int offset = (y - YBase) * XWidth + (x - XBase);
if (Pixels) Pixels[offset] += radiance * filterWt;
if (Alphas) Alphas[offset] += alpha * filterWt;
if (Depths) Depths[offset] += Praster.z * filterWt;
WeightSums[offset] += filterWt;
```

8. Color and Radiometry

This chapter describes the `Spectrum` class which `lrt` uses to represent spectra (i.e. colors) throughought the system. We will also the introduce basic concepts of radiometry and some theory behind light scattering from surfaces necessary to understand how `lrt` represents light scattering at surfaces.

8.1 The Spectrum Class

We just use a simple three-component representation of colors. This is a terrible idea for many reasons. In general, a color spectrum is a continuous spectral distribution defined over the wavelengths of visible light. This may be an arbitrarily complex funciton, so representing it with three samples is a bad idea. However, much of computer graphics is wedded to the fact that images are usually displayed on monitors that have red, green, and blue phosphors which are activated in varying intensities to approximate various spectral distributions.

Our color class, then, pretends to store three samples of the spectral distribution. They happen to be taken at the R, G, and B values of whichever display device is being used.

⟨*Spectrum Constructor Declarations*⟩≡
```
Spectrum() { s[0] = s[1] = s[2] = 0.; }
Spectrum(Float i) { s[0] = s[1] = s[2] = i; }
Spectrum(Float r, Float g, Float b) { s[0] = r; s[1] = g; s[2] = b; }
```

⟨*Spectrum Private Data*⟩≡
```
Float s[3];
```

A variety of arithmetic operations on `Spectrum` objects are supported; the implementations are all quite straightforward. First are operations to add two spectra together.

⟨*Spectrum Method Declarations*⟩+≡
```
Spectrum &operator+=(const Spectrum &s2) {
    s[0] += s2.s[0]; s[1] += s2.s[1]; s[2] += s2.s[2];
    return *this;
}
Spectrum operator+(const Spectrum &s2) {
    return Spectrum(s[0] + s2.s[0], s[1] + s2.s[1], s[2] + s2.s[2]);
}
Spectrum operator-(const Spectrum &s2) {
    return Spectrum(s[0] - s2.s[0], s[1] - s2.s[1], s[2] - s2.s[2]);
}
```

Similarly, multiplication and division of spectra is defined component-wise as well.

⟨*Spectrum Method Declarations*⟩+≡
```
Spectrum operator*(const Spectrum &s2) {
    return Spectrum(s[0] * s2.s[0], s[1] * s2.s[1], s[2] * s2.s[2]);
}
Spectrum operator/(const Spectrum &s2) {
    return Spectrum(s[0] / s2.s[0], s[1] / s2.s[1], s[2] / s2.s[2]);
}
Spectrum operator*(const Spectrum &sp) const {
    return Spectrum(s[0] * sp.s[0], s[1] * sp.s[1], s[2] * sp.s[2]);
}
Spectrum &operator*=(const Spectrum &sp) {
    s[0] *= sp.s[0]; s[1] *= sp.s[1]; s[2] *= sp.s[2];
    return *this;
}
```

We'll define various scaling operations that let `Float`s change the brightness of `Spectrum`s. In the interests of efficiency, we provide them separately here, rather than letting the `Spectrum(Float)` constructor above create a temporary `Spectrum`.

⟨*Spectrum Method Declarations*⟩+≡
```
Spectrum operator*(Float a) const { return Spectrum(s[0]*a, s[1]*a,s[2]*a); }
Spectrum &operator*=(Float a) { s[0] *= a; s[1] *= a; s[2] *= a; return *this; }
friend inline Spectrum operator*(Float a, const Spectrum &s) {
    return Spectrum(s.s[0] * a, s.s[1] * a, s.s[2] * a);
}
Spectrum operator/(Float a) const {
    Float inv = 1. / a;
    return Spectrum(s[0] * inv, s[1] * inv, s[2] * inv);
}
Spectrum &operator/=(Float a) {
    Float inv = 1. / a;
    s[0] *= inv; s[1] *= inv; s[2] *= inv;
    return *this;
}
```

Finally, the obvious equality test.

⟨*Spectrum Method Declarations*⟩+≡
```
bool operator==(const Spectrum &sp) const {
    return (s[0] == sp.s[0] && s[1] == sp.s[1] && s[2] == sp.s[2]);
}
```

Also useful a function that raises the components of a `Spectrum` to a given power, also given as a `Spectrum`.

⟨*Spectrum Method Declarations*⟩+≡
```
Spectrum Pow(const Spectrum &s2) const {
    return Spectrum(pow(max((Float)0., s[0]), s2.s[0]),
           pow(max((Float)0., s[1]), s2.s[1]),
           pow(max((Float)0., s[2]), s2.s[2]));
}
```

And a function that converts to RGB samples for the current display device. With our weak `Spectrum` representation, this is just a no-op.

⟨*Spectrum Method Declarations*⟩+≡
```
void ConvertToRGB(Float *result) const;
```

⟨*Spectrum Method Definitions*⟩+≡
```
void Spectrum::ConvertToRGB(Float *result) const {
    result[0] = s[0]; result[1] = s[1]; result[2] = s[2];
}
```

9.Shading

The low-level BRDFs introduced in Chapter 11 solve only part of the problem of describing how a surface scatters light. In particular, they describe how light is scattered at a particular point on the surface, but we still need to know *which* scattering function (or combination of scattering functions) describes the scattering at a point, and what the parameters to that scattering function are.

In this chapter, we provide a general procedural shading mechanism to do all of the above. 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 appropriate `BRDF`s that describe 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 13) into a single shader. By separating these two pieces, a more flexible system results that is better able to handle new light transport algorithms.

9.1 Surface Functions

Each geometric primitive and each light or surface shader in the scene can have a variety of types of user-supplied data associated with it–the most common example for primitives is shading normals associated with a polygonal object that reduce its faceted appearance. The user might also want to specify arbitrary additional information to be used in shading, like the color of an object, the filename of a texturemap to use, etc. In general, `lrt` allows the user to declare arbitrary variables of a variety of types and bind them to primitives and shaders. These variables represent the values of functions defined over the surface of that primitive. These functions are then used by the shading system to help compute surface properties at intersection points. Functions that don't have constant values over the surface are interpolated from values specified at the object's vertices; as ray intersections are found, primitives interpolate the values of these functions over their surfaces, computing particular values at the intersection point.

All of this data is stored and managed by the `SurfaceFunction` class. The user can

supply variables in a variety of types: floating point values, points, vectors, normals, spectra, and strings. Each type has one of two storage classes:[1]

- uniform. Uniform data has a single value for the entire primitive. (All string data must be `uniform`, since there is no sensible way to interpolate between multiple strings.)
- vertex. There is one vertex data item per vertex of the primitive. For primitives that don't have vertices (e.g. spheres), there are four "vertex" values which are interpolated parametrically based on the (u, v) value at the hit position.

When a `SurfaceFunction` object is created, the number of vertices that the primitive has must be given. If this data is going to be bound to a shader or light source, then `vertex` data is not allowed and this parameter should have the value zero. This makes it possible for us to make sure that variables of storage class `vertex` are provided in appropriate quantities.

⟨*SurfaceFunction Constructor Declarations*⟩≡
```
SurfaceFunction(u_int nv) { nVertex = nv; }
```

⟨*SurfaceFunction Data*⟩≡
```
u_int nVertex;
```

We will be allocating space to store the various types of data that are passed to the `SurfaceFunction` object, so we will define a destructor with a fragment, ⟨*Free SurfaceFunction Memory*⟩ to which we will add code to free this memory as needed.

⟨*SurfaceFunction Method Declarations*⟩≡
```
~SurfaceFunction();
```

⟨*SurfaceFunction Function Definitions*⟩≡
```
SurfaceFunction::~SurfaceFunction() {
    ⟨Free SurfaceFunction Memory⟩
}
```

Finally, we will need to provide a copy constructor for `SurfaceFunction`s. This is needed so that primitives that refine themselves can provide a fresh copy of their `SurfaceFunction` to all their children, since `SurfaceFunction`s are not reference counted. We will fill in this constructor as elements that need to be copied are introduced. Some of you may remember the `SurfaceFunction` copy constructor disaster from the first assignment; this is the fix.

⟨*SurfaceFunction Constructor Declarations*⟩+≡
```
SurfaceFunction(const SurfaceFunction &sf);
```

⟨*SurfaceFunction Function Definitions*⟩+≡
```
SurfaceFunction::SurfaceFunction( const SurfaceFunction &sf ) {
    nVertex = sf.nVertex;
    ⟨SurfaceFunction Copy Constructor⟩
}
```

[1] There are a few differences between the types and storage classes that we support compared to what the RenderMan specification expects. We do not have the matrix type–this is easily implemented, but just doesn't add very much to the exposition in this section. We also do not support user-supplied arrays data. More significantly, there are four possible storage classes that RI defines: constant, uniform, varying, and vertex. The most useful of these two are constant–having a single value over the entire object, and vertex–defined at the vertices of the object. Uniform and varying are of limited use in practice. Therefore, we only support constant and vertex types, though we rename constant to be uniform.

Adding variables and data.

We will now provide methods for binding and accessing `uniform` and `vertex point` data. The code for handling the other types (`Float`s, `Normal`s, etc.) is analogous. We will use `RtToken`s to store the names of the various variables here; see Section C.2 on page 204 for their implementation. For now, consider `RtToken`s to just be zero terminated strings of type `const char *` with the special property that we can test two strings for equality directly by comparing them with the == operator instead of needing to call `strcmp`.

Uniform data is easy to handle; we keep a `vector` that holds pairs of `RtToken`s and their associated values and add new incoming items to the list as they arrive. All of the data that comes to us will be a `Float` pointer, which points at the start of the data. We'll need to dereference the pointer as needed to get the expected data items. (It is the caller's responsibility to make sure that the pointer is valid and points to the appropriate amount of data.)

⟨*SurfaceFunction Method Declarations*⟩+≡
```
void AddUniformPoint(RtToken name, const Float *data);
```

⟨*SurfaceFunction Function Definitions*⟩+≡
```
void SurfaceFunction::AddUniformPoint( RtToken name,
                                       const Float *data) {
    uniformPoints.push_back(
        make_pair(name, Point(data[0], data[1], data[2])));
}
```

⟨*SurfaceFunction Data*⟩+≡
```
vector<pair<RtToken, Point> > uniformPoints;
```

⟨*SurfaceFunction Copy Constructor*⟩≡
```
vector<pair<RtToken, Point> >::const_iterator uniformPointIter;
for (uniformPointIter = sf.uniformPoints.begin();
     uniformPointIter != sf.uniformPoints.end();
     uniformPointIter++) {
    uniformPoints.push_back(
        make_pair( (*uniformPointIter).first,
                   (*uniformPointIter).second ) );
}
```

Per-vertex data is only slightly more complicated. We keep a vector of `RtToken`s and pointers to `point` data; we'll dynamically allocate an appropriate amount of space for each the incoming data item and then copy it there.

⟨*SurfaceFunction Method Declarations*⟩+≡
```
void AddVertexPoint(RtToken name, const Float *data);
```

This method can only be called when the `SurfaceFunction` is bound to a primitive and thus there are vertices to bind data to; we start by asserting that this is the case before moving on.

⟨*SurfaceFunction Function Definitions*⟩+≡
```
void SurfaceFunction::AddVertexPoint(RtToken name,
                                     const Float *data) {
    Assert(nVertex > 0);
    Point *values = new Point[nVertex];
    for (u_int i = 0; i < nVertex; ++i) {
        values[i].x = data[3*i];
        values[i].y = data[3*i+1];
        values[i].z = data[3*i+2];
    }
    vertexPoints.push_back(make_pair(name, values));
}
```

⟨*SurfaceFunction Data*⟩+≡
```
vector<pair<RtToken, Point *> > vertexPoints;
```

⟨*SurfaceFunction Copy Constructor*⟩+≡
```
vector<pair<RtToken, Point *> >::const_iterator vertexPointIter;
for (vertexPointIter = sf.vertexPoints.begin();
     vertexPointIter != sf.vertexPoints.end();
     vertexPointIter++) {
    Point *values = new Point[nVertex];
    memcpy( values,
            (*vertexPointIter).second,
            nVertex*sizeof(*values) );
    vertexPoints.push_back(
        make_pair( (*vertexPointIter).first, values ) );
}
```

Now that we've written some code that allocates memory to store the data, we add a bit to the fragment in the destructor to iterate through all of our `vertex point` data and free that memory.

⟨*Free SurfaceFunction Memory*⟩≡
```
for (u_int i = 0; i < vertexPoints.size(); ++i)
    delete[] vertexPoints[i].second;
```

Now we include functions to add uniform values. The declarations are shown here, but the implementations are tedious and omitted.

⟨*SurfaceFunction Method Declarations*⟩+≡
```
void AddUniformFloat(RtToken name, const Float *data);
void AddUniformHPoint(RtToken name, const Float *data);
void AddUniformVector(RtToken name, const Float *data);
void AddUniformNormal(RtToken name, const Float *data);
void AddUniformColor(RtToken name, const Float *data);
void AddUniformString(RtToken name, const char *data);
```

Looking up data.

Looking up a particular named uniform value is easy; the token with its name is passed in and we check all of our `uniform` variables of the appropriate type and return `NULL` if the item is not found. If it is found, we return a pointer to the value. This pointer is guaranteed to be valid for the lifetime of the `SurfaceFunction` object. The methods for `uniform` variable types other than `point` are analogous and are omitted here.

⟨*SurfaceFunction Method Declarations*⟩+≡
```
const Point *GetUniformPoint(RtToken name) const;
```

⟨*SurfaceFunction Function Definitions*⟩+≡
```
const Point *SurfaceFunction::GetUniformPoint(RtToken name) const {
    for (u_int i = 0; i < uniformPoints.size(); ++i) {
        if (name == uniformPoints[i].first)
            return &uniformPoints[i].second;
    }
    return NULL;
}
```

Similarly, we can look for a pointer to a named bit of `vertex` data: the caller passes in a `RtToken` and a pointer to a `Point` is returned if that item is found. As with `uniform` data, we guarantee that the pointer will remain valid as long as the `SurfaceFunction` object is valid; this way other code can store these pointers away and use to them as needed, avoiding the overhead of repeated calls to `GetVertexPoint` and searches through the data.

⟨*SurfaceFunction Method Declarations*⟩+≡
```
const Point *GetVertexPoint(RtToken name) const;
```

⟨*SurfaceFunction Function Definitions*⟩+≡
```
const Point *SurfaceFunction::GetVertexPoint(RtToken name) const {
    for (u_int i = 0; i < vertexPoints.size(); ++i) {
        if (name == vertexPoints[i].first)
            return vertexPoints[i].second;
    }
    return NULL;
}
```

When looking for data from `SurfaceFunction`, we often do not care if it was found or not — there is a default value for something and want to override it with a `uniform` value from a `SurfaceFunction` if available. These methods take a token and a default value, either returning a new value if it's available or returning the default otherwise.

⟨*SurfaceFunction Method Declarations*⟩+≡
```
Float InitializeFloat(RtToken name, Float def) const;
Point InitializePoint(RtToken name, const Point &def) const;
Vector InitializeVector(RtToken name, const Vector &def) const;
Normal InitializeNormal(RtToken name, const Normal &def) const;
Spectrum InitializeColor(RtToken name, const Spectrum &def) const;
const char *InitializeString(RtToken name, const char *def) const;
```

We only show the `point` version here; the others are similar.

⟨*SurfaceFunction Function Definitions*⟩+≡
```
Point SurfaceFunction::InitializePoint(RtToken name,
                                       const Point &def) const {
    const Point *ptr = GetUniformPoint(name);
    if (ptr != NULL)
        return *ptr;
    else
        return def;
}
```

Interpolating Values at Intersection Points.

When we find intersection with rays and primitives, we'll make an `InterpolatedPrimData` object that holds the interpolated values of the `vertex` data at the hit point as well as a reference to the `uniform` data. This object can later by used by the shading system to look up values for shading computation.

We will store an `InterpolatedPrimData` structure in each `HitInfo`.

⟨*HitInfo Data*⟩+≡
```
InterpolatedPrimData *interpolatedData;
```

⟨*HitInfo Constructor Definition*⟩≡
```
interpolatedData = NULL;
```

⟨*HitInfo Destructor Definition*⟩≡
```
delete interpolatedData;
```

⟨*HitInfo Method Definitions*⟩+≡
```
void HitInfo::SetInterpolatedData(InterpolatedPrimData *id) {
    delete interpolatedData;
    interpolatedData = id;
}
```

Each intersection routine will compute the inverse mapping of an intersection point, and use that to interpolate any user supplied surface functions.

The `InterpolatedPrimData` constructor just stores a pointer to its parent `SurfaceFunction` (for looking up `uniform` data) and then allocates space for the various `vertex` data variables which the geometric primitive should then fill in.

⟨*InterpolatedPrimData Method Declarations*⟩≡
```
InterpolatedPrimData(const SurfaceFunction *pd);
```

⟨*SurfaceFunction Function Definitions*⟩+≡
```
InterpolatedPrimData::InterpolatedPrimData(
        const SurfaceFunction *pd) {
    primitiveData = pd;
    ⟨Allocate space for vertex values⟩
}
```

⟨*InterpolatedPrimData Private Data*⟩≡
```
const SurfaceFunction *primitiveData;
```

We'll need access to the `private` variables of `SurfaceFunction` so that we can just grab `uniform` data variables from there.

⟨*SurfaceFunction Method Declarations*⟩+≡
```
friend class InterpolatedPrimData;
```

We allocate space for one `Point` per `vertex point` variable. Equivalent code for the other types of `vertex` data is similar and is omitted here.

⟨*Allocate space for* `vertex` *values*⟩≡
```
if (primitiveData->vertexPoints.size() > 0) {
    int nvp = primitiveData->vertexPoints.size();
    interpVertexPoints = new Point[nvp];
} else
    interpVertexPoints = NULL;
```

⟨*InterpolatedPrimData Private Data*⟩+≡
```
Point *interpVertexPoints;
```

And in the destructor we need to free up the space that we allocated when we created the `InterpolatedPrimData`.

⟨*InterpolatedPrimData Method Declarations*⟩+≡
```
~InterpolatedPrimData();
```

⟨*SurfaceFunction Function Definitions*⟩+≡
```
InterpolatedPrimData::~InterpolatedPrimData() {
    ⟨Free interpolated vertex value space⟩
}
```

⟨*Free interpolated* `vertex` *value space*⟩≡
```
delete[] interpVertexPoints;
```

`Interpolate` is a method of `SurfaceFunction` that is called by a primitive after an intersection has been found. It creates a new `InterpolatedPrimData` object which is returned. The user should pass in an array of integer offsets into the vertex data arrays and an array of weights associated with each offset. For each of the `vertex` data types, we'll compute an interpolated data value by computing a weighted sum of the relevant items from the vertex arrays weighted by the weights and store it in the `InterpolatedPrimData` object.

⟨*SurfaceFunction Method Declarations*⟩+≡
```
InterpolatedPrimData *Interpolate(int n, const int *offsets,
    const Float *weights) const;
```

For example, when we find the intersection between a ray and a triangle (see 3.3 on page 45), we can use the barycentric coordinates (u, v) to evaluate the surface function at any point on the triangle. Recall that any point $\mathbf{p}$ on the triangle can be expressed as a weighted sum of the three vertices of a triangle:

$$T(u,v) = (1-u-v)V_o + uV_1 + vV_2$$

⟨*Fill in* `HitInfo` *from triangle hit*⟩+≡
```
int interpOffsets[3] = { vertexIndex0, vertexIndex1, vertexIndex2 };
Float interpWeights[3] = { 1. - u - v, u, v };
InterpolatedPrimData *interpolatedData;
interpolatedData =
    mesh->surfaceFunction->Interpolate(3,
                                       interpOffsets,
                                       interpWeights);
hit->SetInterpolatedData(interpolatedData);
```

Similarly, when we intersect a more general surface, we use the inverse mapping coordinates to bilinearly interpolate four values:

⟨*Interpolate four vertex points*⟩≡
```
int interpOffsets[4] = { 0, 1, 2, 3 };
Float interpWeights[4] = { u*v, (1.-u)*v, u*(1.-v), (1.-u)*(1.-v) };
InterpolatedPrimData *interpolatedData;
interpolatedData = surfaceFunction->Interpolate(4,
                                                interpOffsets,
                                                interpWeights);
```

⟨*Fill in* `HitInfo` *from disk hit*⟩+≡
```
⟨Interpolate four vertex points⟩
hit->SetInterpolatedData(interpolatedData);
```

⟨*Fill in* `HitInfo` *from sphere hit*⟩+≡
```
⟨Interpolate four vertex points⟩
hit->SetInterpolatedData(interpolatedData);
```

⟨*SurfaceFunction Function Definitions*⟩+≡
```
InterpolatedPrimData *SurfaceFunction::Interpolate(int n,
        const int *offsets, const Float *weights) const {
    InterpolatedPrimData *ret = new InterpolatedPrimData(this);
    ⟨Interpolate Point data⟩
    ⟨Interpolate other data types⟩
    return ret;
}
```

To perform interpolations for the user, we just loop through all of the point data types (and later the other types). For each one, we compute a weighted sum of each weight times the point value at the appropriate offset.

⟨*Interpolate* `Point` *data*⟩≡
```
for (u_int i = 0; i < vertexPoints.size(); ++i) {
    Point value(0, 0, 0);
    Point *pts = vertexPoints[i].second;
    for (int j = 0; j < n; ++j) {
        int o = offsets[j];
        value.x += weights[j] * pts[o].x;
        value.y += weights[j] * pts[o].y;
        value.z += weights[j] * pts[o].z;
    }
    ret->interpVertexPoints[i] = value;
}
```

The `InterpolatedPrimData` needs to allow `SurfaceFunction` to write into it so that it can set the values of `interpVertexPoints`, etc.

⟨*InterpolatedPrimData Method Declarations*⟩+≡
```
friend class SurfaceFunction;
```

Looking Up Values at Intersection Points.

Now come the methods to look up the values of the user-supplied data at the hit point. `GetPoint` is illustrative of the rest of the methods for the other types. Note that for each method we provide an identical method in `HitInfo` that checks to make sure there is in fact interpolated data. The selectors in `InterpolatedPrimData` are made `private` to make sure that the user writing a shader does not try to access the values directly. .

⟨*InterpolatedPrimData Private Method Declarations*⟩≡
```
friend class HitInfo;
```

⟨*InterpolatedPrimData Private Method Declarations*⟩+≡
```
const Point *GetPoint(RtToken token) const;
```

⟨*HitInfo Method Definitions*⟩+≡
```
const Point *HitInfo::GetPoint(RtToken token) const {
    if (token == RI_P) return &Pobj;
    if (!interpolatedData) return NULL;
    return interpolatedData->GetPoint(token);
}
```

First we check for a `uniform point` with the name in `token`. If there is one, a pointer to its value is returned from our parent `SurfaceFunction` object. Otherwise we need to look through the `vertex` data.

⟨*SurfaceFunction Function Definitions*⟩+≡
```
const Point *InterpolatedPrimData::GetPoint(RtToken token) const {
    const Point *ret = primitiveData->GetUniformPoint(token);
    if (ret) return ret;
    ⟨Search for token in interpolated vertex values⟩
}
```

To look up a given token, we look through the `vertex points` in `SurfaceFunction` once we find it, we use that same offset into `interpVertexPoints` to return a pointer to the data where it's stored here.

⟨*Search for* `token` *in interpolated* `vertex` *values*⟩≡
```
const vector<pair<RtToken, Point *> > &vpp =
    primitiveData->vertexPoints;
for (u_int i = 0; i < vpp.size(); ++i) {
    if (token == vpp[i].first)
        return &interpVertexPoints[i];
}
return NULL;
```

9.2 Attributes for Shading

Primitive Attributes.

We start by adding some fields to `PrimitiveAttributes`. Each object has an intrinsic basic color and opacity associated with it. These can be accessed in the object's surface shader and used however the shader wants to (or not used at all, depending on the shader). Furthermore, the user can bind values to the `vertex colors` `RI_CS` and `RI_OS`, stored in the primitive's `SurfaceFunction` to override these values.

⟨*PrimitiveAttributes Public Data*⟩+≡
```
Spectrum Color, Opacity;
```

⟨*PrimitiveAttributes constructor implementation*⟩+≡
```
Color = Opacity = Spectrum(1.0);
```

This `Material` will be one of the ones defined in this chapter.

⟨*PrimitiveAttributes Public Data*⟩+≡
```
Material *Surface;
```

⟨*PrimitiveAttributes constructor implementation*⟩+≡
```
Surface = NULL;
```

9.3 Shading Context

Before we get into the `Material` classes, we'll define a helper class called `ShadeContext` This class encapsulates a variety of sources of information and puts them behind simple interface. Surface shaders then can just call out to this class to access the information they need to compute their results. After `Integrators` in Chapter 13 find the closest intersection of a ray with the surfaces in the scene, they create a `ShadeContext` by passing in the `HitInfo` at the intersection point and `wo`, the outgoing direction from which the ray arrived at the point being shaded. `wo` should be equal to the negative direction of the ray that intersected the hit point.

⟨*ShadeContext Interface*⟩≡
```
ShadeContext(const HitInfo *, const Vector &wo);
```

⟨*ShadeContext Method Definitions*⟩≡
```
ShadeContext::ShadeContext(const HitInfo *hi, const Vector &w)
        : wo(w.Hat()) {
    hitInfo = hi;
    ⟨Compute geometric normal⟩
    ⟨Compute shading normal⟩
}
```

We leave the incident direction public for easy access, but keep it constant.

⟨*ShadeContext Public Data*⟩≡
```
const Vector wo;
```

⟨*ShadeContext Private Data*⟩≡
```
const HitInfo *hitInfo;
```

The first thing to do is to compute the surface normal at the hit point in world space. We use the transformation supplied in the `PrimitiveAttributes` of the object that we hit, normalizing the result.

We then flip the normal if needed so that it points to the same side of the surface as `wo`. A fast test for if they're pointing on opposite sides of the surface is to examine the sign of their dot product. Solid primitives are supposed to compute surface normals that point to outwards from their interiors, so we can also use the result of this dot product to determine if we're entering or leaving the primitive at the ray intersection point.

⟨*Compute geometric normal*⟩≡
```
Ng = hitInfo->hitPrim->attributes->ObjectToWorld(hitInfo->NgObj);
Ng.Normalize();
if (Dot(Ng, wo) < 0.) {
    Ng = -Ng;
    entering = false;
}
else {
    entering = true;
}
```

⟨*ShadeContext Public Data*⟩+≡
```
Normal Ns, Ng;
bool entering;
```

We also compute a *shading normal* at the hit point. The user may have supplied surface normals at the vertices of a triangle mesh, for example; by interpolating these normals across the faces of the triangles, a smooth-appearing surface results when we shade. We look for an entry called `RI_N` in the interpolated `SurfaceFunction` at the hit point and use that to set `Ns` to the shading normal. Otherwise `Ns` just takes on the value of `Ng`.

⟨*Compute shading normal*⟩≡
```
const Normal *Ni = hitInfo->GetNormal(RI_N);
if (Ni != NULL) {
    Ns = hitInfo->hitPrim->attributes->ObjectToWorld(*Ni).Hat();
    if (Dot(Ns, wo) < 0.)
        Ns = -Ns;
}
else {
    Ns = Ng;
}
```

Shaders all take a set of parameters that describe their operation (e.g. the `PaintedPlastic` shader below takes a string parameter called `texturename` which gives the name of a texture map to use for computing surface colors.) These parameters may come from one of three sources: first, when a shader is written, default values are given to all of its parameters; second, when the shader is first created, values that override the defaults may be supplied; third, when the shader is executed at a specific point, values that have been bound to the actual surface may override the values from the previous two steps.

Here we provide helper functions for the shaders that lets them do the third step of parameter initialization above: when they're running at a particular point, they can call `InitializeFloat` or whatever, passing in `def`, the value that they're planning on using for that particular parameter if the primitive isn't going to override it. These functions look up the value from the primitive if possible and otherwise return the appropriate default.

⟨*ShadeContext Interface*⟩+≡
```
Float InitializeFloat(RtToken token, Float def) const;
Point InitializePoint(RtToken token, const Point &def) const;
Vector InitializeVector(RtToken token, const Vector &def) const;
Normal InitializeNormal(RtToken token, const Normal &def) const;
Spectrum InitializeColor(RtToken token, const Spectrum &def) const;
const char *InitializeString(RtToken token, const char *def) const;
```

Both `InitializeFloat` and `InitializeColor` have some weirdness, since certain parameters can be stored in two places. Here, for example, the user may have specified their own *s* and *t* coordinates for texture mapping as `RI_S` and `RI_T` `vertex float`s. If they haven't done so, then we should just return the *u* and *v* parametric coordinates that the `Primitive` computed, respectively.

⟨*ShadeContext Method Definitions*⟩+≡
```
Float ShadeContext::InitializeFloat(RtToken token, Float def) const {
    const Float *ret = hitInfo->GetFloat(token);
    if (ret)                return *ret;
    if (token == RI_S)      return hitInfo->u;
    else if (token == RI_T) return hitInfo->v;
    return def;
}
```

Similarly, if the user didn't specify per-vertex color and opacity values, we should return the values from the primitive's `PrimitiveAttributes`.

⟨*ShadeContext Method Definitions*⟩+≡
```
Spectrum ShadeContext::InitializeColor(
        RtToken token, const Spectrum &def) const {
    const Spectrum *ret = hitInfo->GetColor(token);
    if (ret)
        return *ret;
    if (token == RI_CS)
        return hitInfo->hitPrim->attributes->Color;
    else if (token == RI_OS)
        return hitInfo->hitPrim->attributes->Opacity;
    return def;
}
```

Finally, the method for accessing `Point` values is pretty vanilla. The methods for strings, vectors, and normals are analogous and are omitted.

⟨*ShadeContext Method Definitions*⟩+≡
```
Point ShadeContext::InitializePoint(
        RtToken token, const Point &def) const {
    const Point *ret = hitInfo->GetPoint(token);
    if (ret) return *ret;
    else     return def;
}
```

9.4 Basic Shaders

Now on to defining the interface for shaders. All surface shaders inherit from `Material`. This takes a `SurfaceFunction` object that should only have `uniform` values bound to it, which is stored away for future use by the particular shader.

⟨*Material Method Declarations*⟩≡
```
Material(SurfaceFunction *ud)
    : surfaceFunction(ud) {
}
```

⟨*Material Protected Data*⟩≡
```
SurfaceFunction *surfaceFunction;
```

⟨*Material Method Declarations*⟩+≡
```
virtual ~Material();
```

⟨*Shading Function Definitions*⟩≡
```
Material::~Material() {
    delete surfaceFunction;
}
```

And this is the interface that surface shaders must provide. A `ShadeContext` (see Section 9.3) is passed in, and a `BRDF` is returned.

⟨*Material Method Declarations*⟩+≡
```
virtual BRDF *Shade(const ShadeContext &sc) const = 0;
```

Matte.

The simplest surface is `Matte`. It describes a diffusely-reflecting surface with a constant reflectivity. `Kd` stores the overall reflectivity of the surface, though in the actual shading function, this is used to scale the color that was computed at the shading point.

⟨*MatteSurface Interface*⟩≡
```
MatteSurface(SurfaceFunction *data)
        : Material(data) {
    Kd = surfaceFunction->InitializeFloat(RI_KD, 1.0);
}
```

⟨*MatteSurface Private Data*⟩≡
```
Float Kd;
```

⟨*MatteSurface Interface*⟩+≡
```
BRDF *Shade(const ShadeContext &sc) const;
```

In the actual shading function, we just look up the current color at the hit point by asking for `RI_CS` (the name that the RenderMan interface gives for the base color at the point being shaded) from our `ShadeContext` and we then check to see if the primitive is going to override the `Kd` value we saved away in our constructor. In any case, we compute the product of these two terms and compute a new `Lambertian` scattering function with the resulting parameters.

⟨*Shading Function Definitions*⟩+≡
```
BRDF *MatteSurface::Shade(const ShadeContext &sc) const {
    Spectrum Cs = sc.InitializeColor(RI_CS, Spectrum(1.0));
    Float surfaceKd = sc.InitializeFloat(RI_KD, Kd);
    return CreateLambertian(surfaceKd * Cs, sc.Ns);
}
```

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 definition, Here we will first define a basic plastic shader and then define one that uses a texture map to choose the surface color.

The parameters to `PlasticSurface` are two reflectivities, `Kd` and `Ks`, which control how much diffuse reflection there is and how much glossy specular reflection there is. Next is a roughness parameter which should range from zero to one that determines the size of the specular highlight; the higher it is, the rougher the surface and the smaller the highlight. Last is a color for the specular highlight, stored in `SpecularColor`.

⟨*PlasticSurface Interface*⟩≡
```
PlasticSurface(SurfaceFunction *data)
        : Material(data) {
    Kd = surfaceFunction->InitializeFloat(RI_KD, .5);
    Ks = surfaceFunction->InitializeFloat(RI_KS, .5);
    Roughness = surfaceFunction->InitializeFloat(RI_ROUGHNESS, .1);
    SpecularColor =
        surfaceFunction->InitializeColor(
            RI_SPECULARCOLOR, Spectrum(1));
}
```

⟨*PlasticSurface Protected Data*⟩≡
```
Float Kd, Ks, Roughness;
Spectrum SpecularColor;
```

We split most of the work of the plastic shader into a separate `PlasticShade` function which just takes a `ShadeContext` and the surface's color. When we implement the plastic shader with texture maps below, it will be able to reuse this function directly.

⟨*PlasticSurface Interface*⟩+≡
```
BRDF *Shade(const ShadeContext &sc) const;
BRDF *PlasticShade(const ShadeContext &sc, const Spectrum &Cs) const;
```

The normal plastic shading function is quite simple, then, just looking up the surface color from the `ShadeContext` and handing that on to the `PlasticShade` function.

⟨*Shading Function Definitions*⟩+≡
```
BRDF *PlasticSurface::Shade(const ShadeContext &sc) const {
    Spectrum Cs = sc.InitializeColor(RI_CS, Spectrum(1.0));
    return PlasticShade(sc, Cs);
}
```

Here we do most of the work. We first override parameters values with values from the shading point and then compute a new `ScatteringMixture` object (see Section 11.2 on page 146) with both diffuse and glossy components.

⟨*Shading Function Definitions*⟩+≡
```
BRDF *PlasticSurface::PlasticShade(const ShadeContext &sc,
        const Spectrum &Cs) const {
    ⟨Possibly override plastic parameters with values from primitive⟩
    ⟨Create new plastic ScatteringMixture⟩
}
```

⟨*Possibly override plastic parameters with values from primitive*⟩≡
```
Float surfKd = sc.InitializeFloat(RI_KD, Kd);
Float surfKs = sc.InitializeFloat(RI_KS, Ks);
Float surfRoughness =
    sc.InitializeFloat(RI_ROUGHNESS, Roughness);
Spectrum surfSpecColor =
    sc.InitializeColor(RI_SPECULARCOLOR, SpecularColor);
```

⟨*Create new plastic* `ScatteringMixture`⟩≡
```
ScatteringMixture *mix = CreateScatteringMixture();
mix->AddFunction(CreateLambertian(surfKd * Cs, sc.Ns));
mix->AddFunction(CreateBlinnGlossy(surfKs * surfSpecColor,
    Clamp(surfRoughness, 0., 1.), sc.Ns, sc.wo));
return mix;
```

Painted Plastic.

The `PaintedPlastic` surface just adds a texture map to the mix. If it can find a valid texture name in the `string texturename` parameter, it uses said texture map to compute the base surface color at the shading point.

⟨*PaintedPlasticSurface Interface*⟩≡
```
PaintedPlasticSurface(SurfaceFunction *data);
```

⟨*Shading Function Definitions*⟩+≡
```
PaintedPlasticSurface::PaintedPlasticSurface(SurfaceFunction *data)
        : PlasticSurface(data) {
    texName = surfaceFunction->InitializeString(RI_TEXTURENAME,
                                NULL);
}
```

⟨*PaintedPlasticSurface Private Data*⟩≡
```
const char *texName;
```

⟨*PaintedPlasticSurface Interface*⟩+≡
```
BRDF *Shade(const ShadeContext &sc) const ;
```

As above, we use `RI_CS` as the default surface color. If something has been bound to `RI_TEXTURENAME`, however, we try to get a pointer to a texture map with that name from the `ShadeContext` If successful, we look up s and t texture coordinates and use them to lookup a color in the texture map (see Section 10.1 on page 137.) The result is passed to the `PlasticShade` function from above to compute a final scattering function.

⟨*Shading Function Definitions*⟩+≡
```
BRDF *PaintedPlasticSurface::Shade(
        const ShadeContext &sc) const {
    Spectrum surfColor = sc.InitializeColor(RI_CS, Spectrum(1.));
    const char *surfTexName =
        sc.InitializeString(RI_TEXTURENAME, texName);
    TextureMap *tmap = sc.GetTexture(surfTexName);

