University of Sheffield
CITY Liberal Studies
Department of Computer Science

FINAL YEAR PROJECT

Photorealistic Rendering System

This report is submitted in partial fulfillment of the requirement for the degree of Bachelor of Science with Honours in Computer Science by

Ioannis Tsiombikas

May, 2006

Approved

Supervisor: Dr. Panagiotis Bamidis

Second examiner: Ms. Anna Sotiriadou

Photorealistic Rendering System

by
Ioannis Tsiombikas

Supervisor
Dr. Panagiotis Bamidis

Abstract

This project is about the creation of a "*renderer*", that is, a program that produces visualizations of virtual 3D environments specified using a mathematical description of surfaces, in a Euclidean 3-dimensional space, along with material properties, and viewing parameters.

Furthermore, a "*photorealistic*" renderer, is a renderer that attempts to facilitate the creation of visualizations which are more or less indistinguishable from reality, by performing more accurate but also time-consuming simulations of the flow of light and its interaction with the objects of the virtual environment.

This project aims at providing advanced features, commonly encountered in big commercial renderers, such as monte carlo ray tracing, programmability through custom plug-in shader programs, which can drive the rendering process, and a flexible client server architecture. All these, while maintaining maximum cross-platform compatibility, and released under free software licensing terms, in order to provide the maximum benefit to the scientific community. In essence, what this project tries to achieve with this amount of flexibility and extensibility, is to provide a platform, which can be used for further research into graphics algorithms and photorealistic rendering.

DECLARATION

All sentences or passages quoted in this thesis from other people's work have been specifically acknowledged by clear cross referencing to author, work and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this thesis and the degree examination as a whole.

Name:

Singed: Date:

Contents

1 Introduction **1**
1.1 Aims of the Project . . . 1
1.2 Rationale . . . 2
1.2.1 Extensibility / Programmability . . . 2
1.2.2 Photorealism . . . 2
1.2.3 Freedom . . . 3
1.2.4 Support for Various Platforms . . . 3
1.3 Definitions . . . 3
1.3.1 Ray Tracing . . . 4
1.3.2 Radiance . . . 4
1.3.3 Diffuse Surface . . . 5
1.3.4 Specular Surface . . . 5
1.3.5 Direct Illumination . . . 5
1.3.6 Indirect Illumination . . . 5
1.3.7 Light Path Notation . . . 6
1.4 Organization of this Document . . . 6

2 Review of Previous Work **7**
2.1 Rendering Techniques . . . 7
2.1.1 Ray Casting . . . 7
2.1.2 Whitted Ray Tracing . . . 7
2.1.3 Radiosity . . . 8
2.1.4 The Rendering Equation . . . 9
2.1.5 Monte Carlo Methods in Ray Tracing . . . 9
2.1.6 Photon Mapping . . . 11
2.2 Reflectance Models . . . 11
2.2.1 Phong/Blinn Model . . . 12
2.2.2 Torrance and Cook . . . 12
2.2.3 Oren and Nayar . . . 12
2.2.4 Schlick's Model . . . 12
2.2.5 Image Based Lighting . . . 13
2.3 Other Renderers . . . 13
2.3.1 RenderMan . . . 13

2.3.2 Mental Ray 13
2.3.3 Final Render 14
2.3.4 POV-Ray 14
2.3.5 Yafray 14

3 Overview of the System 15
3.1 High-Level Design 15

4 Libxray – The Core Library 17
4.1 Design 17
4.1.1 Platform Independence 17
4.1.2 Ease of Use 18
4.1.3 Extensibility 19
4.1.4 Performance 19
4.1.5 Libxray Scene Description File Format 19
4.2 Libxray Implementation 22
4.2.1 Modular Decomposition 23
4.2.2 Programmable Shading 25
4.2.3 Output Data for Each Rendering 28

5 Rendering Daemon, Client, and Other Tools 30
5.1 The xrayd Daemon 30
5.1.1 The Communication Protocol 31
5.1.2 Security Considerations 32
5.2 The Client Program 33
5.3 Additional Tools 33
5.3.1 The Stand-Alone Renderer 33
5.3.2 The 3ds2xray File Format Converter 33
5.3.3 The Simple Image Viewer 34
5.3.4 OpenGL Real-Time Scene Previewer 34
5.3.5 The Fractal Generator 34

6 Testing and Results 35
6.1 Testing 35
6.1.1 Rendering Algorithm Testing 35
6.1.2 Cross-Platform Compatibility Testing 36
6.1.3 Performance Testing 36
6.2 Results 38
6.2.1 Sucesses 38
6.3 Between Success and Failure 41
6.4 Failures 42

7 Conclusion & Future Work **43**
7.1 Future Work 43
7.1.1 Global Illumination 43
7.1.2 Optimization 43
7.1.3 Distributed Rendering 44
7.1.4 Converters and Plug-Ins 44
7.1.5 Add Animation Capabilities to the Scene Format 44
7.2 Conclusion 44

A Student – Supervisor Meeting Logs **49**

B X-Ray Rendering System Installation Manual **52**
B.1 Requirements & Dependencies 52
B.2 Installation Instructions 53

C X-Ray Rendering System User Manual **54**

D The DTD of the XML Scene Description Format **55**

List of Figures

1.1 Image from Turner Whitted's original ray tracing paper in 1980 . . 4

2.1 Simple Whitted ray tracing. Image produced by the rendering system presented in this document 8
2.2 Radiosity handles diffuse global illumination 9
2.3 Distributed or monte carlo ray tracing used for glossy reflection and soft shadows, The picture is from the original paper by Cook, Porter, and Carpenter 10
2.4 Caustic formed by light rays being refracted through a glass of cogniac. Image rendered with Jensen's photon mapping. 11

3.1 high level rendering system design 16

6.1 GameBoy Advance Experiment 37
6.2 Sphere Fractal Rendering 37
6.3 Soft shadows, glossy reflections, and texture modulation of any material attribute, like the typical diffuse color, or the unusual reflectivity coefficient on the shiny sphere 39
6.4 Soft shadows, glossy reflections and refraction in a cornell box . . . 39
6.5 A scene converted from a 3ds file 40
6.6 A more complex scene converted from a 3ds file 40
6.7 Left: the cornell box with direct illumination only. Right: mixed success in the attempt at simulating global illumination 41

Chapter 1

Introduction

Visual perception is one of our more refined senses. “A picture is worth a thousand words”, goes the popular proverb, which is true due to the great efficiency of information retrieval in visual media.

