A Framework for Monte Carlo Image Synthesis

by
Peter Demoreuille and Taylor Shaw

A Thesis submitted in partial fulfillment of the requirements for Honors
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2, 2002

© Copyright 2002 by Peter Demoreuille and Taylor Shaw

This thesis by Peter Demoreuille and Taylor Shaw is accepted in its present form by the Department of Computer Science as satisfying the research requirement for the awardment of Honors.

Date ______________ ____
John Hughes, Reader

Date ______________ ____
Andries van Dam, Reader

Contents

1 Introduction **1**

1.1 Thesis Organization 2

1.2 Image Synthesis 3

1.2.1 Terms and Units 3

1.2.2 Radiometry and Photometry 4

1.2.3 The BRDF, BTDF and BSDF 6

1.2.4 The Rendering Equation 7

1.2.5 Common Simplifications 10

2 Previous Work **11**

2.1 Rendering Frameworks 11

2.2 Rendering Algorithms 13

2.2.1 Ray Tracing 13

2.2.2 Path Tracing 14

2.2.3 Bidirectional Path Tracing 14

2.2.4 Photon Mapping 14

2.2.5 Metropolis Light Transport 15

3 A Framework for Image Synthesis **16**

3.1 Motivating Principles 16

3.1.1 Flexibility 17

3.1.2 Modularity 17

3.1.3 Physical Accuracy 18

3.1.4 Usability 18

3.2 Our Design 18

3.2.1 The `Component` System . . . 20
3.2.2 `Job` . . . 21
3.2.3 `Renderer` . . . 21
3.2.4 `Scene` . . . 22
3.2.5 `SceneObject` . . . 22
3.2.6 `Shader` . . . 23
3.2.7 `EmissionShader` . . . 24
3.2.8 `ObjectMap` . . . 24
3.2.9 `Spectrum` . . . 25
3.2.10 `Camera` . . . 25
3.2.11 `ImageBuffer` . . . 26
3.2.12 `ToneMap` . . . 26
3.2.13 `Output` . . . 26
3.3 Problem Areas . . . 27
3.3.1 Separation of Specular and Diffuse Reflection . . . 27
3.3.2 Estimation of Lighting Components . . . 28
3.3.3 The $G(x, x')$ Term . . . 28
3.3.4 Implementation Details . . . 28

4 Results **30**
4.1 Rendering Algorithms . . . 30
4.2 Problems With The Current Design . . . 32
4.3 Future Directions . . . 33
4.4 Conclusion . . . 34

Bibliography **40**

Chapter 1

Introduction

The goal of this thesis is to create a flexible, robust and extensible framework to aid the implementation of a large class of realistic image synthesis algorithms. The motivation of this work was the realization that most image synthesis algorithms share a great deal of common requirements. We have attempted to identify these common requirements through an analysis of many existing algorithms as well as the equations they approximate, and have used the results to develop an abstract software framework which incorporates these common features.

To limit the scope of this work we have focused our efforts on Monte Carlo image synthesis algorithms that query the scene geometry via ray intersection. We briefly considered incorporating Finite Element algorithms such as radiosity into our framework. However, we determined that the restrictions which these methods place on surface models and geometry make them increasingly unsuitable for the types of scenes which a modern rendering system should be able to accurately render. In addition, the meshing, visibility queries, and other requirements of Finite Element approaches are sufficiently different from the requirements of Monte Carlo algorithms to make incorporating them into the same framework prohibitively difficult.

Our framework can be used to implement many different parts of the rendering process easily and efficiently. We have taken the object-oriented approach of separating all orthogonal aspects of image synthesis into abstract modules which interact through well-defined interfaces. Such a framework is beneficial because a researcher can isolate and modify a specific aspect of the rendering process without changing the

system as a whole, or even knowing the implementation details of other parts of the system. Such an organization helps reduce the amount of time that must be spent learning the software's structure before becoming comfortable with its use, and also helps isolate and identify problems when they occur. The ability to modify isolated parts of the system enables accurate comparison of algorithms without the concern that differing implementations of underlying code will invalidate results. In addition, such a framework provides the opportunity to study the impact of small optimizations and implementation changes on a variety of algorithms without having to modify the rendering algorithms themselves.

The generality of our framework allows for the implementation of a wide variety of state-of-the-art image synthesis algorithms. In some cases, however, the capabilities of our system may be more than is necessary for a given algorithm. We have found that in these cases it is usually easy to adapt the algorithm to use the more general interfaces and data provided by our framework without incurring a significant performance penalty. Although the presence of more general interfaces and data formats does sacrifice some efficiency, we feel that comparing different algorithms using the same framework is more useful than comparing highly optimized algorithms with entirely different implementations. In addition, the use of more precise and accurate data often has a positive impact on the quality of the images produced. Though we have made a significant effort to optimize our implementation wherever possible, our framework is primarily intended to be used in a research environment where short render times are not the highest priority.

1.1 Thesis Organization

The following sections present an overview of the image synthesis problem and define common units and terms which will be used throughout this paper. We also discuss some common assumptions made by image synthesis algorithms to simplify the rendering process.

Chapter 2 presents an overview of related rendering frameworks and several currently popular rendering algorithms. Particular attention is paid to the features a rendering architecture must provide in order to implement each algorithm.

Chapter 3 presents the design of our rendering framework in detail, and discusses the reasoning behind many of the design choices that we made.

Chapter 4 discusses the results of our work, and demonstrates some of the potential uses of our framework. We also indicate many of the problems we have discovered with our framework, and possibilities for future improvement.

1.2 Image Synthesis

The accurate and predictable simulation of light transport in a virtual environment has been the subject of a great deal of research over the last twenty years. With the advent of more sophisticated image synthesis algorithms and the availability of cheaper and faster computers, it is increasingly possible to create photorealistic imagery. So much so, in fact, that the presence of computer generated imagery in film and other media commonly goes unnoticed. But despite the increasing sophistication and realism of computer imagery, all image synthesis algorithms are essentially concerned with calculating an approximation of the way that light interacts with an environment. Thus all rendering algorithms can be reduced to a single equation which describes the physical reflection and transport of light. Each algorithm approximates this equation to varying degrees of accuracy, but fundamentally they all perform a similar calculation. A description of this equation and associated terms and units will form the core of our treatment of image synthesis, and has guided the design of our rendering framework.

We first define a number of common terms from the fields of Radiometry and Photometry which will be used in the remainder of this text. The *Rendering Equation* is then presented and explained in several forms. Finally we discuss several common assumptions made by rendering algorithms, including simplifications of surface properties and frequently ignored properties of light transport.

1.2.1 Terms and Units

A *Solid Angle*, ω, is the three-dimensional analog of a two-dimensional angle. It describes the "area" that a set of directions emanating from a point x occupy. The magnitude of solid angle, measured in *steradians (sr)*, is the ratio of the area of the

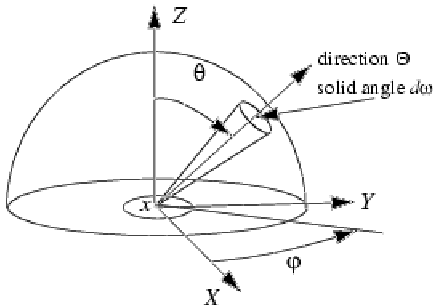

Figure 1.1: Differential solid angle $d\omega$ in direction Θ

sphere that a solid angle occupies divided by the radius of the sphere, $\omega = \frac{S}{r^2}$. From this we know that there are 4π steradians in a sphere.

Conceptually, the solid angle subtended by an object viewed from a point x describes how much of x's field of view is occupied by the object [5].

Often in rendering we are also concerned with the size of an object viewed from a surface, weighted by the angle at which the surface faces the object. *Projected solid angle*, $\omega^{\perp}$, is used for this purpose. The projected solid angle occupied by an object as viewed from a surface point x with normal n may be found by projecting the solid angle subtended by the object onto a plane with normal n. For small solid angles, the projected solid angle may be computed by $(n \cdot v)\omega$, where v is the direction of ω.

Differential solid angle, $d\omega$, is commonly used as an integration variable as it provides a very convenient relationship between the energy and intensity of light. It may be expressed by

$$d\omega = \sin\theta d\theta d\phi \tag{1.1}$$

where θ and ϕ are standard spherical coordinates. This relationship is also very convenient as it provides us with a method of rewriting an integral over differential solid angle (quite commonly computed over the hemisphere, Ω^+) as an integral in spherical coordinates.

$$\int_{\Omega^+} d\omega' = \int_0^{2\pi} \int_0^{\pi/2} \sin\theta d\theta d\phi \tag{1.2}$$

1.2.2 Radiometry and Photometry

Radiometry is the study of quantified light energy, and radiometric quantities describe measurable physical properties of light. The related field of Photometry is the study

of a human observer's non-linear response to light. Photometric quantities describe perceptual response to light energy.