    if (tmap) {
        Float s = sc.InitializeFloat(RI_S, 0.);
        Float t = sc.InitializeFloat(RI_T, 0.);
        surfColor *= tmap->Lookup(s, t);
    }
    return PlasticShade(sc, surfColor);
}
```

Checkered.

Another simple shader is the classic infinite checkerboard. The user specifies the diffuse reflectance and two check colors as well as a floating point number that gives the frequency of switching between the two colors.

⟨*CheckeredSurface Interface*⟩≡
```
CheckeredSurface(SurfaceFunction *data)
        : Material(data) {
    Kd = surfaceFunction->InitializeFloat(RI_KD, 1.0);
    CheckColor1 =
        surfaceFunction->InitializeColor(LRT_CHECKCOLOR1,
                                Spectrum(1,0,0));
    CheckColor2 =
        surfaceFunction->InitializeColor(LRT_CHECKCOLOR2,
                                Spectrum(0,0,1));
    CheckFrequency =
        surfaceFunction->InitializeFloat(LRT_CHECKFREQUENCY,
                                1);
}
```

⟨*CheckeredSurface Private Data*⟩≡
```
Float Kd, CheckFrequency;
Spectrum CheckColor1, CheckColor2;
```

⟨*CheckeredSurface Interface*⟩+≡
```
BRDF *Shade(const ShadeContext &sc) const ;
```

⟨*Shading Function Definitions*⟩+≡
```
BRDF *CheckeredSurface::Shade(const ShadeContext &sc) const {
    ⟨Update checker parameters from surface values⟩
    ⟨Compute checker location and value⟩
}
```

When we shade, we start by doing the usual thing to override parameter values based on values from the primitive that we hit.

⟨*Update checker parameters from surface values*⟩≡
```
Float surfKd = sc.InitializeFloat(RI_KD, Kd);
Spectrum surfCColor1 = sc.InitializeColor(LRT_CHECKCOLOR1,
                                  CheckColor1);
Spectrum surfCColor2 = sc.InitializeColor(LRT_CHECKCOLOR2,
                                  CheckColor2);
Float freq = sc.InitializeFloat(LRT_CHECKFREQUENCY,
                                  CheckFrequency);
```

Now we take the coordinates of the point we're shading and scale them by the checker frequency. We round each of these to the nearest integer and then take modulus 2. This effectively partitions space into cells with value zero or one, where the immediate neighbor across each of the faces of the cell has a different value.

⟨*Compute checker location and value*⟩≡
```
Point Pshade = sc.InitializePoint(RI_P, Point(0,0,0));
Point Pcheck(freq * Pshade.x, freq * Pshade.y, freq * Pshade.z);
if (Mod(Round(Pcheck.x) +
    Round(Pcheck.y) +
    Round(Pcheck.z), 2) == 0) {
    return CreateLambertian(surfKd * surfCColor1, sc.Ns);
} else {
    return CreateLambertian(surfKd * surfCColor2, sc.Ns);
}
```

Glass.

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.

⟨*GlassSurface Interface*⟩≡
```
GlassSurface(SurfaceFunction *data)
        : Material(data) {
    Kr = surfaceFunction->InitializeFloat(RI_KR, .5);
    Kt = surfaceFunction->InitializeFloat(LRT_KT, .5);
    Index = surfaceFunction->InitializeFloat(LRT_INDEX, 1.5);
}
```

⟨*GlassSurface Private Data*⟩≡
```
Float Kr, Kt, Index;
```

⟨*GlassSurface Interface*⟩+≡
```
BRDF *Shade(const ShadeContext &sc) const;
```

As usual, we start by computing new parameters from the primitive's user-supplied values. We then generate a new `ScatteringMixture BRDF` and add reflective and transmissive `BRDFs` as appropriate given the parameter values.

⟨*Shading Function Definitions*⟩+≡
```
BRDF *GlassSurface::Shade(const ShadeContext &sc) const {
    ⟨Override glass parameters from primitive⟩
    if (shadeKr == 0. && shadeKt == 0.) return NULL;

    ScatteringMixture *mix = CreateScatteringMixture();
    if (shadeKr > 0.) {
        mix->AddFunction(CreateSpecularReflection(shadeKr * Cs,
            sc.Ns, sc.wo));
    }
    if (shadeKt > 0.) {
        ⟨Add appropriate transmission scattering function to mixture⟩
    }
    return mix;
}
```

⟨*Override glass parameters from primitive*⟩≡
```
Float shadeKr = sc.InitializeFloat(RI_KR, Kr);
Float shadeKt = sc.InitializeFloat(LRT_KT, Kt);
Float shadeIndex = sc.InitializeFloat(LRT_INDEX, Index);
Spectrum Cs = sc.InitializeColor(RI_CS, Spectrum(0.));
```

The only tricky bit is where we see if we're entering or exiting the solid object that the shader is bound to: depending on which way we're going, we pass the indices of refraction in an appropriate order so that we're going in the right direction from air (with an index of refraction of roughly 1.0) to the object (the given index of refraction.)

⟨*Add appropriate transmission scattering function to mixture*⟩≡
```
if (sc.entering) {
    mix->AddFunction(CreateSpecularTransmission(Kt * Cs,
            sc.Ns, sc.wo,
            1.0, shadeIndex));
} else {
    mix->AddFunction(CreateSpecularTransmission(Kt * Cs,
            sc.Ns, sc.wo,
            shadeIndex, 1.0));
}
```

ShinyMetal.

Another basic combination of scattering functions gives us something that looks like a shiny metal surface. We have both a glossy specular reflection, with reflectance `Ks`, and perfect mirror specular reflection, with reflectance `Kr`.

⟨*ShinyMetalSurface Interface*⟩≡
```
ShinyMetalSurface(SurfaceFunction *data)
        : Material(data) {
    Ks = surfaceFunction->InitializeFloat(RI_KS, 1.0);
    Kr = surfaceFunction->InitializeFloat(RI_KR, 1.0);
    Roughness = surfaceFunction->InitializeFloat(RI_ROUGHNESS, 0.1);
}
```

⟨*ShinyMetalSurface Private Data*⟩≡
```
Float Ks, Kr, Roughness;
```

⟨*ShinyMetalSurface Interface*⟩+≡
```
BRDF *Shade(const ShadeContext &sc) const ;
```

And the usual drill gives us a new mixture scattering function with the two specified components.

⟨*Shading Function Definitions*⟩+≡
```
BRDF *ShinyMetalSurface::Shade(const ShadeContext &sc) const {
    ⟨Override shiny metal parameters from primitive⟩
    ScatteringMixture *mix = CreateScatteringMixture();
    if (shadeKs > 0.) {
        mix->AddFunction(CreateBlinnGlossy(shadeKs * Cs,
                                           shadeRoughness,
                                           sc.Ns,
                                           sc.wo));
    }
    if (shadeKr > 0.)
    {
        mix->AddFunction(CreateSpecularReflection(shadeKr * Cs,
                                                  sc.Ns,
                                                  sc.wo));
    }
    return mix;
}
```

⟨*Override shiny metal parameters from primitive*⟩≡
```
Float shadeKs = sc.InitializeFloat(RI_KS, Ks);
Float shadeKr = sc.InitializeFloat(RI_KR, Kr);
Float shadeRoughness = sc.InitializeFloat(RI_ROUGHNESS, Roughness);
Spectrum Cs = sc.InitializeColor(RI_CS, Spectrum(1.0));
```

ViewST.

Finally, a simple surface that uses the *s* and *t* texture coordinates of the surface to compute the red and green channels of the color. A diffuse surface is returned. This is mostly just useful for debugging the parameterization of primitives.

⟨*ViewSTSurface Interface*⟩≡
```
ViewSTSurface(SurfaceFunction *data)
        : Material(data) {
}
```

⟨*ViewSTSurface Interface*⟩+≡
```
BRDF *Shade(const ShadeContext &sc) const;
```

⟨*Shading Function Definitions*⟩+≡
```
BRDF *ViewSTSurface::Shade(const ShadeContext &sc) const {
    Float sp = sc.InitializeFloat(RI_S, 0.);
    Float tp = sc.InitializeFloat(RI_T, 0.);
    return CreateLambertian(Spectrum(sp, tp, 0.), sc.Ns);
}
```

9.5 Shader Creation And Management

Finally, we provide a single externally-visible function that handles shader creation. The user passes in the name of a surface shader and a set of parameter values that were bound to it; this function handles allocating a new `Material` of the appropriate type and returning it.

⟨*Global Function Declarations*⟩+≡
```
extern Material *MaterialCreate(const char *name,
    SurfaceFunction *bindings);
```

⟨*Shading Function Definitions*⟩+≡
```
Material *MaterialCreate(const char *name,
                         SurfaceFunction *bindings) {
    if (strcmp(name, RI_MATTE) == 0)
        return new MatteSurface(bindings);
    else if (strcmp(name, RI_PLASTIC) == 0)
        return new PlasticSurface(bindings);
    else if (strcmp(name, RI_PAINTEDPLASTIC) == 0)
        return new PaintedPlasticSurface(bindings);
    else if (strcmp(name, RI_SHINYMETAL) == 0)
        return new ShinyMetalSurface(bindings);
    else if (strcmp(name, LRT_VIEWST) == 0)
        return new ViewSTSurface(bindings);
    else if (strcmp(name, LRT_CHECKERED) == 0)
        return new CheckeredSurface(bindings);
    else if (strcmp(name, LRT_GLASS) == 0)
        return new GlassSurface(bindings);
    else {
        Warning("Unknown surface shader: %s,"
                "returning \"matte\"", name);
        return new MatteSurface(bindings);
    }
}
```

10. Texture

10.1 Texture Mapping

We provide some basic functionality for texture mapping. In conjunction with the TIFF image input/output functions described in Appendix B, we here add the functionality for looking up, filtering, and reconstructing texture values at particular points in the texture map.

The class `TextureMap` handles basic texture map management and operations. We take a filename to a TIFF texture, and read it into an array of `Spectrum`s. This can actually be somewhat wasteful, since most TIFFs are stored with 8-bit values in their red, green, and blue channels, while our `Spectrum` class stores spectra with 32-bit floating point values for each channel. For now, however, this will do.

⟨*TextureMap Constructor Declarations*⟩≡
```
TextureMap(const char *filename);
```

⟨*TextureMap Method Definitions*⟩≡
```
TextureMap::TextureMap(const char *filename) {
    Texture = TIFFRead(filename, &width, &height);
}
```

⟨*TextureMap Private Data*⟩≡
```
Spectrum *Texture;
int width, height;
```

And the destructor just needs to free the memory allocated by `TIFFRead`.

⟨*TextureMap Method Declarations*⟩≡
```
~TextureMap();
```

⟨*TextureMap Method Definitions*⟩+≡
```
TextureMap::~TextureMap() {
    delete[] Texture;
}
```

First we'll provide a simple bilinear interpolation texture lookup function. Given continuous floating point texture coordinates with arbitrary values, we compute the coordinates within the texture map that these correspond to. We take the four texels that surround that point and bilinearly interpolate them to compute the texture value.

⟨*TextureMap Method Declarations*⟩+≡
```
Spectrum Lookup(Float u, Float v) const;
```

⟨*TextureMap Method Definitions*⟩+≡
```
Spectrum TextureMap::Lookup(Float u, Float v) const {
    Float weights[4];
    if (!Texture) return Spectrum(1);
    ⟨Compute actual texture coordinates⟩
    ⟨Set up bilinear interpolation weights⟩
    ⟨Look up four corner texels⟩
    return weights[0]*texels[0] + weights[1]*texels[1] +
        weights[2]*texels[2] + weights[3]*texels[3];
}
```

The texture map covers the region in texture coordinates from $(0,0)$ to $(1,1)$. We need to first take the coordinates given by the user and handle the case where they're outside this range. We then compute texture coordinates in the space spanned from $(0,0)$ to $(\text{width},\text{height})$, where width and height are the number of texels in each direction. Finally, we compute the integer texture coordinates of the lower left of the four texels that we'll be using.

⟨*Compute actual texture coordinates*⟩≡
```
⟨Handle out of range texture coordinates⟩
Float xReal = u * (width-1);
Float yReal = v * (height-1);
int x = int(xReal);
int y = int(yReal);
```

For now, we just clamp texture coordinates to the valid range. Another often-useful possibility is to use the `Mod()` function so that the texture repeats infinitely across (u,v) space.

⟨*Handle out of range texture coordinates*⟩≡
```
u = Clamp(u, 0., 1.);
v = Clamp(v, 0., 1.);
```

⟨*Set up bilinear interpolation weights*⟩≡
```
Float dx = xReal - x;
Float dy = yReal - y;
weights[0] = (1.-dx)*(1.-dy);
weights[1] = (1.-dx)*dy;
weights[2] = dx*(1.-dy);
weights[3] = dx*dy;
```

⟨*Look up four corner texels*⟩≡
```
Spectrum texels[4];
texels[0] = Texture[y*[TextureMap::width]width+x];
if (y <height-1 && x < width-1) {
    texels[1] = Texture[(y+1)*[TextureMap::width]width + x];
    texels[2] = Texture[y*[TextureMap::width]width + x + 1];
    texels[3] = Texture[(y+1)*[TextureMap::width]width + x + 1];
} else if (y < height-1) {
    texels[1] = Texture[(y+1)*[TextureMap::width]width + x];
    texels[2] = texels[3] = Spectrum(0);
} else if (x < width-1) {
    texels[2] = Texture[y*[TextureMap::width]width + x + 1];
    texels[1] = texels[3] = Spectrum(0);
} else {
    texels[1] = texels[2] = texels[3] = Spectrum(0);
}
```

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 scene-wide hashtable of texture maps, so that they are only loaded once.

⟨*ShadeContext Private Data*⟩+≡
```
static StringHashTable textures;
```

⟨*ShadeContext Method Definitions*⟩+≡
```
StringHashTable ShadeContext::textures;
```

⟨*ShadeContext Interface*⟩+≡
```
static TextureMap *GetTexture(const char *filename);
```

⟨*ShadeContext Method Definitions*⟩+≡
```
TextureMap *ShadeContext::GetTexture(const char *filename) {
    TextureMap *ret = (TextureMap *)textures.Search(filename);
    if (!ret) {
        ret = new TextureMap(filename);
        textures.Add(filename, ret);
    }
    return ret;
}
```

11. Reflection Models

This chapter defines a set of classes that implement various models for simulating light scattering at surfaces. We will define a `BRDF` class that defines the general interface to these surface reflection functions. BRDF stands for *bidirectional reflectance distribution function*; this is a standard function used to describe scattering at a surface; it is defined so that it gives the fraction of incident energy reflected at the point being shaded for any pair of incident and outgoing directions. We write f_r for the BRDF, and label the direction that incident illumination is arriving from as ω_i and label the outgoing direction where we're interested in outgoing light as ω_o. Thus we have:

$$f_r(\omega_i \rightarrow \omega_o)$$

In source code, we will denote ω_i and ω_o by `wi` and `wo`, respectively.

An important property of BRDFs is that they are *reciprocal*. This means that for all BRDFs and all pairs of angles, $f_r(\omega_i \rightarrow \omega_o) = f_r(\omega_o \rightarrow \omega_i)$. All of the BRDFs described in this chapter obey this property.

The BRDF is itself just an abstract concept that encapsulates a description of light scattering at a surface; how it is expressed or represented is an entirely separate issue. Real world surfaces may have their BRDFs measured directly, with the results stored in large tables or projected to a basis function representation. Alternatively, many authors have come up with simple *parameterized BRDFs*; these are relatively-simple functions that take a small number of parameters (that may have intuitive interpretations, like "roughness"). These functions then turn these parameters into functions that mimic observed BRDFs of real-world surfaces (e.g. plastic, metal, etc.) This chapter describes and has the implementation of a small number of commonly-used parameterized BRDFs.

11.1 The Big Picture

Before we define the BRDF 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 13, 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 BRDF is at a particular point on the surface (see Chapter 9); it returns a newly allocated BRDF object. The integrators then use the BRDF to compute the scattered light at the point, based on the incoming illumination.

We will now define the fundamental interface that all scattering functions provide. The most important method is `fr` which evaluates the BRDF for a given incoming direction `wi`. The outgoing direction, `wo`, doesn't need to be passed to the `fr` function since `wo` is known when the surface shader creates a new BRDF object—`wo` is passed into the BRDF's constructor and each BRDF implementation stores it itself if needed.

⟨*BRDF Method Declarations*⟩+≡
```
virtual Spectrum fr(const Vector &wi) const = 0;
```

11.2 Basic Parameterized Models

Lambertian Reflection.

The simplest scattering function is the Lambertian model; it models a diffuse surface that scatters incident illumination equally in all directions. As such, the particular directions of the incident and outgoing directions make no difference for how much light is scattered. Our Lambertian scattering implementation takes just two parameters, the reflectance of the surface, which is the percentage of incident light that is scattered, and the surface normal at the point being shaded.

⟨*Lambertian Methods*⟩≡
```
Lambertian(const Spectrum &reflectance, const Normal &normal) {
    R = reflectance;
    N = normal;
}
```

⟨*Lambertian Private Data*⟩≡
```
Spectrum R;
Normal N;
```

⟨*Lambertian Methods*⟩+≡
```
Spectrum fr(const Vector &wi) const;
```

The kernel function for `Lambertian` is quite straightforward; we just make sure that the outgoing direction is on the same side of the surface as the surface normal (if it's not, then the point being shaded is facing away from the light source and is thus unilluminated) and return the reflectance.

⟨*BRDF Method Definitions*⟩+≡
```
Spectrum Lambertian::fr(const Vector &wi) const {
    Float costheta = Dot(wi, N);
    if (costheta > 0.) {
        return R;
    } else {
        return Spectrum(0.);
    }
}
```

Glossy Reflection.

A slightly more complex scattering function is a glossy model, due to Jim Blinn. It simulates a glossy specular surface. Unlike perfectly specular surfaces like a mirror or glass, reflections in a glossy surface are blurred out. Plastic is the canonical glossy surface. The Blinn model takes a reflectance and a roughness value. The roughness should be greater than zero and less than or equal to one.

⟨*BlinnGlossy Methods*⟩≡
```
BlinnGlossy(const Spectrum &reflectance, Float roughness,
    const Normal &normal, const Vector &wo);
```

We basically just store the parameters given to us by the user, though we store one over the roughness value, saving a division each time `fr()` is called, since it will generally be called multiple times at each point being shaded. We also divide the roughness value by 8, an empirically chosen value that makes our glossy function better match the one used in the RenderMan-compliant renderers BMRT and prman.

⟨*BRDF Method Definitions*⟩+≡
```
BlinnGlossy::BlinnGlossy(const Spectrum &reflectance,
        Float roughness, const Normal &normal, const Vector &w) {
    R = reflectance;
    roughness /= 8.;  // hack to match BMRT, prman
    invRoughness = 1. / roughness;
    N = normal;
    wo = w;
    costhetao = Dot(wo, N);
}
```

⟨*BlinnGlossy Private Data*⟩≡
```
Spectrum R;
Float invRoughness;
Vector wo;
Normal N;
Float costhetao;
```

⟨*BlinnGlossy Methods*⟩+≡
```
Spectrum fr(const Vector &wi) const;
```

The Blinn model first computes an *half-angle vector*, which is the normalized vector between the incident and outgoing directions—we store this in the variable `H`. It then finds the cosine of the angle between this vector and the surface normal, `costhetah`. If the surface is facing the light source and `costhetah` is greater than zero, it's raised to a power based on the user-supplied roughness value and the product of this with the reflectance is returned to the user.

⟨*BRDF Method Definitions*⟩+≡
```
Spectrum BlinnGlossy::fr(const Vector &wi) const {
    Vector H = (wi + wo).Hat();
    Float costhetah = Dot(N, H);
    Float costhetai = Dot(N, wi);
    if (costhetah > 0. && costhetai > 0.) {
        return R * pow(costhetah, invRoughness) /
                   (costhetai * costhetao);
    }
    else {
        return Spectrum(0.);
    }
}
```

Specular Reflection and Transmission.

We need to make a distinction between scattering functions that are *purely specular* and those that are not. Surfaces like glass or a mirror are perfect specular reflectors: this means that for a given incident direction, there is only a single outgoing direction such that the scattering function is non-zero; as such, their `fr()` function always returns zero, as it's impossible for the caller to randomly choose a `wo` such that the scattering function is non-zero.

Surfaces like these return a number greater than zero in their `SpecularComponents()` method, as appropriate to signify how many separate specular reflection components they have. A mirror would return one specular component, while a glass surface would return two, one for the perfect specular reflection and one for the perfect specular transmission.

⟨*BRDF Method Declarations*⟩+≡
```
virtual int SpecularComponents() const;
```

⟨*BRDF Method Definitions*⟩+≡
```
int BRDF::SpecularComponents() const { return 0; }
```

Scattering functions with specular components also must implement the SampleSpecular method. The caller asks the scattering function to return an outgoing direction for the nth specular component along with the value of the scattering function for that direction.

⟨*BRDF Method Declarations*⟩+≡
```
virtual Spectrum SampleSpecular(int component, Vector *wo) const;
```

⟨*BRDF Method Definitions*⟩+≡
```
Spectrum BRDF::SampleSpecular(int component, Vector *wo) const {
    Severe("SampleSpecular() shouldn't be called for BRDFs "
        "without specular components.");
    return Spectrum(0.);
}
```

SpecularReflection is the first such specular BRDF. It describes perfect reflection from a surface.

⟨*SpecularReflection Methods*⟩≡
```
SpecularReflection(const Spectrum &r,
                   const Normal &N,
                   const Vector &wi);
```

We precompute the reflected direction in the constructor, saving us the repeated work. We use the standard formula that computes the reflected ray given the surface normal and the incident direction, with slight modifications due to our convention that both wi and wo point away from the point being shaded.

⟨*BRDF Method Definitions*⟩+≡
```
SpecularReflection::SpecularReflection(const Spectrum &r,
                                       const Normal &N,
                                       const Vector &wo) {
    R = r;
    reflectedDirection = -wo - 2. * Dot(N, -wo) * Vector(N);
}
```

⟨*SpecularReflection Private Data*⟩≡
```
Spectrum R;
Vector reflectedDirection;
```

The rest of the implementation is completely straightforward; we always return no scattering from fr.

⟨*SpecularReflection Methods*⟩+≡
```
Spectrum fr(const Vector &) const { return Spectrum(0.); }
```

⟨*SpecularReflection Methods*⟩+≡
```
int SpecularComponents() const { return 1; }
```

And instead our scattering is accessed here.

⟨*SpecularReflection Methods*⟩+≡
```
Spectrum SampleSpecular(int component, Vector *wi) const;
```

⟨*BRDF Method Definitions*⟩+≡
```
Spectrum SpecularReflection::SampleSpecular(
        int component, Vector *wi) const {
    Assert(component == 0);
    *wi = reflectedDirection;
    return R;
}
```

The SpecularTransmission class is almost exactly the same as SpecularReflection except that we compute a direction for perfect specular transmission. Additional parameters are indexi and indext, which are the indices of refraction of the medium that the incident ray is in and the medium that the transmitted ray will be entering.

⟨*SpecularTransmission Methods*⟩≡
```
SpecularTransmission(const Spectrum &r,
                     const Normal &N,
                     const Vector &wo,
                     Float indexi,
                     Float indext);
```

We again use the standard formula for computing the refracted ray going from one medium to another, given the incident ray and the normal. In the event of *total internal reflection*, we set the reflectance value to zero.

⟨*BRDF Method Definitions*⟩+≡
```
SpecularTransmission::SpecularTransmission(const Spectrum &r,
        const Normal &N, const Vector &wo, Float indexi,
        Float indext) {
    R = r;
    ⟨Compute specular transmission direction⟩
}
```

⟨*Compute specular transmission direction*⟩≡
```
Float eta = indexi / indext;
Float c1 = Dot(wo, N);
Float c2 = 1 - (eta*eta * (1. - c1));
if (c2 < 0.) {
    // total internal reflection
    R = 0.;
    DirT = wo;
}
else {
    c2 = sqrt(c2);
    DirT = -eta * wo + (eta * c1 - c2) * Vector(N);
}
```

⟨*SpecularTransmission Private Data*⟩≡
```
Spectrum R;
Vector DirT;
```

⟨*SpecularTransmission Methods*⟩+≡
```
Spectrum fr(const Vector &) const { return Spectrum(0.); }
```

⟨*SpecularTransmission Methods*⟩+≡
```
int SpecularComponents() const { return 1; }
```

⟨*SpecularTransmission Methods*⟩+≡
```
Spectrum SampleSpecular(int component, Vector *wo) const;
```

⟨*BRDF Method Definitions*⟩+≡
```
Spectrum SpecularTransmission::SampleSpecular(
        int component, Vector *wi) const {
    Assert(component == 0);
    *wi = DirT;
    return R;
}
```

Mixtures of Scattering Functions.

Since it turns out to be quite handy and since we're all object-oriented and stuff, we also have a scattering function mixture class. It's just a shell when first created, but can hold any number of other `BRDF` objects, which it evaluates the sum of.

⟨*ScatteringMixture Methods*⟩≡
```
ScatteringMixture( ): numSpecular( 0 ) {}
```

But then the creator can call the `AddFunction()` method as much as they'd like, adding as many scattering functions as they'd like. A floating point weight is associated with each one, and scales its contribution relative to the others.

⟨*ScatteringMixture Methods*⟩+≡
```
void AddFunction(BRDF *func, Float weight = 1.0) {
    funcs.push_back(func);
    weights.push_back(weight);
    numSpecular += func->SpecularComponents();
}
```

⟨*ScatteringMixture Protected Data*⟩≡
```
vector<BRDF *> funcs;
vector<Float> weights;
int numSpecular;
```

We need our own destructor so that we free up the scattering functions that we're holding when we're destroyed ourselves.

⟨*ScatteringMixture Methods*⟩+≡
```
~ScatteringMixture();
```

⟨*BRDF Method Definitions*⟩+≡
```
ScatteringMixture::~ScatteringMixture() {
    vector<BRDF *>::const_iterator iter;
    for (iter = funcs.begin(); iter != funcs.end(); ++iter)
        delete *iter;
}
```

Up in `AddFunction` we kept track of the total number of specular components that the scattering functions that we are holding have so that `SpecularComponents` can be implemented efficiently.

⟨*ScatteringMixture Methods*⟩+≡
```
int SpecularComponents() const { return numSpecular; }
```

⟨*ScatteringMixture Methods*⟩+≡
```
Spectrum fr(const Vector &wi) const;
```

To evaluate the scattering kernel, we just walk through our scattering functions and compute their weighted contribution.

⟨*BRDF Method Definitions*⟩+≡
```
Spectrum ScatteringMixture::fr(const Vector &wi) const {
    Spectrum ret(0);
    vector<BRDF *>::const_iterator funcIter = funcs.begin();
    vector<Float>::const_iterator weightIter = weights.begin();

    while (funcIter != funcs.end()) {
        ret += (*weightIter) * (*funcIter)->fr(wi);
        funcIter++, weightIter++;
    }
    return ret;
}
```

⟨*ScatteringMixture Methods*⟩+≡
```
Spectrum SampleSpecular(int component, Vector *wo) const;
```

The SampleSpecular function is somewhat similar: the ScatteringMixture numbers specular components in order based on the ordering of the BRDFs that we're holding. We keep subtracting off the number of components due to the other scattering functions from component until we come to the appropriate scattering function. At this point, component has a value that's appropriate for the scattering function so we just call its SampleSpecular method directly.

⟨*BRDF Method Definitions*⟩+≡
```
Spectrum ScatteringMixture::SampleSpecular(
        int component, Vector *wi) const {
    Assert(component < SpecularComponents());
    vector<BRDF *>::const_iterator funcIter = funcs.begin();
    vector<Float>::const_iterator weightIter = weights.begin();

    while (funcIter != funcs.end()) {
        if (component >= (*funcIter)->SpecularComponents())
            component -= (*funcIter)->SpecularComponents();
        else
            return (*weightIter) *
                   (*funcIter)->SampleSpecular(component, wi);
        ++funcIter, ++weightIter;
    }