Computer graphics is the field of computer science that deals with algorithms for the presentation of visual information through a computer. This ranges from mundane tasks such as drawing graphical user interfaces for human-computer interaction, to visualizing immersive 3-dimensional environments or presenting scientific data.

1.1 Aims of the Project

This project is about the creation of a “rendering system” or a “renderer”. Renderers are programs that given a concise mathematical description of a 3D environment, along with viewing parameters, provide a visualization of that environment in the form of a raster image. This, coupled with the capability to specify the change of various scene parameters over time, also provides the means of creating computer-generated animations.

The significance of the word “photorealistic” in the title of this project refers to the attempt to facilitate the creation of visualizations which are more or less indistinguishable from reality. Photorealism, however, is a function of many parameters, and cannot only be guaranteed by the quality of the simulation performed by the renderer. The expirience of the artist who creates the scene description through some 3D authoring program, the quality of the tools provided by said authoring program, as well as the time allocated to both the creation of the scene and the actual rendering, all play a major role in the level of realism in any computer generated image.

Photorealistic virtual environment visualization is not the only possible application of this system. Another important aspect is the visualization of scientific data. Such visualizations are commonly needed in many scientific fields, like

physics (e.g. visualization of thermodynamic phenomena), biology (visualization of protein folding), or medicine (visualization of CT or MRI scans). Such applications typically differ from most typical 3D environment renderings since they generally require the ability to render volumetric data. The current system, does not provide any built-in functionality to render volumetric data, however due to the extensibility of the design, one could easily add such a capability.

1.2 Rationale

One valid question is, "why another renderer?". The truth is that there are many renderers out there, so what is the reason for yet another one? What does this renderer provides that others don't? This question can be answered by reviewing the major objectives of the project.

Arbitrary subsets of these features are found in other renderers, their intersection however provides a unique oportunity to fill a niche. For instance, there are some free renderers, like POV-ray [1] [1], but they don't support programmable shading as expensive commercial renderers like mental ray [2], do.

1.2.1 Extensibility / Programmability

The renderer is extensible, so that it can be used as a platform for further research in the area of photorealistic image synthesis and scientific visualization.

In essence, the renderer core provides the necessary mechanisms and abstractions needed for the rendering process, which are "driven" by external user-supplied "shader" programs. Of course, the most common operations used for rendering photorealistic images are provided in the form of a library of such shaders, available for the user to use, study and modify.

1.2.2 Photorealism

The renderer should be able to produce, in the hands of a competent artist, images and animations approximating reality to a great extend. This is achieved by providing shaders that implement (with the help of the underlying system of course) some advanced rendering algorithms and illumination models, that are able to simulate as big a subset of all `L(S|D)*E` light paths in a virtual environment as possible. Furthermore, in order to achieve realism, the renderer should be able to use an appropriate reflectance model for each situation, capturing the subtleties of the materials in any given scene. This is why programmability is a major objec-

[1] To be precise, POV-ray is licensed under very restrctive redistribution terms. So although it comes with full source code, it's not really free according to both the Free Software Foundation definition and the Debian Free Software Guidelines

tive, to allow the user to implement any reflectance model is appropriate for the needs of each scene.

1.2.3 Freedom

The fact that this advanced rendering system will be released as free software is also one of the major aspects of this project; to provide such a powerfull and extensible, rendering platform for the community of graphics programmers and researchers to use and build upon, or even modify as they see fit.

To this effect, all parts of the system are going be released as free software [2] under the GNU General Public License (GPL).

This is highly appropriate for an academic project such as this, as the free flow of knowledge, unencumbered by commercialization and proprietarization, is the driving force behind scientific progress.

1.2.4 Support for Various Platforms

The system was designed from the beginning to be totally cross-platform. Parts which depend on operating system services, like process spawning and networking are seperated and interfaced to the rest of the rendering system through wrappers so that implementations can easily be written for each operating system. Thus, all operating systems and computer architectures will be supported through one code base, without introducing forks and ports which are hard to maintain and keep up to date.

All parts of the system should be able to compile and run on a multitude of computer architectures and operating systems, the most relevant of which are deemed to be x86, x86-64, MIPS, PowerPC and UltraSPARC based computers, with GNU/Linux, IRIX, MacOSX, Solaris, and various BSD systems. Due to the platform independence of the code, it should be able to compile and run on MS windows as well, with minimal effort.

1.3 Definitions

Before moving further to the main parts of this document, a number of terms must be defined, for the benefit of readers who are not familiar with the terminology used in the field of computer graphics.

[2] "Free Software" refers to freedom, not price; although the program will also be distributed free-of-charge through the internet.

1.3.1 Ray Tracing

Ray tracing is the basic rendering algorithm used by this renderer (and most photorealistic renderers), to generate a raster image from a mathematical description of an environment.

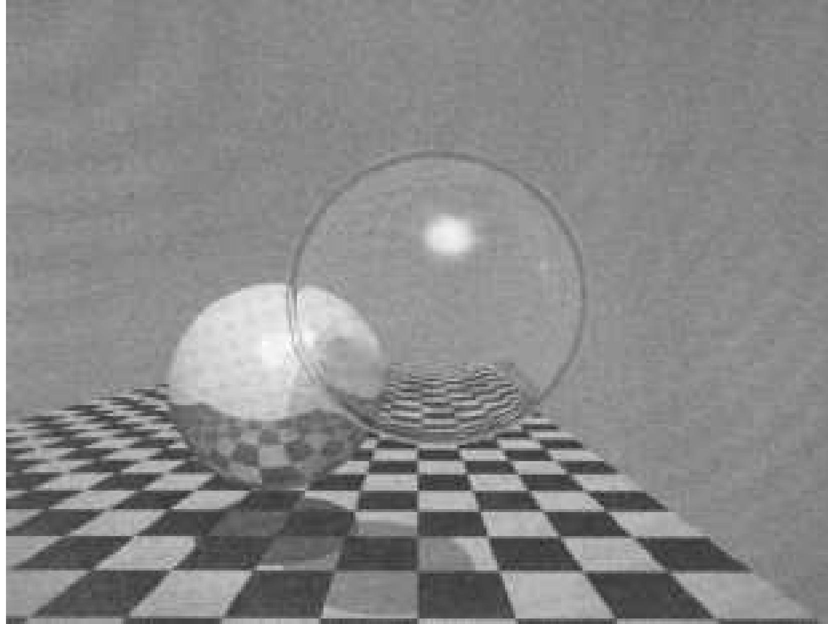

Figure 1.1: Image from Turner Whitted's original ray tracing paper in 1980

In its simplest form, as introduced by Turner Whitted [3], rays are "*shot*" backwards [3] from the camera (for each pixel of the view plane) and any intersections with the various surfaces in the scene are calculated. The closest intersection for each ray is kept (since further intersections are hidden from view by the closest one), shading calculations are performed for that point and secondary reflection/refraction rays are spawned as needed depending on material properties of the surface. The algorithm then proceeds to recursively trace these secondary rays througout the scene, until either a diffuse surface is encountered (no reflection/refraction), or a preset recursion limit is reached. The colors calculated by shading at each intersection throughout this traversal for each pixel are linearly combined and returned as this pixel's color in the raster image.