The basic unit in radiometry is *radiant energy*, which is denoted Q and is measured in $joules$(J).

Radiant flux, or simply *flux*, is defined as the first derivative of energy with respect to time.

$$\Phi = \frac{dQ}{dt} \quad [\mathrm{W}] \tag{1.3}$$

Conceptually, flux is the rate of energy flow though a surface per unit of time.

Irradiance (E) is defined as flux per differential area arriving at a point x.

$$E(x) = \frac{d\Phi(x)}{dA(x)} \quad [\mathrm{W{\cdot}m^{-2}}] \tag{1.4}$$

The related terms *radiant exitance* or *radiosity* (which are equivalent) are commonly used to denote the flux per differential area leaving x. Note that these three terms are defined with respect to a point x which has normal n_x.

Radiance, the most widely used radiometric term in image synthesis, is defined as the flux per solid angle per projected area at a point x with surface normal n.

$$L(x, \omega) = \frac{d^2\Phi}{dA^{\perp}\ d\omega} = \frac{d^2\Phi}{(n \cdot \omega)\ dA\ d\omega} \quad [\mathrm{W \cdot m^{-2} \cdot sr^{-1}}] \tag{1.5}$$

This term implicitly measures the number of photons leaving a small area ($dA^{\perp}$) in all the directions contained within a small solid angle ($d\omega$) per unit time. Radiance has the property that its value does not change along a ray in 3-space (given the absence of participating media such as fog).

Radiometric terms may be converted to corresponding photometric terms by integrating with the standard observer response curve (also the curve for the CIE Y component), which describes the response of the human eye to light of different wavelengths. Many photometric analogs of the standard radiometric terms exist, but the most important of these is *luminance*, which is the analog of *radiance*. As most images are meant to be viewed by humans, luminance is a very useful measurement unit. Using luminance instead of radiance often results in images that have less human-visible noise. For more information about photometry and color science, consult [33].

1.2.3 The BRDF, BTDF and BSDF

A BRDF, or *bidirectional reflectance distribution function*, describes the relationship between incoming and reflected light at a surface. The BRDF, $f_r(x, \omega, \omega')$ is defined as

$$f_r(x, \omega, \omega') = \frac{dL_r(\omega')}{dE(\omega)} = \frac{dL_r(\omega')}{L_i(\omega)d\omega^{\perp}} = \frac{dL_r(\omega')}{L_i(\omega)d\omega|n \cdot \omega|} \tag{1.6}$$

Intuitively the BRDF represents the ratio between reflected radiance and incoming irradiance (note the difference in units, and how the definition "divides out" the fundamental $\cos(\theta)$ term from the BRDF). If the incoming irradiance field at a particular point is known, using the BRDF we may describe the radiance leaving the surface in any direction.

As the BRDF models the physical phenomena of light scattering at a surface, there are several requirements that it must satisfy in order to be physically accurate. The most important requirement is energy conservation, which is described by the condition

$$\int_{\Omega^+} f_r(x, \omega, \omega')d\omega'^{\perp} \leq 1 \quad \text{for all } \omega \tag{1.7}$$

This simply states that no more energy may be reflected than is incident on the surface. The second property that must be maintained is symmetry, described by the condition:

$$f_r(x, \omega, \omega') = f_r(x, \omega', \omega) \qquad \text{for all } \omega, \omega' \tag{1.8}$$

This states that the BRDF should reflect the same amount of energy for both possible directions of light flow. For this reason, the BRDF is commonly written $f_r(x, \omega \leftrightarrow \omega')$ to explicitly denote its symmetrical nature. Many of the rendering algorithms discussed in chapter 2 use this property and trace paths of light in both directions. Unfortunately some commonly used techniques in computer graphics, such as the use of shading normals, create BRDFs where symmetry is not maintained, and action must be taken by the algorithm to correct this. More information about non-symmetric BRDFs may be found in [27] and [28].

A concept similar to the BRDF, but which instead describes the transmission of light through a surface is the BTDF, or *bidirectional transmission distribution function*, and is denoted $f_t(x, \omega, \omega')$. Because it often is more convenient to only use one function to describe the reflectance and transmittance properties of a surface, the

notion of a BSDF has recently become more popular. The BSDF, or *bidirectional scattering distribution function*, is simply the union of the BRDF and BTDF, and is denoted $f_s(x, \omega, \omega')$. This unification of terms makes it possible to express transport and scattering equations much more compactly, and often makes implementation of rendering algorithms less complex.

The BSDF in turn is a simplification of a more complex function, the BSSRDF, which also includes subsurface scattering by separately denoting the entry and exit surface points in the reflection geometry [8]. In order to accurately render materials where subsurface scattering plays an important role (such as marble and human skin), full modeling of the BSSRDF is necessary.

1.2.4 The Rendering Equation

In his seminal paper, Kajiya presented the *Rendering Equation*, which describes the balance of the flow of light between surfaces [10]. Kajiya also showed that many popular rendering techniques, including radiosity and Whitted-style ray tracing were implicit approximations of this equation. This continues to be true, as most of the Monte Carlo rendering algorithms developed since, such as Photon Mapping and Bidirectional Path Tracing, base their formulation of light transport on the Rendering Equation. The original equation presented in Kajiya's paper is:

$$L_r(x, x') = g(x, x') \left[L_e(x, x') + \int_S f_r(x, x', x'')\ L_r(x', x'')\ dx'' \right] \tag{1.9}$$

The rendering equation states that the light reflected by a point x in direction $d = \overrightarrow{x' - x}$ is the sum of self-emitted light (the $L_e(x, x')$ term) and the light reflected by all other surfaces in the scene towards this point (the irradiance at this point), multiplied by a term which captures the reflective properties of the surface (the BRDF). Note that this equation is implicitly defined per wavelength, and is typically approximated by three samples corresponding to the red, green and blue response of the human eye. As this equation takes the form of an integral over the surface of all the objects in the scene, it is a little unwieldy and is certainly difficult to approximate. Because of this, the Rendering Equation is commonly expressed using a more convenient integration domain which removes all of the surfaces not visible from a certain point, and uses units which are directly applicable to image creation. This change in variables is

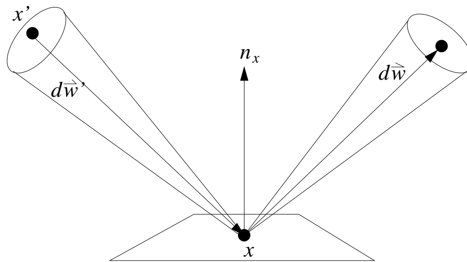

Figure 1.2: Geometry of the Rendering Equation

shown in equation 1.10, and the meaning of the terms in this equation is illustrated in figure 1.2.

$$L_r(x,\omega) = L_e(x,\omega) + \int_{\Omega^+} L_r(x',-\omega')\ f(x,\omega \rightarrow \omega')\ |n_x \cdot \omega'|\ d\omega' \qquad (1.10)$$

This recursive equation is the basis of many Monte Carlo rendering algorithms. Although there are other formulations of the rendering equation, most notably the path integral formulation used in Metropolis Light Transport, this is the form which is most often used.

As the Rendering Equation may appear somewhat impenetrable when first presented, an explanation of its derivation will be given, as well as common variants frequently used in rendering algorithms.

In order to compute the reflected radiance in direction ω from a point x, we must first find the irradiance at point x. Recall from section 1.2.3 that if the irradiance at a point x is known, we may use the BRDF of the surface to find the radiance reflected in any direction. To find the irradiance $E(x)$, we may simply integrate the incoming radiance at x over the entire hemisphere. Note the units in this equation agree, as you are integrating radiance over the hemisphere, removing the "per solid angle" and yielding irradiance.

$$E(x) \quad = \quad \int_{\Omega^+} L_r(x',-\omega)\ |n \cdot \omega|\ d\omega$$

Now that we have a method of estimating the irradiance at a given point, we must convert this into reflected radiance in a certain direction. To do this, we simply need

to factor in the BRDF in this computation, and add the emitted light.

$$L_r(x,\omega) \quad = \quad Ł_e(x,\omega) + \int_{\Omega^+} f_r(x,\omega,\omega')\ L_r(x',\omega')\ |n \cdot \omega'|\ d\omega'$$

This yields the Rendering Equation as presented in equation 1.10.