    // We shouldn't get to this point.
    Severe("Major logic error in"
           " ScatteringMixture::SampleSpecular()");
    return Spectrum(0);
}
```

11.3 BRDF Creation

⟨*Global Function Declarations*⟩+≡
```
extern BRDF *CreateLambertian(const Spectrum &reflectance,
                              const Normal &normal);
extern BRDF *CreateBlinnGlossy(const Spectrum &reflectance,
                               Float roughness,
                               const Normal &normal,
                               const Vector &wo);
extern BRDF *CreateSpecularReflection(const Spectrum &r,
                                      const Normal &N,
                                      const Vector &wo);
extern BRDF *CreateSpecularTransmission(const Spectrum &r,
                                        const Normal &N,
                                        const Vector &wo,
                                        Float indexi,
                                        Float indext);
extern ScatteringMixture *CreateScatteringMixture();
```

⟨*BRDF Function Definitions*⟩≡

```
BRDF *CreateLambertian(const Spectrum &reflectance,
                       const Normal &normal) {
    return new Lambertian(reflectance, normal);
}

BRDF *CreateBlinnGlossy(const Spectrum &reflectance,
                        Float roughness,
                        const Normal &normal,
                        const Vector &wo) {
    return new BlinnGlossy(reflectance, roughness, normal, wo);
}

BRDF *CreateSpecularReflection(const Spectrum &r,
                               const Normal &N,
                               const Vector &wo) {
    return new SpecularReflection(r, N, wo);
}

BRDF *CreateSpecularTransmission(const Spectrum &r,
                                 const Normal &N,
                                 const Vector &wo,
                                 Float indexi,
                                 Float indext) {
    return new SpecularTransmission(r, N, wo, indexi, indext);
}

ScatteringMixture *CreateScatteringMixture() {
    return new ScatteringMixture;
}

#if 0
BRDF *CreateLafortune(int nLobes, const LafortuneLobe lobes[], const Vector &wi,
        const Normal &N, const Vector &dPdu) {
    Assert(1 == 0);
    return NULL;
    // need to implement the sampling methods
//  return new Lafortune(nLobes, lobes, wi, N, dPdu);
}
#endif
```

12. Light Sources

A scene full of geometry doesn't do much good without lights to illuminate it. This chapter has the general interface that sources of illumination in the scene conform to as well as definitions of a number of interesting types of lights. In addition to basic light sources like point lights (a point in space that emits light in all directions in equal intensity), we also include lights like spot lights, that concentrate illumination through a cone in space, leaving other areas dark.

12.1 Light Attributes

First we define the attributes that are managed by the API and bound to each light source. The first two are `CastsShadows` and `NSamples`. `CastsShadows` makes it possible to flag light sources that don't cast shadows in the scene–when deciding if a point in space is illuminated, we don't bother tracing a shadow ray if `CastsShadows` is `false`. This is completely wrong and non-physical, but can be useful when trying to achieve a particular lighting effect.

`NSamples` is used for area light sources (see Sections 12.4 and 13.4), where more than one shadow ray may be traced toward the light source. This gives a mechanism for the user the specify the number of rays to be traced.

⟨*LightAttributes Method Declarations*⟩≡
```
LightAttributes() {
    ⟨LightAttributes Constructor Implementation⟩
}
```

⟨*LightAttributes Public Data*⟩≡
```
bool CastsShadows;
int NSamples;
```

⟨*LightAttributes Constructor Implementation*⟩≡
```
CastsShadows = false;
NSamples = 1;
```

We also store two transformations with each light, to go back and forth between world space and the coordinate space that was active when the light was first defined.

⟨*LightAttributes Public Data*⟩+≡
```
Transform WorldToLight, LightToWorld;
```

12.2 Light Interface

All light sources inherit from the abstract `Light` class, which defines the interface that they must adhere to.

All lights store both `SurfaceFunction` and `LightAttributes`. The attributes are described above, and `SurfaceFunction` was introduced in Section 9.1 on page 117. `SurfaceFunction` is used to store values to the parameters of the light such as the color that it emits that the user specified when the light was created.

⟨*Light Interface*⟩≡
```
Light(SurfaceFunction *ud, LightAttributes *attr )
    : attributes(attr), surfaceFunction(ud) { }
```

⟨*Light Protected Data*⟩≡
```
LightAttributes *attributes;
SurfaceFunction *surfaceFunction;
```

⟨*Light Interface*⟩+≡
```
virtual ~Light();
```

⟨*Light Method Definitions*⟩≡
```
Light::~Light() {
    delete surfaceFunction;
    attributes->Dereference();
}
```

And now we define the interface that `Light`s expose. A boolean function records if the light casts shadows or not; it returns a value depending on the `LightAttributes`.

⟨*Light Interface*⟩+≡
```
bool CastsShadows() const { return attributes->CastsShadows; }
```

The `dE` method is the most important function in the light interface. The user passes in the point being shaded in world space, `PShade`, and a pointer to return a point on the light source. The light source computes the differential irradiance (typically denoted by dE) arriving at `Pshade` from `Plight` and returns it to the caller.

⟨*Light Interface*⟩+≡
```
virtual Spectrum dE(const Point &Pshade, Point *Plight) const = 0;
```

Finally, a way to get at the number of samples to be taken on the light. This is really only useful for area lights, where more than one sample may be taken.

⟨*Light Interface*⟩+≡
```
int NumSamples() const { return attributes->NSamples; }
```

⟨*Light Interface*⟩+≡
```
virtual const Primitive *GetPrimitive() const { return NULL; }
```

Scene Methods.

⟨*Scene Method Declarations*⟩+≡
```
bool Unoccluded(const Point &, const Point &) const;
bool Unoccluded(const Ray &) const;
```

⟨*Scene Methods*⟩+≡
```
bool Scene::Unoccluded(const Point &p1, const Point &p2) const {
    if (!accelerator) return true;
    ⟨Check visibility between points⟩
    return true;
}
```

⟨*Check visibility between points*⟩≡
```
⟨Update finite shadow ray statistics⟩
Float tmin = 1e-6;
Float tmax = 1.-tmin;
if (accelerator->IntersectClosest(Ray(p1, p2-p1), tmin, &tmax,
        NULL)) {
    ++shadowRayOccluded;
    return false;
}
```

⟨*Update finite shadow ray statistics*⟩≡
```
static int shadowRayChecks = 0, shadowRayOccluded = 0;
if (shadowRayChecks == 0)
    StatsRegisterRatio(STATS_DETAILED, "Integration",
        "Finite Shadow Ray Checks",
        &shadowRayOccluded, &shadowRayChecks);
++shadowRayChecks;
```

⟨*Scene Methods*⟩+≡
```
bool Scene::Unoccluded(const Ray &r) const {
    if (!accelerator) return true;
    ⟨Check infinite shadow ray⟩
    return true;
}
```

⟨*Check infinite shadow ray*⟩≡
```
⟨Update infinite shadow ray statistics⟩
Float tmin = 1e-6;
Float tmax = INFINITY;
Ray ray = r;
ray.D.Normalize();
if (accelerator->IntersectClosest(ray, tmin, &tmax, NULL)) {
    ++shadowRayOccluded;
    return false;
}
```

⟨*Update infinite shadow ray statistics*⟩≡
```
static int shadowRayChecks = 0, shadowRayOccluded = 0;
if (shadowRayChecks == 0)
    StatsRegisterRatio(STATS_DETAILED, "Integration",
        "Infinite Shadow Ray Checks",
        &shadowRayOccluded, &shadowRayChecks);
++shadowRayChecks;
```

12.3 Basic Light Types

Now onward to some light source implementations. We will start with some of the simplest ones, in order to make the interfaces and responsibilities of light sources more familiar.

Isotropic Point Lights.

A useful type of light is the *point light*; illumination is emitted from a point in space equally and in all directions. The `IsotropicPointLight` implements such a light.

⟨*IsotropicPointLight Methods*⟩≡
```
IsotropicPointLight(SurfaceFunction *ud, LightAttributes *attr);
```

In the constructor, we need to compute the values of two variables: `I`, which is the intensity being emitted by the light, and `lightPos`, the light's position in world space.

⟨*IsotropicPointLight Method Definitions*⟩≡
```
IsotropicPointLight::IsotropicPointLight(SurfaceFunction *ud,
        LightAttributes *attr) : Light(ud, attr) {
    ⟨Compute point light position⟩
    ⟨Compute light intensity⟩
}
```

⟨*IsotropicPointLight Private Data*⟩≡
```
Spectrum I;
Point lightPos;
```

By default, the light is centered at the origin $(0,0,0)$. However, if the user bound a value to the light's `from` position, we grab it from our `SurfaceFunction` and store it in `lightPos`. In either case, we use the our light to world space transformation to compute the light's position in world space.

⟨*Compute point light position*⟩≡
```
lightPos = surfaceFunction->InitializePoint(RI_FROM, Point(0, 0, 0));
lightPos = attr->LightToWorld(lightPos);
```

There are two (slightly redundant) knobs to turn to specify the light's brightness and color. `lightcolor` is a spectrum that describes the overall spectrum that is emitted by the light and `intensity` is a floating-point value that the light color is multiplied by.

⟨*Compute light intensity*⟩≡
```
Spectrum lightColor =
    surfaceFunction->InitializeColor(RI_LIGHTCOLOR,
                        Spectrum(1.0));
Float intensity =
    surfaceFunction->InitializeFloat(RI_INTENSITY, 1.0);
I = lightColor * intensity;
```

⟨*IsotropicPointLight Methods*⟩+≡
```
Spectrum dE(const Point &Ps, Point *Plight) const;
```

Now our task when the user asks for illumination from the light is quite simple. We just store the light's world space position in the output `Plight` variable and return our intensity divided by the square of the distance between the light and the point being illuminated. This adjustment is necessary to account for the r^2 fall-off of intensity with distance.

⟨*IsotropicPointLight Method Definitions*⟩+≡
```
Spectrum IsotropicPointLight::dE(const Point &Ps,
        Point *Plight) const {
    *Plight = lightPos;
    return I / DistanceSquared(lightPos, Ps);
}
```

Directional Lights.

Another straightforward 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.

⟨*DirectionalLight Methods*⟩≡
```
DirectionalLight(SurfaceFunction *ud, LightAttributes *attr);
```

Similarly to the point source, we compute the light's intensity and direction in the constructor. We reuse the ⟨*Compute light intensity*⟩ fragment from the point light, since all of the relevant variables have the same names.

⟨*IsotropicPointLight Method Definitions*⟩+≡
```
DirectionalLight::DirectionalLight(SurfaceFunction *ud,
        LightAttributes *attr) : Light(ud, attr) {
    Point pFrom, pTo;
    ⟨Compute light direction vector⟩
    ⟨Compute light intensity⟩
}
```

⟨*DirectionalLight Private Data*⟩≡
```
Spectrum I;
Vector lightDir;
```

By default, directional lights point from the origin to the point $(0,0,1)$ in light space. These two points (the `from` and `to` points) can be overridden by the user, however, so we grab their values from `SurfaceFunction` if possible. We then transform these points from light space to world space and compute the normalized light direction vector that they define.

⟨*Compute light direction vector*⟩≡
```
pFrom = surfaceFunction->InitializePoint(RI_FROM, Point(0, 0, 0));
pTo = surfaceFunction->InitializePoint(RI_TO, Point(0, 0, 1));
pFrom = attr->LightToWorld(pFrom);
pTo = attr->LightToWorld(pTo);
lightDir = (pFrom - pTo).Hat();
```

⟨*DirectionalLight Methods*⟩+≡
```
Spectrum dE(const Point &, Point *Plight) const;
```

Now to sample a directional light, our interface wants us to compute a point on the light source. Well, perhaps the interface should be revisited, but so it goes the day before class starts. We compute a point that has the right effect by offsetting from the point being shaded a fair distance along the light vector.

⟨*DirectionalLight Method Definitions*⟩≡
```
Spectrum DirectionalLight::dE(const Point &Ps,
        Point *Plight) const {
    *Plight = Ps + 1000. * lightDir; // Ye Olde Hack O' Rama
    return I;
}
```

Spot Light.

A more interesting light is a spot light: light originates from a single point, but the light is oriented so that it's facing in a particular direction. Illumination is present inside a particular cone around that direction and the rest of space is not lit.

⟨*SpotLight Methods*⟩≡
```
SpotLight(SurfaceFunction *ud, LightAttributes *attr);
```

We re-use the ⟨*Compute light direction vector*⟩ and ⟨*Compute light intensity*⟩ fragments from the distant and point lights, respectively.

⟨*SpotLight Method Definitions*⟩≡
```
SpotLight::SpotLight(SurfaceFunction *ud, LightAttributes *attr)
        : Light(ud, attr) {
    ⟨Compute light direction vector⟩
    ⟨Compute spot light cone angles⟩
    ⟨Compute light intensity⟩
}
```

⟨*SpotLight Private Data*⟩≡
```
Spectrum I;
Point pFrom, pTo;
Vector lightDir;
```

The final parameters that we store describe the distribution of light in the spot light. First is `coneAngle`; this gives the angular width of the cone of light, measured from its center. We convert the user-supplied value to radians for more efficient computation in the shader. Next, `coneDeltaAngle` defines a region along the inside of the cone where the light falls off. When the angle `coneAngle - coneDeltaAngle` is reached, the intensity of the light starts to fall off, reaching zero at `coneAngle`. Finally, `bemaDistribution` defines another, more gradual fall-off from the center of the beam.

⟨*Compute spot light cone angles*⟩≡
```
coneAngle =
    Radians(surfaceFunction->InitializeFloat(RI_CONEANGLE, 30.));
coneDeltaAngle =
    Radians(surfaceFunction->InitializeFloat(RI_CONEDELTAANGLE, 5.));
beamDistribution =
    surfaceFunction->InitializeFloat(RI_BEAMDISTRIBUTION, 2.0);
coneAngle = Clamp( coneAngle, 0, 2*M_PI );
coneDeltaAngle = Clamp(coneDeltaAngle, 0, coneAngle );
```

⟨*SpotLight Private Data*⟩+≡
```
Float coneAngle, coneDeltaAngle, beamDistribution;
```

⟨*SpotLight Methods*⟩+≡
```
Spectrum dE(const Point &Ps, Point *Plight) const;
```

To evaluate the spot light, we first stuff our position in `Plight`. We then compute the cosine of the angle between the vector from the point being shaded to the vector along the center of the spotlight. We then compute an attenuation factor that scales the light's intensity at `Ps`, the point being lit. First we raise `cosangle` to the `beamDistribution` power: this makes the light brightest in the center of the cone, gradually falling off to the edges. Then we attenuate this further, using `SmoothStep` to ramp the intensity from one down to zero through the transition zone around the edge of the cone. Finally we multiply this by the light's intensity and divide by r^2, returning incident differential irradiance at the point being lit.

⟨*SpotLight Method Definitions*⟩+≡
```
Spectrum SpotLight::dE(const Point &Ps, Point *Plight) const {
    *Plight = pFrom;
    Float cosangle =
        max(Dot((pFrom - Ps).Hat(), lightDir), (Float)0.);
    Float atten = pow( cosangle, beamDistribution );
    atten *= SmoothStep(cos(coneAngle),
                        cos(coneAngle - coneDeltaAngle),
                        cosangle);
    //atten *= SmoothStep(cos(coneAngle+coneDeltaAngle),
    //                    cos(coneAngle),
    //                    cosangle);
    return atten * I / DistanceSquared(pFrom, Ps);
}
```

12.4 Area Lights

We'll now start to provide some functionality for *area lights*; these are associated with actual pieces of geometry in the scene that emit light out into the scene. This part of the implementation is just concerned with computing the outgoing light from particular points on the light source; how those points are chosen for the various types of `Primitive`s is the topic of Section 13.4.

⟨*Area Light Function Definitions*⟩≡
```
AreaLight::AreaLight(SurfaceFunction *ud, LightAttributes *attr)
        : Light(ud, attr) {
    primitive = NULL;
}
```

`AreaLight`s have an additional method, `SetGeometry`. For each primitive that is an emitter, a pointer to that primitive is passed to the currently active area light shader (if any).

⟨*AreaLight Interface*⟩≡
```
void SetGeometry(const Primitive *prim) {
    if (primitive != NULL) {
        Severe("Only one primitive per area light allowed.");
    }
    primitive = prim;
}
```

⟨*AreaLight Protected Data*⟩≡
```
const Primitive *primitive;
```

⟨*AreaLight Interface*⟩+≡
```
const Primitive *GetPrimitive() const { return primitive; }
```

Finally, we provide a method that evaluates the area light's emitted radiance (usually denoted in formulae by L_e) at a point on the surface of the light. This takes a reference to a `ShadeContext` object that holds information about the geometry of the point being shaded and the outgoing direction (see Section 9.3 on page 126).

⟨*AreaLight Interface*⟩+≡
```
virtual Spectrum Le(const ShadeContext &sc) const = 0;
```

Diffuse Area Light.

Onward now to a particular type of area light: the diffuse area light emits a uniform intensity in all directions across the surface of the object. In the constructor, we just compute `Lo`, the outgoing radiance at each point on the surface.

⟨*Area Light Function Definitions*⟩+≡
```
DiffuseAreaLight::DiffuseAreaLight(SurfaceFunction *ud,
        LightAttributes *attr) : AreaLight(ud, attr) {
    Spectrum lightColor =
        surfaceFunction->InitializeColor(RI_LIGHTCOLOR,
        Spectrum(1.0));
    Float intensity =
        surfaceFunction->InitializeFloat(RI_INTENSITY, 1.0);
    Lo = lightColor * intensity;
}
```

⟨*DiffuseAreaLight Private Data*⟩≡
```
Spectrum Lo;
```

We will defer the implementation of `DiffuseAreaLight`'s `dE()` method until Section 13.4, where Monte Carlo integration and area light sampling is explained.

⟨*DiffuseAreaLight Interface*⟩≡
```
Spectrum Le(const ShadeContext &) const;
```

We'll trust the caller that the given point is on the light source.

⟨*Area Light Function Definitions*⟩+≡
```
Spectrum DiffuseAreaLight::Le(const ShadeContext &) const {
    return Lo;
}
```

12.5 Light Source Creation

To make it easier for the interface layer to create appropriate `Light` objects, we provide a short creation function that maps from light names to light objects.

⟨*Global Function Declarations*⟩+≡
```
extern Light *PointLightCreate(const char *name,
                               SurfaceFunction *data,
                               LightAttributes *attr);
```

⟨*Light Creation Function Definitions*⟩≡
```
Light *PointLightCreate(const char *name,
                        SurfaceFunction *data,
                        LightAttributes *attr) {
    if (strcmp(name, RI_POINTLIGHT) == 0)
        return new IsotropicPointLight(data, attr);
    else if (strcmp(name, RI_DISTANTLIGHT) == 0)
        return new DirectionalLight(data, attr);
    else if (strcmp(name, RI_SPOTLIGHT) == 0)
        return new SpotLight(data, attr);
    else
        return NULL;
}
```

⟨*Global Function Declarations*⟩+≡
```
extern AreaLight *AreaLightCreate(const char *name,
                                  SurfaceFunction *data,
                                  LightAttributes *attr);
```

⟨*Light Creation Function Definitions*⟩+≡
```
AreaLight *AreaLightCreate(const char *name,
                           SurfaceFunction *data,
                           LightAttributes *attr) {
    if (strcmp(name, LRT_DIFFUSE) == 0)
        return new DiffuseAreaLight(data, attr);
    else
        return NULL;
}
```

13. Light Transport

This chapter finally brings all of the preceding chapters together into the service of image synthesis. 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 eye rays, they are handed off to the `Integrator` that the user selected; the integrator is then responsible for doing appropriate shading lighting computations to compute the radiance scattered back along the ray. We will provide a few different `Integrator`s, each providing a different level of accuracy in its modelling of light transport.

We start by adding some entries to the `PrimitiveAttribute` class. First, for each geometric primitive, we keep a list of `Light`s; these are all of the lights in the scene that are illuminating the primitive.

⟨*PrimitiveAttributes Public Data*⟩+≡
```
list<Light *> Lights;
```

We also add a field to the scene that describes which integrator is to be used.

⟨*Scene Public Data*⟩+≡
```
RtToken IlluminationIntegrator;
```

By default, we do simple recursive Whitted-style raytracing.

⟨*Initialize Default Scene Options*⟩≡
```
IlluminationIntegrator = LRT_WHITTED;
```

Now we can describe the interface that all `Integrator`s must adhere to. There is just a single function that they implement, `Integrate`. The parameters are the following:

1. `ray`: the ray along which the scattered radiance should be evaluated.
2. `hitInfo`: this non-`NULL` pointer to a `HitInfo` object is used to store information about the intersection point; this may be used by the caller after the function returns.
3. `hitDist`: this is both a caller-supplied upper bound on the maximum distance to search for an intersection as well as an output variable where the parametric distance to the intersection should be returned.
4. `alpha`: the opacity of the surface that was hit should be set in this output variable; it should be zero if no surface was hit.

`Integrate` returns a `Spectrum` that holds the radiance along the ray.

⟨*Integrator Method Declarations*⟩+≡
```
virtual Spectrum Integrate(const Ray &ray, HitInfo *hitInfo,
    Float *hitDist, Float *alpha) const = 0;
```

13.1 Color Integrator

Our first integrator is the simplest integrator possible. It finds the first surface visible along the ray, and returns the color that was bound to the surface (see Chapter 9). It is mostly useful for debugging various other parts of the system, but is also illustrative in showing how integrators operate.

This is its implementation in its entirety. We call the scene's `IntersectClosest` method to find the first intersection along the ray. This updates `hitDist` and `hitInfo` appropriately if an intersection was found. If there was in fact an intersection, we set the alpha value to one and look up the color that was bound to the surface. This color is either available via `hitInfo`, which looks for it in the color values bound at the hit point, or in the `Color` field of the `PrimitiveAttributes`, reflecting a constant color over the entire surface.

⟨*ColorIntegrator Method Definitions*⟩≡
```
Spectrum ColorIntegrator::Integrate(const Ray &ray,
        HitInfo *hitInfo, Float *hitDist,
        Float *alpha) const {
    if (scene->Intersect(ray, 0., hitDist, hitInfo)) {
        *alpha = 1.;
        const Spectrum *Csp = hitInfo->GetColor(RI_CS);
        if (Csp) {
            return *Csp;
        } else {
            return hitInfo->hitPrim->attributes->Color;
        }
    } else {
        *alpha = 0.;
        return Spectrum(0,0,0);
    }
}
```

13.2 Ray-Casting with Shadows

The next `Integrator` shows how to compute basic shading at the hit point, including shadows cast by the light sources. As such, we'll call it the `RayCastingIntegrator`.

The basic integration function has the same general form as the one for `ColorIntegrator` If the ray intersects geometry in the scene, we run the object's surface shader at the intersection point to compute the scattering function there. We then process the light sources to compute incoming illumination at the hit point and from that, the reflected radiance along `ray`.

Note that the fragments ⟨*Evaluate BRDF and create surface shader*⟩ and ⟨*Compute reflection by integrating over the lights*⟩ are defined way back in chapter 1.

⟨*RayCastingIntegrator Method Definitions*⟩≡
```
Spectrum RayCastingIntegrator::Integrate(const Ray &ray,
        HitInfo *hitInfo, Float *hitDist,
        Float *alpha) const {
    if (scene->Intersect(ray, 0., hitDist, hitInfo)) {
        ShadeContext shadeContext(hitInfo, -ray.D);
        Spectrum L(0.);
        *alpha = 1.;
        ⟨Compute emitted light if an area light source⟩
        ⟨Evaluate BRDF and create surface shader⟩
        ⟨Compute reflection by integrating over the lights⟩
        ⟨Clean up from integration⟩
        return L;
    }
    else {
        *alpha = 0.;
        return Spectrum(0.);
    }
}
```

If the ray happened to hit an area light source, we need to add the emitted radiance

⟨*Compute emitted light if an area light source*⟩≡
```
if (hitInfo->hitPrim->attributes->LightShader != NULL) {
    L +=
     hitInfo->hitPrim->attributes->LightShader->Le(shadeContext);
}
```

When we're done, all that there is to do is to free the scattering function that the shader returned.

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

13.3 Whitted Integrator

A small generalization of the ray casting integrator above allows much more interesting pictures. 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 modelled with the ray-tracing algorithm. When a specularly reflective or transmissive object is hit by a ray, new rays are also traced in the reflected and refracted directions in addition to spawning rays to the light sources. The radiance along these 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`.

13.4 Monte Carlo Integrator

Now on to the skeleton implementation that you'll be extending for the fifth assignment. Here we'll add some interfaces for Monte Carlo sampling of scattering functions and light sources as well as the skeleton implementation of a Monte Carlo `Integrator`. In this section, we will base this framework on the *rendering equation*, as expressed in a few different forms, depending on which is most convenient. Recall from class that the basic Monte Carlo estimator is:

$$\int f(x)\mathrm{d}x = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)} \tag{13.4.1}$$

where f is the function that we're integrating, N is the number of samples that we're talking, x_i are the sample locations we chose (in some as-yet undefined manner) and $p(x_i)$ is the probability of choosing sample x_i.

Sampling Reflection Functions.

First we will talk about using importance sampling to compute integrals with the reflection functions from Chapter 11. Given some point on a surface, we wish to compute the integral

$$L_o(x,\omega_o) = \int_\Omega f_r(x,\omega_i \to \omega_i)\, L_i(\omega_i)\cos\theta_i \mathrm{d}\omega_i. \tag{13.4.2}$$

In other words, we're integrating the incident radiance L_i over the hemisphere surrounding the point x that we're shading. We want to compute $L_o(x,\omega_o)$, the outgoing radiance at x in a direction ω_o, given a known BRDF, f_r. We could choose directions ω_i entirely randomly over the hemisphere to compute the above integral, but we can compute a more accurate estimate using *importance sampling*. If we can choose ω_i from a distribution that approximately matches one of the terms in the integral, we will allocate our samples more efficiently.

We'll add a `Sample()` method to BRDFs that lets us sample directions based on the shape of the BRDF. A pair of random numbers ranging from zero to one are passed in the `u` variable; this lets the caller provide a set of *stratified* sample points.

`Sample()` returns the value of the BRDF for the direction chosen, which is stored in `*wi`. The probability with which `*wi` was chosen should be stored in `*pdf` (which stands for "probability density function"). If a valid sample wasn't generated, `*pdf` should be set to zero. The integrator should to check for this case before using the returned sample.

⟨*BRDF Method Declarations*⟩+≡
```
virtual Spectrum Sample(Float u[2], Vector *wi, Float *pdf) const = 0;
```

We'll start by defining a simple importance sampling function which samples directions from a cosine-weighted distribution over the hemisphere. This is appropriate for `Lambertian` surfaces because we know that the integral in Equation 13.4.2 weights the result by a cosine term. Therefore, we will generate `wi` directions that tare more likely to be close to the top of the hemisphere than the bottom, where the cosine term has a small value.

⟨*Lambertian Methods*⟩+≡
```
Spectrum Sample(Float u[2], Vector *wi, Float *pdf) const;
```

⟨*BRDF MC Methods*⟩≡
```
Spectrum Lambertian::Sample(Float u[2], Vector *wi,
        Float *pdf) const {
    ⟨Remap sample positions to -1 to 1⟩
    ⟨Compute local coordinate system axes⟩
    ⟨Compute outgoing direction and pdf⟩
    return fr(*wi);
}
```

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.

We start by remapping the two random numbers from the user to the range $[-1,1]$. We then make sure that the resulting coordinate (u_0,u_1) is inside the unit disk; if it's not, we just *reject* it. We tell the user that there's no sample to be taken for this pair of random numbers by setting the `pdf` to zero. Note that we could uniformly sample the disk by directly computing an equi-areal sampling in polar coordinates, but this would be slow. Since you probably have such a sampler from your depth-of-field assignment, try replacing this code with yours to see the effect on the speed. The disadvantage of rejection sampling is that the caller may not re-generate a sample; In the extreme case of one sample per estimation ($N = 1$), many points will not be lit at all.

⟨*Remap sample positions to -1 to 1*⟩≡
```
Float u0 = 2*u[0] - 1.;
Float u1 = 2*u[1] - 1.;
if (u0*u0 + u1*u1 > 1.) {
    *pdf = 0.;
    return Spectrum(0.);
}
```

Now we compute a coordinate system so that we can do the projection up from the disk to the sphere centered around the normal. We already have the normal direction, `N`, stored in the `Lambertian` class. We now need two more unit vectors that are orthogonal to `N`. These three vectors specify a little coordinate system around the point being shaded. A few cross products get us there.

In the usual case below, the `else` clause will be followed. We first pick some vector that won't be parallel to `N` by permuting `N`'s components. The cross product of this with `N` gives us a vector that's perpendicular to `N`, `Du`. Taking the cross product of `Du` and `N`, we get another vector that's perpendicular to *both* of them, thus giving us the coordinate system.

⟨*Compute local coordinate system axes*⟩≡
```
Vector Du, Dv;
if (N.x == 0. && N.y == 0.) {
    Du = Vector(1, 0, 0);
    Dv = Vector(0, 1, 0);
}
else {
    Du = Cross(Vector(N.y, N.x, 0.), Vector(N)).Hat();
    Dv = Cross(Du, Vector(N)).Hat();
}
```

Now we just do the projection to compute the direction `wi`. All that's left is computing the probability density for the direction that we chose: in order for the Monte Carlo estimate of Equation 13.4.1 to be correct, it needs to reflect the fact that we were less likely to choose directions close to the horizon.

We know that Malley's method generates samples in a cosine distribution. We first need to normalize this distribution so that it's a valid probability density function. This is done by finding a constant c such that the pdf integrates to exactly one over our domain.

$$1 = c\int_{\omega}\cos\omega d\omega \tag{13.4.3}$$

$$\frac{1}{c} = \int_0^{2\pi}\int_0^{\pi/2}\cos\theta\sin\theta d\theta d\phi \tag{13.4.4}$$

$$c = \frac{1}{\pi} \tag{13.4.5}$$

Thus, the pdf for a given direction ω is $\frac{1}{\pi}\cos\omega$.

⟨*Compute outgoing direction and pdf*⟩≡
```
*wi = u0 * Du + u1 * Dv + sqrt(1. - (u0*u0 + u1*u1)) * Vector(N);
*pdf = Dot(N, *wi) / M_PI;
```

"Now you do it." Here is the skeleton for importance sampling the `BlinnGlossy` scattering function. As described in the assignment, you want to importance sample a half-angle direction `H` based on the `invRoughness` specular exponent. The incident direction `wi` is computed by reflecting `wo` about the `H` vector.

Computing the probability density function for this is tricky because `H` is being sampled from a different distribution than the hemisphere centered around the normal. Instead of Equation 13.4.2, we're instead computing the integral:

$$L_o(x,\omega_o) = \int_\Omega f_r(x,\omega_i \to \omega_i) L_i(\omega_i) \cos\theta_i d\omega_H.$$

We thus need to compute a correction term $\frac{\delta\omega_i}{\delta\omega_H}$ to convert back to the measure of the original integral. Recall that $d\omega = \sin\theta d\theta d\phi$, then apply the geometry of the situation and basic trig identities.

⟨*BlinnGlossy Methods*⟩+≡
```
Spectrum Sample(Float u[2], Vector *wi, Float *pdf) const;
```

⟨*BRDF MC Methods*⟩+≡
```
Spectrum BlinnGlossy::Sample(Float u[2],
        Vector *wi, Float *pdf) const {
    // YOUR CODE HERE
    return Spectrum(0.);
}
```

To wrap up, we'll provide the code for sampling `ScatteringMixture` scattering functions; we just randomly pick one of the constituent scattering functions and sample that.[1].

⟨*ScatteringMixture Methods*⟩+≡
```
Spectrum Sample(Float u[2], Vector *wi, Float *pdf) const;
```

Two details to note: first, we divide the pdf that the individual scattering function returned by `funcs.size()`, the total number of scattering functions. This is necessary to re-normalize the pdf since we're actually integrating over more items than `funcs[i]` realizes we are. Second, we call the `ScatteringMixture` `fr` function to compute the value, rather than just returning the `Spectrum` that `funcs[i]->Sample()` returned.

⟨*BRDF MC Methods*⟩+≡
```
Spectrum ScatteringMixture::Sample(Float u[2],
        Vector *wi, Float *pdf) const {
    int fnum = int(RandomFloat() * funcs.size());
    funcs[fnum]->Sample(u, wi, pdf);
    *pdf /= funcs.size();
    return fr(*wi);
}
```

[1] Multiple importance sampling could be used to do this much more efficiently. However, for simplicity, we will just do the most obvious implementation

Sampling Area Light Sources.

Another way to write the rendering integral is as an integral over all of the surfaces in the scene (or, when computing direct lighting, over all of the emitting objects in the scene):

$$L_o(x,\omega_o) = \int_A f_r(x,\omega_i \rightarrow \omega_i) L_e(x',\omega_o) \frac{\cos\theta_i \cos\theta_o}{r^2} V(x,x') \mathrm{d}A' \tag{13.4.6}$$

Where $V(x,x')$ is the visibility function that returns one if the two points x and x' are mutually visible and zero otherwise.

In order to be able to evaluate this integral, we need to be able to sample points on the area light sources in the scene. To be able to do this, we need to be able to sample points on all of the various types of geometric primitives that may be in the scene. We add a `Sample()` method to the `Primitive` class, and to `Sphere`s, `Cylinder`s, and `Cone`s.

The `Sample()` method, like the `Sample()` method for `BRDF`s takes pair of random numbers ranging from zero to one. It fills in a `HitInfo` with information about the sampled point on the light source and returns the probability density function with respect to the area measure on the light that the point is sampled.

⟨*Primitive Interface*⟩+≡
```
virtual Float Sample(Float u[2], HitInfo *hitinfo) const;
```

⟨*Primitive Methods*⟩+≡
```
Float Primitive::Sample(Float[2], HitInfo *) const {
    Severe("Sample() method unimplemented for primitive");
    return 0.;
}
```

⟨*Sphere Interface*⟩+≡
```
Float Sample(Float u[2], HitInfo *) const;
```

⟨*Cylinder Interface*⟩+≡
```
Float Sample(Float u[2], HitInfo *) const;
```

⟨*Cone Interface*⟩+≡
```
Float Sample(Float u[2], HitInfo *) const;
```

For this assignment, the points chosen should uniformly sample the surface of the light source. This means that it should be equally probable that any point on the surface is sampled and thus that the pdf returned should be a constant. The pdf should be normalized so that the integral over the area of the light source $\int_A \mathrm{pdf}(x')\mathrm{d}A'$ is equal to one.

⟨*Primitive Sampling Functions*⟩≡

```
Float Sphere::Sample(Float u[2], HitInfo *hitInfo) const {
    // YOUR CODE HERE: sample a point P, and compute
    // the inverse mapping to find uu and vv.  Then
    // pass this and the Disk's Normal, (0,0,1) into
    // the (misnamed) RecordHit() method of HitInfo.