1.3.2 Radiance

Radiance is the energy arriving to a specific point by a specific direction. It's a 5-dimensional function of position and direction, typically denoted by $L(x, \omega)$, x

[3] Backwards in respect to the normal flow of photons in the scene.

being a 3-dimensional position vector, and ω a direction in spherical coordinates.

1.3.3 Diffuse Surface

A surface is characterized as diffuse, if it behaves as a Lambertian diffuser, spreading incoming light to all directions of the hemisphere above. In other words, the outgoing radiance of a perfect diffuser is constant for all directions, and only depends on the angle between the light and the normal of the surface at any specific point. Specifically, $L_d(x) = N_x \cdot L$, which due to the fact that we are dealing with unit-length vectors, is equivalent to the cosine of the angle between the normal and the light direction vector *(also known as Lambert's cosine law)*.

1.3.4 Specular Surface

A surface is characterized as specular, if it reflects light, or lets light be transmitted through it. A perfect mirror, or the interface between water and air, are good examples of surfaces exibiting specular behaviour.

1.3.5 Direct Illumination

Direct illumination, refers to illumination arriving directly from a light source to a surface. This, of course, requires a clear line of sight between the surface and the light source.

1.3.6 Indirect Illumination

Indirect illumination, refers to light arriving to a surface indirectly by bouncing off a nearby surface. It can be categorized in two important categories based on the nature of the interaction prior to arriving to the surface of interest.

- If light bounces off a diffuse surface before arriving at the surface of interest, the phenomenon is called diffuse indirect illumination. Such illumination is encountered in reality for example in a room with windows in mid-day, where the sun rays do not actually enter the room directly, but still arive indirectly to illuminate the walls in the room.

- If light is reflected off a reflective object, or transmitted through a translucent object before reaching its destination, the phenomenon is called specular indirect illumination. The most prominent visual effect of this phenomenon in nature, is encountered in "caustics" formed by concentrated light through a lens or curved mirror.

1.3.7 Light Path Notation

A notation introduced by Heckbert [4] to describe possible paths of light simulated by various rendering algorithms. It consists of a string of characters from the set $\{L, S, D, E\}$, which stands for *light*, *specular interaction*, *diffuse interaction*, and *eye* respectively. By using combinations of these letters in a sequence we can describe various interaction of light in an environment. For example a diffuse sphere seen through a mirror, corresponds to a `LDSE` path, while a caustic formed on a table by light concentrated by a lens, corresponds to an `LSDE` path.

Heckbert also used a regular expression notation to describe classes of possible light paths, for example, a simple ray tracer can simulate zero or one diffuse reflection, followed by zero or more specular interactions before arriving at the eye; such a class of paths can be denoted by the expression `LD?S*E`, while all possible light paths in a scene are really described by `L(D|S)*E`.

1.4 Organization of this Document

After this lenghty introduction, the following chapter reviews the most significant previous academic and commerical work in the area, as seen from the viewpoint of this project. So, for example, real-time rendering algorithms, or texture synthesis methods are left out as they are out of the scope of this project.

The rendering system is discussed in detail chapters 3 to 5, begining with an overview of the system and continuing in an in depth discussion of each important part.

Chapter 6 discusses the results obtained along with testing methods used for this project. Which of the objectives were satisfied and what elements were dropped or changed shape in the course of the project. A retrospective view of the design and implementation decisions is also provided, based on insight gained from the process.

Finally, chapter 7 concludes this document, and discusses possible directions of future work.

Chapter 2

Review of Previous Work

Research in the field of computer graphics is vast, and I can't hope to do much more than pinpoint the major contributors to ideas behind this project, seperating them into logical groups from the viewpoint of this rendering system. Also other similar rendering systems are presented in the appropriate section below.

However, before going on with this review, let me note that apart from the various papers mentioned here, a lot of condensed knowledge of the field is available in the many books that can be found in the references. Their contribution to the ideas and algorithms behind this project is invaluable.

2.1 Rendering Techniques

In this subsection, the major contribution in respect to rendering algorithms will be presented. Some of them are not strictly relevant to the project itself, in the sense that they will not be all necessarily implemented. However, they are essential in order to understand and evaluate the various rendering algorithms implemented by this renderer.

2.1.1 Ray Casting

In 1968, the first paper on rendering solid objects with a computer was presented by Arthur Appel. In his paper, "Some Techniques for Shading Machine Renderings of Solids," [5] he presented experiments with tracing rays through a virtual environment and plotting the result with a black & white plotter. This method, often called *ray casting*, provided the basis for further development of the recursive ray tracer.

2.1.2 Whitted Ray Tracing

One of the biggest early breakthroughs in computer graphics, was Turner Whitted's model of a recursive backward ray-tracer [3]. He presented a method to simulate

the interaction of light in a 3D environment by tracing rays backwards from the viewer to the light. This model is still the basis of most modern photorealistic rendering systems.

Whitted's ray-tracer could simulate any number of specular interactions between the light and the observer, thus providing a simple elegant method to solve the specular global illumination problem. As mentioned previously, in light path notation, Whitted ray tracers can simulate `LD?S*E` paths.