If all that is needed is an estimate of the amount of light reflected by a surface which comes directly from a luminaire (commonly referred to as the *direct lighting* component of the integral), we can simply replace the L_r term in the integral with an L_e as shown below.

$$L_{r,direct}(x,\omega) \quad = \quad Ł_e(x,\omega) + \int_{\Omega^+} f_r(x,\omega,\omega')\ L_e(x',\omega')\ |n \cdot \omega'|\ d\omega'$$

This (non-recursive!) equation finds the direct-illumination reflected by a surface in a given direction. Note, however, that the integral is performed over the entire hemisphere, most of which almost certainly does not contain luminaires. We may more efficiently evaluate this integral by removing the part of the domain which we know will evaluate to zero, and only integrate over the solid angle subtended by all the light sources at x, Ω_l. This is shown in equation 1.11.

$$L_{r,direct}(x,\omega) = Ł_e(x,\omega) + \int_{\Omega_l} f_r(x,\omega,\omega')\ L_e(x',\omega')\ |n \cdot \omega'|\ d\omega' \tag{1.11}$$

As the solid angle subtended by the light sources at a point may be difficult to find (consider the case of partially occluded light sources, etc), we may reformulate the integral to integrate over the surfaces of all the lights as shown in equation 1.12.

$$L_{r,direct}(x,\omega) = Ł_e(x,\omega) + \int_{S_l} f_r(x,\omega,x'-x)\ L_e(x',x-x')\ |n \cdot x'-x|\ G(x,x')\ dx' \tag{1.12}$$

The term $G(x,x')$ is often referred to as the "geometric" term, and is defined to be

$$G(x,x') = V(x,x')\ \frac{|n' \cdot (x-x')|}{||x-x'||^2} \tag{1.13}$$

Where $V(x,x')$ is zero if x and x' are visible, one otherwise. There exist very efficient methods of selecting samples while estimating $L_{r,direct}$ for various luminaire shapes. For more information on these Monte Carlo direct illumination techniques, consult [24], [31] and [30].

Because we can efficiently sample the direct lighting contribution to the rendering equation, we can improve the efficiency of sampling the entire rendering equation by separating the equation into direct and indirect terms.

$$L_r(x,\omega) = L_{r,direct}(x,\omega) + L_{r,indirect}(x,\omega) \tag{1.14}$$

This powerful optimization is known as *next event estimation*, and is presented in more detail in [12].

1.2.5 Common Simplifications

Geometric Optics

Although the full behavior of light may only be modeled with a theory that incorporates both the wave and particle natures of light, almost all rendering algorithms model only those effects which may be simulated using geometric optics. This means that effects caused by interference, coherence, diffraction and polarization will not be captured unless specifically added into the algorithm.

This is a simplification that we have made as well when designing our rendering system, and no particular consideration has been given to how these effects may be added to an algorithm implemented in our framework.

Surface Models

Rendering algorithms commonly approximate the BSSRDF by using a BSDF, and therefore ignore or merely approximate subsurface scattering effects. In addition, it is also quite common for rendering algorithms to make assumptions about the BRDFs used in a scene, and even to restrict them to be one of several hard-coded functions (commonly perfect diffuse, perfect specular and phong). As this is so common, some algorithms make the assumption that the reflectance properties of a surface may be characterized by two constants, p_s and p_d, which represent the specular and diffuse coefficients of reflection respectively.

No assumptions about a surface's BRDF have been made in our framework, although there are functions to query whether a surface has a specular or diffuse component, to find its average reflectance, and so forth.

Chapter 2

Previous Work

This chapter describes several of the existing rendering frameworks and algorithms which we examined while designing our system. We have attempted to identify and generalize the aspects common to many of these systems, and our design is a product of this analysis.

2.1 Rendering Frameworks

Most of the earlier rendering frameworks were designed for use with a specific image synthesis algorithm. For instance, the *Reyes* architecture [2], used in Pixar's "Photorealistic RenderMan" software, only supports a single type of scanline rendering. While the algorithm used is extremely flexible, *Reyes* does not currently provide support for global illumination, and is not object-oriented[1].

The *Ray Tracing Kernel* [11] introduced an object-oriented framework for describing scene geometry, but is limited to ray tracing and is not physically-based. Our work incorporates many of the design and organizational principles introduced in this system.

Many universities which are involved with global illumination research have developed general rendering frameworks to support multiple algorithms. Cornell's *Testbed for Image Synthesis* [26] is one of the largest existing suites of global illumination

[1] Release 11 of Pixar's Photorealistic Renderman (PRMan), due in Q4 2002, will include Global Illumination support.

algorithms. It divides the rendering process into a number of modules written in C which can be arranged in arbitrary ways. It is not object-oriented, and breaks up the rendering process by algorithm, rather than independent parts of rendering algorithms, as in our system. Thus changing a specific part of a rendering algorithm in Cornell's testbed necessitates rewriting large portions of the algorithm from scratch.

The *RenderPark* system developed at Katholieke Universiteit Leuven is another extremely flexible framework which supports many different types of global illumination algorithms [16]. It is written in a combination of C and C++, and is freely available for download.

Vision is a rendering system developed at the University of Erlangen-Nuremberg aimed at providing a general system for physically-based image synthesis [25]. Although Vision provides support for Monte Carlo rendering algorithms, much of the system focuses on finite element approaches, and it is evident that the requirements of finite element algorithms dominated the design process. However, Vision's architecture has many aspects desirable for Monte Carlo image synthesis, and the several papers describing its design have greatly informed our work.

EFFIGI, developed at Trinity College Dublin [14], is a rendering framework which uses object-oriented methods to abstract both geometric and mathematical concepts. It describes rendering algorithms as simply a series of mathematical operations. It is therefore extremely general and configurable. Its concept of function objects which can be plugged together in arbitrary configurations has influenced the design of our component system a great deal.

The *Ray Tracing Framework for Global Illumination* [23] is another object-oriented framework for Monte Carlo image synthesis. It is physically-based and capable of supporting many different algorithms.

Glassner's *Spectrum* architecture is an object-oriented framework based on signal processing [4]. Its rendering computations are driven by "active" geometric objects which are responsible for casting rays, evaluating illumination, and so forth. This design model restricts the flexibility of the architecture because implementing a new algorithm necessitates rewriting all of the participating objects.

Radiance [32] is a suite of programs primarily written for the analysis and visualization of architectural lighting. It is freely available for non-commercial use, source

packages and related data may be obtained at `http://radsite.lbl.gov/radiance`.

Radiance can produce excellent global illumination solutions for various lighting situations because it uses a very general algorithm based on a combination of Finite Element and Monte Carlo approaches. It uses physically correct units and places few constraints on scene descriptions. However, the Radiance framework is strongly tied its rendering algorithm, and it seems very difficult to extend it to support other global illumination algorithms. In addition, the Radiance source is in C, which gives the package enhanced portability but increases the difficulty of maintaining a generic software system that emphasizes replaceable modules.

2.2 Rendering Algorithms

2.2.1 Ray Tracing

The most simple global illumination algorithm is *ray tracing*, introduced by Whitted in [19]. A simple ray tracer shoots a single ray through each pixel on the image plane. When a surface is hit, a ray is shot in the direction of each light source to estimate the direct illumination incident on the surface at that point. In addition, if the surface is specular, reflection and refraction rays are spawned. This process continues recursively until a maximum recursion depth is reached or no objects are hit.

The only feature that a framework must provide to support a ray tracer implementation is the ability to query ray intersections with the environment. It places no restrictions on scene geometry or color representations. However, the classical ray tracing algorithm is only capable of rendering scenes with point lights and very simple BRDFs.

These issues are addressed by *distributed ray tracing* [3] which shoots more than one ray per pixel and can spawn additional rays from surface points to more accurately estimate global illumination. Distributed ray tracing is capable of rendering soft shadows from area lights, depth of field, glossy specular reflection and other effects not possible in a classical ray tracer. Like Whitted's ray tracing, the only feature that a framework must provide in order to support a distributed ray tracer is the ability to query ray intersections.

2.2.2 Path Tracing

Presented by Kajiya in the same paper which described the Rendering Equation, path tracing was the first algorithm to compute an unbiased approximation to the solution of the full rendering equation. Developed as an extension of distribution ray tracing, path tracing using a random walk to sample the space of all possible paths from the eye to light sources in the scene, using Russian roulette to determine when a particular path should be terminated. Several optimizations and variance reduction techniques are typically included in an implementation of path tracing, including next event estimation, importance sampling and stratification of reflected rays. Although it suffers from a large amount of noise and typically requires a large number of samples per pixel, its implementation is usually straightforward.