    // hitInfo->RecordHit(P, N, uu,
    //                    vv, attributes, surfaceFunction);
    // return Pdf;
    return 0.;
}

Float Cylinder::Sample(Float u[2], HitInfo *hitInfo) const {
    // YOUR CODE HERE: sample a point P, and compute
    // the inverse mapping to find uu and vv.  Then
    // pass this and the Disk's Normal, (0,0,1) into
    // the (misnamed) RecordHit() method of HitInfo.

    // hitInfo->RecordHit(P, N, uu,
    //                    vv, attributes, surfaceFunction);
    // return Pdf;
    return 0.;
}

Float Cone::Sample(Float u[2], HitInfo *hitInfo) const {
    // YOUR CODE HERE: sample a point P, and compute
    // the inverse mapping to find uu and vv.  Then
    // pass this and the Disk's Normal, (0,0,1) into
    // the (misnamed) RecordHit() method of HitInfo.

    // hitInfo->RecordHit(P, N, uu,
    //                    vv, attributes, surfaceFunction);
    // return Pdf;
    return 0.;
}
```

⟨*DiffuseAreaLight Interface*⟩+≡

```
Spectrum dE(const Point &Pshade, Point *Plight) const;
```

Now we can implement the `DiffuseAreaLight`'s `dE` method. We generate a pair of random numbers and pass them into the appropriate `AreaLightSampler` object, which chooses a point on the light source, We transform the point and normal back out of light space and into world space, and compute and return the incident differential irradiance at the point being shaded.

⟨*Area Light Function Definitions*⟩+≡
```
Spectrum DiffuseAreaLight::dE(const Point &Pshade,
        Point *Plight) const {
    Float u[2] = { RandomFloat(), RandomFloat() };
    HitInfo hitInfo;
    Float pdf = primitive->Sample(u, &hitInfo);
    if (pdf != 0.) {
        *Plight =
            primitive->attributes->ObjectToWorld(hitInfo.Pobj);
        Normal N =
            primitive->attributes->ObjectToWorld(hitInfo.NgObj);
        Float costhetao =
            fabs(Dot(N.Hat(), (Pshade - *Plight).Hat()));
        return Lo * costhetao /
                    (DistanceSquared(Pshade, *Plight) * pdf);
    } else {
        return Spectrum(0.);
    }
}
```

Multiple Importance Sampling.

Finally, we need some infrastructure for multiple importance sampling. Recall that the basic Monte Carlo estimator of Equation 13.4.1 can be expressed as:

$$\int f(x)\mathrm{d}x = \sum_{i=1}^{N} \frac{1}{n_i} \sum_{j=1}^{n_i} w(x_{i,j}) \frac{f(x_{i,j})}{p(x_{i,j})} \tag{13.4.7}$$

Where we are using N different importance sampling methods and each one takes n_i samples. A variety of weighting functions w can be used as long as some basic requirements are met (see Section 9.2.1.1 of Veach's thesis.)

For all of the effective weighting methods, we need to be able to generate a sample using one importance sampling method and then compute the probability density that each of the other methods would have generated it. Thus, we need a few new methods.

First is `Pdf()`, for `BRDF`'s. It returns the probability density with respect to solid angle that a particular outgoing angle would have been sampled. Once the importance sampling functions of Section 13.4 have been implemented, these are quite easy to implement.

⟨*BRDF Method Declarations*⟩+≡
```
virtual Float Pdf(const Vector &wi) const = 0;
```

⟨*Lambertian Methods*⟩+≡
```
Float Pdf(const Vector &wi) const;
```

⟨*BRDF MC Methods*⟩+≡
```
Float Lambertian::Pdf(const Vector &wi) const {
    Float costheta = Dot(wi, N);
    if (costheta > 0.) {
        return 1. / (M_PI * costheta);
    } else {
        return 0.;
    }
}
```

And here is the skeleton for the `BlinnGlossy` pdf.

⟨*BlinnGlossy Methods*⟩+≡
```
Float Pdf(const Vector &wi) const;
```

⟨*BRDF MC Methods*⟩+≡
```
Float BlinnGlossy::Pdf(const Vector &wi) const {
    // YOUR CODE HERE
    return 0.;
}
```

For scattering mixtures, the pdf is the average of all of the constituent pdfs. This corresponds to the sub-optimal approach of using constant weighting functions in Veach's multiple importance sampling framework.

⟨*ScatteringMixture Methods*⟩+≡
```
Float Pdf(const Vector &wi) const;
```

⟨*BRDF MC Methods*⟩+≡
```
Float ScatteringMixture::Pdf(const Vector &wi) const {
    Float pdfSum = 0.;
    for (u_int i = 0; i < funcs.size(); ++i)
        pdfSum += funcs[i]->Pdf(wi);
    return pdfSum / funcs.size();
}
```

Here we'll just provide inert methods for the specular reflection and transmission scattering functions.

⟨*SpecularTransmission Methods*⟩+≡
```
Spectrum Sample(Float u[2], Vector *wi, Float *pdf) const
    { *pdf = 0.; return Spectrum(0.); }
Float Pdf(const Vector &wi) const { return 0.; }
```

⟨*SpecularReflection Methods*⟩+≡
```
Spectrum Sample(Float u[2], Vector *wi, Float *pdf) const
    { *pdf = 0.; return Spectrum(0.); }
Float Pdf(const Vector &wi) const { return 0.; }
```

Now we need to have similar methods for the light sources. Given an outgoing ray from a point being shaded (in world space), they should return the probability density of sampling the point on the light where that ray intersects the light. If the ray doesn't intersect the light geometry, the returned pdf should be zero; otherwise it should be the uniform probability density of sampling a point on the light from Section 13.4. The cosine of the angle that the ray makes with the surface of the light at the hit point should be stored in `costhetao`, and the *world space* location of the hit point should be stored in `Plight`.

⟨*Primitive Interface*⟩+≡
```
virtual Float Pdf(const Ray &ray,
                  Float *costhetao,
                  Point *Plight) const;
```

Run-time-error-time if the user tries to make an area light out of anything other than the primitives we care about for this assignment.

⟨*Primitive Methods*⟩+≡
```
Float Primitive::Pdf(const Ray &, Float *, Point *) const {
    Severe("Pdf() method unimplemented for primitive");
    return 0.;
}
```

⟨*Sphere Interface*⟩+≡
```
Float Sphere::Pdf(const Ray &ray,
                  Float *costhetao,
                  Point *Plight) const;
```

⟨*Cylinder Interface*⟩+≡
```
Float Cylinder::Pdf(const Ray &ray,
                    Float *costhetao,
                    Point *Plight) const;
```

⟨*Cone Interface*⟩+≡
```
Float Cone::Pdf(const Ray &ray,
                Float *costhetao,
                Point *Plight) const;
```

⟨*Primitive Sampling Functions*⟩+≡
```
Float Sphere::Pdf(const Ray &ray,
                  Float *costhetao,
                  Point *Plight) const {
    // YOUR CODE HERE
    return 0.;
}
```

⟨*Primitive Sampling Functions*⟩+≡
```
Float Cylinder::Pdf(const Ray &ray,
                    Float *costhetao,
                    Point *Plight) const {
    // YOUR CODE HERE
    return 0.;
}
```

⟨*Primitive Sampling Functions*⟩+≡
```
Float Cone::Pdf(const Ray &ray,
                Float *costhetao,
                Point *Plight) const {
    // YOUR CODE HERE
    return 0.;
}
```

Integrator Implementation.

Now we can tie all this together an implement a Monte Carlo integrator. It should support sampling the BRDF only, sampling the light sources only, and multiple importance sampling with both the BRDF and the lights. It has the usual form as the rest of the integrators.

⟨*MC Integrator Methods*⟩≡
```
Spectrum MCIntegrator::Integrate(const Ray &ray, HitInfo *hitInfo,
        Float *hitDist, Float *alpha) const {
    if (scene->Intersect(ray, 1e-4, hitDist, hitInfo)) {
        ShadeContext shadeContext(hitInfo, -ray.D);
        Spectrum L(0.);
        *alpha = 1.;
        ⟨Compute emitted light if an area light source⟩
        ⟨Evaluate BRDF and create surface shader⟩
        ⟨Do Monte Carlo lighting integration⟩
        ⟨Clean up from integration⟩
        return L;
    }
    else {
        *alpha = 0.;
        return Spectrum(0.);
    }
}
```

Now on to the part to be implemented. Depending on which integration method was specified for the primitive that we hit, the appropriate method should be used.

One final caveat for the combination integrator that implements multiple importance sampling: the BRDFs return pdfs with respect to solid angle, and the light sources return pdf with respect to area. The two need to be reconciled before the weighting functions can be computed. In particular, to convert a pdf expressed in terms of area to one in terms of solid angle, multiply by:

$$\frac{r^2}{\cos\theta_i \cos\theta_o}$$

where θ_i is the angle between the outgoing ray and the normal at the point being shaded, θ_o is the angle between the ray leaving the light source and the light's surface normal, and r^2 is the distance between the point on the light and the point being shaded.

⟨*Do Monte Carlo lighting integration*⟩≡

```
if (hitInfo->hitPrim->attributes->Sampling ==
    PrimitiveAttributes::SampleSurface) {
    // YOUR CODE HERE: integrate by sampling the BRDF
}
else if (hitInfo->hitPrim->attributes->Sampling ==
         PrimitiveAttributes::SampleLight) {
    // YOUR CODE HERE: integrate by sampling the Light source
    // for each light...
    //    for i = 0 to light->NumSamples()
    //        compute (estimate of rendering equation) / NumSamples
}
else if (hitInfo->hitPrim->attributes->Sampling ==
         PrimitiveAttributes::SampleCombination) {
    // YOUR CODE HERE: integrate by sampling a little bit
    // from both and then combining the results intelligently
}
```

14.Summary and Conclusion

A. Utilities

We will now define some routines that will be generally useful throughout the rest of the system. First is a set of routines for error reporting; these are used for things ranging from reporting invalid input from the user to reporting fundamental bugs in `lrt`. By gathering all error reporting in a single place, we make it easy to change how errors of various severities are handled. Next are routines for gathering statistics about the performance of the ray-tracer. Again, by gathering all of this data through a common set of interfaces, it's easy to adjust the detail of statistics reported to the user. Finally are a set of miscellaneous short mathematical functions; these provide some primitive operations that have wide application.

A.1 The C++ standard library

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` and `list` classes from the C++ standard library are parametrized container classes. `vector` is similar to an array, though it can automatically grow as items are added to it. `list` is a doubly-linked list class. As they are template classes, a vector of `int`s (for example) is declared as `vector<int> v;`.

To add a new item to a vector or list, a `push_back` method is available:

```
vector<int> vec;
for (int i = 0; i < 10; ++i)
    vec.push_back(i);
```

We can't say `vec[i] = i` in the above loop, since the vector needs to be informed that the user needs it to grow bigger, so that space may need to be allocated if needed. The `list` class also supports `push_back`, as well as a `push_front` call that adds the item to the head of the list. `vector`s do *not* support `push_front`, since this would be an expensive operation for a long vector, since all of the items already in the vector would need to be moved ahead one space.

A useful operation supported by vectors (but not lists) 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.)

However, we can quickly access the first and last items in a list, using the front and back methods, respectively. Similarly, pop_front and pop_back remove the first and last items from the list.

Items in lists and vectors are accessed through a general mechanism based on *iterators*. An iterator acts like a pointer to an item in a list or vector sequence. Just like a normal pointer, it can be dereferenced, using the * or -> operators, and it can be incremented and decremented to move through the items in the sequence. Each sequence supports a begin method, which returns an iterator pointing at the start of the sequence, and an end method, which points one past the end. Putting this all together, we can walk through the entries in a list, printing out each one like this:

```
list<int> l;
list<int>::iterator iter = l.begin();
while (iter != l.end()) {
    printf ("%d\n", *iter);
    ++iter;
}
```

Both container classes provide a size method, which returns the total number of items inside of them. For the vector class, this can be used in conjunction with the [] operator to access items in the vector directly, rather than using iterators:

```
for (int i = 0; i < vec.size(); ++i)
    printf ("%d\n", vec[i]);
```

After a vector has been filled (e.g. with push_back), its members an be modified with the [] operator as well. lists do not provide this method of accessing their items, since the lookup would take $O(n)$ time, rather than $O(1)$ as it does with vectors.

Both sequences 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,

```
l.erase(l.begin(), l.end());
```

empties a list completely.

Finally, the pair template class will be occasionally used; it provides a convenient way to construct a new object that holds two other objects. For example, if we're filling a hash table and are storing an array of pointers to hashed objects Foo with their integer hash values, we might declare an array of pair<Foo *, int>. Given a variable p that is a pair of objects, the constituent objects can be accessed as p.first and p.second. We can create a pair object with the make_pair function:

```
int i = 0, Foo *foop = NULL;
pair<Foo *, int> p = make_pair(foop, i);
p.first = new Foo;
```

A.2 Error Reporting

We provide four functions for reporting error conditions. In increasing severity, they are `Info`, `Warning`, `Error`, and `Severe`. All of them take a formatting string as their first argument and then a variable number of arguments providing values for the format. The syntax is identical to that used by the `printf` family of functions. For example,

```
Info("Now tracing ray number %d\n", rayNum);
```

Some compilers have non-portable ways of indicating that particular functions take a formatting string like `printf` with a variable number of arguments. These compilers can then verify that the types of the extra arguments after the formatting string are appropriate for the format. Thus, code like:

```
int FrameNum;
Info("Finished rendering frame number %f\n", FrameNum);
```

can be properly flagged as incorrect, since the formatting string indicates that `FrameNum` is a `double`, while it is actually an `int`. We define `PRINTF_FORMAT` here depending on which compiler is being used; for those where it's not possible to enable this type of syntax check, `PRINTF_FORMAT` just has an empty definition.

⟨*Setup printf format*⟩≡
```
#ifdef __GNUG__
#define PRINTF_FORMAT __attribute__ ((__format__ (__printf__, 1, 2)))
#else
#define PRINTF_FORMAT
#endif // __GNUG__
```

Now we can declare the four error reporting functions, using `PRINTF_FORMAT` if available.

⟨*Global Function Declarations*⟩+≡
```
⟨Setup printf format⟩
extern void Info(const char *, ...) PRINTF_FORMAT;
extern void Warning(const char *, ...) PRINTF_FORMAT;
extern void Error(const char *, ...) PRINTF_FORMAT;
extern void Severe(const char *, ...) PRINTF_FORMAT;
```

Because all four of these functions do almost the same thing—first format the error string and then do something with it—all of them call a common function, passing along the error information from the user as well as information about what to do with the message. It may be ignored, in which case the message is discarded; it may be printed and then program execution may continue, or it may be an error of such severity that it's impossible to go on and the program must abort.

⟨*Error Reporting Definitions*⟩≡
```
#define ERROR_IGNORE 0
#define ERROR_CONTINUE 1
#define 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 `ERROR_IGNORE`, `ERROR_CONTINUE`, or `ERROR_ABORT`.

⟨*Error Reporting Functions*⟩≡
```
static void processError(const char *format, va_list args,
                         const char *message, int disposition) {
    ⟨Format error string⟩
    ⟨Report error⟩
}
```

First we need to take the formatting string and the additional arguments passed by the user giving values to be substituted in the formatting string and turn it into a new string with those substitutions performed. Thankfully, the `vsprintf` function in the standard C library takes care of this for us.

⟨*Format error string*⟩≡
```
#define ERR_BUF_SZ 1024
static char errorBuf[ERR_BUF_SZ];
vsprintf(errorBuf, format, args);
```

Now that we have the error message in `errorBuf`, we print it or not, and exit the program it the error was a big one.

⟨*Report error*⟩≡
```
switch (disposition) {
case ERROR_IGNORE:
    return;
case ERROR_CONTINUE:
    fprintf(stderr, "%s: %s\n", message, errorBuf);
    break;
case ERROR_ABORT:
    fprintf(stderr, "%s: %s\n", message, errorBuf);
    abort();
}
```

We can now define the four globally-visible error functions. All are identical, except for how they prefix the error message and how it is disposed of. `Severe` is the only one that aborts execution; code that calls the other error reporting functions must therefore be able to recover from any error that is reported by `Error`, etc. These functions are quite straightforward. They use the standard C functions for getting ready to process a variable number of function arguments; after `va_start` is called, the `args` variable encapsulates information about the remaining arguments to the function. However, rather than calling the `va_arg` function to examine the subsequent arguments, we just pass the `args` variable into `processError`. It then passes it in to `vsprintf`, which handles unpacking the arguments.

⟨*Error Reporting Functions*⟩+≡
```
void Info(const char *format, ...) {
    va_list args;
    va_start(args, format);
    processError(format, args, "Notice", ERROR_CONTINUE);
    va_end(args);
}
```

⟨*Error Reporting Functions*⟩+≡
```
void Warning(const char *format, ...) {
    va_list args;
    va_start(args, format);
    processError(format, args, "Warning", ERROR_CONTINUE);
    va_end(args);
}
```

⟨*Error Reporting Functions*⟩+≡
```
void Error(const char *format, ...) {
    va_list args;
    va_start(args, format);
    processError(format, args, "Error", ERROR_CONTINUE);
    va_end(args);
}
```

⟨*Error Reporting Functions*⟩+≡
```
void Severe(const char *format, ...) {
    va_list args;
    va_start(args, format);
    processError(format, args, "Fatal Error", ERROR_ABORT);
    va_end(args);
}
```

We also define our own version of the standard `assert` macro. This asserts that an expression's value is true; if not, `Severe` is called with information about where the assertion failed.

⟨*Global Inline Functions*⟩≡
```
#ifdef NDEBUG
#define Assert(expr) ((void)0)
#else
#define Assert(expr) \
    ((expr) ? (void)0 : Severe("Assertion " #expr " failed in %s, line %d", \
                               __FILE__, __LINE__))
#endif // NDEBUG
```

A.3 Statistics

We also provide a unified interface for gathering statistics. This way, various parts of the program call into a single point where they can register what sorts of statistics they will be recording. At program termination, a single function call causes all such statistics to be printed out.

Two types of statistics can be gathered:

- *Counters*: These provide a way to count the frequency of something—e.g. the total number of rays that are traced while making an image.
- *Ratios*: This records the ratio of the frequency of two events—e.g. the number of successful ray-triangle intersection tests versus the total number of ray-triangle intersection tests.

We differentiate between two types of statistics: basic statistics that should always be printed at the end of rendering, and detailed statistics that should only be printed when the user expressly asks for all of them. The detailed statistics are likely to only be useful to a programmer who understands something of the workings of the system.

⟨*Global Constants*⟩+≡
```
#define STATS_NONE 0
#define STATS_BASIC 1
#define STATS_DETAILED 2
```

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.

⟨*Statistics Definitions*⟩≡
```
struct StatTracker {
    StatTracker(const char *cat, const char *n, int l,
                int *pa, int *pb = NULL);
    ~StatTracker();

    char *category;
    char *name;
    int level;
    int *ptra, *ptrb;
};
```

To construct a `StatTracker`, then, we just copy the strings the user passed in to us and store the appropriate pointers.

⟨*Statistics Functions*⟩≡
```
StatTracker::StatTracker(const char *cat, const char *n, int l,
                         int *pa, int *pb) {
    category = Strdup(cat);
    name = Strdup(n);
    level = l;
    ptra = pa;
    ptrb = pb;
}
```

Destroying a `StatTracker` just involves freeing up the memory that `Strdup` allocated.

⟨*Statistics Functions*⟩+≡
```
StatTracker::~StatTracker() {
    delete[] category;
    delete[] name;
}
```

All of the `StatTracker`s are stored in a `static vector`.

⟨*Statistics Definitions*⟩+≡
```
static vector<StatTracker *> trackers;
```

We'll define a short function that takes care of adding a `StatTracker` to the `trackers` array; it first looks through all of the already-registered `StatTracker`s and makes sure that this isn't a duplicate. If it is, an error message is printed and it isn't added again. The caller should ensure that each statistic is only reported to the statistics system once.

⟨*Statistics Definitions*⟩+≡
```
static void addTracker(StatTracker *newTracker) {
    for (u_int i = 0; i < trackers.size(); ++i) {
        if (strcmp(newTracker->category, trackers[i]->category) == 0 &&
            strcmp(newTracker->name, trackers[i]->name) == 0) {
                Error("Statistic %s/%s multiply reported. Discarding.",
                    newTracker->category, newTracker->name);
                return;
        }
    }
    trackers.push_back(newTracker);
}
```

This brings us to the declaration and definition of `StatsRegisterCounter`; the counter is passed in via the `ptr` parameter.

⟨*Global Function Declarations*⟩+≡
```
extern void StatsRegisterCounter(int level, const char *category,
                                 const char *name, int *ptr);
```

The `StatsRegisterCounter` function is quite simple. We just create a new `StatsTracker` and hand it off to `addTracker`.

⟨*Statistics Functions*⟩+≡
```
void StatsRegisterCounter(int level, const char *category,
                          const char *name, int *ptr) {
    addTracker(new StatTracker(category, name, level, ptr));
}
```

When we are notified of a ratio to track, we are given two `int` pointers–one for each of the two events. We store both of them in the `StatTracker` and add it to the `trackers` array.

⟨*Global Function Declarations*⟩+≡
```
extern void StatsRegisterRatio(int level, const char *category,
                               const char *name, int *pa, int *pb);
```

⟨*Statistics Functions*⟩+≡
```
void StatsRegisterRatio(int level, const char *category,
                        const char *name, int *pa, int *pb) {
    addTracker(new StatTracker(category, name, level, pa, pb));
}
```

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`. A `level` is passed in as well; this gives the maximum level of statistic to be printed (see `STATS_BASIC` and `STATS_DETAILED` above.)

⟨*Global Function Declarations*⟩+≡
```
extern void StatsPrint(int level, FILE *dest);
```

⟨*Statistics Functions*⟩+≡
```
void StatsPrint(int level, FILE *dest) {
    if (level == STATS_NONE) return;
    fprintf(dest, "Statistics (%s)\n",
        level == STATS_BASIC ? "Basic" : "Detailed");
    for (u_int i = 0; i < trackers.size(); ++i) {
        if (trackers[i]->level <= level) {
            ⟨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*⟩≡
```
fprintf(dest, "%s/%s", trackers[i]->category, trackers[i]->name);
⟨Pad out to results column⟩
if (trackers[i]->ptrb == NULL)
    fprintf(dest, "%d\n", *trackers[i]->ptra);
else {
    if (*trackers[i]->ptrb > 0) {
        Float ratio = (Float)*trackers[i]->ptra / (Float)*trackers[i]->ptrb;
        fprintf(dest, "%d:%d (%f%%)\n", *trackers[i]->ptra, *trackers[i]->ptrb,
                100. * ratio);
    }
    else
        fprintf(dest, "%d:%d\n", *trackers[i]->ptra, *trackers[i]->ptrb);
}
```

After printing the name and category, we print enough spaces so that all of the statistic values start in the column `resultsColumn`.

⟨*Pad out to results column*⟩≡
```
int textLength = strlen(trackers[i]->category) + 1 + strlen(trackers[i]->name);
int resultsColumn = 52;
int paddingSpaces = resultsColumn - textLength;
while (paddingSpaces--)
    putc(' ', dest);
```

When the program is freeing up memory when it's about to exit, it can call the `StatsCleanup` function, which frees the `StatsTracker`s that we've created.

⟨*Global Function Declarations*⟩+≡
```
extern void StatsCleanup();
```

⟨*Statistics Functions*⟩+≡
```
void StatsCleanup() {
    ⟨Reset user-supplied statistics pointers⟩
    for (u_int i = 0; i < trackers.size(); ++i)
        delete trackers[i];
    trackers.erase(trackers.begin(), trackers.end());
}
```

We reset the various counter pointers that the user gave us to zero before we destroy the trackers; this way, if the renderer runs again before the program exits, all of the various statistics will start counting from zero again.

⟨*Reset user-supplied statistics pointers*⟩≡
```
for (u_int i = 0; i < trackers.size(); ++i) {
    trackers[i]->ptra = 0;
    if (trackers[i]->ptrb)
        trackers[i]->ptrb = 0;
}
```

A.4 Matrix Inversion

⟨*Global Function Declarations*⟩+≡
```
void MatrixInvert( Float m[4][4], Float minv[4][4] );
```

⟨*Matrix Functions*⟩≡
```
void MatrixInvert( Float m[4][4], Float minv[4][4] ) {
    int indxc[4], indxr[4];
    int ipiv[4] = { 0, 0, 0, 0 };

    memcpy( minv, m, 4*4*sizeof(Float) );

    for (int i = 0; i < 4; i++) {
        int irow, icol;
        Float big = 0.;
        ⟨Choose pivot⟩
        ++ipiv[icol];
        ⟨Swap rows irow and icol for pivot⟩
        ⟨Set m[icol][icol] to one by scaling row icol appropriately⟩
        ⟨Subtract this row from others to zero out their columns⟩
    }
    ⟨Swap columns to reflect permutation⟩
}
```

⟨*Choose pivot*⟩≡
```
for (int j = 0; j < 4; j++) {
    if (ipiv[j] != 1) {
        for (int k = 0; k < 4; k++) {
            if (ipiv[k] == 0) {
                if (fabs(minv[j][k]) >= big) {
                    big = Float(fabs(minv[j][k]));
                    irow = j;
                    icol = k;
                }
            }
            else if (ipiv[k] > 1)
                Error("Singular matrix in MatrixInvert");
        }
    }
}
```

⟨*Swap rows* irow *and* icol *for pivot*⟩≡
```
if (irow != icol) {
    for (int k = 0; k < 4; ++k)
        swap(minv[irow][k], minv[icol][k]);
}
indxr[i] = irow;
indxc[i] = icol;
if (minv[icol][icol] == 0.)
    Error("Singular matrix in MatrixInvert");
```

⟨*Set m*[*icol*][*icol*] *to one by scaling row* icol *appropriately*⟩≡
```
Float pivinv = 1. / minv[icol][icol];
minv[icol][icol] = 1.;
for (int j = 0; j < 4; j++)
    minv[icol][j] *= pivinv;
```

⟨*Subtract this row from others to zero out their columns*⟩≡
```
for (int j = 0; j < 4; j++) {
    if (j != icol) {
        Float save = minv[j][icol];
        minv[j][icol] = 0;
        for (int k = 0; k < 4; k++)
            minv[j][k] -= minv[icol][k]*save;
    }
}
```

⟨*Swap columns to reflect permutation*⟩≡
```
for (int j = 3; j >= 0; j--) {
    if (indxr[j] != indxc[j]) {
        for (int k = 0; k < 4; k++)
            swap(minv[k][indxr[j]], minv[k][indxc[j]]);
    }
}
```

A.5 Miscellaneous 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 `pos` parameter. When `pos` is zero, the result is `start`; when `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 `Lerp` returns *exactly* the values `start` and `end` when `pos` has values 0 and 1, respectively, with our imlementation. This isn't necessarily the case with the other formulation, again due to floating point roundoff.

⟨*Global Inline Functions*⟩+≡
```
inline Float Lerp(Float pos, Float start, Float end) {
    return (1. - pos) * start + pos * end;
}
```

`Clamp` clamps a value `val` to be between the values `low` and `high`. If `val` is out of that range, `low` or `high` is returned as appropriate.

⟨*Global Inline Functions*⟩+≡
```
inline Float Clamp(Float val, Float low, Float high) {
    if (val < low) return low;
    else if (val > high) return high;
    else return val;
}
```

A simple function that can make expressions more clear is `Step`; if takes a point at which the step happens and a value and returns zero if the value is less than the position and a one if it's greater than the position. Thus, conditionals can be cleanly folded into expressions as:

```
Float cutoff = ComputeCutoff(x);
return 2 + Step(cutoff, 3 * CurrentValue() / 2.) * M_PI;
```

Instead of needing to write

```
Float cutoff = ComputeCutoff(x);
if (cutoff < 3 * CurrentValue() / 2)
    return 2 + Step(cutoff, 3 * CurrentValue() / 2.) * M_PI;
else
    return 2
```

⟨*Global Inline Functions*⟩+≡

```
inline int Step(Float stepPos, Float val) {
    if (val < stepPos) return 0;
    else               return 1;
}
```

Another useful related function is `SmoothStep`; it takes a minimum and maximum value and a point at which to evaluate the step function. If the point is below the minimum, zero is returned, and if it's above the maximum, one is returned. Otherwise it smoothly interpolates between zero and one.

⟨*Global Inline Functions*⟩+≡

```
inline Float SmoothStep(Float min, Float max, Float value) {
    Float v = Clamp((value - min) / (max - min), 0., 1.);
    return -2. * v * v * v + 3. * v * v;
}
```

`Round` rounds a floating point value to the nearest integer, using the built in math library function `rint`.

⟨*Global Inline Functions*⟩+≡

```
inline int Round(Float val) {
    return (int)rint(val);
}
```

`Mod` computes the remainder of a/b. This function is handy since it behaves predictably and reasonably with negative numbers—the C and C++ standards leave the behavior of the `%` operator undefined in that case.

⟨*Global Inline Functions*⟩+≡