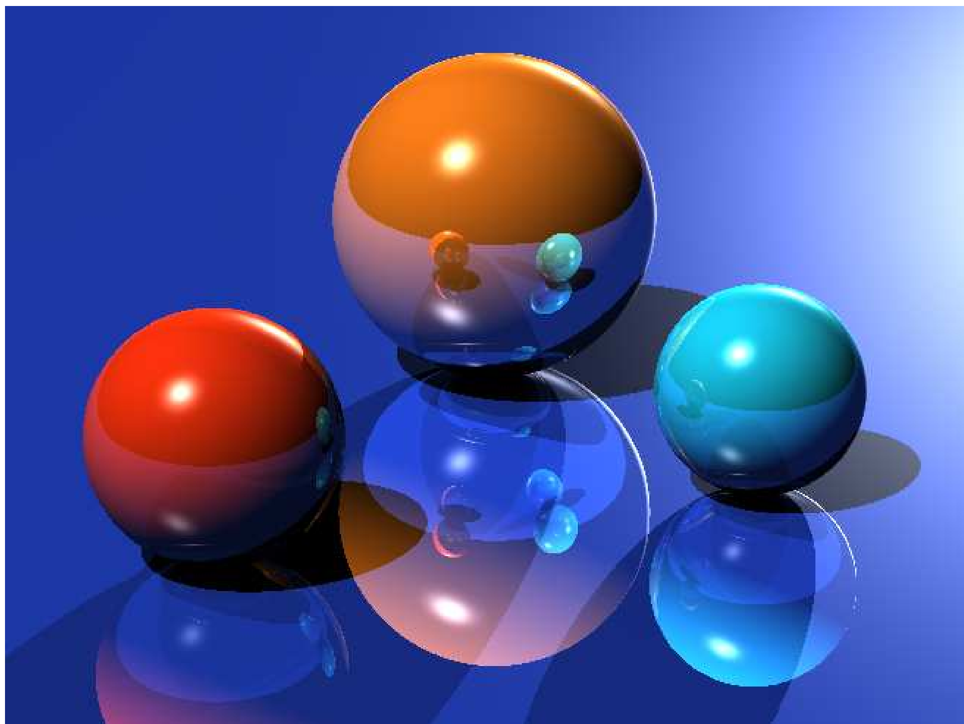

Figure 2.1: Simple Whitted ray tracing. Image produced by the rendering system presented in this document

2.1.3 Radiosity

Later, a number of researchers started using principles of thermodynamics [6], to implement a diffuse global illumination solution, called radiosity, with spectacular results. However in its standard form this algorithm could not handle any specular interactions (only `LD*E` paths), and is often mixed with whitted ray tracing to produce photorealistic images.

Figure 2.2: Radiosity handles diffuse global illumination

2.1.4 The Rendering Equation

In SIGGRAPH 1986, James Kajiya published his seminal paper "The rendering equation" [7], which unifies all aspects of photorealistic rendering in a single light transport equation:

$$I(x, x') = g(x, x') \ \left[e(x, x') + \int_S \rho(x, x', x'') \ I(x', x'') \ dx'' \right]$$

He also provided an algorithm, called path tracing to evaluate this equation using monte-carlo integration. Although rather slow and inefficient this method provides an excellent reference for comparing results of other rendering methods. Kajiya's technique captures all possible light transport paths, providing a complete `L(S|D)*E` solution for the first time.

2.1.5 Monte Carlo Methods in Ray Tracing

Robert Cook, Thomas Porter and Loren Carpenter presented a number of related enhancements [8] to the standard whitted ray-tracer, based on monte carlo methods, to be able to calculate a number of photorealistic phenomena. Namely these phenomena are:

- Reflection and refraction from imperfect specular surfaces, i.e. glossiness. To achieve this, instead of sampling (tracing a ray) towards the mirror direction after hitting a specular surface, a number of random samples are taken over a solid angle of directions, thus numerically integrating incoming radiance over the solid angle.

- Depth of field. The effect of having a specific focal distance, while everything further becomes gradually defocused and blurry. This was achieved by taking the origin of each primary ray to be randomly distributed on an imaginary lens around the view point, instead of having each ray start from exactly the same point [1].
- Penumbra regions in shadows. The simple ray tracing model assumes that lights are zero-area points in space, and as a result, it produces hard-edged shadows, with no penumbra region. By using area lights, and a monte carlo integration technique to determine the projected area of the light source for each point on a surface, soft edged shadows with penumbra regions can be simulated properly.
- Motion blur, is also simulated by having rays distributed evenly in a temporal interval around each frame time. However this complicates ray-object intersections, especially if we use bounding volumes to fast-reject negatives from the object database. The complication stems from the fact that in any arbitrary time interval, a bounding sphere of a moving object is no longer a sphere, but a, possibly bent if the motion isn't linear, capsule.

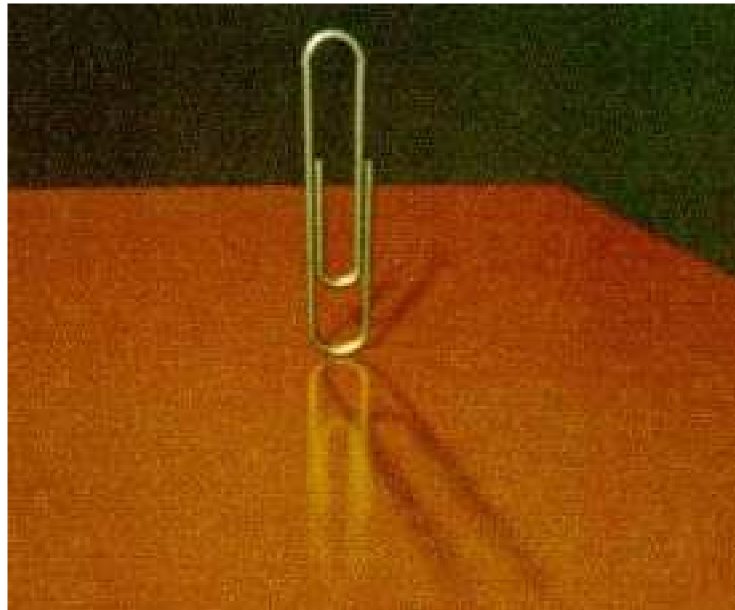

Figure 2.3: Distributed or monte carlo ray tracing used for glossy reflection and soft shadows, The picture is from the original paper by Cook, Porter, and Carpenter

[1]Also known as the pinhole camera model

These methods where named "distributed ray tracing" by the authors. However, I will not use this term due to ambiguity with distributed processing, i.e breaking up calculations to multiple hosts over a computer network which is also very relevant to rendering systems, and I will refer to it simply as monte carlo methods in ray-tracing.

2.1.6 Photon Mapping

Finally, a major contribution that affects this project is Henrik Wann Jensen's work on "photon mapping" [9], a preprocessing technique which can be used to either provide a direct estimate of illumination in a 3D environment, or guide the evaluation of light transport via monte-carlo integration, in essence to importance sample Kajiya's integral. In either case, the full illumination of a scene can be captured with varying levels of accuracy, or in path notation, `L(S|D)*E` paths.