As the only visibility query that is required is ray-casting, no particular restrictions are placed on the scene geometry. Similarly there are no restrictions on the BRDFs used by the surfaces in the scene. The only requirements necessary to implement path tracing are the ability to perform ray intersection queries, query a BRDF for its reflectance, and evaluate a surface's BRDF. The ability to importance sample each BRDF and to separately select the light emitting surfaces in the scene is desirable in order to implement various optimizations.

2.2.3 Bidirectional Path Tracing

Bi-Directional Path Tracing, presented independently in both [12] and [28], is an extension of path tracing where light is traced both from the eye and lights. However, the basic requirements of the algorithm do not change, and it is straightforward to implement bidirectional path tracing in a framework which supports path tracing.

2.2.4 Photon Mapping

Photon Mapping, described in [7] and [6], is a very popular and efficient method of extending distribution ray tracing to efficiently calculate a variety of effects including global illumination, participating media, and subsurface scattering. Photon Mapping is a two-pass algorithm, where a first pass computes the *photon map* by tracing photons emitted from lights through the scene and recording photon information

whenever a photon interacts with a surface. The second pass renders the scene using the photon map to compute irradiance estimates using density estimation.

Although Photon Mapping is a biased algorithm, it is consistent and results improve when the number of photons used is increased. The algorithm does, however, make a number of assumptions about the surfaces in the scene. In order to compute an accurate irradiance estimate using density estimation, it is desirable for all the samples in the photon map to have equal energy. To accomplish this, photon mapping extensively uses Russian roulette, and relies on the fact that all surfaces have two constants p_s and p_d which describe their specular and diffuse reflection properties. In addition it assumes that each surface has an RGB triple describing its color. Although it is easily extensible to handle more samples, this does requires explicit manipulation of the color representation, requiring modifications to the algorithm if the color representation changes.

2.2.5 Metropolis Light Transport

Metropolis Light Transport, introduced in Eric Veach's 1997 PhD thesis [28] and [29], is based on the path integral formulation of light transport and the metropolis integration method. MLT is a physically correct, unbiased algorithm. It is particularly efficient at rendering scenes with very difficult lighting, such as strong indirect illumination, caustics caused by indirect illumination, and light traveling through narrow openings. In addition, the algorithm can be modified to efficiently handle different lighting effects and conditions by designing new mutation strategies.

The algorithm only requires ray intersection queries, and makes no assumptions about the BRDF models used besides the fact that they are physically correct, although modifications to the algorithm are provided to properly handle BRDFs with non-symmetric scattering. In order to compute color images, MLT requires that the luminance of a color sample be computed. For improved efficiency when implementing the algorithm, it is desirable to be able to efficiently importance sample BRDFs, know the size of the image being produced, and have access to the raw pixel data of the resulting image buffer.

Chapter 3

A Framework for Image Synthesis

This chapter describes the design of our rendering framework and discusses the motivation for many of the design decisions which were made. An overview is given of all the main parts of the system, and particular attention is paid to potentially confusing design choices. We conclude with a list of difficulties which we encountered either while designing suitably generic interfaces, or while implementing the interfaces we had designed.

3.1 Motivating Principles

The goal of this work is to create a flexible rendering architecture capable of producing physically correct still images using any Monte Carlo rendering technique. The system should not impose any undue limits on the types of rendering algorithms that can be implemented, and should be capable of rendering scenes of arbitrary geometric and material complexity. The framework should also be reasonably easy to learn, use, and extend to other purposes. A researcher should be able to configure and modify the system with a minimal amount of programming.

In designing this architecture we were not concerned with supporting animation or any time-dependent effects. We also decided to ignore non-geometric optical phenomena such as diffraction. In addition, we chose not to support any non-physically-correct lighting approximations such as bump-mapping and the interpolation of normals to simulate curved surfaces. While these techniques are very useful in many

rendering systems, we believe that the bias they add to global illumination simulations makes them undesirable in the context of our framework.

The following sections discuss several of the characteristics which we believe are essential for any modern rendering architecture.

3.1.1 Flexibility

A general rendering architecture should be capable of supporting any Monte Carlo image synthesis algorithm. The system should not be closely tied to any particular algorithm or optimized for specific uses. It must be general enough to support the capabilities of all existing and future Monte Carlo algorithms. In addition, the framework must not limit the set of possible implementations of any part of the system. It should be sufficiently flexible to allow different algorithms to be substituted for any part of the rendering process without modifying too much code.

Flexibility is important while developing new algorithms because different implementations of an algorithm can be compared quickly and efficiently. It also allows simpler, or faster, versions of any part of the rendering process to be substituted for more complex implementations if rendering time is a concern.

Increasing the flexibility and generality of a rendering framework often increases its complexity, and impedes its performance. We have frequently had to balance generality with performance concerns in our implementation.

3.1.2 Modularity

Our belief is that a flexible rendering framework can be achieved through a modular architecture that divides the rendering process into a set of independent components. All orthogonal aspects of the rendering process can be separated into independent modules which are concerned with performing a single task, and which communicate via well-defined interfaces.

The benefit of a modular architecture is that changes to specific parts of the rendering system are isolated to a single module and do not affect the system as a whole. Thus it is possible to easily plug in a new implementation of a module without modifying very much code.

Care must be taken to limit the interdependencies between modules, and to make sure that they only have local knowledge of the rendering process. While some parts of any global illumination algorithm must necessarily have global knowledge of the environment, global knowledge should be limited wherever possible.

3.1.3 Physical Accuracy

Modern rendering algorithms are capable of computing highly accurate approximations of physical light transport. It is therefore necessary for a rendering architecture to be capable of consistently supporting physically meaningful calculations in real-world units. All data values should be related to well-defined radiometric or photometric quantities.

3.1.4 Usability

While we are interested in creating an extremely general and flexible rendering architecture, it is important not to sacrifice ease of use for generality. The interfaces provided by a system should be easy to understand and implement. Throughout the design process we have stressed simplicity and elegance over complexity, and have attempted to only provide the minimal features necessary for an architecture of this type.

3.2 Our Design

Based on our analysis of the requirements of image synthesis algorithms, we have designed an object-oriented rendering framework which adheres to the general design principles discussed in the preceding section. We have divided the rendering process into a set of logically-distinct components which communicate via well-defined interfaces. Each component type is defined by an abstract class which implements the `Component` interface. Subclasses of each component type provide implementations of their parent's interface.

In choosing how to divide up the image synthesis computation, we have attempted

to balance the generality of systems based on low-level function objects, such as EFFIGI, with the efficiency of systems which perform their decomposition at a very high level, such as the Cornell Testbed. We settled on a design in which each component is responsible for a set of related operations. For example, our BSDF representation can calculate the value of a BSDF but is also responsible for knowing how to efficiently sample itself. In this respect, the rendering framework which our system most resembles is the Vision architecture. However, because Vision supports finite element methods, our framework is significantly less complicated.

We have tried to limit the interdependencies of components wherever possible. The majority of the components in our system only have knowledge about themselves and the data they need to perform their computations. Global knowledge is limited to the `Scene`, which stores all geometric objects, and the `Renderer`, which is responsible for actually computing global illumination.

All our components are passive in the rendering process. They simply provide information when it is requested. A render is only driven by the `Renderer` class, and is not directed by a set of active objects as in the Spectrum architecture.

To make our system easily configurable and extensible, we do not use scene description files which merely define the geometry in the scene. Instead we have developed a *render description language* which allows all aspects of a render process to be specified in a single file. A render description file specifies all the geometry and related surface properties of a scene, but also describes other aspects of the render such as the rendering algorithm, the type of image reconstruction filter to use, the raycasting acceleration structure to place geometry in, and so forth. Any component type can be instantiated in our file format. Thus it is possible to modify virtually any aspect of a render job without recompiling the program. It is as simple as changing a line in a file.

To support this ability to specify any aspect of a render job from a file, we have designed a system which allows every subclass of `Component` to specify arbitrary attributes which can be parsed and saved without the parser having any prior knowledge of them. This system allows the entire state of the a render process to be saved to a file and reloaded at a later time. Thus a render can be described using a graphical interface, saved, and then rendered in batch. The system also facilitates incorporating

implementations of new algorithms into the framework because all instance variables of any `Component` subclass are automatically parsable. If a class implements the `Component` interface, it can be integrated into our system without changing any code outside of the new class. The `Component` system is discussed in greater detail in the following section.

In the following sections we give a detailed description of the `Component` system and then discuss many of the most important abstract `Component` subclasses used in our framework.

3.2.1 The `Component` System

In order to facilitate various aspects of parsing, dynamic interface creation, memory management and several other tasks which become tedious and error-prone in a large software architecture, we created a modular component system which automates many of these tasks. All major parts of the rendering framework inherit from a top-level `Component` class, which provides the necessary methods to enable dynamic component parsing, reference counting and specific C++ language extensions.