```
inline int Mod(int a, int b) {
    int n = int(a/b);
    a -= n*b;
    if (a < 0)
        a += b;
    return a;
}
```

Finally, simple functions that compute the minimum or maximum of two values and a function that swaps the values of two variables. We just use the appropriate functions provided by the standard C++ library.

⟨*Global Include Files*⟩+≡
```
#include <algorithm>
using std::min;
using std::max;
using std::swap;
```

Unfortunately, not all system `math.h` files store the value of π in `M_PI`. If it is not defined, we do it ourself.

⟨*Global Constants*⟩+≡
```
#ifndef M_PI
#define M_PI           3.14159265358979323846
#endif
```

⟨*Global Constants*⟩+≡
```
#define INV_255 .00392156862745098039
```

We define a generally-useful `INFINITY` value using `HUGE_VAL` from the standard math library, which is the largest representable floating point number.

⟨*Global Constants*⟩+≡
```
#define INFINITY HUGE_VAL
```

Two simple functions convert from angles expressed in degrees to radians, and vice versa.

⟨*Global Inline Functions*⟩+≡
```
inline Float Radians(Float deg) { return (M_PI/180.) * deg; }
inline Float Degrees(Float rad) { return (180./M_PI) * rad; }
```

Finally, a string operation. A commonly-performed string operation is the duplication of an existing string; `Strdup` takes a pointer to a C-style NULL terminated string and allocates space for a new string, into which it copies the original string.

⟨*Global Inline Functions*⟩+≡
```
inline char *Strdup(const char *str) {
    char *ret = new char[strlen(str) + 1];
    strcpy(ret, str);
    return ret;
}
```

A.6 Random Numbers

We will provide a pseudo-random number generation function for the system. This is useful because it allows us to ensure that the system produces the same results regardless of machine architecture and C library implementation. This is particularly helpful since many systems provide random number generation routines with poor statistical distributions.

The random number generate we choose is the "Mersenne Twister" by Makoto Matsumoto and Takuji Nishimura. The code to the random number generater is very involved and complex, and we will not present it here. Pointers to details on the algorithm can be found at the end of this section. Their algorithm provides two functions, `sgenrand` and `genrand`. The former is used to seed the random number generator, and the second returns a random `double` between zero and one.

⟨*Global Function Declarations*⟩+≡
```
extern Float RandomFloat(Float min = 0., Float max = 1.);
```

⟨*Random Number Functions*⟩+≡
```
Float RandomFloat(Float min, Float max) {
    return Lerp(genrand(), min, max);
}
```

A.7 Reference Counted Objects

Sometimes we will have objects where many other objects hold pointers to them. In order to simplify the process of keeping track of these shared objects and knowing when we can free their memory, we provide a `ReferenceCounted` class that these shared objects can inherit from. This keeps a track of how many references there are to the object.

⟨*ReferenceCounted Protected Interface*⟩≡
```
ReferenceCounted() { References = 1; }
```

⟨*ReferenceCounted Private Data*⟩≡
```
mutable int References;
```

When another object makes a copy of the pointer, it should call the `Reference` method of the object that it's sharing. Similarly, when it's done with the pointer, it should call the `Dereference` method.

⟨*ReferenceCounted Interface*⟩≡
```
void Reference() const { ++References; }
void Dereference() const { if (--References == 0) delete this; }
```

⟨*ReferenceCounted Interface*⟩+≡
```
int GetReferences() const { return References; }
```

We'll make the destructor of the base class `protected` in an effort to prevent direct deletion of reference counted objects.

⟨*ReferenceCounted Protected Interface*⟩+≡
```
~ReferenceCounted() { }
```

A.8 Hash Tables

For certain operations it will be useful to have an efficient mapping from keys to data. We implement a simple hashtable that keys from strings to `void *`s.

⟨*Global Classes*⟩+≡
```
class StringHashTable {
public:
    ⟨StringHashTable Methods⟩
private:
    ⟨StringHashTable Private Data⟩
};
```

⟨*StringHashTable Private Data*⟩≡
```
static const int NUM_BUCKETS = 1047;
typedef list<pair<const char *, void *> > ItemType;
ItemType buckets[NUM_BUCKETS];
```

⟨*StringHashTable Methods*⟩≡
```
u_int Hash(const char *str) const;
```

⟨*StringHashTable Method Definitions*⟩≡
```
u_int StringHashTable::Hash(const char *str) const {
    u_int hashValue = 0;
    while (*str) {
        hashValue <<= 1;
        ++str;
    }
    return hashValue % NUM_BUCKETS;
}
```

⟨*StringHashTable Methods*⟩+≡
```
void *Search(const char *key) const;
```

⟨*StringHashTable Method Definitions*⟩+≡
```
void *StringHashTable::Search(const char *key) const {
    u_int index = Hash(key);
    ItemType::const_iterator iter;
    for (iter = buckets[index].begin(); iter != buckets[index].end(); iter++) {
        if(strcmp(key, iter->first) == 0)
            return iter->second;
    }
    return NULL;
}
```

⟨*StringHashTable Methods*⟩+≡
```
void Add(const char *key, void *data);
```

⟨*StringHashTable Method Definitions*⟩+≡

```
void StringHashTable::Add(const char *key, void *data) {
    u_int index = Hash(key);
    ItemType::iterator iter = buckets[index].begin();
    while (iter != buckets[index].end()) {
        if (strcmp(key, iter->first) == 0)
            buckets[index].erase(iter);
        ++iter;
    }
    buckets[index].push_front(make_pair(strdup(key), data));
}
```

Further Reading

Detailed information about the random number generator we are using, including the original paper from ACM Transactions on Modelling and Computer Simulation are available at
`http://www.math.keio.ac.jp/~matumoto/emt.html`.

B. TIFF Input and Output

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

⟨*Global Function Declarations*⟩+≡

```
extern Spectrum *TIFFRead(const char *name, int *xSize, int *ySize);
```

The function that reads TIFFs deals with three main types of TIFFs:

- Standard RGB eight-bit per pixel TIFFs.
- TIFFs with *colormaps*: an array of RGB colors where each pixel is represented by an index into the array (this can reduce storage needs when there are a small number of colors in the image).
- TIFFs with floating-point RGB pixel values.

We always return the pixel data as a `Spectrum` array that `TIFFRead()` allocates. This can be a wasteful representation; for an eight-bit per pixel TIFF, this is four times bigger than the original data. However, it greatly simplifies our task.

⟨*TIFF Function Definitions*⟩≡
```
Spectrum *TIFFRead(const char *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*⟩≡
```
TIFF *tiff = TIFFOpen(name, "r");
if (!tiff) {
    Error("Unable to open TIFF %s", name);
    return NULL;
}
```

We first determine the resolution of the TIFF and the number of samples per pixel.

⟨*Get basic information from TIFF header*⟩≡
```
short int nSamples;
TIFFGetField(tiff, TIFFTAG_IMAGEWIDTH, xSize);
TIFFGetField(tiff, TIFFTAG_IMAGELENGTH, ySize);
TIFFGetField(tiff, TIFFTAG_SAMPLESPERPIXEL, &nSamples);
```

Now things get a little complicated. We find out how many bits each sample has and in what format they're stored in. We require that either each sample is 32 bits wide and stored as a floating point value, *or* that it's 8 bits and stored as unsigned integer values. If the above is not true, we head to the fragment ⟨*Clean up after TIFF reading error*⟩, which will clean up any memory that's been allocated, close the file, and return `NULL`.

Finally we make sure that the RGB samples are interleaved (that is, as RGBRGBRGB along a scanline.) TIFFs also support images where each of the channels is stored in a separate contiguous part of the file; we don't support these.

⟨*Make sure this is a TIFF we can read*⟩≡

```
short int bitsPerSample, sampleFormat;
if (!TIFFGetField(tiff, TIFFTAG_BITSPERSAMPLE, &bitsPerSample)) {
    Error("TIFFRead: bits per sample not set in TIFF");
    ⟨Clean up after TIFF reading error⟩
}
if (!TIFFGetField(tiff, TIFFTAG_SAMPLEFORMAT, &sampleFormat)) {
    if (bitsPerSample == 32)
        sampleFormat = SAMPLEFORMAT_IEEEFP;
    else
        sampleFormat = SAMPLEFORMAT_UINT;
}

if (bitsPerSample == 32 && sampleFormat != SAMPLEFORMAT_IEEEFP) {
    Error("TIFFRead: 32 bit TIFF not stored in floating point format");
    ⟨Clean up after TIFF reading error⟩
}
else {
    if (bitsPerSample != 8) {
        Error("TIFFRead: more than 8 bits per sample unsupported");
        ⟨Clean up after TIFF reading error⟩
    }
    if (sampleFormat != SAMPLEFORMAT_UINT) {
        Error("TIFFRead: 8 bit TIFFs must be stored as unsigned ints");
        ⟨Clean up after TIFF reading error⟩
    }
}

if (nSamples * *xSize != TIFFScanlineSize(tiff)) {
    Error("TIFFRead: RGB not interleaved in TIFF %s", name);
    ⟨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 `PHOTOMETRIC` field doesn't indicate that there is a palette (aka colormap) stored with the image, we just give up, saving the code for that obscure case. If it is there, we store pointers to the colormap in `mapR`, `mapG`, and `mapB`.

⟨*Read TIFF colormap if present*⟩≡
```
u_short *mapR = 0, *mapG = 0, *mapB = 0;
if (nSamples == 1) {
    short photoType;
    TIFFGetField(tiff, TIFFTAG_PHOTOMETRIC, &photoType);
    if (photoType != PHOTOMETRIC_PALETTE) {
        Error("TIFFRead: colormap not found in one-sample image");
        ⟨Clean up after TIFF reading error⟩
    }
    TIFFGetField(tiff, TIFFTAG_COLORMAP, &mapR, &mapG, &mapB);
}
```

Now we can allocate space for the resulting pixels and for buffers for reading the image. We allocate `ubuf` or `fbuf` as appropriate for the format of the image that we're reading.

⟨*Allocate space for* `pixels` *and buffers*⟩≡
```
pixels = new Spectrum[*xSize * *ySize];
Spectrum *pixelp = pixels;

if (bitsPerSample == 32) fbuf = new float[nSamples * *xSize];
else                     ubuf = new u_char[nSamples * *xSize];
```

⟨*Read a TIFF scanline*⟩≡
```
if (fbuf) {
    ⟨Read floating point TIFF scanline⟩
}
else {
    ⟨Read 8-bit TIFF scanline⟩
}
```

We read the scanline into `fbuf` and then copy it into `pixels`. Because we're reading the image from top-to-bottom, we end up going through `pixels` in order from start to end. Thus, we just increment `pixelp` after each pixel is processed. We also keep a pointer into the data read from the file, `fbufp`; this is incremented by `nSamples` after each pixel to get to the next pixel.

⟨*Read floating point TIFF scanline*⟩≡
```
float *fbufp = fbuf;
if (TIFFReadScanline(tiff, fbuf, y, 1) == -1) {
    ⟨Clean up after TIFF reading error⟩
}
for (int x = 0; x < *xSize; ++x) {
    *pixelp = Spectrum(fbufp[0], fbufp[1], fbufp[2]);
    ++pixelp;
    fbufp += nSamples;
}
```

Similarly, we can do similar tricks with `ubuf` and `ubufp` when reading eight-bit TIFFs.

⟨*Read 8-bit TIFF scanline*⟩≡
```
u_char *ubufp = ubuf;
if (TIFFReadScanline(tiff, ubuf, y, 1) == -1) {
    ⟨Clean up after TIFF reading error⟩
}
for (int x = 0; x < *xSize; ++x) {
    if (nSamples == 1) {
        ⟨Decode TIFF colormap entry⟩
    }
    else {
        ⟨Convert standard 8-bit TIFF pixel⟩
    }
    ++pixelp;
    ubufp += nSamples;
}
```

If there is a colormap, we just use the sample value to index into the colormap for each of red, green, and blue. We scale by 1./255. so that the returned image values lie between zero and one.

⟨*Decode TIFF colormap entry*⟩≡
```
int mapOffset = *ubufp;
*pixelp = Spectrum(mapR[mapOffset] * INV_255 * INV_255,
    mapG[mapOffset] * INV_255 * INV_255, mapB[mapOffset] * INV_255 * INV_255);
```

And reading a normal pixel is easy; we just need to scale by 1./255..

⟨*Convert standard 8-bit TIFF pixel*⟩≡
```
*pixelp = Spectrum(ubufp[0] * INV_255, ubufp[1] * INV_255, ubufp[2] * INV_255);
```

⟨*Close TIFF and return*⟩≡
```
delete[] ubuf;
delete[] fbuf;
TIFFClose(tiff);
return pixels;
```

⟨*Clean up after TIFF reading error*⟩≡
```
delete[] pixels;
delete[] ubuf;
delete[] fbuf;
TIFFClose(tiff);
return NULL;
```

B.2 Output

We will provide two functions for writing TIFF data: the difference between them is the format used for storing RGB 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 6.2).

The second format stores the RGB 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 **??** on page ??.) Unfortunately, few widely-used image display programs support floating point TIFF images.

⟨*Global Function Declarations*⟩+≡
```
extern void TIFFWrite8Bit(const char *name, Float *RGB, Float *alpha, Float *depth,
    int XRes, int YRes);
```

Both of the output routines have similar structures. The file is opened, individual scanlines of pixels are written, and the file is closed.

⟨*TIFF Function Definitions*⟩+≡
```
void TIFFWrite8Bit(const char *name, Float *RGB, Float *alpha, Float *depth,
    int XRes, int YRes) {
    Assert(RGB);
    ⟨Open 8-bit TIFF file for writing⟩
    ⟨Write 8-bit scanlines⟩
    ⟨Close 8-bit TIFF file⟩
}
```

Actually, we should use `TIFFSetErrorHandler()` and `TIFFSetWarningHandler()` and funnel that stuff to our own warning/error routines.

After opening the image (similarly to the `fopen()` call), we set a variety of flags which tell the library exactly what kind of TIFF we're going to give it, how to encode the samples, etc. Most of these should be reasonably self-explanatory. See the TIFF documentation (XXX URL?) for a full explanation.

⟨*Open 8-bit TIFF file for writing*⟩≡
```
TIFF *tiff = TIFFOpen(name, "w");
if (!tiff) {
    Error("Unable to open TIFF %s for writing", name);
    return;
}

⟨Compute and set up samples per pixel⟩
TIFFSetField(tiff, TIFFTAG_IMAGEWIDTH, XRes);
TIFFSetField(tiff, TIFFTAG_IMAGELENGTH, YRes);
TIFFSetField(tiff, TIFFTAG_BITSPERSAMPLE, 8);
TIFFSetField(tiff, TIFFTAG_PHOTOMETRIC, PHOTOMETRIC_RGB);
⟨Set Generic TIFF Fields⟩
```

⟨*Compute and set up samples per pixel*⟩≡
```
int sampleCount = 0;
if (RGB) sampleCount += 3;
if (alpha) ++sampleCount;
if (depth) Warning("Depth channel to be discarded in TIFF"); // for now
TIFFSetField(tiff, TIFFTAG_SAMPLESPERPIXEL, sampleCount);
if (alpha) {
    short int extra[] = { EXTRASAMPLE_ASSOCALPHA };
    TIFFSetField(tiff, TIFFTAG_EXTRASAMPLES, (short)1, extra);
}
```

There are a few fields that are set the same way for both eight-bit and floating point TIFF files; we'll set them in a single fragment that can be shared.

⟨*Set Generic TIFF Fields*⟩≡
```
TIFFSetField(tiff, TIFFTAG_ROWSPERSTRIP, 1L);
TIFFSetField(tiff, TIFFTAG_XRESOLUTION, 1.0);
TIFFSetField(tiff, TIFFTAG_YRESOLUTION, 1.0);
TIFFSetField(tiff, TIFFTAG_RESOLUTIONUNIT, 1);
TIFFSetField(tiff, TIFFTAG_COMPRESSION, COMPRESSION_NONE);
TIFFSetField(tiff, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG);
TIFFSetField(tiff, TIFFTAG_ORIENTATION, (int)ORIENTATION_TOPLEFT);
```

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 `RGB` array linearly from start to finish, we traverse it scanline-by-scanline, from top-to-bottom–exactly the order that we're going to write it out in. Thus we can do pointer arithmetic with `pixelp` to go through the pixels.

⟨*Write 8-bit scanlines*⟩≡
```
u_char *buf = new u_char[sampleCount * XRes];
Float *pixelp = RGB, *alphap = alpha;

for (int y = 0; y < YRes; ++y) {
    u_char *bufp = buf;
    for (int x = 0; x < XRes; ++x) {
        ⟨Pack 8-bit RGB samples into buf⟩
        ⟨Pack 8-bit alpha samples into buf⟩
    }
    TIFFWriteScanline(tiff, buf, y, 1);
}
```

⟨*Pack 8-bit RGB samples into buf*⟩≡
```
if (pixelp)
    for (int s = 0; s < 3; ++s) {
        *bufp = (u_char)(*pixelp);
        ++bufp;
        ++pixelp;
    }
```

⟨*Pack 8-bit alpha samples into buf*⟩≡
```
if (alphap) {
    *bufp = (u_char)(*alphap);
    ++bufp;
    ++alphap;
}
```

⟨*Close 8-bit TIFF file*⟩≡
```
delete[] buf;
TIFFClose(tiff);
```

Writing out a floating point TIFF file is quite similar. The only differences are in some of the flags we set (which now say that it's a floating-point image), and how we write the data out.

⟨*Global Function Declarations*⟩+≡
```
extern void TIFFWriteFloat(const char *name, Float *RGB, Float *alpha, Float *depth,
    int XRes, int YRes);
```

⟨*TIFF Function Definitions*⟩+≡
```
void TIFFWriteFloat(const char *name, Float *RGB, Float *alpha, Float *depth,
    int XRes, int YRes) {
    if (alpha || depth)
        Warning("Alpha and/or depth channels to be discarded in TIFF");
    Assert(RGB);
    ⟨Open Float TIFF file for writing⟩
    ⟨Write Float scanlines⟩
    ⟨Close Float TIFF file⟩
}
```

⟨*Open Float TIFF file for writing*⟩≡
```
TIFF *tiff = TIFFOpen(name, "w");
if (!tiff) {
    Error("Unable to open TIFF %s for writing", name);
    return;
}

TIFFSetField(tiff, TIFFTAG_IMAGEWIDTH, XRes);
TIFFSetField(tiff, TIFFTAG_IMAGELENGTH, YRes);
TIFFSetField(tiff, TIFFTAG_SAMPLESPERPIXEL, 3);
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 typedef'd to `double`, then we would need to allocate a temporary buffer and convert to `float`, as we did above for `unsigned char`. We'll just assert that this hasn't happened, and write out the pixel data as given.

⟨*Write Float scanlines*⟩≡
```
Float *pixelp = RGB;
for (int y = 0; y < YRes; ++y) {
    TIFFWriteScanline(tiff, pixelp, y, 1);
    pixelp += 3 * XRes;  // ahead to the start of the next scanline
}
```

⟨*Close Float TIFF file*⟩≡
```
TIFFClose(tiff);
```

C. The RenderMan Interface

In this chapter we will describe our implementation of an external interface to `lrt`. One 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. Ideally, there are a variety of renderers that all support the same scene description method; this makes it easy for modelling programs, which then only need to know how to output scenes in a single manner.

For interactive graphics, the OpenGL interface has been successful in this regard. By providing a single interface to a variety of graphics hardware from a variety of vendors, OpenGL has made it possible to write interactive applications once and run them on many graphics accelerators. For printers, the PostScript language has had similar benefits, making it easy for applications to lay out pages of text and data without worrying about what particular vendor's printer is being used.

For high-end renderers, there has not been a single interface that has been as widely-adopted as PostScript or OpenGL. In the late 1980s, however, Pixar Animation Studios developed the *RenderMan Interface* (RI). RI is a general interface to rendering systems that was intended function like a 3D version of PostScript. In comparison to most other graphics interfaces such as OpenGL, RI is a higher-level scene description designed for high-quality rendering. Some of its basic features are:

- Direct support for a variety of curved surface representations.
- Flexible and powerful ways of describing complex materials and light sources.
- The ability to describe the motion of objects and the camera over time.

A variety of modellers support RI, making it an excellent API (*application programmer's interface*) for `lrt` to support. By not inventing a new scheme for providing data to the renderer, it's possible to use many scenes that have already been created with `lrt`. RI is currently supported by Pixar's *Photorealistic RenderMan* renderer; all of the data for the frames of Pixar's computer generated animated movies passed through the RenderMan interface on the way to being rendered. Larry Gritz's *Blue*

Moon Rendering Tools is a freely-downloadable raytracer that also supports RI; it provides a much more complete implementation of all of the nooks and crannies of the specification than `lrt`, though source code is not available.

For `lrt`, we will support a carefully chosen subset of RI. Various features are not supported for two main reasons:

- Pedagogical: we do not include some features that are very similar to others that are implemented, if there is little to be learned from seeing them. Furthermore, features that would take many pages to implement well but may not merit a treatment that detailed are omitted. For example, `lrt` doesn't support all of the geometric primitives described in the RenderMan specification (in particular, there is no support for spline-based curved surfaces, since we chose to support the more general paradigm of subdivision surfaces for describing curved surfaces.)
- Disagreement: some parts of RI are showing their age; for instance, some parts of the specification make it difficult to support physically-based rendering in the manner that we would like do. In these situations, we have tried to make the smallest-possible changes to the interface that solve the particular problems.

C.1 Underpinnings and Structure

The RenderMan interface includes two ways of communicating with the renderer: the procedural interface–a set of procedure calls that can be called from a program written in some language (such as C), and the *bytestream protocol* (RIB)–a text-based format. We support both, parsing RIB and turning it into corresponding RI procedural calls. This chapter describes most of the procedural interface; the RIB parser is presented separately.

The RenderMan interface starts by defining a set of basic types that will be used by the procedural interface. These types are all prefixed with *Rt*. The actual procedural interface names start with *Ri*.

⟨*RI Types*⟩≡
```
typedef short   RtBoolean;
typedef int     RtInt;
typedef float   RtFloat;
typedef char   *RtToken;
```

⟨*RI Types*⟩+≡
```
typedef RtFloat RtColor[3];
typedef RtFloat RtPoint[3];
typedef char   *RtString;
typedef void   *RtPointer;
typedef void    RtVoid;
```

RI matrices are the transpose of our matrix representation: we'll need to handle this later as they come into the system.

⟨*RI Types*⟩+≡
```
typedef RtFloat RtMatrix[16];
```

⟨*RI Constants*⟩≡
```
#define RI_FALSE    0
#define RI_TRUE     1
#define RI_INFINITY (RtFloat)1.0e38
#define RI_EPSILON  (RtFloat)1.0e-10
#define RI_NULL     NULL
```

Basic structure.

The rendering system is initialized by a call to `RiBegin()` (see Section C.4.) After this, general rendering options like the camera position and the image resolution can be set (Section **??** on page ??). `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.

Many of the RI procedures take a variable number of arguments, terminated by `RI_NULL`. Consider a function `RiFoo(RtInt n, ...)`: it always takes an integer argument `n`, which is then followed by a variable argument list holding user-supplied *named parameters* and their values. There are three ways to call `RiFoo()`. First, the user can supply a set of parameter names and their values. For parameters that describe a set of values (e.g. `pts`, below), the number of expected items in the array is determined by the semantics of the particular procedure. It is an error for the caller to not provide a pointer to a sufficient number of values.

```
RtPoint pts[2] = {{1, 2, 3}, {0, 0, 0}};
RtFloat f = 0.5;
RiFoo(512, (RtToken)"P", (RtPointer)pts, (RtToken)"Kd", (RtPointer)&f,
RI_NULL);
```

For each such function like `RiFoo(RiInt, ...)`, there is also a vector-based procedure call `RiFooV()` that takes three arguments in addition to the fixed arguments (if any). These three arguments are a count of the number of parameters, pointers to the tokens that name the parameters, and pointers to the values of the parameters. Section C.2 describes the token mechanism for naming procedure parameters in detail.

```
RtToken tokens[2];
RtPointer values[2];
tokens[0] = "P";
values[0] = (RtPointer)pts;
tokens[1] = "Kd";
values[1] = (RtPointer)&f;
RiFooV(512, 2, tokens, values);
```

Finally, all of the procedural interface calls have an equivalent in the bytestream protocol. The above `RiFoo()` calls correspond to the following RIB:

```
Foo 512 "P" [1 2 3 0 0 0] "Kd" [.5]
```

State tracking.

Because almost all of the RI calls are illegal before `RiBegin()` is called and because most of the others are only legal before or after `RiWorldBegin()`, we will provide some facilities for tracking what state the API is in. We use a module static variable `currentApiState`. It starts out with value `STATE_UNINITIALIZED` and is updated by `RiBegin()`, `RiWorldBegin()`, and `RiEnd()`.

⟨*API Static Data*⟩≡

```
#define STATE_UNINITIALIZED  0
#define STATE_BEGIN          1
#define STATE_WORLD_BEGIN    2
static int currentApiState = STATE_UNINITIALIZED;
```

Now, all RI procedures that are only valid in particular states call the VERIFY_STATE macro, passing the state that they expect us to be in as well as a string that is their procedure name. If the states don't match, we print an error message and return immediately from the function.

⟨*API Macros*⟩≡
```
#define VERIFY_STATE(s, func) \
    if (currentApiState != s) { \
        Error("Must have called %s before calling %s().  Ignoring.", \
            missingStateCall[s], func); \
        return; \
    } \
    else /* swallow trailing semicolon */
```

Through some array indexing trickery, we can take the expected state value s, and find the string name of the procedure that needs to be called before the current function can be used.

⟨*API Static Data*⟩+≡
```
static const char *missingStateCall[] = { "RiEnd()", "RiBegin()", "RiWorldBegin()" };
```

C.2 Tokens and Declarations

The RI specification defines a fair number of built-in parameters to the standard shaders, light sources, etc., each declared to have a particular type (e.g. point data) These are all initialized by lrt in Section D.1 on page 239 and variables holding their names are available (e.g. RI_P). In addition to all of these built-in parameters, 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 RtToken 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 RtToken is passed, rather than a const char * pointing to the name, better performance may be possible.

Tokens are stored in a small hash table. Strings are hashed to compute an offset into the table and then we walk through the list of tokens at that position to find the token being looked for. In addition to the string that is their name, we store an int with each one where we encode its type.

⟨*API Static Data*⟩+≡
```
#define TOKEN_HASH_SIZE 1024
static list<pair<RtToken, int> > TokenTable[TOKEN_HASH_SIZE];
```

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. (See Section 9.1 for a discussion of uniform versus vertex data.) A token can be declared repeatedly with no ill effect; however, it is an error to redeclare a token with a different type than was originally used to declare it.

⟨*RI Function Declarations*⟩≡
```
extern RtToken RiDeclare(char *name, char *type);
```

⟨*RI Function Definitions*⟩+≡
```
RtToken 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 `RI_NULL`.

⟨*Compute integer type code for variable*⟩≡
```
int tokenType = stringToType(type);
if (tokenType == TOKEN_TYPE_ERROR)
    return RI_NULL;
```

Now we'll look through the linked-list at the appropriate hash table position and see if a token with this name has already been declared. If so, we make sure that the user isn't redeclaring a variable with a new type, printing a warning if so.

⟨*Search for token in token table*⟩≡
```
u_int hashValue = hashTokenString(name);
list<pair<RtToken, int> >::iterator iter = TokenTable[hashValue].begin();
while (iter != TokenTable[hashValue].end()) {
    if (strcmp(iter->first, name) == 0) {
        ⟨Complain if token was redeclared to a different type⟩
        return iter->first;
    }
    ++iter;
}
```

We compute a basic hash function based on the string: for each character in the string, we shift the hash value left by one bit and then compute the exclusive or of the previous hash value with the character's value.

⟨*API Static Methods*⟩≡
```
u_int hashTokenString(const char *name) {
    u_int hashValue = 0;
    while (*name) {
        hashValue <<= 1;
        hashValue ^= *name;
        ++name;
    }
    return hashValue % TOKEN_HASH_SIZE;
}
```

We just have to directly compare the integer token types to see if the two are different. If they are, we use the function `typeToString()` to turn those integer types back into printable strings, suitable for a warning message.

⟨*Complain if token was redeclared to a different type*⟩≡
```
if (iter->second != tokenType) {
    char str1[80], str2[80];
    typeToString(iter->second, str1);
    typeToString(tokenType, str2);
    Warning("RiDeclare: Token '%s' redeclared from %s to %s.", name, str1, str2);
}
```

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.

⟨*Add token to table if not found*⟩≡
```
char *newToken = Strdup(name);
TokenTable[hashValue].push_front(make_pair(newToken, tokenType));
return newToken;
```

These are the functions that turn declared variable types into integers and back again.

⟨*API Static Methods*⟩+≡
```
static int stringToType(const char *strType);
static void typeToString(int type, char string[]);
```

And these constants help decode the meaning of the integers. Each integer is constructed by and-ing together the code for a storage class (uniform or vertex) and the code for a type (float, etc.)

⟨*API Static Data*⟩+≡
```
#define TOKEN_TYPE_ERROR       -1
#define TOKEN_TYPE_UNIFORM (1<<0)
#define TOKEN_TYPE_VERTEX  (1<<1)
#define TOKEN_TYPE_FLOAT   (1<<2)
#define TOKEN_TYPE_POINT   (1<<3)
#define TOKEN_TYPE_VECTOR  (1<<4)
#define TOKEN_TYPE_NORMAL  (1<<5)
#define TOKEN_TYPE_STRING  (1<<6)
#define TOKEN_TYPE_COLOR   (1<<7)
#define TOKEN_TYPE_VOID    (1<<8)
#define TOKEN_TYPE_HPOINT  (1<<9)
```

The user may pass in `RI_NULL` as the type when a parameter is declared; this indicates that it uses no storage–it's `void`.

⟨*API Static Methods*⟩+≡
```
static int stringToType(const char *strType) {
    if (strType == NULL) return TOKEN_TYPE_VOID;
    int type;
    ⟨Decide if the type is uniform or vertex⟩
    ⟨Decode the storage class of the type⟩
    return type;
}
```

We take the string given by the user and skip over leading whitespace. Then we try to match the storage class, as either `uniform` or `vertex`. If neither matches, we assume that it's `uniform` and try to go on and decode the type.

⟨*Decide if the type is* `uniform` *or* `vertex`⟩≡
```
const char *strp = strType;
while (*strp && isspace(*strp))
    ++strp;
if (!*strp) return TOKEN_TYPE_ERROR;
if (strncmp("uniform", strp, strlen("uniform")) == 0) {
    type = TOKEN_TYPE_UNIFORM;    strp += strlen("uniform");
}
else if (strncmp("constant", strp, strlen("constant")) == 0) {
    type = TOKEN_TYPE_UNIFORM;    strp += strlen("constant");
}
else if (strncmp("vertex", strp, strlen("vertex")) == 0) {
    type = TOKEN_TYPE_VERTEX;     strp += strlen("vertex");
}
else
    type = TOKEN_TYPE_UNIFORM;
```

And we again skip over white space and then try to decode the type.

⟨*Decode the storage class of the type*⟩≡
```
while (*strp && isspace(*strp)) ++strp;
if (!*strp) return TOKEN_TYPE_ERROR;

#define TRY_DECODING_TYPE(name, mask) \
    if (strncmp(name, strp, strlen(name)) == 0) { \
        type |= mask; strp += strlen(name); \
    }