Figure 2.4: Caustic formed by light rays being refracted through a glass of cogniac. Image rendered with Jensen's photon mapping.

2.2 Reflectance Models

Reflectance models, are models describing the behaviour of light when it interacts with a surface. Traditionally, reflectance models (or illumination models), are used to calculate radiance leaving the surface towards the view point, i.e. the intensity and color of light reflected off a surface at a specific point towards the viewer. However, recently there is a lot of effort into the development of reflectance models that can be easily importance sampled, to make monte carlo ray tracing more efficient.

The reflectance function, also called BRDF for Bi-directional Reflectance Distribution Function, provides a measure of reflected radiance towards a specific direction, based on the direction of incoming radiance (irradiance).

2.2.1 Phong/Blinn Model

In 1973 Bui Tuong Phong presented an empirical model for specular reflection [10] that closely matches the appearence of imperfect shiny surfaces. This model although not physically accurate, still enjoys a lot of popularity, expecially in real-time graphics applications. Actually, the most popular version of this reflectance model, is a slightly simplified form introduced by Jim Blinn [11]. The model owes its popularity to its simplicity, and mostly to the ability to be calulated very rapidly, which is essential for real-time graphics programs.

2.2.2 Torrance and Cook

Torrance and Cook [12] presented at the annual ACM SIGGRAPH conference of 1981, their research on a physically accurate reflectance model that can achieve a very high level of realism for many different materials, and most importantly metallic ones which the phong illumination model does not handle very well. The Torrance and Cook model is based on a statistical distribution of hypothetical microfacets on the surface, and take into consideration Fresnel's law to calculate the intensity and wavelength of the reflected light in respect to its angle with the surface normal.

2.2.3 Oren and Nayar

In the paper "Generalization of Lambert's Reflectance Model" by Oren and Nayar [13], another important reflectance model was presented, this time targeted towards diffuse (lambertian) reflectors.

2.2.4 Schlick's Model

A simple but physically plausible, empirical reflectance model was introduced in 1993 by Christophe Schilck [14]. Its advantages are the high degree of parametrization with an intuitive set of parameters, and the fact that it's cheaper, computationally, to calculate than full physically based reflectance models, while providing plausible results (as opposed to phong/blinn model which is physically incorrect). Also a big advantage is that it supports importance sampling.

2.2.5 Image Based Lighting

Finally, although not really a reflectance model Paul Debevec's [15] work on image based lighting should be mentioned here as an alternative way to illumination of a 3D surface. Debevec's approach uses high dynamic range textures to illuminate synthetic objects exactly as they would be lit in a real environment, using a "light probe" to capture the light arriving at a specific point in that environment in a panoramic image.

2.3 Other Renderers

As mentioned earlier, a number of rendering systems are available. In this section a brief overview of the most important such systems is provided.

2.3.1 RenderMan

The RenderMan system [16] was developed in the 80's by Pixar inc. This was the first renderer to feature a shading language which was used to drive the whole process of rendering, which is similar to one of the objectives of this project as well.

RenderMan provides its own C-like language for this, which is a defacto standard on its own. This language acting as both shading and scene description language, can be used to easily integrate renderman with any animation suite, since the renderer itself is a seperate process running as a server. This client-server architecture is another great design decision for RenderMan that is also used for this project.

RenderMan is a very powerful renderer, maybe even *the* most powerful renderer available. It is still developed and used to produce Pixar's excellent 3D animated films. RenderMan is also licensed (in proprietary terms) to various animation studios outside pixar, and a lot of the special effects in movies are rendered with RenderMan.

There are also 3rd party renderers, implementing the RenderMan language and programming interface, but are not as complete or full-featured as the original Pixar version.

2.3.2 Mental Ray

Mental Ray [2] is, along with renderman, one of the two most powerful commercial renderers available. It also supports programmable shading, a hallmark of the top rendering systems, but in a lesser degree than renderman. Mental Ray provides a C API for its programmable shading, which is faster and more efficient than the use of interpreted languages for this purpose.

Mental Ray was initially developed as a renderer for the Maya 3D animation suite by Alias—Wavefront for SGI machines running IRIX. Now it works on IRIX, GNU/Linux, MacOS X and Windows, and can be integrated with other animation programs through the use of properietary plugins. In the latest version, IRIX support was dropped due to the declining popularity of the platform in recent years, due to the availability of very powerful and cheap x86-based computers running free UNIX systems like GNU/Linux.

2.3.3 Final Render

Not so powerful as the previous two, this [17] commercial renderer is able to produce high quality images and animations. It begun as a plug-in renderer for 3D Studio MAX on MS Windows, now it supports Maya and Cinema4D as well, and is able to run on UNIX systems. This renderer does not provide any programmable shading unlike the previous two.

2.3.4 POV-Ray

One of the very few free software renderers. POV-Ray [1] provides a powerful and open, scene description language, and relatively good rendering capabilities. However it does not provide any programmable shading facilities such as those encountered in the top renderers in the field. Also as mentioned in the introduction, its classification as free software is disputed, due to its restrictive redistribution licensing terms.

2.3.5 Yafray

Finally a newcomer and trully free renderer, licensed under the LGPL. Yafray [18] supports a lot of the advanced rendering techniques that are some of the objectives of this project. However ther isn't any support for programmable shading.

Chapter 3

Overview of the System

This chapter will first provide a high-level overview of the rendering system as a whole, and then go on to discuss the major design decisions. Then the following chapters will provide an in-depth discussion of each of the elements of the system.

3.1 High-Level Design

From the initial conception of the project, it was aparent that a clean seperation between the subsystems of the renderer will maximize the usefullness, usability, maintenability, and platform independence of the system, while also having the advantage of being elegant and easier to develop. Also central to the design of the system, was that of programmability through shader programs; more on that in the corresponding section further on.

Figure 3.1 illustrates the high-level architecture of the renderer, and is shown here to aid the understanding in the discussion during the rest of this chapter. The next couple of paragraphs present an overview of the major components of the rendering system, which will be presented in detail in the following sections.

The core of the renderer is implemented in the form of an easy to use reusable library, called "*libxray*" which may be linked with any program requiring high quality rendering and visualization. This aspect alone makes the system extremely flexible. The library, which is discussed in detail further on, provides all the necessary algorithms needed to perform photorealistic rendering, and it is really a manner of calling a few high-level functions for any program to be able to visualize a high quality 3D environment.