Each subclass of `Component` uses several short macros to declare which of its instance variables are dynamically available at runtime. These instance variable can be of any standard data type, or instances of other components. Any subclass of `Component` may be queried for its `class_info` structure, which provides the name of the class, its parent, a list of the class's member variables and their types, and a method to create an instance of the class if it is not abstract. Using this information, any of the member variables may then be accessed or modified. There are also additional methods available for obtaining the information for a particular class name and finding all the subclasses of a particular class.

Having access to all this information greatly simplifies parsing render configuration files, as the parser has no need to be modified every time a new parsable class is implemented. It simply needs to obtain the `class_info` structures for the specified class and use the information contained to create an instance and verify that all the attributes specified are member variables of the correct type. In addition, the ability to dynamically query classes for their types and variables has enabled various GUI widgets to be written which dynamically create configuration panels tailored to a

particular subclass of `Component`.

The `Component` class also has two methods, `reference()` and `release()`, to support reference counting. In most cases classes only need to call the `release()` method instead of using the `delete` operator, and do not have to bother with incrementing reference counts. Due to the design of the parsing process, proper references are automatically set by the configuration file parser when all objects are created.

3.2.2 `Job`

The `Job` is the top-level `Component` which contains all of the data associated with a particular rendering process. It stores the current `Renderer`, `Camera`, `Scene`, `ImageBuffer`, `ToneMap` and `Output` classes and manages the flow of data between these components. It also takes care of several housekeeping chores that must be done before and after a render is started, such as initialization and finalization of all the `Components` and creation of all the rendering threads. User interfaces interact with the `Job` to monitor and control the progress of a render.

3.2.3 `Renderer`

The `Renderer` component is the superclass of all image synthesis algorithms. Renderers determine the method by which the rendering process approximates the rendering equation. Thus, each renderer is responsible for sampling the environment geometry by interacting with the `Scene`, calculating illumination, and finally contributing color values to the `ImageBuffer`. The `Renderer` interface consists of two methods which are called by the `Job`. The first, `render()`, begins the rendering process, and the second, `is_mt()`, returns true if the renderer is capable of multithreaded operation. If the renderer does support multi-threaded operation and the current machine has more than one processor, multiple rendering threads will be started by the `Job` and each one will subsequently call the `render()` method.

Example subclasses of `Renderer` are implementations of classical “Whitted” style ray tracing, distribution ray tracing, path tracing, and photon mapping.

3.2.4 Scene

The `Scene` component is responsible for storing all scene geometry and performing ray intersection with the environment. Rendering algorithms query the scene geometry by calling the `Scene`'s `intersect()` method. `intersect()` returns a structure containing the closest object along a given ray, and information about the intersection such as the intersection point and normal. `Scene` is also responsible for keeping track of all the luminaires and providing access to them.

Subclasses of `Scene` determine how scene geometry is stored and can implement various ray tracing acceleration data structures. For example, we have implemented a `Scene` which performs brute-force raycasting, and an Octree scene data structure based on [18] and [1].

3.2.5 SceneObject

The `SceneObject` component is an abstract superclass for all geometric objects which can be rendered. It encapsulates a linear transformation matrix, an object-space bounding-box and various surface properties of the object. These include the material's index of refraction, a `Shader` and an `EmissionShader` (see sections 3.2.6 and 3.2.7). Objects which emit light are not represented differently in our framework. An object is considered a luminaire simply if it has an non-`NULL` `EmissionShader`.

Each `SceneObject` provides an `intersect()` method which the `Scene` uses to query the intersection of the object with a ray. `intersect()` returns a structure which contains information about the intersection, such as the intersection point and the normal and texture coordinates at that point. For efficiency reasons, all `SceneObject`s delay calculation of as many of these quantities as possible until they are explicitly requested by calling the `get_surface_attr()` method.

Every `SceneObject` also provides methods for generating either random points over its entire surface or random points which are visible from another point in world space. The latter method is provided to enable more efficient direct lighting calculations, such as those presented in [24]. Each of these sampling methods returns the probability density of selecting the chosen point with respect to the luminaire's surface area.

We have currently implemented many `SceneObject`s, including a variety of implicit surfaces, polygons and triangle meshes.

3.2.6 `Shader`

The `Shader` component is the representation of both a BSDF and surface color. It has methods to evaluate the BSDF given incoming and outgoing rays, to importance sample the BSDF given an incoming ray, and to evaluate the probability of taking a particular sample. The `Shader` also allows for subsurface scattering, as a `Shader` implementation is responsible for setting the object-space location of the outgoing ray from the surface. Although this information is available, a particular rendering algorithm need not use this extra data and will simply loose all subsurface scattering effects. Because sampling and evaluating `Shader`s is an extremely common operation and must be performed as quickly as possible, a method may optionally be implemented which *both* samples and then evaluates the BSDF given an incoming ray. This method allows these two operations to be calculated more quickly than would be possible when sampling and evaluating the BSDF separately.

If the `Shader` has both specular and diffuse components it is responsible for importance sampling the scattering type as efficiently as possible. When a selection is made and a sample selected, it informs the rendering algorithm which choice was made by setting an argument variable. Both the evaluation and sampling probability methods take a corresponding boolean argument which specifies which scattering component of the BSDF should be evaluated. Using this variable, the rendering algorithm may select a particular scattering type if desired. It may also query the `Shader` to see if it has a non-zero coefficient of specular or diffuse scattering by using the `is_specular` and `is_diffuse` methods of the `Shader`. The constants themselves are not available as they may be computed dynamically depending on surface position, the incoming ray, or any other variable.

When a `Shader` is sampled or queried for the probability density of sampling a particular outgoing ray, it is responsible for providing this value with respect to solid angle. This value may be easily converted to area measure if needed by the rendering algorithm.

When the shader is evaluated it returns a `Spectrum` object (see section 3.2.9)

rather than a single value. As a surface's reflectance properties indirectly describe its color, this coupling is somewhat natural, and more importantly allows a `Shader` much more flexibility when computing scattering. The superclass of all `Shader`s has an `ObjectMap` (see 3.2.8) which represents the surface's color. The `Shader` is then responsible for using this if desired when evaluating the `Spectrum` for a particular scattering event. Note this coupling of surface color and `Shader` evaluation allows for shaders which may simulate effects such as diffraction and dispersion, and will function regardless of the rendering algorithm's support for these effects.

Several `Shader` models been implemented, including Lambertian, perfect specular, dielectric, and energy-conserving phong. Each of these uses the `Component` system to provide access to the various coefficients involved. Several others have been started, such as those described in [21], [22] and [13].

3.2.7 `EmissionShader`

The `EmissionShader` describes how rays are emitted from a luminaire. If a `SceneObject` has a non-`NULL` `EmissionShader`, this indicates that the `SceneObject` is a luminaire.

The `EmissionShader` has a similar interface to the `Shader`, but is not concerned with the direction of incoming rays (as they do not exist). It has methods to evaluate the emitted `Spectrum` given an outgoing ray, to sample the outgoing distribution, and to find the probability that a particular sample was taken. It also has a method to retrieve the power of the light, `power()`, which returns a value in Joules per area. Note that this choice of units will alter the total emitted energy of the light based on its surface area, but is necessary as the computation of the surface area of an arbitrary `SceneObject` is not currently possible. This issue is discussed further in section 4.2.

3.2.8 `ObjectMap`

An `ObjectMap` is an abstraction of a function defined over the surface of a `SceneObject`. It has two `sample()` methods, both of which take the values computed by object intersection (object space point and normal, world space point and normal, and texture coordinate) and return either a scalar or a vector. This abstraction has proved to be

extremely powerful and allows many complex effects to be created without specialized code. For example, a `Shader` may use the result of an `ObjectMap` evalution to determine the "roughness" of the surface at a point, and use this quantity to evaluate a BSDF model.

Several basic subclasses of `ObjectMap` have been written, including implementations which return a constant, the result of a texture lookup, the evaluation of various types of Perlin noise [17] [9], and the normal of the object. In addition to these, a multi-map has been implemented which combines two to three `ObjectMap`s using various texturing operations such as multiply, add and lerp.

3.2.9 `Spectrum`

The `Spectrum` class is the representation of color information in our system. Colors are represented as point-sampled spectra where the number and spacing of sample points over the visible range is fixed for each run of an algorithm. The XYZ or RGB color values of the spectrum can be queried, and color multiplication and addition operators are provided.