     TRY_DECODING_TYPE("float",  TOKEN_TYPE_FLOAT)
else TRY_DECODING_TYPE("point",  TOKEN_TYPE_POINT)
else TRY_DECODING_TYPE("hpoint", TOKEN_TYPE_HPOINT)
else TRY_DECODING_TYPE("vector", TOKEN_TYPE_VECTOR)
else TRY_DECODING_TYPE("normal", TOKEN_TYPE_NORMAL)
else TRY_DECODING_TYPE("string", TOKEN_TYPE_STRING)
else TRY_DECODING_TYPE("color",  TOKEN_TYPE_COLOR)
else {
    Error("RiDeclare: unable to decode type \"%s\"", strType);
    return TOKEN_TYPE_ERROR;
}
⟨Warn if there is additional non-whitespace⟩
```

⟨*Warn if there is additional non-whitespace*⟩≡
```
while (*strp && isspace(*strp)) ++strp;
if (*strp)
    Warning("RiDeclare: unknown text at end of declared type \"%s\".", strType);
```

The reverse procedure is very simple; we just see what bits in `type` are set and construct an appropriate string.

⟨*API Static Methods*⟩+≡
```
static void typeToString(int type, char *string) {
    if (type == TOKEN_TYPE_VOID) {
        string = strcat(string, "RI_NULL");
        return;
    }
    if (type & TOKEN_TYPE_UNIFORM)     string = strcat(string, "uniform ");
    else if (type & TOKEN_TYPE_VERTEX) string = strcat(string, "vertex ");
    else
        Severe("typeToString: unknown type %d", type);

    if (type & TOKEN_TYPE_FLOAT)  string = strcat(string, "float");
    else if (type & TOKEN_TYPE_POINT)  string = strcat(string, "point");
    else if (type & TOKEN_TYPE_HPOINT) string = strcat(string, "hpoint");
    else if (type & TOKEN_TYPE_VECTOR) string = strcat(string, "vector");
    else if (type & TOKEN_TYPE_NORMAL) string = strcat(string, "normal");
    else if (type & TOKEN_TYPE_STRING) string = strcat(string, "string");
    else if (type & TOKEN_TYPE_COLOR)  string = strcat(string, "color");
    else                          Severe("typeToString: unknown type %d", type);
}
```

This is a utility function for the rest of the RI implementation: given a C-style string, it returns the token for that string if it has already been declared. Otherwise `RI_NULL` is returned. Tokens are considered to be the same regardless of case.

⟨*API Static Methods*⟩+≡
```
static RtToken lookupToken(const char *name) {
    if (name == NULL) return RI_NULL;
    u_int offset = hashTokenString(name);
    list<pair<RtToken, int> >::iterator iter = TokenTable[offset].begin();
    while (iter != TokenTable[offset].end()) {
        if (strcmp(iter->first, name) == 0)
            return iter->first;
        ++iter;
    }
    return RI_NULL;
}
```

And this takes a name and returns the type of a declared variable.

⟨*API Static Methods*⟩+≡
```
static bool lookupTokenAndType(const char *name, RtToken *token, int *type) {
    Assert(name != NULL);
    ⟨Decode inline parameter declaration⟩
    ⟨Lookup token and type in token table⟩
}
```

⟨*Decode inline parameter declaration*⟩≡
```
bool tryInline = false;
⟨Decide if this looks like an inline declaration⟩
if (tryInline) {
    ⟨Make a copy of the relevant substring⟩
    ⟨Decode the substring⟩
}
```

⟨*Make a copy of the relevant substring*⟩≡
```
const char *end = &name[strlen(name) - 1];
while (!isspace(*end))
    --end;
char *buf = (char *)alloca(end - name + 1);
strncpy(buf, name, end-name);
buf[end-name] = '\0';
```

⟨*Decode the substring*⟩≡
```
*type = stringToType(buf);
*token = (char *)end + 1;
if (*type != TOKEN_TYPE_ERROR) return true;
```

⟨*Decide if this looks like an inline declaration*⟩≡
```
const char *p = name;
while (*p) {
    if (isspace(*p)) {
        tryInline = true;
        break;
    }
    ++p;
}
```

⟨*Lookup token and type in token table*⟩≡
```
u_int offset = hashTokenString(name);
list<pair<RtToken, int> >::iterator iter = TokenTable[offset].begin();
while (iter != TokenTable[offset].end()) {
    if (iter->first == name) {
       *token = iter->first;
       *type = iter->second;
       return true;
    }
    ++iter;
}
iter = TokenTable[offset].begin();
while (iter != TokenTable[offset].end()) {
    if (strcmp(iter->first, name) == 0) {
       *token = iter->first;
       *type = iter->second;
       return true;
    }
    ++iter;
}
return false;
```

C.3 Argument List Processing

We will also provide some general facilities for dealing with variable numbers of arguments to RI functions. The general strategy for functions like `RiFoo()` will be to process the variable arguments into a form that's suitable for the vector `RiFooV()` function and then call that from `RiFoo()`.

The first function in this effort is `vectorizeParameters()`. It takes a `va_list` variable, which is set using the standard C library `va_start()` function call; it is essentially an iterator that lets us look at the variable argument list one item at a time. We go through the arguments until we reach `RI_NULL`, signifying the end of the list. For each token we find, we grab the its associated values and store a pointer to them, too.

⟨*API Includes*⟩≡
```
#include <stdarg.h>
```

⟨*API Static Methods*⟩+≡
```
static void vectorizeParameters(va_list args) {
    argsCount = 0;
    const char *name;
    while ((name = va_arg(args, const char *)) != RI_NULL) {
        if (argsCount >= argsAllocated) {
            ⟨Expand the arg arrays⟩
        }
        ⟨Add argument and value to arg arrays⟩
    }
}
```

We store all of these parameters and pointers to their user-supplied values in the following module-local variables. Because none of the RI calls are re-entrant, it's safe to just squirrel them away here, rather than returning them from `vectorizeParameters()`.

⟨*API Static Data*⟩+≡
```
static int argsCount;
static int argsAllocated;
static RtToken *argTokens;
static RtPointer *argParams;
```

If need be, we expand the `argTokens` and `argParams` arrays.

⟨*Expand the arg arrays*⟩≡
```
int newArgs;
if (argsAllocated == 0) newArgs = 10;
else                    newArgs = 2 * argsAllocated;
argTokens = (RtToken *)realloc(argTokens, newArgs * sizeof(RtToken));
argParams = (RtPointer *)realloc(argParams, newArgs * sizeof(RtPointer));
argsAllocated = newArgs;
```

Now we look up a parameter's token and its type in the token table and add it and a pointer to its values to the arrays. If the name has not previously been declared with `RiDeclare()`, we don't know how to interpret the values associated with it, so we skip it.

⟨*Add argument and value to arg arrays*⟩≡
```
argTokens[argsCount] = (RtToken)name;
argParams[argsCount] = va_arg(args, RtPointer);
if (argParams[argsCount] == RI_NULL)
    Error("Null parameter value for argument %s?!?", name);
else
    ++argsCount;
```

When `lrt` is done, we free up the token and parameter space that we allocated.

⟨*System-wide cleanup*⟩≡
```
argsAllocated = 0;
if (argTokens) { free(argTokens); argTokens = NULL; }
if (argParams) { free(argParams); argParams = NULL; }
```

Here are two utility functions; the first lets us easily initialize floating-point variables from user-supplied parameter lists: the caller should pass in the token that names the parameter of interest and a default value for it. We look through the arguments from the user, returning the value of the appropriate one if present and otherwise returning the supplied default value.

⟨*API Static Methods*⟩+≡
```
static Float InitializeFloat(RtToken tokenName, Float defaultValue, int nArgs,
        RtToken tokens[], RtPointer params[]) {
    ⟨Verify that we are indeed looking for a float parameter⟩
    for (int i = 0; i < nArgs; ++i)
        if (tok == tokens[i])
            return *((Float *)params[i]);
    for (int i = 0; i < nArgs; ++i)
        if (strcmp(tok, tokens[i]) == 0)
            return *((Float *)params[i]);
    return defaultValue;
}
```

⟨*Verify that we are indeed looking for a float parameter*⟩≡
```
RtToken tok;
int type;
if (!lookupTokenAndType(tokenName, &tok, &type))
    Severe("Token %s unknown in InitializeFloat() in RI", tokenName);
if (!(type & TOKEN_TYPE_FLOAT) && !(type & TOKEN_TYPE_UNIFORM)) {
    char buf[128];
    typeToString(type, buf);
    Severe("Token %s declared to be non-uniform float type %s",
        tokenName, buf);
}
```

Another useful utility function goes through the user supplied parameters and prints a warning about the ones that we didn't use. This takes the name of the function calling it, the vectorized parameters, and then a variable number of tokens terminated by `RI_NULL`, one for each of the tokens that we did use in the calling function.

⟨*API Static Methods*⟩+≡
```
void reportUnusedParameters(const char *funcName, int nArgs, RtToken tokens[],
        ...) {
    ⟨Make a copy of the parameters that were processed⟩
    ⟨Report the parameters that were not used⟩
}
```

⟨*Make a copy of the parameters that were processed*⟩≡
```
vector<RtToken> usedParameters;
va_list args;
va_start(args, tokens);
RtToken token;
while ((token = va_arg(args, RtToken)) != RI_NULL)
    usedParameters.push_back(token);
va_end(args);
```

⟨*Report the parameters that were not used*⟩≡
```
for (int i = 0; i < nArgs; ++i) {
    for (u_int j = 0; j < usedParameters.size(); ++j)
        if (tokens[i] == usedParameters[j])
            goto tokenUsed;
    for (u_int j = 0; j < usedParameters.size(); ++j)
        if (strcmp(tokens[i], usedParameters[j]) == 0)
            goto tokenUsed;
    Warning("%s(): parameter \"%s\" is unknown and will be ignored.",
        funcName, argTokens[i]);
tokenUsed: ;
}
```

Another useful little utility function creates the appropriate `SurfaceFunction` object that encapsulates all of the parameters and values that were given.

⟨*API Static Methods*⟩+≡
```
static SurfaceFunction *initSurfaceFunction(int nVertices, int n,
        RtToken tokens[], RtPointer params[]) {
    SurfaceFunction *surffunc = new SurfaceFunction(nVertices);
    for (int i = 0; i < n; ++i) {
        if (strcmp(tokens[i], RI_P) == 0) continue;
        if (strcmp(tokens[i], RI_PW) == 0) continue;
        ⟨Initialize data for nth parameter⟩
    }
    return surffunc;
}
```

⟨*Initialize data for nth parameter*⟩≡
```
int paramType;
RtToken token;
if (lookupTokenAndType(tokens[i], &token, &paramType)) {
    if (paramType & TOKEN_TYPE_UNIFORM) {
        ⟨Initialize uniform primitive data⟩
    }
    else if (nVertices == 0) {
        ⟨Complain about vertex data⟩
    }
    else {
        Assert(paramType & TOKEN_TYPE_VERTEX);
        ⟨Initialize vertex primitive data⟩
    }
}
else
    Warning("Unknown parameter '%s'.  Ignoring it.", tokens[i]);
```

⟨*Initialize uniform primitive data*⟩≡
```
if (paramType & TOKEN_TYPE_FLOAT)
    surffunc->AddUniformFloat(token, (Float *)params[i]);
else if (paramType & TOKEN_TYPE_POINT)
    surffunc->AddUniformPoint(token, (Float *)params[i]);
else if (paramType & TOKEN_TYPE_HPOINT)
    surffunc->AddUniformHPoint(token, (Float *)params[i]);
else if (paramType & TOKEN_TYPE_VECTOR)
    surffunc->AddUniformVector(token, (Float *)params[i]);
else if (paramType & TOKEN_TYPE_NORMAL)
    surffunc->AddUniformNormal(token, (Float *)params[i]);
else if (paramType & TOKEN_TYPE_COLOR)
    surffunc->AddUniformColor(token, (Float *)params[i]);
else if (paramType & TOKEN_TYPE_STRING)
    surffunc->AddUniformString(token, *((const char **)params[i]));
```

If we were called with `nVertices` zero, the caller should be a surface shader or a light source creation function. If there is in fact `vertex` data present, we'll complain.

⟨*Complain about vertex data*⟩≡
```
Warning("Ignoring vertex data '%s' being bound to light or surface shader",
    token);
```

⟨*Initialize vertex primitive data*⟩≡
```
if (paramType & TOKEN_TYPE_FLOAT)
    surffunc->AddVertexFloat(token, (Float *)params[i]);
else if (paramType & TOKEN_TYPE_POINT)
    surffunc->AddVertexPoint(token, (Float *)params[i]);
else if (paramType & TOKEN_TYPE_HPOINT)
    surffunc->AddVertexHPoint(token, (Float *)params[i]);
else if (paramType & TOKEN_TYPE_VECTOR)
    surffunc->AddVertexVector(token, (Float *)params[i]);
else if (paramType & TOKEN_TYPE_NORMAL)
    surffunc->AddVertexNormal(token, (Float *)params[i]);
else if (paramType & TOKEN_TYPE_COLOR)
    surffunc->AddVertexColor(token, (Float *)params[i]);
```

C.4 Setup and World and Frame

We can now move forward and start to define more of the RI functions. We'll start with the first function that should be called as well as the last: `RiBegin()` and `RiEnd()`.

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiBegin(RtToken), RiEnd();
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiBegin(RtToken tok) {
    if (currentApiState != STATE_UNINITIALIZED) {
        Error("RiBegin() has already been called.");
        return;
    }
    currentApiState = STATE_BEGIN;

    ⟨RtToken Initialization⟩

    for (int i = 0; i < MOTION_LEVELS; ++i)
        curTransform[i] = Transform();
    scene = new Scene;
    curLightAttributes = new LightAttributes;

    if (tok != RI_NULL) Warning("Unknown renderer name %s in RiBegin()", tok);
}
```

⟨*API Static Data*⟩+≡
```
#define MOTION_LEVELS 2
static int motionLevel = 0;
static bool inMotionBlock = false;
static Transform curTransform[MOTION_LEVELS];
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiEnd() {
    currentApiState = STATE_UNINITIALIZED;
    ⟨System-wide cleanup⟩
    curLightAttributes->Dereference();
    curLightAttributes = NULL;
    delete scene;
    scene = NULL;
    StatsCleanup();
}
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiFrameBegin(RtInt), RiFrameEnd();
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiFrameBegin(RtInt num) {
}
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiFrameEnd() {
}
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiWorldBegin(), RiWorldEnd();
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiWorldBegin() {
    VERIFY_STATE(STATE_BEGIN, "RiWorldBegin");
    currentApiState = STATE_WORLD_BEGIN;

    ⟨Set world to camera transformation⟩

    curPrimitiveAttributes = new PrimitiveAttributes;
    curPrimitiveAttributes->Surface = MaterialCreate("matte", new SurfaceFunction(0));
    curMaterialAttributes = new MaterialAttributes;
}
```
xxx

⟨*API Static Data*⟩+≡
```
static vector<Primitive *> primitives;
static list<Light *> lights;
static list<list<Light *> > lightStack;
static PrimitiveAttributes *curPrimitiveAttributes;
```

⟨*API Static Methods*⟩+≡
```
static void AddPrimitive(Primitive *prim) {
    if (!prim) return;
    primitives.push_back(prim);
    if (curPrimitiveAttributes->LightShader)
        curPrimitiveAttributes->LightShader->SetGeometry(prim);
}
```
Set world2camera from current xform, set current to identity. Now current is the object2world transformation. Now we can accept primitives.

⟨*Set world to camera transformation*⟩≡
```
for (int i = 0; i < MOTION_LEVELS; ++i) {
    scene->camera->WorldToCamera[i] = curTransform[i];
    curTransform[i] = Transform();
}
```

Finish rendering image before it returns. Nuke lights and retained objects defined since world begin.

C.5 Managing Graphics State

Options.

Options: overall system-wide stuff; can't be set after world-begin Camera Output

Camera.

Most camera definition management is completely straightforward; all of the following functions just directly set the appropriate fields in `camera` with the parameters passed to them (see Section XXXX). As there is nothing more to them, their implementation is not included here.

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiPixelSamples(RtFloat x, RtFloat y);
extern RtVoid RiFormat(RtInt x, RtInt y, RtFloat aspect);
extern RtVoid RiFrameAspectRatio(RtFloat aspect);
extern RtVoid RiScreenWindow(RtFloat left, RtFloat right, RtFloat bottom,
    RtFloat top);
extern RtVoid RiCropWindow(RtFloat left, RtFloat right, RtFloat bottom, RtFloat top);
extern RtVoid RiClipping(RtFloat hither, RtFloat yon);
extern RtVoid RiShutter(RtFloat time0, RtFloat time1);
extern RtVoid RiDepthOfField(RtFloat fstop, RtFloat focallen, RtFloat focaldist);
```

The first interesting RI function that will make use of the variable argument processing code will be `RiProjection()`. Like all other interface functions that take variable arguments, a vector form (`RiProjectionV()`) is provided as well.

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiProjection(RtToken name, ...);
extern RtVoid RiProjectionV(RtToken name, RtInt nArgs, RtToken tokens[],
    RtPointer params[]);
```

The user passes a transformation type `name`, which can be `RI_ORTHO`, which requests an orthographic transformation, `RI_PERSPECTIVE`, which requests a perspective transformation, or `RI_NULL`, which says that the user has constructed their own projection matrix (which is in `curTransform`), which we should just use for the transformation.

We will just turn the variable parameter list into `argTokens` and friends and then use those arrays of parameters and values to call `RiProjectionV()` which will do all of the work.

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiProjection(RtToken name, ...) {
    va_list args;
    va_start(args, name);
    vectorizeParameters(args);
    RiProjectionV(name, argsCount, argTokens, argParams);
    va_end(args);
}
```

We first verify that `RiBegin()` has been called but not yet `RiWorldBegin()`. We then do the appropriate initialization based on `name` and finish by resetting the current transformation to the identity.

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiProjectionV(RtToken name, RtInt nArgs, RtToken tokens[],
        RtPointer params[]) {
    VERIFY_STATE(STATE_BEGIN, "RiProjection");
    ⟨Handle various projection types⟩
    curTransform[0] = Transform();
}
```

⟨*Handle various projection types*⟩≡
```
if (name == RI_ORTHOGRAPHIC || strcmp(name, RI_ORTHOGRAPHIC) == 0) {
    ⟨Initialize orthographic projection⟩
}
else if (name == RI_PERSPECTIVE || strcmp(name, RI_PERSPECTIVE) == 0) {
    ⟨Initialize perspective projection⟩
}
else if (name == RI_NULL) {
    ⟨Initialize NULL projection⟩
}
else
    Error("Unknown projection %s specified in RiProjection()", name);
```

We concatenate a basic orthographic transformation (see Section **??**) to the current transformation and store the result in `CameraToScreen`.

⟨*Initialize orthographic projection*⟩≡
```
scene->camera->CameraToScreen = curTransform[0] *
    Orthographic(scene->camera->ClipHither, scene->camera->ClipYon);
scene->camera->ProjectionType = Camera::Orthographic;
reportUnusedParameters("RiProjection", nArgs, tokens, RI_NULL);
```

The perspective projection is similar, though we look for a user-supplied parameter called "fov"; if that's found, we use that as the field of view for the perspective projection. Otherwise, we use a default 90 degree field of view.

⟨*Initialize perspective projection*⟩≡
```
Float fov = InitializeFloat(RI_FOV, 90., nArgs, tokens, params);
scene->camera->CameraToScreen = curTransform[0] *
    Perspective(fov, 1. / scene->image->PixelAspectRatio, scene->camera->ClipHither,
        scene->camera->ClipYon) * Scale(-1, -1, 1);
scene->camera->ProjectionType = Camera::Perspective;
reportUnusedParameters("RiProjection", nArgs, tokens, RI_FOV, RI_NULL);
```

⟨*Initialize NULL projection*⟩≡
```
scene->camera->CameraToScreen = curTransform[0];
scene->camera->ProjectionType = Camera::Perspective; // guess
reportUnusedParameters("RiProjection", nArgs, tokens, RI_NULL);
```

Display.

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiExposure(RtFloat gain, RtFloat gamma);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiExposure(RtFloat gain, RtFloat gamma) {
    scene->image->Gain = gain;
    scene->image->Gamma = gamma;
}
```

filter and widths

⟨*RI Types*⟩+≡
```
typedef RtFloat (*RtFilterFunc)(RtFloat, RtFloat, RtFloat, RtFloat);
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiPixelFilter(RtFilterFunc filter, RtFloat xwidth, RtFloat ywidth);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiPixelFilter(RtFilterFunc filter, RtFloat xwidth, RtFloat ywidth) {
    scene->sampler->Filter = filter;
    scene->sampler->FilterXWidth = xwidth;
    scene->sampler->FilterYWidth = ywidth;
}
```

⟨*RI Function Declarations*⟩+≡
```
extern RtFloat RiBoxFilter(RtFloat x, RtFloat y, RtFloat xwidth, RtFloat ywidth);
extern RtFloat RiTriangleFilter(RtFloat x, RtFloat y, RtFloat xwidth, RtFloat ywidth);
extern RtFloat RiCatmullRomFilter(RtFloat x, RtFloat y, RtFloat xwidth, RtFloat ywidth)
extern RtFloat RiGaussianFilter(RtFloat x, RtFloat y, RtFloat xwidth, RtFloat ywidth);
extern RtFloat RiSincFilter(RtFloat x, RtFloat y, RtFloat xwidth, RtFloat ywidth);
extern RtFloat RiMitchellFilter(RtFloat x, RtFloat y, RtFloat xwidth, RtFloat ywidth);
```

For these, we just call the appropriate function from Section 7.3 on page 108.

The imager shader is set with the `RiImager()` call (see Section ?? for a discussion of tone reproduction with imager shaders). Like `RiProjection()`, there is a variable argument version and a vector version. We will omit the first version, which just bundles up the arguments like `RiProjection()` does and then calls the second version.

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiImager(RtToken name, ...);
extern RtVoid RiImagerV(RtToken name, RtInt n, RtToken tokens[],
    RtPointer params[]);
```

For now, no imagers anyway...

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiImagerV(RtToken name, RtInt n, RtToken tokens[], RtPointer params[]) {
    VERIFY_STATE(STATE_BEGIN, "RiImager");
    scene->image->Imager = name;
    reportUnusedParameters("RiImager", n, tokens, RI_NULL);
}
```

The `RiQuantize()` funciton sets parameter values for the color and depth quantization parts of the imaging pipeline (see Section 6.2.) 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 Declarations*⟩+≡
```
extern RtVoid RiQuantize(RtToken type, RtInt one, RtInt minimum, RtInt maximum,
        RtFloat ditheramp);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiQuantize(RtToken type, RtInt one, RtInt minimum, RtInt maximum,
        RtFloat ditheramp) {
    if (type == RI_RGBA || strcmp(type, RI_RGBA) == 0) {
        scene->image->ColorQuantOne = one;
        scene->image->ColorQuantMin = minimum;
        scene->image->ColorQuantMax = maximum;
        scene->image->ColorQuantDither = ditheramp;
    }
    else if (type == RI_Z || strcmp(type, RI_Z) == 0) {
        scene->image->DepthQuantOne = one;
        scene->image->DepthQuantMin = minimum;
        scene->image->DepthQuantMax = maximum;
        scene->image->DepthQuantDither = ditheramp;
    }
    else
        Error("Unknown type %s passed to RiQuantize()", type);
}
```

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? LUV?) 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 RtVoid RiDisplay(char *name, RtToken type, RtToken mode, ...);
extern RtVoid RiDisplayV(char *name, RtToken type, RtToken mode, int nArgs,
    RtToken tokens[], RtPointer parameters[]);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiDisplayV(char *name, RtToken type, RtToken mode, int nArgs,
        RtToken tokens[], RtPointer parameters[]) {
    ⟨Complain about unknown type or mode⟩
    ⟨Verify display mode parameter⟩
    ⟨Initialize display options⟩
    ⟨Process display variable argument list⟩
}
```

⟨*Complain about unknown* `type` *or* `mode`⟩≡
```
if (type != RI_FILE && strcmp(type, RI_FILE) != 0) {
    Error("Display type %s not supported.", type);
    return;
}
RtToken displayMode = lookupToken(mode);
if (displayMode == RI_NULL) {
    Error("Display mode %s unknown.", mode);
    return;
}
```

⟨*Verify display mode parameter*⟩≡
```
if (displayMode != RI_RGB && displayMode != RI_RGBA && displayMode != RI_RGBZ &&
    displayMode != RI_RGBAZ && displayMode != RI_A && displayMode != RI_AZ &&
    displayMode != RI_Z) {
    Error("Display mode %s not supported.", mode);
    return;
}
```

⟨*Initialize display options*⟩≡
```
scene->image->DisplayType = type;
scene->image->DisplayName = Strdup(name);
scene->image->DisplayMode = displayMode;
```

We don't use any additional arguments, so we'll complain if there are any.

⟨*Process display variable argument list*⟩≡
```
reportUnusedParameters("RiDisplay", nArgs, tokens, RI_NULL);
```

Miscellaneous.

hider

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiHider(RtToken type, ...);
extern RtVoid RiHiderV(RtToken type, RtInt n, RtToken tokens[], RtPointer parms[]);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiHiderV(RtToken type, RtInt nArgs, RtToken tokens[], RtPointer parms[]) {
    if (type == RI_HIDDEN || strcmp(type, RI_HIDDEN) == 0 || type == RI_NULL) {
        for (int i = 0; i < nArgs; ++i) {
            if (tokens[i] == LRT_JITTER || strcmp(tokens[i], LRT_JITTER)== 0) {
                Float on = *((RtFloat *)(parms[i]));
                scene->sampler->JitterSamples = on;
            }
        }
        reportUnusedParameters("RiHider", nArgs, tokens, LRT_JITTER, RI_NULL);
    }
    else {
        if (type == RI_PAINT || strcmp(type, RI_PAINT) == 0)
            Error("Paint hider not supported by RiHider()");
        else
            Error("Unknown type %s passed to RiHider()", type);
        reportUnusedParameters("RiHider", nArgs, tokens, RI_NULL);
    }
}
```

implementation-specific stuff

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiOption(RtToken name, ...);
extern RtVoid RiOptionV(RtToken name, RtInt n, RtToken tokens[], RtPointer parms[]);
```

⟨*RI Function Definitions*⟩+≡

```
RtVoid RiOptionV(RtToken name, RtInt n, RtToken tokens[], RtPointer parms[]) {
    if (name == LRT_DISPLAY || strcmp(name, LRT_DISPLAY) == 0) {
        for (int i = 0 ; i < n ; i++) {
            if (tokens[i] == LRT_IMAGEVIEWER ||
                strcmp(tokens[i], LRT_IMAGEVIEWER) == 0) {
                char **arr = (char **) parms[i];
                scene->image->ImageViewer = Strdup(arr[0]);
            }
            else
                Warning("Ignoring unknown option 'display' token %s", tokens[i]);
        }
    }
    else if (name == LRT_RENDER || strcmp(name, LRT_RENDER) == 0) {
        ⟨Parse rendering options⟩
    }
    else if (name == RI_CAMERA || strcmp(name, RI_CAMERA) == 0) {
        ⟨Parse camera options⟩
    }
    else
        Warning("RiOption: unknown option class %s", name);
}
```

⟨*Parse rendering options*⟩≡

```
for (int i = 0; i < n; ++i) {
    if (tokens[i] == LRT_INTEGRATOR || strcmp(tokens[i], LRT_INTEGRATOR) == 0) {
        RtToken param = *((RtToken *)(parms[i]));
        if (strcmp(param, LRT_COLOR) == 0)
            scene->IlluminationIntegrator = LRT_COLOR;
        else if (strcmp(param, LRT_RAYCAST) == 0)
            scene->IlluminationIntegrator = LRT_RAYCAST;
        else if (strcmp(param, LRT_WHITTED) == 0)
            scene->IlluminationIntegrator = LRT_WHITTED;
        else if (strcmp(param, LRT_MONTECARLO) == 0)
            scene->IlluminationIntegrator = LRT_MONTECARLO;
        else if (strcmp(param, LRT_RMAN) == 0)
            scene->IlluminationIntegrator = LRT_RMAN;
        else
            Error("Unknown integrator type \'%s\' passed to RiOption", param);
    }
    else
        Error("Unknown rendering option \"%s\" passed to RiOption", tokens[i]);
}
```

⟨*Parse camera options*⟩≡

```
for (int i = 0; i < n; ++i) {
    if (tokens[i] == LRT_TYPE || strcmp(tokens[i], LRT_TYPE) == 0) {
        RtToken param = *((RtToken *)(parms[i]));
        if (strcmp(param, LRT_PINHOLE) == 0) {
            delete scene->camera;
            scene->camera = new PinholeCamera;
        }
        else if (strcmp(param, LRT_MBDOF) == 0)
        {
            delete scene->camera;
            scene->camera = new MbDOFCamera;
        }
        else
            Error("Unknown camera type \'%s\' passed to RiOption", param);
    }
    else if (tokens[i] == LRT_SHUTTER || strcmp(tokens[i], LRT_SHUTTER) ==  0) {
        RtToken param = *((RtToken *)(parms[i]));
        if (strcmp(param, LRT_IRIS) == 0)
            scene->camera->ShutterType = LRT_IRIS;
        else if (strcmp(param, LRT_STRIPE) == 0)
            scene->camera->ShutterType = LRT_STRIPE;
        else
            Error("Unknown shutter type \'%s\' passed to RiOption", param);
    }
    else if (tokens[i] == LRT_IRIS_RATE || strcmp(tokens[i], LRT_IRIS_RATE) == 0) {
        Float param = *((Float *)(parms[i]));
        scene->camera->IrisRate = param;
    }
    else if (tokens[i] == LRT_STRIPE_WIDTH || strcmp(tokens[i], LRT_STRIPE_WIDTH) == 0)
        Float param = *((Float *)(parms[i]));
        scene->camera->StripeWidth = param;
    }
    else if (tokens[i] == LRT_STRIPE_DIRECTION || strcmp(tokens[i], LRT_STRIPE_DIRECTIO
        RtToken param = *((RtToken *)(parms[i]));
        if (strcmp(param, LRT_DOWN) == 0)
            scene->camera->StripeDirection = LRT_DOWN;
        if (strcmp(param, LRT_UP) == 0)
            scene->camera->StripeDirection = LRT_UP;
        if (strcmp(param, LRT_LEFT) == 0)
            scene->camera->StripeDirection = LRT_LEFT;
        if (strcmp(param, LRT_RIGHT) == 0)
            scene->camera->StripeDirection = LRT_RIGHT;
        else
            Error("Unknown shutter type \'%s\' passed to RiOption", param);
    }
    else
        Error("Unknown rendering option \"%s\" passed to RiOption", tokens[i]);
}
```

Attributes.

Attributes: attribute stack: push/pop, etc. Current shaders Light sources

current transformation matrix: transforms all incoming geometry. defines the current (or object) coordinate system. all geometry is defined in the current coordinate system. new incoming transformations are applied to the end of it–thus they are the first to be applied to geometry.

transformation stack, named coordinate systems

Our implementation of the interface tracks this state as it's specified by RI procedure calls. Whenever a geometric primitive is specified, it is bundled up with the current state and stored in a list of primitives. When the entire scene has been specified, all that is passed into the renderer is the `Options` object, which specifies overall system-wide options, a list of `Primitive`s (each of which holds an `PrimitiveAttributes` object), and a list of `Light`s. Thus the rendering system is insulated from the details of the hierarchical scene provided by the user.