The actual renderer, which uses libxray, is implemented in the form of a networked rendering daemon (server). This choice is significant as it allows extremely simple rendering front-end programs to be written, tailored to the needs of a specific user / class of users, or organization. But most importantly, because it allows the actual work (which is very processor-intensive) to be performed on powerful dedicated rendering servers, accessible by the users who may be working on

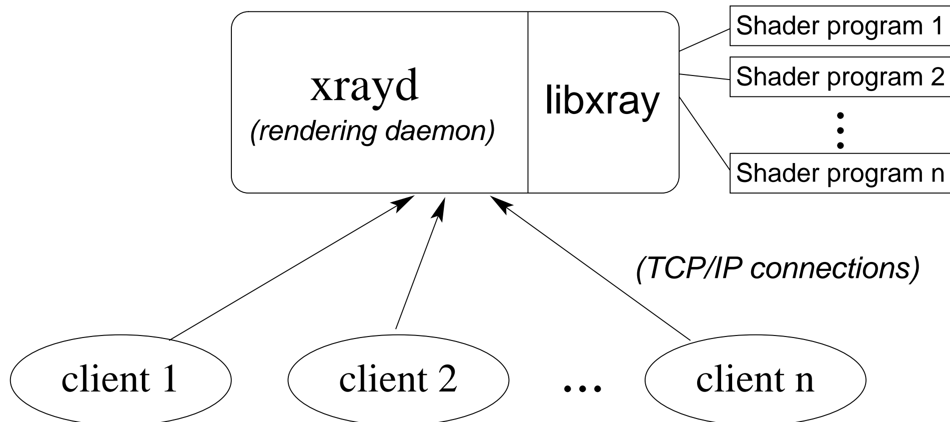

Figure 3.1: high level rendering system design

inexpensive workstations.

Finally, the libxray, provides a C++ API to write user-defined shader programs, that drive the whole rendering process. Thus giving the users, the power to perform custom modifications to the standard rendering algorithm, or even implement completely new rendering algorithms according to their needs. This kind of flexibility, which is an element encountered on very few cutting edge renderers, essentially transforms this system into a platform suited for research and experimentation on new rendering techniques. Furthermore, the way this programmability is implemented, combines the best elements of both interpreted shaders and compiled code.

In addition to all the elements described above, a number of simple but important tools were developed to support the use of this rendering system, ranging from file format converters, and other specialized tools useful for working with the system, to one-off programs to aid testing and debugging.

Chapter 4

Libxray – The Core Library

This chapter is dedicated to the most important piece of the rendering system, namely the libxray library. Libxray, handles the real work of the rendering system providing all the necessary functionality and data structures necessary to support the representation and rendering of 3D environments. It is designed to be highly efficient, easily reusable, and totally platform independent. This is the heart of the system, and contains the bulk of the code of the whole project.

4.1 Design

The major goals in designing libxray, was ease of use, platform independence, efficiency, and extensibility, each of those goals will be considered now in turn to shed light into the design of libxray, followed by the design of the scene definition file format.

4.1.1 Platform Independence

The bulk of the library is implemented in ISO standard C++, with some parts in ANSI/ISO C. The attention to standards compliancy is crucial (but not sufficient), to make the library extremely portable, and usable under any operating system, and computer architecture.

It is almost impossible for any non-trivial program to be completely self-reliant, and avoid all interactions with operating system facilities or other libraries. This is true for libxray as well, but a substantial amount of effort was put into minimizing, seperating and wrapping the platform-dependent parts, under platform-independent wrapper code, which is conditionally compiled after detecting the platform during compilation.

In fact, interaction with the operating system was only needed in spawning child processes or threads, and compiling shader programs (more on that later, in the appropriate shader details section). Also, dependencies to external libraries

was almost completely eliminated, leaving only a few critical and, above all, platform independent libraries for image loading (*libpng and jpeglib*) and XML parsing (*expat*).

An important design decision that allowed almost complete decoupling from operating system facilities, was the decision to not provide any form of "*presentation*" functionality in libxray. This means that if a program that uses libxray needs to perform rendering and then *show the result to the user*, it will have to assume the responsibility of dealing with the underlying window system (e.g. X11) in order to create a window, present the picture, and handle the necessary events.

Attention to the language standards and minimization of dependencies from OS facilities and external (possible non-portable) libraries, is the biggest step towards achieving the goal of portability. However "the devil is in the details" and an unweary or inexperienced programmer may be trapped in some of the small differences between various operating systems and computer architectures.

The most common pitfall for UNIX programmers, is not taking into account that the MS Windows system, treats text file streams in a special way, translating `\r\n` to `\n` on reading, and the inverse on writing to files. Thus attention must be paid to file input and output, to ensure the correct mode is used for each case. Also, a side-effect of this pecculiarity makes "*filter programs*", that need to write binary data to standard output, useless on MS Windows, since the windows shell performs file redirections through translated/text streams, thus mangling the binary output by inserting arbitrary `\r` characters. Such a bug was discovered in this program when testing a subset of it under windows.

Another common pitfall for the inexperienced programmer is different byte-orders between processor architectures. The problem manifests itself when performing binary input/output or bit manipulations on multi-byte data. In essence, some architectures store the data with the most significant byte first (at the lowest address) and the least significant byte last (higher address), while other architectures follow the opposite convention. The first is called big-endian byte order, and is used by MIPS, PowerPC, Motorolla, and SPARC processors, while the second is called little-endian, and is used by Intel microprocessors. This did not present a problem for this project since I was prepared and handled such discrepancies early on through compile-time architecture detection and conditional compilation.

4.1.2 Ease of Use

Ease of use is an important objective for libxray. That is, to be able to easily integrate high quality 3D rendering capabilities into any application by linking with libxray, with the minimum of effort from the client program, was a major consideration behind the design of this library.

To demonstrate that this was indeed achieved, it suffices to take a look at the simple testing driver program (`fray/fray.cc`) that was used througout the

development of the library. The whole source is less than 30 lines of C++ code, including empty lines, error handling and output of various messages to the console!

Essentially, the program only needs to instruct the library to read rendering options from a configuration file, call a function to read the scene definition, another function to perform the rendering, and two final calls to get the resulting pixels, and save them in an image file.

Even a more complicated program that would be able to utilize the full advanced data output capabilities of libxray to perform compositing, relighting, exposure control and other advanced processes, will only need a couple more calls to the library itself, and of course a lot of code to interface with the window system for presentation and user interaction, tasks which are out of the scope of the renderer.