Color operations are some of the most frequently called methods in any rendering algorithm, so this is one area where we have chosen to emphasize efficiency over flexibility. Thus, we currently do not allow different color sampling methods to be combined by an algorithm. In addition, to avoid virtual method calls, `Spectrum` has no virtual methods, and we only provide a single implementation. So far this lack of generality has not presented any difficulty, but it can be easily changed if the need arises.

3.2.10 `Camera`

The `Camera` component is responsible for mapping pixel positions on the film plane to rays in world space. Each object which implements the `Camera` interface is responsible for performing this mapping in both directions (produce a world space ray from a pixel coordinate, and a pixel coordinate from a world space ray which presumably intersects the film plane), and to sample a "path" through the camera's lens system. This last method is quite similar to the ray from pixel method, but provides some algorithms

with a slightly better interface and presents an opportunity for more efficient sampling of the camera.

3.2.11 `ImageBuffer`

The `ImageBuffer` component is responsible for storing the output of a render. Various utilities are available for per pixel or scanline access, scaling the image data by arbitrary matrices, and gathering statistics about the pixel values. The buffer stores each pixel value as a three component vector, representing the X, Y, and Z components of the CIE XYZ color space. Values are added to the buffer using its `contribute()` method, which takes an (x, y) coordinate on the film plane, a `Spectrum` and a sample weight. `contribute()` adds this sample value to the buffer using an image reconstruction method determined by the subclass. For example, we have implemented `ImageBuffer`s which use a Gaussian filter, and the reconstruction filter described by Mitchell and Netravali [15], among others.

3.2.12 `ToneMap`

The `ToneMap` component is responsible for performing any post-processing on the values stored in the `ImageBuffer` after a render has completed and prior to writing the data to a file or displaying it on the screen. The `ToneMap` interface provides one method, `tonemap()`, which takes a source `ImageBuffer` and writes values to a destination `ImageBuffer`. Typically the `ToneMap` is responsible for converting the XYZ values stored in the render buffer into RGB values suitable for output, and normalizing them to lie between zero and one.

We have experimented with many types of tone mapping including straight normalization, square- and cube-root tone mapping operators and the tone map operator described by Schlick [20]. In addition, we provide a NULL tone map which does nothing to the values and which is used to save raw render data.

3.2.13 `Output`

The `Output` component writes the values stored in an `ImageBuffer` to a file of a format specified by specific subclass implementations. Its interface provides a single

method, `write_image()`, which takes the `ImageBuffer` to write out. Most `Output` implementations assume that all values in the buffer have been appropriately scaled to lie between zero and one by a tone mapping process. The one exception is `OutputRaw` which writes out raw image buffer data as doubles.

Currently we are able to save files in TGA and PNG formats, as well as a proprietary raw image format.

3.3 Problem Areas

When designing and implementing the framework as described above, many difficulties arose. Several of these are concerned with the mathematical aspects of the light transport equations, and others are issues encountered when implementing particular components of the framework.

3.3.1 Separation of Specular and Diffuse Reflection

As specular and diffuse scattering are physically different phenomena, each component of the reflection should be sampled separately. It is much more efficient to directly sample the specular component of reflection, so much so that direct illumination calculations should not be performed on a purely specular surface. In addition, when estimating direct lighting at a surface with a non-zero diffuse component, only the diffuse component of the BSDF should be evaluated when computing the estimate. This will prevent the direct illumination from being contributed twice and the surface from becoming too bright.

It is important to note that the intensity of a luminaire indirectly viewed in a specular surface is independent of the distance from the light. As the light moves further away, the size of the reflected luminaire will fall off according to its subtended solid angle (at a rate proportional to $\frac{1}{d^2}$), and this falloff in size is normally responsible for the perceived drop in intensity. This realization helped verify the correctness of resulting images.

3.3.2 Estimation of Lighting Components

Some confusion arose when creating the estimators for the direct lighting and indirect lighting at a point. In order to estimate an integral such as 1.12, a estimator similar to the following must be used:

$$L_r(x,\vec{\omega}) = S_l \frac{1}{n} \sum^{n} \frac{f_r(x,\vec{\omega},x'-x)\ L_e(x',x-x')\ |\vec{n}\cdot(x'-x)|G(x,x')}{p(x')} \tag{3.1}$$

This approximation calculates an estimate of the average emitted radiance from points on the luminaire to the surface point, and then multiplies this by the total surface area of the light. This yields an approximation of the irradiance at x due to direct emission from luminaires.

This also holds when estimating indirect illumination, where the domain is the entire hemisphere and is thus scaled by 2π steradians (as the indirect illumination is sampled by choosing sample directions rather than points on luminaires). Particular care must be taken when computing the indirect illumination estimate, as the implementation must be careful not to contribute direct lighting at a surface twice.

3.3.3 The $G(x,x')$ Term

Used to convert between area and solid angle measures, the $G(x,x')$ term (defined in 1.13) contains a division by distance squared. As this distance may be arbitrarily small, the $G(x,x')$ term is unbounded. The nature of this calculation may originally lead one to believe that the radiance values from points very close to luminaires are also unbounded, but this is not the case. If the $G(x,x')$ term is very small over the entire surface of the light, the light itself must be very small. As the direct illumination is estimated by a summation such as 3.1, it can be seen that this small area will cancel the very large $G(x,x')$ term.

3.3.4 Implementation Details

As is well-known, creating software which is immune to floating point precision issues is very difficult. We have tried to carefully implement code which must deal with floating point numbers of varying magnitude and attempt to alleviate some of these

problems. The majority of these problems, however, are normally encountered in the rendering algorithms themselves, and thus their implementors must be careful.

We have also found implementing some aspects of color representation which uses sampled spectra to be non trivial. In particular the computation of an RGB to XYZ conversion matrix for particular monitor phosphors has proven to be a difficult exercise, as well as creating suitable spectrum values for rendering from XYZ or RGB values.

Chapter 4

Results

Our image synthesis framework consists of about fifteen abstract classes which define the interfaces used by each part of the rendering system. In addition we have written over fifty other classes which either implement these interfaces or provide various utility functions such as user interfaces and model loaders.

The framework has been tested and used successfully by students in a graduate-level computer graphics class. The students had access to the framework's interface but could not read any of the implementation code. Still, most students had few difficulties understanding how to use the interface, and were able to implement sophisticated global illumination rendering algorithms in the space of only a few weeks.

In the remainder of this chapter we describe some of the image synthesis algorithms which we implemented to test our system. In particular we focus on our experience using the framework we have created, and suggest what types of algorithms are well-suited to being implemented using it. We then discuss many of the problems we have encountered and shortcomings of our framework. We conclude by outlining future directions for research and ways that we would like to improve the current system.

4.1 Rendering Algorithms

In order to test the completed framework, we implemented several rendering algorithms and observed how difficult it was to adapt their implementations to the requirements imposed by our system.

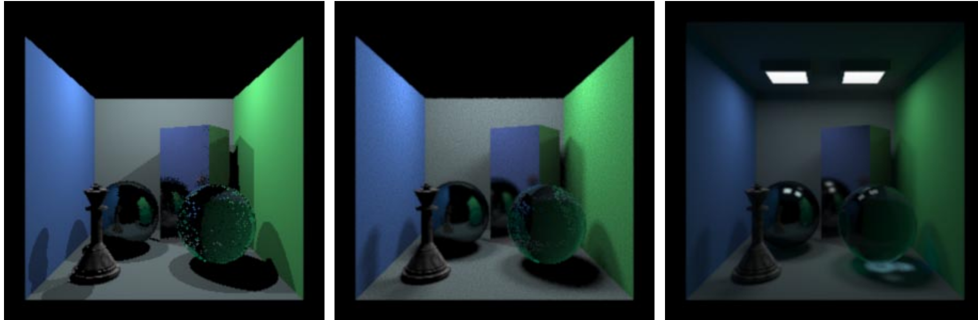

Figure 4.1: The same scene rendered with (from l-r) our classical Whitted ray tracer, our distributed ray tracer with 6 samples per pixel, and our path tracer with 1000 samples per pixel. Note the soft shadows in the distributed ray traced and path traced images, and the caustics in the path traced image.

Several algorithms presented little trouble. Whitted-style ray tracing was the first completed, and due to its simplicity was trivial to implement. However, as it did not even use the `Shader` interfaces, it was not a suitable test for much of the framework.

Next we completed a distribution ray tracer, which exercised nearly all of the framework, including the `Shader` and `EmissionShader`. As the distribution ray tracer evaluated direct lighting and reflections separately, measures had to be taken to avoid counting illumination twice. To solve this and also reduce other artifacts, evaluation and sampling of the specular and diffuse components of `Shader`s had to be properly tracked in order to sample each component of the BSDF properly and avoid introducing extensive amounts of noise.