⟨*API Static Data*⟩+≡
```
static LightAttributes *curLightAttributes;
static MaterialAttributes *curMaterialAttributes;
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiTransformBegin();
extern RtVoid RiTransformEnd();
extern RtVoid RiAttributeBegin();
extern RtVoid RiAttributeEnd();
```

⟨*API Static Data*⟩+≡
```
static list<char> hierarchicalState;
static list<Transform> transformStack[MOTION_LEVELS];
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiTransformBegin() {
    for (int i = 0; i < MOTION_LEVELS; ++i)
        transformStack[i].push_back(curTransform[i]);
    hierarchicalState.push_back('t');
}

RtVoid 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();
}
```

⟨*API Static Data*⟩+≡
```
static list<PrimitiveAttributes *> geomAttrStack;
static list<LightAttributes *> lightAttrStack;
static list<MaterialAttributes *> surfaceAttrStack;
```

⟨*RI Function Definitions*⟩+≡

```
RtVoid RiAttributeBegin() {
    RiTransformBegin();
    VERIFY_STATE(STATE_WORLD_BEGIN, "RiAttributeBegin");

    geomAttrStack.push_back(curPrimitiveAttributes);
    curPrimitiveAttributes = new PrimitiveAttributes(*curPrimitiveAttributes);
    lightAttrStack.push_back(curLightAttributes);
    curLightAttributes = new LightAttributes(*curLightAttributes);
    surfaceAttrStack.push_back(curMaterialAttributes);
    curMaterialAttributes = new MaterialAttributes(*curMaterialAttributes);
    lightStack.push_back(lights);

    hierarchicalState.push_back('a');
}

RtVoid RiAttributeEnd() {
    VERIFY_STATE(STATE_WORLD_BEGIN, "RiAttributeEnd");
    if (!geomAttrStack.size() || hierarchicalState.back() != 'a') {
        Error("Unmatched RiAttributeEnd encountered.  Ignoring it.");
        return;
    }
    curPrimitiveAttributes->Dereference();
    curPrimitiveAttributes = geomAttrStack.back();
    geomAttrStack.pop_back();
    curLightAttributes->Dereference();
    curLightAttributes = lightAttrStack.back();
    lightAttrStack.pop_back();
    curMaterialAttributes->Dereference();
    curMaterialAttributes = surfaceAttrStack.back();
    surfaceAttrStack.pop_back();
    hierarchicalState.pop_back();
    lights = lightStack.back();
    lightStack.pop_back();
    RiTransformEnd();
}
```

XXX fixme for motion

⟨*Prepare geometric attributes for binding*⟩≡

```
if ((curTransform[0] != curPrimitiveAttributes->WorldToObject ||
    lights != curPrimitiveAttributes->Lights) &&
    curPrimitiveAttributes->GetReferences() > 1) {
    PrimitiveAttributes *newAttr = new PrimitiveAttributes(*curPrimitiveAttributes);
    curPrimitiveAttributes->Dereference();
    curPrimitiveAttributes = newAttr;
}
curPrimitiveAttributes->ObjectToWorld = curTransform[0];
curPrimitiveAttributes->WorldToObject = Transform(curTransform[0].GetInverse());
curPrimitiveAttributes->Lights = lights;
curPrimitiveAttributes->Reference();
```

⟨*Update geometric attributes for changing value*⟩≡
```
if (curPrimitiveAttributes->GetReferences() > 1) {
    PrimitiveAttributes *newAttr = new PrimitiveAttributes(*curPrimitiveAttributes);
    curPrimitiveAttributes->Dereference();
    curPrimitiveAttributes = newAttr;
}
```

⟨*Update light attributes for changing value*⟩≡
```
if (curLightAttributes->GetReferences() > 1) {
    LightAttributes *newAttr = new LightAttributes(*curLightAttributes);
    curLightAttributes->Dereference();
    curLightAttributes = newAttr;
}
```

⟨*Prepare light attributes for binding*⟩≡
```
if (curTransform[0] != curLightAttributes->WorldToLight &&
    curLightAttributes->GetReferences() > 1) {
    LightAttributes *newAttr = new LightAttributes(*curLightAttributes);
    curLightAttributes->Dereference();
    curLightAttributes = newAttr;
}
curLightAttributes->LightToWorld = curTransform[0];
curLightAttributes->WorldToLight = Transform(curTransform[0].GetInverse());
curLightAttributes->Reference();
```

⟨*Prepare surface attributes for binding*⟩≡
```
if (curTransform[0] != curMaterialAttributes->WorldToSurface &&
    curMaterialAttributes->GetReferences() > 1) {
    MaterialAttributes *newAttr = new MaterialAttributes(*curMaterialAttributes);
    curMaterialAttributes->Dereference();
    curMaterialAttributes = newAttr;
}
curMaterialAttributes->SurfaceToWorld = curTransform[0];
curMaterialAttributes->WorldToSurface = Transform(curTransform[0].GetInverse());
curMaterialAttributes->Reference();
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiColor(RtColor Cs);
extern RtVoid RiOpacity(RtColor Cs);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiColor(RtColor Cs) {
        ⟨Update geometric attributes for changing value⟩
        curPrimitiveAttributes->Color = Spectrum(Cs[0], Cs[1], Cs[2]);
}
RtVoid RiOpacity(RtColor Cs) {
        ⟨Update geometric attributes for changing value⟩
        curPrimitiveAttributes->Opacity = Spectrum(Cs[0], Cs[1], Cs[2]);
}
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiOrientation(RtToken orientation);
extern RtVoid RiReverseOrientation();
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiOrientation(RtToken orientation) { RI_UNIMP(); }
RtVoid RiReverseOrientation() { RI_UNIMP(); }
```

⟨*Global Inline Functions*⟩+≡
```
#ifdef __GNUC__
#define RI_UNIMP() { static int first = 1 ; if (first) { first = 0; Warning( "call to u
#else
#define RI_UNIMP() { static int first = 1 ; if (first) { first = 0; Warning( "call to u
#endif
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiSides(RtInt sides);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiSides(RtInt sides) { RI_UNIMP(); }
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiAttribute(RtToken name, ...);
extern RtVoid RiAttributeV(RtToken name, RtInt n, RtToken tokens[],
    RtPointer parms[]);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiAttributeV(RtToken name, RtInt n, RtToken tokens[], RtPointer parms[]) {
    if (name == LRT_LIGHT || strcmp(name, LRT_LIGHT) == 0) {
        ⟨Process light attributes⟩
    }
    else if (name == LRT_RENDER || strcmp(name, LRT_RENDER) == 0) {
        ⟨Process MC rendering attributes⟩
    }
    else
        Warning("Ignoring unknown attribute %s", name);
}
```

⟨*Process light attributes*⟩≡
```
for (int i = 0; i < n; ++i) {
    if (tokens[i] == LRT_SHADOWS || strcmp(tokens[i], LRT_SHADOWS) == 0) {
        const char *param = *((const char **)(parms[i]));
        if (strcmp(param, "on") == 0) {
            ⟨Update light attributes for changing value⟩
            curLightAttributes->CastsShadows = true;
        }
        else if (strcmp(param, "off") == 0) {
            ⟨Update light attributes for changing value⟩
            curLightAttributes->CastsShadows = false;
        }
        else
            Warning("Unknown on/off value for attribute shadows %s", param);
    }
    else if (tokens[i] == LRT_NSAMPLES || strcmp(tokens[i], LRT_NSAMPLES) == 0) {
        Float count = *((RtFloat *)(parms[i]));
        if (count < 1.)
            Warning("Ignoring < 1 number of light samples %f", count);
        else {
            ⟨Update light attributes for changing value⟩
            curLightAttributes->NSamples = int(count);
        }
    }
    else
        Error("Ignoring unknown light attribute %s", tokens[i]);
}
```

⟨*PrimitiveAttributes Public Data*⟩+≡
```
enum { SampleSurface, SampleLight, SampleCombination } Sampling;
```

⟨*PrimitiveAttributes constructor implementation*⟩+≡
```
Sampling = SampleCombination;
```

⟨*Process MC rendering attributes*⟩≡
```
for (int i = 0; i < n; ++i) {
    if (tokens[i] == LRT_SAMPLE || strcmp(tokens[i], LRT_SAMPLE) == 0) {
        const char *param = *((const char **)(parms[i]));
        ⟨Update geometric attributes for changing value⟩
        if (strcmp(param, LRT_SURFACE) == 0)
            curPrimitiveAttributes->Sampling = PrimitiveAttributes::SampleSurface;
        else if (strcmp(param, LRT_LIGHT) == 0)
            curPrimitiveAttributes->Sampling = PrimitiveAttributes::SampleLight;
        else if (strcmp(param, LRT_COMBINATION) == 0)
            curPrimitiveAttributes->Sampling = PrimitiveAttributes::SampleCombination;
        else
            Warning("Unknown 'sample' value, '%s' for attribute render", param);
    }
    else
        Error("Ignoring unknown rendering attribute '%s'", tokens[i]);
}
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiSurface(RtToken name, ...);
extern RtVoid RiSurfaceV(RtToken name, RtInt n, RtToken tokens[],
    RtPointer parms[]);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiSurfaceV(RtToken name, RtInt n, RtToken tokens[], RtPointer params[]) {
    ⟨Prepare surface attributes for binding⟩
    SurfaceFunction *surfData = initSurfaceFunction(0, n, tokens, params);
    ⟨Update geometric attributes for changing value⟩
    curPrimitiveAttributes->Surface = MaterialCreate(name, surfData);
}
```

C.6 Transformations

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiIdentity();
extern RtVoid RiTransform(RtMatrix transform);
extern RtVoid RiConcatTransform(RtMatrix transform);
extern RtVoid RiPerspective(RtFloat fov);
extern RtVoid RiTranslate(RtFloat dx, RtFloat dy, RtFloat dz);
extern RtVoid RiRotate(RtFloat angle, RtFloat dx, RtFloat dy, RtFloat dz);
extern RtVoid RiScale(RtFloat sx, RtFloat sy, RtFloat sz);
```

XXXXXXXXX Ok, so if no motion, need to set all of them the same way. Otherwise, if motion, set the appropriate one and increment to the next motion level. XXXXXXXXXXXXXXXXX

⟨*Prepare for motion transform*⟩≡
```
int xform = 0;
if (inMotionBlock)
    xform = motionLevel;
if (motionLevel > MOTION_LEVELS) {
    Warning("Only %d motion levels are supported. Ignoring.",
        motionLevel);
    return;
}
```

⟨*Update transform for motion block*⟩≡
```
if (inMotionBlock)
    ++motionLevel;
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiIdentity() {
    ⟨Prepare for motion transform⟩
    curTransform[xform] = Transform();
    ⟨Update transform for motion block⟩
}

RtVoid RiTransform(RtMatrix tr) {
    ⟨Prepare for motion transform⟩
    curTransform[xform] = Transform(tr[0], tr[4], tr[8], tr[12],
         tr[1], tr[5], tr[9], tr[13],
         tr[2], tr[6], tr[10], tr[14],
         tr[3], tr[7], tr[11], tr[15]);
    ⟨Update transform for motion block⟩
}

RtVoid RiConcatTransform(RtMatrix tr) {
    ⟨Prepare for motion transform⟩
    curTransform[xform] = curTransform[xform] *
        Transform(tr[0], tr[4], tr[8], tr[12],
                  tr[1], tr[5], tr[9], tr[13],
                  tr[2], tr[6], tr[10], tr[14],
                  tr[3], tr[7], tr[11], tr[15]);
    ⟨Update transform for motion block⟩
}

RtVoid RiPerspective(RtFloat fov) {
    ⟨Prepare for motion transform⟩
    // XXX?
    curTransform[xform] = curTransform[xform] * Perspective(fov, 1.0,
        scene->camera->ClipHither, scene->camera->ClipYon);
    ⟨Update transform for motion block⟩
}

RtVoid RiTranslate(RtFloat dx, RtFloat dy, RtFloat dz) {
    ⟨Prepare for motion transform⟩
    curTransform[xform] = curTransform[xform] * Translate(Vector(dx, dy, dz));
    ⟨Update transform for motion block⟩
}

RtVoid RiRotate(RtFloat angle, RtFloat dx, RtFloat dy, RtFloat dz) {
    ⟨Prepare for motion transform⟩
    curTransform[xform] = curTransform[xform] * Rotate(angle, Vector(dx, dy, dz));
    ⟨Update transform for motion block⟩
}

RtVoid RiScale(RtFloat sx, RtFloat sy, RtFloat sz) {
    ⟨Prepare for motion transform⟩
    curTransform[xform] = curTransform[xform] * Scale(sx, sy, sz);
    ⟨Update transform for motion block⟩
}
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiCoordinateSystem(RtToken);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiCoordinateSystem(RtToken) { RI_UNIMP( ); }
```

C.7 Geometric Primitives

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiPolygon(RtInt nverts, ...);
extern RtVoid RiPolygonV(RtInt nverts, RtInt n, RtToken tokens[],
    RtPointer parms[]);
```

The `RiPolygon` interface guarantees that the polygon is concave. In this case, we will tesselate the given polygon and create a `TriangleMesh` primitive.

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiPolygonV(RtInt nverts, RtInt n, RtToken tokens[], RtPointer params[]) {
    ⟨Prepare geometric attributes for binding⟩
    ⟨Get vertex positions from parameters⟩
    ⟨Tesselate single convex polygon⟩
}
```

⟨*Get vertex positions from parameters*⟩≡
```
Point *P = NULL;
for (int i = 0; i < n; ++i) {
    if (tokens[i] == RI_P || strcmp(tokens[i], RI_P) == 0)
        P = (Point *)params[i];
}
```

⟨*Tesselate single convex polygon*⟩≡
```
SurfaceFunction *surffunc = initSurfaceFunction(nverts, n, tokens, params);
RtInt *indices = new RtInt[ 3*(nverts-2) ];
for (int i = 0 ; i < nverts-2 ; i++) {
    indices[3*i] = 0;
    indices[3*i+1] = i+2;
    indices[3*i+2] = i+1;
}
AddPrimitive(new TriangleMesh(nverts-2, nverts, indices, P,
    curPrimitiveAttributes, surffunc));
```

⟨*RI Function Declarations*⟩+≡
```
RtVoid RiSphere(RtFloat radius, RtFloat zmin, RtFloat zmax,
                RtFloat thetamax, ...);
RtVoid RiSphereV(RtFloat radius, RtFloat zmin, RtFloat zmax,
                 RtFloat thetaMax, RtInt n, RtToken tokens[],
                 RtPointer params[]);
```

⟨*RI Function Declarations*⟩+≡
```
RtVoid RiPointsPolygons( RtInt npolys, RtInt nvertices[], RtInt vertices[],
                         ...);
RtVoid RiPointsPolygonsV( RtInt npolys, RtInt nvertices[], RtInt vertices[],
                          RtInt n, RtToken tokens[], RtPointer params[]);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiPointsPolygonsV( RtInt npolys, RtInt nvertices[], RtInt vertices[],
        RtInt n, RtToken tokens[], RtPointer params[])  {
    ⟨Prepare geometric attributes for binding⟩
    ⟨Get vertex positions from parameters⟩
    ⟨Tesselate convex polygon mesh⟩
    AddPrimitive(new TriangleMesh(nIndices/3, nVertices, indices,
        P, curPrimitiveAttributes, surffunc));
}
```

⟨*Tesselate convex polygon mesh*⟩≡
```
int nVertices = 0;
int nIndices = 0;
int vIndex = 0;
for (int i = 0; i < npolys; ++i) {
    nIndices += 3*(nvertices[i]-2);
    for (int j = 0 ; j < nvertices[i] ; j++) {
        nVertices = max(nVertices, vertices[vIndex] + 1);
        vIndex++;
    }
}

SurfaceFunction *surffunc = initSurfaceFunction( nVertices, n, tokens, params );
RtInt *indices = new RtInt[ nIndices ];

int whichIndex = 0;
vIndex = 0;
for (int i = 0; i < npolys; ++i) {
    int anchorIndex = vIndex;
    for (int j = 0 ; j < nvertices[i]-2 ; j++) {
        indices[whichIndex++] = vertices[anchorIndex];
        indices[whichIndex++] = vertices[vIndex+1];
        indices[whichIndex++] = vertices[vIndex+2];
        vIndex++;
    }
    vIndex += 2;
}
```

Yay, here's the sphere function...

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiSphereV(RtFloat radius, RtFloat zmin, RtFloat zmax,
        RtFloat thetaMax, RtInt n, RtToken tokens[],
        RtPointer params[]) {
    ⟨Prepare geometric attributes for binding⟩
    SurfaceFunction *surffunc = initSurfaceFunction(4, n, tokens, params);
    AddPrimitive(new Sphere(radius, zmin, zmax, thetaMax,
        curPrimitiveAttributes, surffunc));

}
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiDisk(RtFloat height, RtFloat radius, RtFloat tmax, ...);
extern RtVoid RiDiskV(RtFloat height, RtFloat radius, RtFloat tmax,
    RtInt n, RtToken tokens[], RtPointer parms[]);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiDiskV(RtFloat height, RtFloat radius, RtFloat thetamax,
    RtInt n, RtToken tokens[], RtPointer params[]) {
    ⟨Prepare geometric attributes for binding⟩
    SurfaceFunction *surffunc = initSurfaceFunction(4, n, tokens, params);
    AddPrimitive(new Disk(height, radius, thetamax,
        curPrimitiveAttributes, surffunc));
}
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiCone(RtFloat height, RtFloat radius, RtFloat tmax, ...);
extern RtVoid RiConeV(RtFloat height, RtFloat radius, RtFloat tmax,
    RtInt n, RtToken tokens[], RtPointer parms[]);
extern RtVoid RiCylinder(RtFloat radius, RtFloat zmin, RtFloat zmax,
    RtFloat tmax, ...);
extern RtVoid RiCylinderV(RtFloat radius, RtFloat zmin, RtFloat zmax, RtFloat tmax,
    RtInt n, RtToken tokens[], RtPointer parms[]);
extern RtVoid RiHyperboloid(RtPoint point1, RtPoint point2, RtFloat tmax, ...);
extern RtVoid RiHyperboloidV(RtPoint point1, RtPoint point2, RtFloat tmax,
    RtInt n, RtToken tokens[], RtPointer parms[]);
extern RtVoid RiParaboloid(RtFloat rmax, RtFloat zmin, RtFloat zmax,
    RtFloat tmax, ...);
extern RtVoid RiParaboloidV(RtFloat rmax, RtFloat zmin, RtFloat zmax, RtFloat tmax,
    RtInt n, RtToken tokens[], RtPointer parms[]);
extern RtVoid RiTorus(RtFloat majrad, RtFloat minrad, RtFloat phimin,
    RtFloat phimax, RtFloat tmax, ...);
extern RtVoid RiTorusV(RtFloat majrad, RtFloat minrad, RtFloat phimin,
    RtFloat phimax, RtFloat tmax, RtInt n, RtToken tokens[],
    RtPointer parms[]);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiCone(RtFloat height, RtFloat radius, RtFloat tmax, ...) {
    va_list args;
    va_start(args, tmax);
    vectorizeParameters(args);
    RiConeV(height, radius, tmax, argsCount, argTokens, argParams);
    va_end(args);
}

RtVoid RiConeV(RtFloat height, RtFloat radius, RtFloat tmax,
    RtInt n, RtToken tokens[], RtPointer params[]) {
    ⟨Prepare geometric attributes for binding⟩
    SurfaceFunction *surffunc = initSurfaceFunction( 4, n, tokens, params );
    AddPrimitive(new Cone(height, radius, tmax, curPrimitiveAttributes, surffunc));
}

RtVoid RiCylinder(RtFloat radius, RtFloat zmin, RtFloat zmax,
    RtFloat tmax, ...) {
    va_list args;
    va_start(args, tmax);
    vectorizeParameters(args);
    RiCylinderV(radius, zmin, zmax, tmax, argsCount, argTokens, argParams);
    va_end(args);
}

RtVoid RiCylinderV(RtFloat radius, RtFloat zmin, RtFloat zmax, RtFloat tmax,
    RtInt n, RtToken tokens[], RtPointer params[]) {
    ⟨Prepare geometric attributes for binding⟩
    SurfaceFunction *surffunc = initSurfaceFunction( 4, n, tokens, params );
    AddPrimitive(new Cylinder(radius, zmin, zmax, tmax, curPrimitiveAttributes,
        surffunc));
}

RtVoid RiHyperboloid(RtPoint point1, RtPoint point2, RtFloat tmax, ...) {
    va_list args;
    va_start(args, tmax);
    vectorizeParameters(args);
    RiHyperboloidV(point1, point2, tmax, argsCount, argTokens, argParams);
    va_end(args);
}

RtVoid RiHyperboloidV(RtPoint p1, RtPoint p2, RtFloat tmax,
    RtInt n, RtToken tokens[], RtPointer params[]) {
    ⟨Prepare geometric attributes for binding⟩
    SurfaceFunction *surffunc = initSurfaceFunction( 4, n, tokens, params );
    AddPrimitive(new Hyperboloid(Point(p1[0], p1[1], p1[2]),
        Point(p2[0], p2[1], p2[2]), tmax, curPrimitiveAttributes, surffunc));
}

RtVoid RiParaboloid(RtFloat rmax, RtFloat zmin, RtFloat zmax,
    RtFloat tmax, ...) {
    va_list args;
    va_start(args, tmax);
    vectorizeParameters(args);
```

```
    RiParaboloidV(rmax, zmin, zmax, tmax, argsCount, argTokens, argParams);
    va_end(args);
}

RtVoid RiParaboloidV(RtFloat rmax, RtFloat zmin, RtFloat zmax, RtFloat tmax,
    RtInt n, RtToken tokens[], RtPointer params[]) {
    ⟨Prepare geometric attributes for binding⟩
    SurfaceFunction *surffunc = initSurfaceFunction( 4, n, tokens, params );
    AddPrimitive(new Paraboloid(rmax, zmin, zmax, tmax, curPrimitiveAttributes,
        surffunc));
}

RtVoid RiTorus(RtFloat majrad, RtFloat minrad, RtFloat phimin,
    RtFloat phimax, RtFloat tmax, ...) {
    RI_UNIMP();
}

RtVoid RiTorusV(RtFloat majrad, RtFloat minrad, RtFloat phimin,
    RtFloat phimax, RtFloat tmax, RtInt n, RtToken tokens[],
    RtPointer params[]) {
    RI_UNIMP();
}
```

⟨*RI Types*⟩+≡

```
typedef RtVoid (*RtFunc)();
```

⟨*RI Function Declarations*⟩+≡

```
extern RtVoid RiProcedural(RtPointer data, RtBound bound, RtFunc subdivfunc,
    RtFunc freefunc);
```

Note that these implicitly save and restore attributes

⟨*RI Function Declarations*⟩+≡

```
extern RtVoid RiSolidBegin(RtToken);
extern RtVoid RiSolidEnd();
```

⟨*RI Function Definitions*⟩+≡

```
RtVoid RiSolidBegin(RtToken) { RI_UNIMP(); }
RtVoid RiSolidEnd() { RI_UNIMP(); }
```

⟨*RI Function Declarations*⟩+≡

```
extern RtVoid RiGeneralPolygon(RtInt nloops, RtInt nverts[], ...);
extern RtVoid RiGeneralPolygonV(RtInt nloops, RtInt nverts[], RtInt n,
    RtToken tokens[], RtPointer parms[]);
extern RtVoid RiPointsGeneralPolygons(RtInt npolys, RtInt nloops[], RtInt nverts[],
    RtInt verts[], ...);
extern RtVoid RiPointsGeneralPolygonsV(RtInt npolys, RtInt nloops[],
    RtInt nverts[], RtInt verts[], RtInt n, RtToken tokens[],
    RtPointer parms[]);
```

⟨*RI Function Definitions*⟩+≡

```
RtVoid RiGeneralPolygon(RtInt nloops, RtInt nverts[], ...) { RI_UNIMP(); }
RtVoid RiGeneralPolygonV(RtInt nloops, RtInt nverts[], RtInt n,
    RtToken tokens[], RtPointer parms[]) { RI_UNIMP(); }
RtVoid RiPointsGeneralPolygons(RtInt npolys, RtInt nloops[], RtInt nverts[],
    RtInt verts[], ...) { RI_UNIMP(); }
RtVoid RiPointsGeneralPolygonsV(RtInt npolys, RtInt nloops[],
    RtInt nverts[], RtInt verts[], RtInt n, RtToken tokens[],
    RtPointer parms[]) { RI_UNIMP(); }
```

C.8 Light Sources

⟨*RI Types*⟩+≡
```
typedef RtPointer RtLightHandle;
```

⟨*RI Function Declarations*⟩+≡
```
extern RtLightHandle RiLightSource(RtToken name, ...);
extern RtLightHandle RiLightSourceV(RtToken name, RtInt n, RtToken tokens[],
    RtPointer parms[]);
```

⟨*RI Function Definitions*⟩+≡
```
RtLightHandle RiLightSourceV(RtToken name, RtInt n, RtToken tokens[],
          RtPointer params[]) {
    SurfaceFunction *data = initSurfaceFunction(0, n, tokens, params);
    ⟨Prepare light attributes for binding⟩
    Light *lt;
    lt = PointLightCreate(name, data, curLightAttributes);
    if (lt == RI_NULL) {
        ⟨Complain about unknown point light type⟩
    }
    else
        lights.push_back(lt);
    return lt;
}
```

⟨*Complain about unknown point light type*⟩≡
```
curLightAttributes->Dereference();
delete data;
Error("RiLightSource: point light type '%s' unknown.", name);
```

⟨*RI Function Declarations*⟩+≡
```
extern RtLightHandle RiAreaLightSource(RtToken name, ...);
extern RtLightHandle RiAreaLightSourceV(RtToken name, RtInt n, RtToken tokens[],
    RtPointer parms[]);
```

⟨*PrimitiveAttributes Public Data*⟩+≡
```
AreaLight *LightShader;
```

⟨*PrimitiveAttributes constructor implementation*⟩+≡
```
LightShader = NULL;
```

⟨*RI Function Definitions*⟩+≡
```
RtLightHandle RiAreaLightSourceV(RtToken name, RtInt n, RtToken tokens[],
        RtPointer params[] ) {
    SurfaceFunction *data = initSurfaceFunction(0, n, tokens, params);
    ⟨Prepare light attributes for binding⟩
    AreaLight *lt = AreaLightCreate(name, data, curLightAttributes);
    ⟨Update geometric attributes for changing value⟩
    curPrimitiveAttributes->LightShader = lt;
    if (lt == RI_NULL) {
        ⟨Complain about unknown area light type⟩
    }
    else
        lights.push_back(lt);
    return lt;
}
```

⟨*Complain about unknown area light type*⟩≡
```
curLightAttributes->Dereference();
delete data;
Error("RiAreaLightSource: area light type '%s' unknown.", name);
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiIlluminate(RtLightHandle light, RtBoolean on);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiIlluminate(RtLightHandle light, RtBoolean on) {
    if (on) {
        ⟨Add the light to the light list if not there already⟩
    }
    else {
        ⟨Remove the light from the light list⟩
    }
}
```

⟨*Add the light to the light list if not there already*⟩≡
```
list<Light *>::iterator iter = lights.begin();
while (iter != lights.end()) {
    if (*iter == light) {
        Info("Trying to Illuminate light that is already on.");
        return;
    }
    ++iter;
}
lights.push_front((Light *)light);
```

⟨*Remove the light from the light list*⟩≡
```
list<Light *>::iterator iter = lights.begin();
while (iter != lights.end()) {
    if (*iter == light) {
        lights.erase(iter);
        return;
    }
    ++iter;
}
Error("Trying to de-Illuminate an unknown light!");
```

C.9 Motion and Animation

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiMotionBegin(RtInt N, ...) {
    RtFloat times[16];
    Assert(N < 16);
    va_list args;
    va_start(args, N);
    for (int i = 0; i < N; ++i)
        times[i] = va_arg(args, double);
    RiMotionBeginV(N, times);
}
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiMotionBeginV(RtInt N, RtFloat times[]) {
    Assert(!inMotionBlock);
    inMotionBlock = true;
    motionLevel = 0;
    if (N > 2)
        Warning("Only two levels in motion block will be used.");
}
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiMotionEnd() {
    if (!inMotionBlock)
        Error("Unmatched MotionEnd statement");
    inMotionBlock = false;
}
```

D.RI Details

D.1 Default RI Tokens

⟨*RI Constants*⟩+≡

```
extern RtToken RI_A, RI_ABORT, RI_AMBIENTLIGHT, RI_AMPLITUDE, RI_AZ,
       RI_BACKGROUND, RI_BEAMDISTRIBUTION, RI_BICUBIC, RI_BILINEAR, RI_BLACK,
       RI_BUMPY, RI_CAMERA, RI_CLAMP, RI_COMMENT, RI_CONEANGLE, RI_CONEDELTAANGLE,
       RI_CONSTANT, RI_CS, RI_DEPTHCUE, RI_DIFFERENCE, RI_DISTANCE,
       RI_DISTANTLIGHT, RI_FILE, RI_FLATNESS, RI_FOG, RI_FOV, RI_FRAMEBUFFER,
       RI_FROM, RI_HANDLER, RI_HIDDEN, RI_IDENTIFIER, RI_IGNORE, RI_INSIDE,
       RI_INTENSITY, RI_INTERSECTION, RI_KA, RI_KD, RI_KR, RI_KS, RI_LH,
       RI_LIGHTCOLOR, RI_MATTE, RI_MAXDISTANCE, RI_METAL, RI_MINDISTANCE, RI_N,
       RI_NAME, RI_NONPERIODIC, RI_NP, RI_OBJECT, RI_ORIGIN, RI_ORTHOGRAPHIC,
       RI_OS, RI_OUTSIDE, RI_P, RI_PAINT, RI_PAINTEDPLASTIC, RI_PERIODIC,
       RI_PERSPECTIVE, RI_PLASTIC, RI_POINTLIGHT, RI_PRIMITIVE, RI_PRINT, RI_PW,
       RI_PZ, RI_RASTER, RI_RGB, RI_RGBA, RI_RGBAZ, RI_RGBZ, RI_RH, RI_ROUGHNESS,
       RI_S, RI_SCREEN, RI_SHADINGGROUP, RI_SHINYMETAL, RI_SMOOTH,
       RI_SPECULARCOLOR, RI_SPOTLIGHT, RI_ST, RI_STRUCTURE, RI_T, RI_TEXTURENAME,
       RI_TO, RI_TRIMDEVIATION, RI_UNION, RI_WORLD, RI_Z;
```

⟨*API Global Data*⟩≡

```
RtToken RI_A, RI_ABORT, RI_AMBIENTLIGHT, RI_AMPLITUDE, RI_AZ,
       RI_BACKGROUND, RI_BEAMDISTRIBUTION, RI_BICUBIC, RI_BILINEAR, RI_BLACK,
       RI_BUMPY, RI_CAMERA, RI_CLAMP, RI_COMMENT, RI_CONEANGLE, RI_CONEDELTAANGLE,
       RI_CONSTANT, RI_CS, RI_DEPTHCUE, RI_DIFFERENCE, RI_DISTANCE,
       RI_DISTANTLIGHT, RI_FILE, RI_FLATNESS, RI_FOG, RI_FOV, RI_FRAMEBUFFER,
       RI_FROM, RI_HANDLER, RI_HIDDEN, RI_IDENTIFIER, RI_IGNORE, RI_INSIDE,
       RI_INTENSITY, RI_INTERSECTION, RI_KA, RI_KD, RI_KR, RI_KS, RI_LH,
       RI_LIGHTCOLOR, RI_MATTE, RI_MAXDISTANCE, RI_METAL, RI_MINDISTANCE, RI_N,
       RI_NAME, RI_NONPERIODIC, RI_NP, RI_OBJECT, RI_ORIGIN, RI_ORTHOGRAPHIC,
       RI_OS, RI_OUTSIDE, RI_P, RI_PAINT, RI_PAINTEDPLASTIC, RI_PERIODIC,
       RI_PERSPECTIVE, RI_PLASTIC, RI_POINTLIGHT, RI_PRIMITIVE, RI_PRINT, RI_PW,
       RI_PZ, RI_RASTER, RI_RGB, RI_RGBA, RI_RGBAZ, RI_RGBZ, RI_RH, RI_ROUGHNESS,
       RI_S, RI_SCREEN, RI_SHADINGGROUP, RI_SHINYMETAL, RI_SMOOTH,
       RI_SPECULARCOLOR, RI_SPOTLIGHT, RI_ST, RI_STRUCTURE, RI_T, RI_TEXTURENAME,
       RI_TO, RI_TRIMDEVIATION, RI_UNION, RI_WORLD, RI_Z;
```

⟨*RtToken Initialization*⟩≡

```
RI_A                = RiDeclare("a", RI_NULL);
RI_ABORT            = RiDeclare("abort", RI_NULL);
RI_AMBIENTLIGHT     = RiDeclare("ambientlight", RI_NULL);
RI_AMPLITUDE        = RiDeclare("amplitude", RI_NULL);
RI_AZ               = RiDeclare("az", RI_NULL);
RI_BACKGROUND       = RiDeclare("background", RI_NULL);
RI_BEAMDISTRIBUTION = RiDeclare("beamdistribution", RI_NULL);
RI_BICUBIC          = RiDeclare("bicubic", RI_NULL);
RI_BILINEAR         = RiDeclare("bilinear", RI_NULL);
RI_BLACK            = RiDeclare("black", RI_NULL);
RI_BUMPY            = RiDeclare("bumpy", RI_NULL);
RI_CAMERA           = RiDeclare("camera", RI_NULL);
RI_CLAMP            = RiDeclare("clamp", RI_NULL);
RI_COMMENT          = RiDeclare("comment", RI_NULL);
RI_CONEANGLE        = RiDeclare("coneangle", "float");
RI_CONEDELTAANGLE   = RiDeclare("conedeltaangle", "float");
RI_CONSTANT         = RiDeclare("constant", RI_NULL);
RI_CS               = RiDeclare("Cs", "vertex color");
RI_DEPTHCUE         = RiDeclare("depthcue", RI_NULL);
RI_DIFFERENCE       = RiDeclare("difference", RI_NULL);
RI_DISTANCE         = RiDeclare("distance", RI_NULL);
RI_DISTANTLIGHT     = RiDeclare("distantlight", RI_NULL);
RI_FILE             = RiDeclare("file", RI_NULL);
RI_FLATNESS         = RiDeclare("flatness", RI_NULL);
RI_FOG              = RiDeclare("fog", RI_NULL);
RI_FOV              = RiDeclare("fov", "float");
RI_FRAMEBUFFER      = RiDeclare("framebuffer", RI_NULL);
RI_FROM             = RiDeclare("from", "point");
RI_HANDLER          = RiDeclare("handler", RI_NULL);
RI_HIDDEN           = RiDeclare("hidden", RI_NULL);
RI_IDENTIFIER       = RiDeclare("identifier", "string");
RI_IGNORE           = RiDeclare("ignore", RI_NULL);
RI_INSIDE           = RiDeclare("inside", RI_NULL);
RI_INTENSITY        = RiDeclare("intensity", "float");
RI_INTERSECTION     = RiDeclare("intersection", RI_NULL);
RI_KA               = RiDeclare("Ka", "float");
RI_KD               = RiDeclare("Kd", "float");
RI_KR               = RiDeclare("Kr", "float");
RI_KS               = RiDeclare("Ks", "float");
RI_LH               = RiDeclare("lh", RI_NULL);
RI_LIGHTCOLOR       = RiDeclare("lightcolor", "color");
RI_MATTE            = RiDeclare("matte", RI_NULL);
RI_MAXDISTANCE      = RiDeclare("maxdistance", "float");
RI_METAL            = RiDeclare("metal", RI_NULL);
RI_MINDISTANCE      = RiDeclare("mindistance", "float");
RI_N                = RiDeclare("N", "vertex normal");
RI_NAME             = RiDeclare("name", "string");
RI_NONPERIODIC      = RiDeclare("nonperiodic", RI_NULL);
RI_NP               = RiDeclare("Np", RI_NULL);
RI_OBJECT           = RiDeclare("object", RI_NULL);
RI_ORIGIN           = RiDeclare("origin", "point");
RI_ORTHOGRAPHIC     = RiDeclare("orthographic", RI_NULL);
RI_OS               = RiDeclare("Os", "float");
```