4.1.3 Extensibility

Extensibility, which is a major objective of this project, is achieved by providing the ability to write custom user-defined shader programs, which drive the actual rendering process. Through these programs the user can experiment with new rendering algorithms, improve some aspects of the built-in shaders, and in essence customize the rendering process to his or her needs.

These rendering programs are defined on a material basis. This means that whenever an object with an attached shader is encountered during the rendering process, that shader is called to calculate lighting on the point of interest, and perform the next step, which may include further recursive tracing of new rays.

All the details on the implementation of the concept of programmable shading, in libxray, are discussed in the appropriate section, further on.

4.1.4 Performance

Unfortunately, this is an aspect of the original design of libxray, which did not make it into the system in its original form, due to time constraints. The more elaborate space subdivision optimization scheme that was originally planed, was not implemented, and a simple bounding volume technique was used in its place (see results and future work chapters for more details).

4.1.5 Libxray Scene Description File Format

Many possible alternatives were considered during the initial design phase of the project, for the scene description file format. A few of these alternatives and their advantages/disadvantages are presented in the following paragraphs.

4.1.5.1 Existing File Formats

The first point of consideration was wether it would be a good idea to use an existing file format for the scene description, instead of inventing a new one. Some of the existing file formats that were considered are the following:

The "3ds" file format, is compact (due to its binary format), relatively well known and easy to export scenes in this form, since it's directly supported by a major 3D authoring suite (3D Studio MAX). Also I had prior experience with it, as I'm using it as the main scene input file format for my real-time 3d engine. However it has some fatal flaws, which make it inappropriate for this project. It's old and clunky, there are many "features" which show its age and lineage. It cannot support meshes with more than 2^{16} polygons or vertices and it turncates texture image filenames to a maximum of 12 characters (the old DOS 8.3 format). Furthermore, it does not really support even the basic features needed by a photorealistic ray tracer, namely: flexible material attributes, index of refraction, and of course no have a shader filename as a material property. Also, the format only stores triangular meshes, which means that other kinds of surfaces like quadratic surfaces or voxel fields cannot be represented. Finally, due to the fact that it's a binary format, it's really hard to create or manipulate without special tools.

The POV-ray file format, is much nicer and full-featured. It is text-based, which is a disirable attribute, and supports everything that a ray tracer needs *except* shaders. It might be able though to add support for shader references as an extension to the file format. Also a lot of scenes are available in this format which is good as there is a lot of testing material. Still, it's not without disadvantages. For one, it's quite hard to parse, a full specialized parser must be written to be able to read the file format, and even then it has primitives totally out of the scope of a scene description format, providing rudimentary scripting for procedural scene generation directly in the scene file. Apart from the complications in supporting this file format, there is also the issue that the way it's structured is completely incompatible with the internals of this rendering system.

The RenderMan scene description, is not really a scene file format, rather than a programming language doubling as both the shading language and scene description language for the renderer, which makes it totally incompatible with the design of this rendering system.

4.1.5.2 The Design of a New File Format

Using an existing file format was deemed a bad idea, as it would not provide the features needded, or provide them in a totally different manner that would not fit the design of this system. Thus, it was desided to design a new file format,

providing all the the features needed, in a manner appropriate with the design of this rendering system.

The main objectives for this new design, apart from the similarity with the design of the renderer, were the following:

1. It should be easy to write a loader for this file format, and easy to modify in order to support new capabilities, as they are introduced into the rendering system.

2. It should be able to support all the required features implemented by the rendering system. For example it must be able to attach a shader program to a specific material of the scene, to be used whenever this material is enountered.

3. It should be text-based, to be able to create and manipulate scene descriptions with simple tools (text editors, scripting languages, etc).

4. It should be as compact as possible to minimize transmition times and storage overhead (this partly contradicts the second point, but the second point takes precedence).

5. Tools to convert from other existing formats to this file format should be easy to write.

So, in order to satisfy most of these objectives, an XML-based file format was chosen. It's easy to create and modify with simple tools, it's human-readable, and it's very easy to write a program that converts some other format to a form of XML. It's also very easy to parse with the help of any XML parsing library. Another added benefit is that a very concise formal definition of an XML file format can be written, and then used to check the correctness of any XML file according to that definition, by automated tools. This formal definition is called an XML DTD (Document Type Definition).

The DTD of the scene description file format for this project, is included in the appendices. Here, a short example scene definition will be included to get an idea of how is the file format structured.

Figure 4.1, presents such a minimal XML file format example. In the first line, the doctype of the file is declared, so that an XML checker may validate the file. Then "*root tag*" of the file is encountered, which is the "*scene*" tag. In the scene we have a material called "Material01" which references the "phong.cc" shader. This means that if a ray hits an object with this material, it will call the specified shader to calculate lighting at that point, and direct the next step in the rendering process, possibly spawning two new rays for the reflection and refraction directions, and calling trace again (indirect recursion). We also have an object named "ball" which has a spherical surface of radius 1, a light and a camera (which is used as the view point for the rendering).

Listing 4.1: Small XML file format example

```
<!DOCTYPE scene SYSTEM "xray.dtd">
<scene name="example scene">
  <mat-list>
    <material name="Material01" shader="phong.cc"/>
      <attr name="diffuse" val="1, 0.3, 0.1" tex="foo.png"/>
      <attr name="specular" val="1, 1, 1"/>
      <attr name="shininess" val="50"/>
      <attr name="reflect" val="0.5"/>
      <attr name="transparency" val="0.5"/>
    </material>
  </mat-list>

  <object name="ball">
    <matref ref="Material01"/>
    <xform pos="0, 0, 0"/>
    <sphere rad="1"/>
  </object>

  <light name="light01" type="point" pos="-10, 6, -10"/>
  <camera name="cam01" pos="0, 5, -8" target="0, 0, 0" fov="45"/>
</scene>
```

4.1.5.3 Animation Support

At this point, although the rendering system fully supports keyframed animation for every object, camera or light source, this is not yet included into the scene definition file format. This means that a program *can* use these animation facilities if it assigns keyframes explicitly, but there is no way to just load an animation and render a sequence from the rendering server (see further work chapter).