We also developed a Path Tracer using the distribution ray tracer as a template, and many of the same issues applied. Balancing the illumination given by next event estimation and indirect illumination estimation proved tricky at times, as our framework lacks a method of finding the surface area of a luminaire (see 4.2).

More difficult to integrate was an implementation of Photon Mapping, as p_s and p_d values are not explicitly available in our framework. In addition, the color representation which we use is not RGB as is required by the original photon mapping algorithm, and we wished to keep the `Spectrum` class data private in order to enforce the separation of algorithms and support code. As a result, we were forced to estimate irradiance using the photon map without Russian roulette, which generated noticeably poorer results. We would like improve our implementation of this algorithm as

well as modify the framework to make its implementation slightly easier.

An implementation of Metropolis Light Transport is nearly completed, and the framework has proven itself to be very well designed for the implementation of this sophisticated rendering algorithm.

4.2 Problems With The Current Design

In order to address some issues encountered when estimating direct illumination and specifying luminaire emittance, we need to add methods to query the world-space surface area of an object. This would necessitate adding an additional abstract method to the `SceneObject` class which returns surface area, and requiring all subclasses that could possibly be a luminaire to implement this method.

The current implementation of master objects and instancing is quite poor. A redesign of how this system functions would be very advantageous and would certainly improve rendering time immensely. These changes would probably be most easily implemented if the `Scene` component also implemented the interface of `SceneObject` as recommended in [11]. This would allow a master object to simply be an instance of a `Scene` which may be then inserted multiple times into other `Scene`s. In addition, this would solve issues where certain objects force spatial subdivision for the entire scene to be higher than necessary, as these problematic objects may be placed into other scenes which have separate spatial subdivision parameters.

The presence of participating media and volumetric objects is noticeably absent from the design of our framework. We would like to add support for participating media and either find methods of adapting the `Shader` interface to support volume scattering operations or creating a new `VolumeShader` which will support these computations.

The treatment of single-sided objects in the framework is currently not very well-defined. The appearance of the reverse side of open surfaces such as planes, discs and tubes are all dependent on the particular rendering algorithm being used. We would like to make this consistent, by either modifying the intersection routines to return positive normals on both sides, removing open surfaces from the framework, or some other solution which does not depend on the implementation details of specific

rendering algorithms.

The framework currently does not support transparent objects placed within other transparent objects if proper refraction is required. As the `Shader` interface does not include the index of refraction of the current medium, shaders must make the assumption that a ray is either in a vacuum or inside the current object based on its dot product with the normal of the surface. This may not always be true if refractive objects are nested inside of each other, and the angle of refraction will be incorrect as a result.

4.3 Future Directions

The addition of CSG operations is something that we would like to implement, but it may necessitate a large number of additional intersection operations being added in order to avoid an adverse effect on rendering speed. It may prove to be more efficient to simply improve the speed of mesh intersection and use highly tessellated meshes rather than CSG objects.

We would like to implement more efficient sampling methods for more luminaires. Currently only spheres have an efficient sampling method for direct lighting computations, and we would like to add cylinders, triangles and various other shapes to this list. We would also like to add support to the triangle mesh class for uniform sampling over the surface area of triangle meshes, as this would enable meshes to be luminaires.

The scenes currently handled by the framework are somewhat simple, as common surfaces such as bezier patches and NURBs are not supported. We would like to add support for these objects, using exact solution methods to avoid approximation errors. We want to increase the efficiency of triangle mesh intersection to enable larger meshes to be rendered. The addition of displacement shaders would also enable much more complex scenes to be rendered, and would serve as a substitute for bump mapping, which we have avoided because it is not physically correct.

We would like to implement a shading language complier that is able to dynamically create new `Shader`s, `EmissionShader`s and `ObjectMap`s. A language as complex as that used in Pixar's Photorealistic Renderman is not necessary and would probably

incur large execution costs, but a simple language tailored to the implementation of procedural textures and shaders would add a great deal of flexibility to the framework.

Further integration with modeling packages would be very desirable, as the current method of scene creation by hand is extremely tedious. At the very least, additional file format parsers should be written to enable the framework to import scenes from other common modeling packages such as MAX, Maya and Lightwave.

We would also like to add support to the framework for rendering animation and time-dependent effects. This may require significant changes to the interfaces of many classes, as a time variable must be present in all of the intersection routines and related code. The design of the data structures which would contain the animation data is not clear to us at this time, and may require some additional changes to the framework. Significant changes to the acceleration structures would also be needed to incorporate moving objects.

4.4 Conclusion

We believe that we have created an extremely flexible rendering framework capable of producing physically-correct images of very high quality. Our experience using the system indicates that it is reasonably easy to use, and does not place undue restrictions on the types of Monte Carlo rendering algorithms which can be implemented. In addition, it has been successfully used by third parties to implement complicated image synthesis algorithms. While the system does have some shortcomings, we believe that the framework we have developed is a solid foundation for future global illumination research.

We conclude this paper with some images produced using our framework.

Figure 4.2: Three teapots illustrating different `Shader` types. From left to right are a perfectly specular transparent shader, a diffuse shader using a 3D Perlin noise texture to simulate marble, and a phong shader with an exponent of 2048. Rendered with a path tracer using 400 samples per pixel. Note the very soft shadows, indirect illumination and caustic cast by the leftmost teapot.

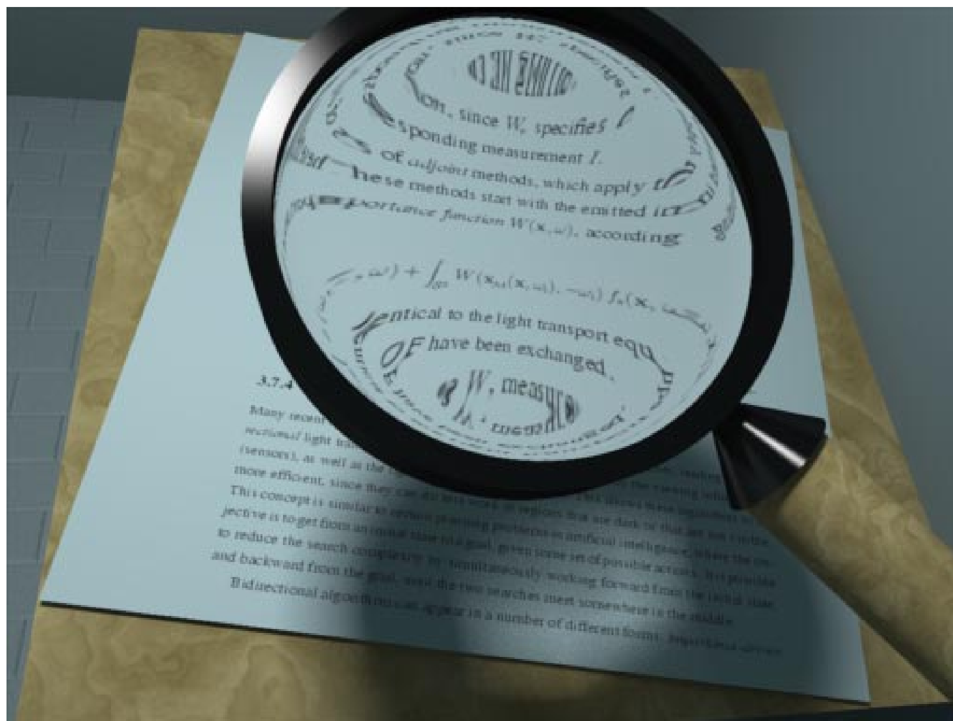

Figure 4.3: A magnifying glass created with a scaled sphere. The lens itself uses a dielectric shader, the shiny ring a phong shader with very low exponent, and the remaining objects are diffuse. Computed with a path tracer using 1000 samples per pixel.

Figure 4.4: A box scene with various objects, showing glossy reflections, caustics and strong indirect illumination. Note the brighter area of the ceiling above the glossy box and the several places where a light is indirectly viewed through two or more glossy bounces. Path Tracing with 1250 samples per pixel.