```
RI_OUTSIDE          = RiDeclare("outside", RI_NULL);
RI_P                = RiDeclare("P", "vertex point");
RI_PAINT            = RiDeclare("paint", RI_NULL);
RI_PAINTEDPLASTIC   = RiDeclare("paintedplastic", RI_NULL);
RI_PERIODIC         = RiDeclare("periodic", RI_NULL);
RI_PERSPECTIVE      = RiDeclare("perspective", RI_NULL);
RI_PLASTIC          = RiDeclare("plastic", RI_NULL);
RI_POINTLIGHT       = RiDeclare("pointlight", RI_NULL);
RI_PRIMITIVE        = RiDeclare("primitive", RI_NULL);
RI_PRINT            = RiDeclare("print", RI_NULL);
RI_PW               = RiDeclare("Pw", "vertex hpoint");
RI_PZ               = RiDeclare("Pz", "vertex float");
RI_RASTER           = RiDeclare("raster", RI_NULL);
RI_RGB              = RiDeclare("rgb", RI_NULL);
RI_RGBA             = RiDeclare("rgba", RI_NULL);
RI_RGBAZ            = RiDeclare("rgbaz", RI_NULL);
RI_RGBZ             = RiDeclare("rgbz", RI_NULL);
RI_RH               = RiDeclare("rh", RI_NULL);
RI_ROUGHNESS        = RiDeclare("roughness", "float");
RI_S                = RiDeclare("s", "vertex float");
RI_SCREEN           = RiDeclare("screen", RI_NULL);
RI_SHADINGGROUP     = RiDeclare("shadinggroup", RI_NULL);
RI_SHINYMETAL       = RiDeclare("shinymetal", RI_NULL);
RI_SMOOTH           = RiDeclare("smooth", RI_NULL);
RI_SPECULARCOLOR    = RiDeclare("specularcolor", "color");
RI_SPOTLIGHT        = RiDeclare("spotlight", RI_NULL);
//RI_ST               = RiDeclare("st", "vertex float[2]");  // XXX
RI_STRUCTURE        = RiDeclare("structure", RI_NULL);
RI_T                = RiDeclare("t", "vertex float");
RI_TEXTURENAME      = RiDeclare("texturename", "string");
RI_TO               = RiDeclare("to", "point");
RI_TRIMDEVIATION    = RiDeclare("trimdeviation", "float");
RI_UNION            = RiDeclare("union", RI_NULL);
RI_WORLD            = RiDeclare("world", RI_NULL);
RI_Z                = RiDeclare("z", RI_NULL);
```

D.2 LRT Defined Tokens

⟨*RI Constants*⟩+≡

```
extern RtToken LRT_IMAGEVIEWER;
extern RtToken LRT_RENDER, LRT_DISPLAY;
extern RtToken LRT_INTEGRATOR, LRT_SHADOWS, LRT_COLOR;
extern RtToken LRT_WHITTED, LRT_RMAN, LRT_MONTECARLO;
extern RtToken LRT_SURFACE, LRT_COMBINATION, LRT_SAMPLE;
extern RtToken LRT_LIGHT, LRT_RAYCAST, LRT_NSAMPLES;
extern RtToken LRT_VIEWST, LRT_GLASS, LRT_INDEX, LRT_KT;
extern RtToken LRT_CHECKERED, LRT_CHECKCOLOR1, LRT_CHECKCOLOR2;
extern RtToken LRT_CHECKFREQUENCY, LRT_DIFFUSE;
extern RtToken LRT_TYPE, LRT_SHUTTER;
extern RtToken LRT_PINHOLE, LRT_MBDOF;
extern RtToken LRT_IRIS, LRT_STRIPE;
extern RtToken LRT_IRIS_RATE, LRT_STRIPE_WIDTH, LRT_STRIPE_DIRECTION;
extern RtToken LRT_DOWN, LRT_UP, LRT_LEFT, LRT_RIGHT;
```

⟨*API Global Data*⟩+≡

```
RtToken LRT_IMAGEVIEWER;
RtToken LRT_RENDER, LRT_DISPLAY;
RtToken LRT_INTEGRATOR, LRT_SHADOWS, LRT_COLOR;
RtToken LRT_WHITTED, LRT_RMAN, LRT_MONTECARLO;
RtToken LRT_SURFACE, LRT_COMBINATION, LRT_SAMPLE;
RtToken LRT_LIGHT, LRT_RAYCAST, LRT_NSAMPLES;
RtToken LRT_VIEWST, LRT_GLASS, LRT_INDEX, LRT_KT;
RtToken LRT_CHECKERED, LRT_CHECKCOLOR1, LRT_CHECKCOLOR2, LRT_CHECKFREQUENCY;
RtToken LRT_DIFFUSE;
RtToken LRT_TYPE, LRT_SHUTTER;
RtToken LRT_PINHOLE, LRT_MBDOF;
RtToken LRT_IRIS, LRT_STRIPE;
RtToken LRT_IRIS_RATE, LRT_STRIPE_WIDTH, LRT_STRIPE_DIRECTION;
RtToken LRT_DOWN, LRT_UP, LRT_LEFT, LRT_RIGHT;
```

⟨*RtToken Initialization*⟩+≡

```
LRT_RAYCAST = RiDeclare("raycast", RI_NULL);
LRT_CHECKCOLOR1 = RiDeclare("checkcolor1", "color");
LRT_CHECKCOLOR2 = RiDeclare("checkcolor2", "color");
LRT_CHECKERED = RiDeclare("checkered", RI_NULL);
LRT_CHECKFREQUENCY = RiDeclare("checkfrequency", "float");
LRT_COLOR = RiDeclare("color", RI_NULL);
LRT_DISPLAY = RiDeclare("display", RI_NULL);
LRT_GLASS = RiDeclare("glass", RI_NULL);
LRT_IMAGEVIEWER = RiDeclare("imageviewer", "string");
LRT_INDEX = RiDeclare("index", "float");
LRT_INTEGRATOR = RiDeclare("integrator", "string");
LRT_KT = RiDeclare("Kt", "float");
LRT_LIGHT = RiDeclare("light", RI_NULL);
LRT_NSAMPLES = RiDeclare("nsamples", "float");
LRT_RENDER = RiDeclare("render", RI_NULL);
LRT_SHADOWS = RiDeclare("shadows", "string");
LRT_VIEWST = RiDeclare("viewst", RI_NULL);
LRT_WHITTED = RiDeclare("whitted", RI_NULL);
LRT_MONTECARLO = RiDeclare("montecarlo", RI_NULL);
LRT_RMAN = RiDeclare("renderman", RI_NULL);
LRT_DIFFUSE = RiDeclare("diffuse", RI_NULL);
LRT_SURFACE = RiDeclare("surface", RI_NULL);
LRT_COMBINATION = RiDeclare("combination", RI_NULL);
LRT_SAMPLE = RiDeclare("sample", RI_NULL);
LRT_PINHOLE = RiDeclare("pinhole", "string" );
LRT_MBDOF = RiDeclare("mbdof", "string" );
LRT_IRIS = RiDeclare("iris", "string" );
LRT_STRIPE = RiDeclare("stripe", "string" );
LRT_TYPE = RiDeclare("type", "string" );
LRT_SHUTTER = RiDeclare("shutter", "string" );
LRT_IRIS_RATE = RiDeclare("iris_rate", "float" );
LRT_STRIPE_WIDTH = RiDeclare("stripe_width", "float" );
LRT_STRIPE_DIRECTION = RiDeclare("stripe_direction", "string" );
LRT_DOWN = RiDeclare( "down", "string" );
LRT_UP = RiDeclare( "up", "string" );
LRT_LEFT = RiDeclare( "left", "string" );
LRT_RIGHT = RiDeclare( "right", "string" );
```

D.3 Tokens for compatibility with BMRT and PRMAN

⟨*RI Constants*⟩+≡

```
extern RtToken LRT_FORMAT, LRT_BUCKETSIZE, LRT_GRIDSIZE, LRT_TEXTUREMEMORY;
extern RtToken LRT_ZTHRESHOLD, LRT_EXTREMEDISPLACEMENT, LRT_EYESPLITS;
extern RtToken LRT_GEOMMEMORY, LRT_BIAS0, LRT_BIAS1, LRT_SHADER, LRT_TEXTURE;
extern RtToken LRT_VFXMASTER, LRT_VFXINSTANCE, LRT_ARCHIVE, LRT_RESOURCE;
extern RtToken LRT_MINSAMPLES, LRT_MAXSAMPLES, LRT_MAX_RAYLEVEL, LRT_BRANCH_RATIO;
extern RtToken LRT_MAX_BRANCH_RATIO, LRT_MINSHADOWBIAS, LRT_STEPS, LRT_MINPATCHSAMPLES;
extern RtToken LRT_VERBOSITY, LRT_INCLUDE, LRT_REFRESH, LRT_FULLCOLOR;
extern RtToken LRT_SETPOINTS, LRT_NAME, LRT_SHADINGGROUP, LRT_BINARY;
extern RtToken LRT_COORDINATESYSTEM, LRT_SPHERE, LRT_SENSE, LRT_AVERAGECOLOR;
extern RtToken LRT_EMISSIONCOLOR, LRT_PATCHSIZ, LRT_ELEMSIZE, LRT_MINSIZE;
extern RtToken LRT_ZONAL, LRT_CASTS_SHADOWS, LRT_PATCH_MAXLEVEL, LRT_PATCH_MINLEVEL;
extern RtToken LRT_PATCH_MULTIPLIER, LRT_TRUEDISPLACEMENT, LRT_CACHE;
extern RtToken LRT_UDIVISIONS, LRT_VDIVISIONS, LRT_ORIGIN;
extern RtToken LRT_MERGE, LRT_COMPRESSION, LRT_RESOLUTION, LRT_RESOLUTIONUNIT;
extern RtToken LRT_JITTER, LRT_PDISC, LRT_FLATNESS, LRT_WIDTH, LRT_CONSTANTWIDTH;
```

⟨*API Global Data*⟩+≡

```
RtToken LRT_FORMAT, LRT_BUCKETSIZE, LRT_GRIDSIZE, LRT_TEXTUREMEMORY;
RtToken LRT_ZTHRESHOLD, LRT_EXTREMEDISPLACEMENT, LRT_EYESPLITS;
RtToken LRT_GEOMMEMORY, LRT_BIAS0, LRT_BIAS1, LRT_SHADER, LRT_TEXTURE;
RtToken LRT_VFXMASTER, LRT_VFXINSTANCE, LRT_ARCHIVE, LRT_RESOURCE;
RtToken LRT_MINSAMPLES, LRT_MAXSAMPLES, LRT_MAX_RAYLEVEL, LRT_BRANCH_RATIO;
RtToken LRT_MAX_BRANCH_RATIO, LRT_MINSHADOWBIAS, LRT_STEPS, LRT_MINPATCHSAMPLES;
RtToken LRT_VERBOSITY, LRT_INCLUDE, LRT_REFRESH, LRT_FULLCOLOR;
RtToken LRT_SETPOINTS, LRT_NAME, LRT_SHADINGGROUP, LRT_BINARY;
RtToken LRT_COORDINATESYSTEM, LRT_SPHERE, LRT_SENSE, LRT_AVERAGECOLOR;
RtToken LRT_EMISSIONCOLOR, LRT_PATCHSIZE, LRT_ELEMSIZE, LRT_MINSIZE;
RtToken LRT_ZONAL, LRT_CASTS_SHADOWS, LRT_PATCH_MAXLEVEL, LRT_PATCH_MINLEVEL;
RtToken LRT_PATCH_MULTIPLIER, LRT_TRUEDISPLACEMENT, LRT_CACHE;
RtToken LRT_UDIVISIONS, LRT_VDIVISIONS, LRT_ORIGIN;
RtToken LRT_MERGE, LRT_COMPRESSION, LRT_RESOLUTION, LRT_RESOLUTIONUNIT;
RtToken LRT_JITTER, LRT_PDISC, LRT_FLATNESS, LRT_WIDTH, LRT_CONSTANTWIDTH;
```

⟨*RtToken Initialization*⟩+≡

```
  LRT_FORMAT              = RiDeclare( "format", "string" );
  LRT_BUCKETSIZE          = RiDeclare( "bucketsize", /*int*/ "float" /*[2]*/);
  LRT_GRIDSIZE            = RiDeclare( "gridsize", /*int*/ "float" /*[2]*/);
  LRT_TEXTUREMEMORY       = RiDeclare( "texturememory", /*int*/ "float" );
  LRT_ZTHRESHOLD          = RiDeclare( "zthreshold", "point" );
  LRT_EXTREMEDISPLACEMENT = RiDeclare( "extremedisplacement", /*int*/ "float" );
  LRT_EYESPLITS           = RiDeclare( "eyesplits", /*int*/ "float" );
  LRT_GEOMMEMORY          = RiDeclare( "geommemory", /*int*/ "float" );
  LRT_BIAS0               = RiDeclare( "bias0", "float" );
  LRT_BIAS1               = RiDeclare( "bias1", "float" );
  LRT_SHADER              = RiDeclare( "shader", "string" );
  LRT_TEXTURE             = RiDeclare( "texture", "string" );
  LRT_VFXMASTER           = RiDeclare( "vfxmaster", "string" );
  LRT_VFXINSTANCE         = RiDeclare( "vfxinstance", "string" );
  LRT_ARCHIVE             = RiDeclare( "archive", "string" ); /* Added for PRMan 3.8 */
  LRT_RESOURCE            = RiDeclare( "resource", "string" ); /* Missing item added. */
  LRT_MINSAMPLES          = RiDeclare( "minsamples", /*int*/ "float" );
  LRT_MAXSAMPLES          = RiDeclare( "maxsamples", /*int*/ "float" );
  LRT_MAX_RAYLEVEL        = RiDeclare( "max_raylevel", /*int*/ "float" );
  LRT_BRANCH_RATIO        = RiDeclare( "branch_ratio", /*int*/ "float" );
  LRT_MAX_BRANCH_RATIO    = RiDeclare( "max_branch_ratio", /*int*/ "float" );
  LRT_MINSHADOWBIAS       = RiDeclare( "minshadowbias", "float" );
  LRT_STEPS               = RiDeclare( "steps", /*int*/ "float" );
  LRT_MINPATCHSAMPLES     = RiDeclare( "minpatchsamples", /*int*/ "float" );
  LRT_VERBOSITY           = RiDeclare( "verbosity", "string" );
  LRT_INCLUDE             = RiDeclare( "include", "string" );
  LRT_REFRESH             = RiDeclare( "refresh",   /*int*/ "float" );  /* See [PIXA93b]. */
  LRT_FULLCOLOR           = RiDeclare( "fullcolor", /*int*/ "float" );  /* See [PIXA93b]. */
  LRT_SETPOINTS           = RiDeclare( "setpoints", /*int*/ "float"/*[2]*/ );  /* See [PIXA98]. */
  LRT_NAME                = RiDeclare( "name", "string" );
      /* already declared as RI_NULL?? */
  /*LRT_SHADINGGROUP      = RiDeclare( "shadinggroup", "string" );*/

  LRT_BINARY              = RiDeclare( "binary", /*int*/ "float" );
  LRT_COORDINATESYSTEM    = RiDeclare( "coordinatesystem", "string" );
  LRT_SPHERE              = RiDeclare( "sphere", "float" );
  LRT_SENSE               = RiDeclare( "sense", "string" );
  LRT_AVERAGECOLOR        = RiDeclare( "averagecolor", "color" );
  LRT_EMISSIONCOLOR       = RiDeclare( "emissioncolor", "color" );
  LRT_PATCHSIZE           = RiDeclare( "patchsize", /*int*/ "float" );
  LRT_ELEMSIZE            = RiDeclare( "elemsize", /*int*/ "float" );
  LRT_MINSIZE             = RiDeclare( "minsize", /*int*/ "float" );
  LRT_ZONAL               = RiDeclare( "zonal", "string" );
  LRT_CASTS_SHADOWS       = RiDeclare( "casts_shadows", "string" );
  LRT_PATCH_MAXLEVEL      = RiDeclare( "patch_maxlevel", /*int*/ "float" );
  LRT_PATCH_MINLEVEL      = RiDeclare( "patch_minlevel", /*int*/ "float" ); /* 2.3.5 */
  LRT_PATCH_MULTIPLIER    = RiDeclare( "patch_multiplier", /*int*/ "float" ); /* 2.3.6? */
  LRT_TRUEDISPLACEMENT    = RiDeclare( "truedisplacement", /*int*/ "float" ); /* 2.3.5 */
  LRT_CACHE               = RiDeclare( "cache", "string" );
  LRT_UDIVISIONS          = RiDeclare( "udivisions", /*int*/ "float" ); /* See [PIXA93b]. */
  LRT_VDIVISIONS          = RiDeclare( "vdivisions", /*int*/ "float" ); /* See [PIXA93b]. */

      /* already declared??*/
```

```
/*LRT_ORIGIN                = RiDeclare( "origin", int "float[2]" );*/

LRT_MERGE               = RiDeclare( "merge", /*int*/ "float" );
LRT_COMPRESSION         = RiDeclare( "compression",    "string" );   /* See [PIXA93a].
LRT_RESOLUTION          = RiDeclare( "resolution",     /*int*/ "float" /*[2]*/);  /* Se
LRT_RESOLUTIONUNIT      = RiDeclare( "resolutionunit", "string" );   /* See [PIXA93a].
LRT_JITTER              = RiDeclare( "jitter", /*int*/ "float" );
LRT_PDISC               = RiDeclare( "pdisc", /*int*/ "float" );
/*LRT_FLATNESS          = RiDeclare( "flatness", "float" );*/ /* already declared?*/
LRT_WIDTH               = RiDeclare( "width", "vertex float" );
LRT_CONSTANTWIDTH       = RiDeclare( "constantwidth", "constant float" );
```

D.4 Unsupported RI Calls

Explain why these guys aren't supported...

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiPixelVariance(RtFloat variation);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiPixelVariance(RtFloat) { }
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiTextureCoordinates(RtFloat s1, RtFloat t1, RtFloat s2, RtFloat t2,
        RtFloat s3, RtFloat t3, RtFloat s4, RtFloat t4);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiTextureCoordinates(RtFloat s1, RtFloat t1, RtFloat s2, RtFloat t2,
        RtFloat s3, RtFloat t3, RtFloat s4, RtFloat t4) { RI_UNIMP( ); }
```

⟨*RI Function Declarations*⟩+≡
```
extern RtVoid RiShadingRate(RtFloat r);
extern RtVoid RiShadingInterpolation(RtToken type);
```

⟨*RI Function Definitions*⟩+≡
```
RtVoid RiShadingRate(RtFloat r) { }
RtVoid RiShadingInterpolation(RtToken type) { }
```

⟨*RI Variable Declarations*⟩≡
```
extern RtInt RiLastError;
```

⟨*RI Types*⟩+≡
```
typedef RtVoid (*RtErrorHandler)(RtInt code, RtInt severity, char *msg);
```

E.A NURBS Primitive

⟨*NURBS Methods*⟩≡
```
NURBS(int nu, int uorder, Float *uknot, Float umin, Float umax,
    int nv, int vorder, Float *vknot, Float vmin, Float vmax,
    Float *P, bool isHomogeneous, PrimitiveAttributes *attributes,
    SurfaceFunction *surf);
```

⟨*NURBS Definitions*⟩≡
```
NURBS::NURBS(int numu, int uo, Float *uk, Float u0, Float u1,
    int numv, int vo, Float *vk, Float v0, Float v1,
    Float *p, bool homogeneous, PrimitiveAttributes *attr,
    SurfaceFunction *surf)
        : Primitive(attr, surf) {
    nu = numu;    uorder = uo;
    umin = u0;    umax = u1;
    nv = numv;    vorder = vo;
    vmin = v0;    vmax = v1;
    isHomogeneous = homogeneous;
    if (isHomogeneous) {
        P = new Float[4*nu*nv];
        memcpy(P, p, 4*nu*nv*sizeof(Float));
    } else {
        P = new Float[3*nu*nv];
        memcpy(P, p, 3*nu*nv*sizeof(Float));
    }
    uknot = new Float[nu + uorder];
    memcpy(uknot, uk, (nu + uorder) * sizeof(Float));
    vknot = new Float[nv + vorder];
    memcpy(vknot, vk, (nv + vorder) * sizeof(Float));
}
```

⟨*NURBS Data*⟩≡
```
int nu, uorder, nv, vorder;
Float umin, umax, vmin, vmax;
Float *uknot, *vknot;
bool isHomogeneous;
Float *P;
```

⟨*NURBS Methods*⟩+≡
```
~NURBS();
```

⟨*NURBS Definitions*⟩+≡
```
NURBS::~NURBS() {
    delete[] P;
    delete[] uknot;
    delete[] vknot;
}
```

⟨*NURBS Methods*⟩+≡
```
virtual BBox BoundObjectSpace() const;
virtual BBox BoundWorldSpace() const;
```

⟨*NURBS Definitions*⟩+≡
```
BBox NURBS::BoundObjectSpace() const {
    if (!isHomogeneous) {
        ⟨Compute object-space bound of non-homogeneous NURBS⟩
    } else {
        ⟨Compute object-space bound of homogeneous NURBS⟩
    }
}
```

⟨*Compute object-space bound of non-homogeneous NURBS*⟩≡
```
Float *pp = P;
BBox bound = Point(pp[0], pp[1], pp[2]);
for (int i = 0; i < nu*nv; ++i, pp += 3)
    bound = Union(bound, Point(pp[0], pp[1], pp[2]));
return bound;
```

⟨*Compute object-space bound of homogeneous NURBS*⟩≡
```
Float *pp = P;
BBox bound = Point(pp[0] / pp[3], pp[1] / pp[3], pp[2] / pp[3]);
for (int i = 0; i < nu*nv; ++i, pp += 4)
    bound = Union(bound, Point(pp[0] / pp[3], pp[1] / pp[3], pp[2] / pp[3]));
return bound;
```

⟨*NURBS Definitions*⟩+≡
```
BBox NURBS::BoundWorldSpace() const {
    if (!isHomogeneous) {
        ⟨Compute world-space bound of non-homogeneous NURBS⟩
    } else {
        ⟨Compute world-space bound of homogeneous NURBS⟩
    }
}
```

⟨*Compute world-space bound of non-homogeneous NURBS*⟩≡
```
Float *pp = P;
Point pt = attributes->ObjectToWorld(Point(pp[0], pp[1], pp[2]));
BBox bound = pt;
for (int i = 0; i < nu*nv; ++i, pp += 3) {
    pt = attributes->ObjectToWorld(Point(pp[0], pp[1], pp[2]));
    bound = Union(bound, pt);
}
return bound;
```

⟨*NURBS Methods*⟩+≡
```
virtual bool CanIntersect() const { return false; }
```

⟨*NURBS Methods*⟩+≡
```
virtual void Refine(vector<Primitive *> *refined) const;
```

⟨*NURBS Definitions*⟩+≡
```
void NURBS::Refine(vector<Primitive *> *refined) const {
    ⟨Compute NURBS dicing rates⟩
    ⟨Evaluate NURBS over grid of points⟩
    ⟨Generate points-polygons mesh⟩
    ⟨Cleanup from NURBS refinement⟩
}
```

⟨*Compute NURBS dicing rates*⟩≡
```
int diceu = 30, dicev = 30;
Float *ueval = new Float[diceu];
Float *veval = new Float[dicev];
Point *evalPs = new Point[diceu*dicev];
Point *evalNs = new Point[diceu*dicev];
int i;
for (i = 0; i < diceu; ++i)
    ueval[i] = Lerp((Float)i / (Float)(diceu-1), umin, umax);
for (i = 0; i < dicev; ++i)
    veval[i] = Lerp((Float)i / (Float)(dicev-1), vmin, vmax);
```

⟨*Evaluate NURBS over grid of points*⟩≡
```
memset(evalPs, 0, diceu*dicev*sizeof(Point));
memset(evalNs, 0, diceu*dicev*sizeof(Point));
```

⟨*Generate points-polygons mesh*⟩≡
```
int nTris = 2*(diceu-1)*(dicev-1);
int *vertices = new int[3 * nTris];
int *vertp = vertices;
⟨Compute the vertex offset numbers for the triangles⟩
refined->push_back(new TriangleMesh(nTris, diceu*dicev, vertices,
    evalPs, attributes, NULL));
// XXX Need to make a new surface function and hold the N values
```

⟨*Compute the vertex offset numbers for the triangles*⟩≡
```
for (int v = 0; v < dicev-1; ++v) {
    for (int u = 0; u < diceu-1; ++u) {
#define VN(u,v) ((v)*diceu+(u))
        *vertp++ = VN(u,   v);
        *vertp++ = VN(u+1, v);
        *vertp++ = VN(u+1, v+1);

        *vertp++ = VN(u,   v);
        *vertp++ = VN(u+1, v+1);
        *vertp++ = VN(u,   v+1);
#undef VN
    }
}
```

¡¡Complain about `vertex` data that we'll be ignoring¿¿

⟨*Cleanup from NURBS refinement*⟩≡
```
delete[] ueval;
delete[] veval;
delete[] evalPs;
delete[] evalNs;
```

⟨*Turn NURBS into triangles*⟩≡
```
    Pw = (Homogeneous3 *)alloca(nu*nv*sizeof(Homogeneous3));
    for (int i = 0; i < nu*nv; ++i) {
        Pw[i].x = P[i].x;
        Pw[i].y = P[i].y;
        Pw[i].z = P[i].z;
        Pw[i].w = 1.;
    }
}
for (int i = 0; i < NP; ++i) {
    for (int j = 0; j < NP; ++j) {
        Vector dPdu, dPdv;
        Point pt = NURBSEvaluateSurface(uorder, uknot, nu, u,
            vorder, vknot, nv, v, Pw, &dPdu, &dPdv);
        pts[i][j][0] = pt.x;
        pts[i][j][1] = pt.y;
        pts[i][j][2] = pt.z;

        Normal N = Normal(Cross(dPdu, dPdv).Hat());
        nrms[i][j][0] = N.x;
        nrms[i][j][1] = N.y;
        nrms[i][j][2] = N.z;
    }
}
```

⟨*Complain about* `vertex` *data that we'll be ignoring*⟩≡
```
#if 0
// BASE it on the surface func...
for (int i = 0; i < n; ++i) {
    if (tokens[i] != RI_P && strcmp(tokens[i], RI_P) != 0 &&
        tokens[i] != RI_PW && strcmp(tokens[i], RI_PW) != 0) {
        RtToken t;
        int type;
        if (lookupTokenAndType(tokens[i], &t, &type))
            if (type & TOKEN_TYPE_VERTEX)
                Warning("Vertex parameter \'%s\' is being ignored for spline patch.",
                    tokens[i]);
    }
}
#endif
```

E.1 Evaluation Routines

⟨*NURBS Evaluation Functions*⟩≡
```
static int KnotOffset(const float *knot, int order, int np, float t) {
    int firstKnot = order - 1;
    int lastKnot = np;

    int knotOffset = firstKnot;
    while (t > knot[knotOffset+1])
    ++knotOffset;
    assert(knotOffset < lastKnot);
    assert(t >= knot[knotOffset] && t <= knot[knotOffset + 1]);
    return knotOffset;
}

// doesn't handle flat out discontinuities in the curve...

struct Homogeneous3 {
Homogeneous3() { x = y = z = w = 0.; }
Homogeneous3(Float xx, Float yy, Float zz, Float ww) {
    x = xx; y = yy; z = zz; w = ww;
}
Float x, y, z, w;
};
```

⟨*NURBS Evaluation Functions*⟩+≡

```
static Homogeneous3
NURBSEvaluate(int order, const float *knot, const Homogeneous3 *cp, int np,
          int cpStride, float t, Vector *deriv = NULL) {
//    int nKnots = np + order;
    float alpha;

    int knotOffset = KnotOffset(knot, order, np, t);
    knot += knotOffset;

    int cpOffset = knotOffset - order + 1;
    assert(cpOffset >= 0 && cpOffset < np);

    Homogeneous3 *cpWork =
    (Homogeneous3 *)alloca(order * sizeof(Homogeneous3));
    for (int i = 0; i < order; ++i)
    cpWork[i] = cp[(cpOffset+i) * cpStride];

    for (int i = 0; i < order - 2; ++i)
    for (int j = 0; j < order - 1 - i; ++j) {
        alpha = (knot[1 + j] - t) /
        (knot[1 + j] - knot[j + 2 - order + i]);
        assert(alpha >= 0. && alpha <= 1.);

        cpWork[j].x = cpWork[j].x * alpha + cpWork[j+1].x * (1. - alpha);
        cpWork[j].y = cpWork[j].y * alpha + cpWork[j+1].y * (1. - alpha);
        cpWork[j].z = cpWork[j].z * alpha + cpWork[j+1].z * (1. - alpha);
        cpWork[j].w = cpWork[j].w * alpha + cpWork[j+1].w * (1. - alpha);
    }

    alpha = (knot[1] - t) / (knot[1] - knot[0]);
    assert(alpha >= 0. && alpha <= 1.);

    Homogeneous3 val(cpWork[0].x * alpha + cpWork[1].x * (1. - alpha),
             cpWork[0].y * alpha + cpWork[1].y * (1. - alpha),
             cpWork[0].z * alpha + cpWork[1].z * (1. - alpha),
             cpWork[0].w * alpha + cpWork[1].w * (1. - alpha));

    if (deriv) {
        float factor = (order - 1) / (knot[1] - knot[0]);
        Homogeneous3 delta((cpWork[1].x - cpWork[0].x) * factor,
               (cpWork[1].y - cpWork[0].y) * factor,
               (cpWork[1].z - cpWork[0].z) * factor,
               (cpWork[1].w - cpWork[0].w) * factor);

        deriv->x = delta.x / val.w - (val.x * delta.w / (val.w * val.w));
        deriv->y = delta.y / val.w - (val.y * delta.w / (val.w * val.w));
        deriv->z = delta.z / val.w - (val.z * delta.w / (val.w * val.w));
    }

    return val;
}
```

⟨*NURBS Evaluation Functions*⟩+≡

```
static Point
NURBSEvaluateSurface(int uOrder, const float *uKnot, int ucp, float u,
             int vOrder, const float *vKnot, int vcp, float v,
             const Homogeneous3 *cp, Vector *dPdu, Vector *dPdv) {
    Homogeneous3 *iso = (Homogeneous3 *)alloca(max(uOrder, vOrder) *
                          sizeof(Homogeneous3));

    int uOffset = KnotOffset(uKnot, uOrder, ucp, u);
    int uFirstCp = uOffset - uOrder + 1;
    assert(uFirstCp >= 0 && uFirstCp + uOrder - 1 < ucp);

    for (int i = 0; i < uOrder; ++i)
        iso[i] = NURBSEvaluate(vOrder, vKnot, &cp[uFirstCp + i], vcp,
                    ucp, v);

    int vOffset = KnotOffset(vKnot, vOrder, vcp, v);
    int vFirstCp = vOffset - vOrder + 1;
    assert(vFirstCp >= 0 && vFirstCp + vOrder - 1 < vcp);

    Homogeneous3 P = NURBSEvaluate(uOrder, uKnot, iso - uFirstCp, ucp,
                    1, u, dPdu);

    if (dPdv) {
        for (int i = 0; i < vOrder; ++i)
            iso[i] = NURBSEvaluate(uOrder, uKnot, &cp[(vFirstCp+i)*ucp], ucp,
                    1, u);
        (void)NURBSEvaluate(vOrder, vKnot, iso - vFirstCp, vcp, 1, v, dPdv);
    }
    return Point(P.x/P.w, P.y/P.w, P.z/P.w);;
}
```

F. Glossary

Index of Identifiers