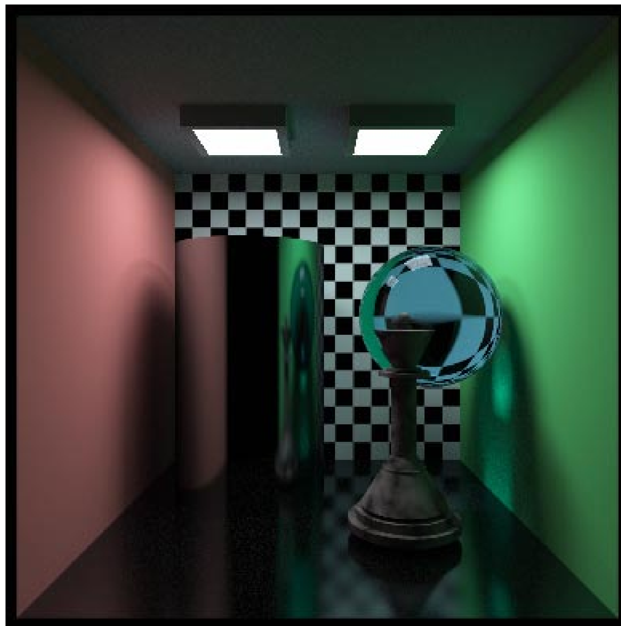

Figure 4.5: A similar scene showing a very glossy floor. Path Tracing with 1250 samples per pixel.

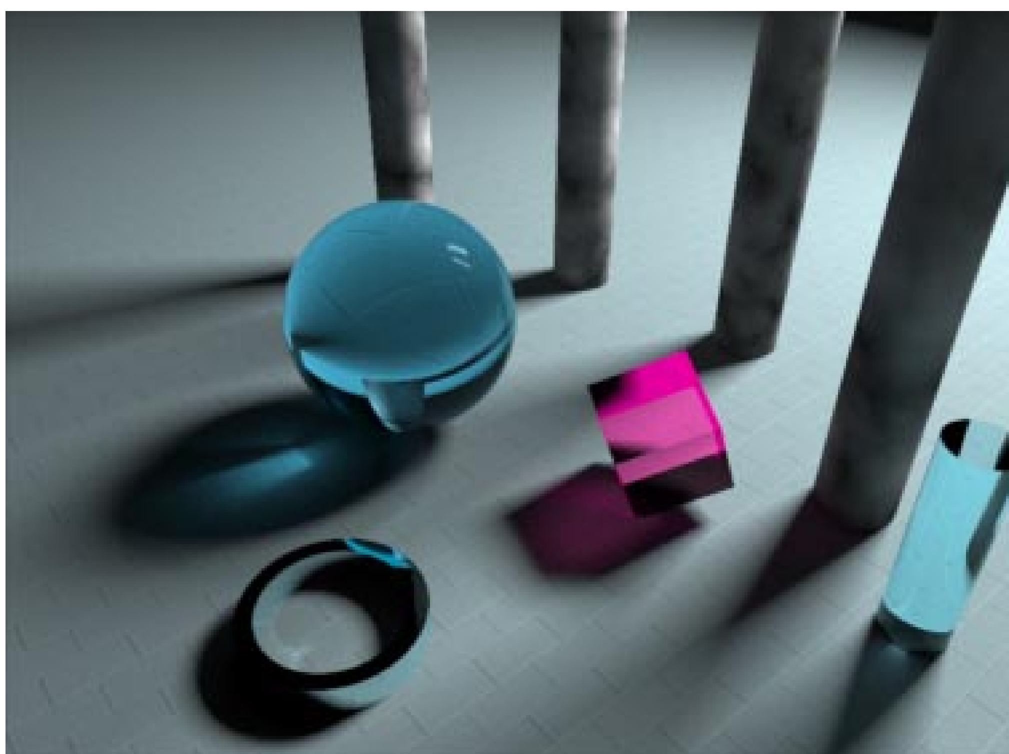

Figure 4.6: Scene illustrating caustics cast by various object types by a distant light.

Figure 4.7: A wall lamp containing a small planar luminaire. The lamp surface is very glossy, as well as the ball in the right corner. A large amount of the illumination in the scene is indirect. Path Tracing with 500 samples per pixel.

Figure 4.8: Note the mysterious character illuminated entirely via indirect illumination in the photograph on the wall.

Figure 4.9: Three cows with varying material properties.

Bibliography

[1] J. Arvo. Linear-time voxel walking for octrees. In *Ray Tracing News, v2*, March 1988.

[2] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, number 4, pages 95–102, Anaheim, California, July 1987.

[3] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 137–145, Minneapolis, Minnesota, July 1984.

[4] A. Glassner. Spectrum - a proposed image synthesis architecture. In *EUROGRAPHICS Tutorial Note 1*, pages 33–135. 1991.

[5] Andrew S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann, San Francisco, California, 1995. ISBN 1-55860-276-3.

[6] Henrick Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, Natick, Massachusetts, 2001. ISBN 1-56881-147-0.

[7] Henrik Wann Jensen. Global illumination using photon maps. In *Eurographics Rendering Workshop 1996*, pages 21–30, Porto, Portugal, June 1996. Eurographics / Springer Wien. ISBN 3-211-82883-4.

[8] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 511–518. ACM Press / ACM SIGGRAPH, August 2001. ISBN 1-58113-292-1.

[9] Eric Hoffert K. Perlin. Hypertexture. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, 1989.

[10] James T. Kajiya. The rendering equation. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, volume 20, pages 143–150, Dallas, Texas, August 1986.

[11] David Kirk and James Arvo. The ray tracing kernel. In *Proceedings of Ausgraph '88*, pages 75–82, 1988.

[12] Eric Lafortune. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering*. PhD thesis, Katholieke Universiteit Leuven, February 1996.

[13] Peter Shirley Helen Hu Brian Smits Eric P. Lafortune. A practitioners' assessment of light reflection models. In *Pacific Graphics '97*, Seoul, Korea, October 1997.

[14] W. Leeson, C. O'Sullivan, and S. Collins. EFFIGI: An efficient framework for implementing global illumination. In *Eighth International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media WSCG 2000)*, Plzen, Czech Republic, 2000.

[15] Don P. Mitchell and Arun N. Netravali. Reconstruction filters in computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, volume 22, pages 221–228, Atlanta, Georgia, August 1988.

[16] Computer Graphics Research Group of Katholieke Universiteit Leuven. http://www.cs.kuleuven.ac.be/cwis/research/graphics/RENDERPARK/.

[17] K. Perlin. An image synthesizer. In *Computer Graphics*, volume 19, 1988.

[18] J. Revelles, C. Urena, and M. Lastra. An efficient parametric algorithm for octree traversal. In *WSCG 2002 Conference Proceedings*, pages 212–219, February 2000.

[19] Steven M. Rubin and J. Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *Computer Graphics (Proceedings of SIGGRAPH 80)*, volume 14, pages 110–116, Seattle, Washington, July 1980.

[20] Christophe Schlick. Quantization techniques for the visualization of high dynamic range pictures. pages 7–20.

[21] Christophe Schlick. An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246, 1994.

[22] Christophe Schlick. A survey of shading and reflectance models. *Computer Graphics Forum*, 13(2):121–131, June 1994.

[23] Peter Shirley, Kelvin Sung, and William Brown. A ray tracing framework for global illumination systems. In *Graphics Interface '91*, pages 117–128. Canadian Information Processing Society, June 1991.

[24] Peter Shirley, Changyaw Wang, and Kurt Zimmerman. Monte carlo techniques for direct lighting calculations. *ACM Transactions on Graphics*, 15(1):1–36, January 1996. ISSN 0730-0301.

[25] P. Slusallek and Hans-Peter Siedel. Vision - an architecture for global illumination calculations. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):77–96, March 1995. ISSN 1077-2626.

[26] B. Trumbore, W. Lytle, and Donald P. Greenberg. A testbed for image synthesis. In *Eurographics '91*, pages 467–480. North-Holland, September 1991.

[27] Eric Veach. Non-symmetric scattering in light transport algorithms. In *Eurographics Rendering Workshop 1996*, pages 81–90, Porto, Portugal, June 1996. Eurographics / Springer Wien. ISBN 3-211-82883-4.

[28] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, December 1997.

[29] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 65–76, Los Angeles, California, August 1997. ACM SIGGRAPH / Addison Wesley. ISBN 0-89791-896-7.

[30] Changyaw Wang. Physically correct direct lighting for distribution ray tracing. In *Graphics Gems III*, pages 307–313, 562–568. Academic Press, Boston, 1992. ISBN 0-12-409673-5.

[31] Changyaw Wang. *The Direct Lighting Computation in Global Illumination Methods*. PhD thesis, Indiana University, February 1994.

[32] Gregory J. Ward. The radiance lighting simulation and rendering system. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 459–472, Orlando, Florida, July 1994. ACM SIGGRAPH / ACM Press. ISBN 0-89791-667-0.

[33] Gunter Wyszecki and W. S. Stiles. *Color Science: Concepts and Methods, Quantitative Data and Formula*. Wiley-Interscience, 1982. ISBN: 047102106.