However animations *can* be created even now, in a rather more complicated manner, by generating somehow a series of scene files, one for each frame, with slightly different positioning each time. The scene files (and thust the frames rendered) should be 25 per second of animation (PAL) or 30 pes second (NTSC) to achieve smooth motion. The resulting rendered frames can be combined into video file using any available compression codec, with any video encoding program. Sample animations was created for the needs of this project by using the mencoder, command-line video encoder program.

4.2 Libxray Implementation

This section will provide an overview of the source-level organization of libxray, along with implementation details for the major parts.

4.2.1 Modular Decomposition

The source tree of libxray is inside the "`src`" subdirectory. The main part of the code is inside this directory, while there are some subdirectories with various "modules" or parts of the code. All of the code in the source tree is compiled and linked directly into "`libxray.a`" (a static library) and "`libxray.so`" (a shared library), from the top-level makefile.

4.2.1.1 The Main Part

In the main source directory of libxray, there is code to represent and handle materials, 3D objects and surfaces (defined as triangular meshes or mathematical equations), textures and materials, scene loading from the special XML file format, viewport configurations and supersampling, cameras and lights, and finally transformations in 3D space, animation using keyframes and controllers, and in general every little thing that needs to be done to represent 3D environments mathematically, and trace rays through them.

Also the code to load, compile, and manage shaders in the renderer is also in this main directory, with the only exception of the shader API headers and support code, that are inside the "`sdr`" subdirectory.

This main part of the library is all in C++, and makes use of object oriented primitives wherever it makes sense. For example there is a surface hierarchy with an abstract data type defining the notion of a surface, and subclasses for specific surfaces like meshes, geometrical spheres, infinite planes, etc, all of them implementing a ray test virtual member function.

However, I personally frown upon the missuse of OOP for things that are not inherently object oriented, which leads to gross overengineering and in the end fails to accomplish the first rule of design, that form follows function, or in other words, the solution must resemble the problem. Thus, for example, the texture manager is just a collection of functions to request a texture, store and retrieve textures in a shared texture pool, and load textures off the disk.

This design principle was followed throughout the whole project, keeping the implementation as simple as possible, and only using more elaborate constructions and class hierarchies, where it makes sense to do so.

4.2.1.2 Mathematical Library

The most important module after the main code, is without doubt the mathematical library, which resides in the subdirectory "`vmath`". It is also implemented in C++ and makes heavy use of operator overloading, in order to provide a natural way to handle all the mathematical primitives in the regular infix notation with the regular arithmetic operators.

It provides functionality for handling 2D, 3D, or 4D vectors, 3x3 and 4x4 square matrices, quaternions, rays, orthonormal basis, transformations, spherical

coordinates, numerical integration, real-valued random numbers, and many related facilities.

Definitely the oldest piece of code in the project, it was developed and polished over a series of years of personal involvment in graphics programming, since most of the functionality is essential for doing both realtime and non-realtime, 2D or 3D graphics programs. A number of additions have been done for this project, mainly ray handling and spherical coordinates.

4.2.1.3 Image File Module

This part of the code, written in plain C, provides unified reading and writing functionality for PNG (Portable Netwok Graphics), JPEG, PPM (Portable PixMap), and TGA (Targa) image file formats. The first two are implemented through the use of *libpng* and *jpeglib*, while the last two are implemented from scratch. The code provides file format detection through file identification blocks (not filename extensions), in order to be able to load any of the supported formats with a single `load_image` function call.

This code is called only by the texture manager (`texman.cc`), every other part of the rendering system uses the higher level functionality of the texture manager instead of handling images and image files directly. Also the texture manager loads each image only once, independently of how many different materials reference it.

4.2.1.4 Shader API

The directory "`sdr`" holds the headers and supporting code of the shader interface. It essentially provides wrappers to simplify the writing of shaders and accessing the various elements of the scene without digging into the internals of the various renderer data structures (although this can also be done if it is deemed necessary).

The initial idea was to provide a seperate C API for shader programs, and insulate them completely from the renderer internals. However, during implementation, this proved to be a bad idea, as it made shader writing actually *more* complicated, and incured a lot of implementation overhead by having to essentially duplicate a big part of the core functionality.

So in the end, this approach was abandoned in favour of providing full access to the core primitives of the renderer to the shader, thus essentially exposing a C++ API to the shader writer. This approach proved to be much more powerful, and coupled with the provision of various wrapper functions to simplify most common shader operations, actually provides an even easier interface for writing shaders than would ever be possible with a C API (mainly due to the availability of operator overloading for the mathematical primitives).

4.2.1.5 Common Code

A final module is contained in the subdirectory "`common`". There we have various source files providing some generic facilities used throughout the rendering system, such as hash tables, message logging, byteorder determination and endianess-independent input/output functions, as well as two different configuration file parsers and some code to determine optimal locations for placing supporting files by detecting the operating system and reading some environment variables.

4.2.2 Programmable Shading

One part of the libxray implementation needs special mention is the programmable shading capability. In the previous section the shader API was discussed, that is, what facilities are provided to the shader writer for interfacing with the core of the renderer, accessing the scene state, etc. Here we are mostly concerned on the other side of the coin, i.e. how does libxray handle shader programs.

As was already mentioned, libxray provides a C++ API to the shader. What is implied by that statement of course is that the implementation language of shaders should be C++ as well, to be able to use the API. This is the major reasons why initially the idea was to provide a C API, in order to be able to use it from different languages. That approach however has proved impractical as discussed previously.

The shader source files must contain an entry function "`shade`" defined with a specific signature, which is called by the libxray core whenever during ray tracing, an intersection is encountered with an object whose material contains a reference to that shader.

In order for this to work, the shader source file is compiled and linked in the form of a dynamic shared library, which is dynamically loaded at runtime by libxray.

Some implications of implementing shaders as C++ (or C for that matter) programs, are the typically the following (these do *not* apply in this case as will be explained immediately afterwards):

1. Efficient execution. This is true mainly because C++ is a compiled directly into machine code, but also because there are very good optimizing compilers for C++ due to the hardware-friendly design, and popularity of the language.

2. Hard to use for non-programmers. This is essentially true for every possible language that is full-featured enough to be useful to implement graphics algorithms. However C and C++ are not very "forgiving" to the novice.

3. Generally not suited for quick tests and experimentation due to the "write-compile-use" cycle. In this aspect interpreted languages, although not as efficient, excel by providing the flexibility to execute directly without the

overhead of compiling, linking, or even having to set up a development environment, finding the library files to give to the linker, and all the accompanying hassle.

4. The resulting binary is non-portable. Remembering that the system is based on a client-server cross-platform architecture, consider the scenario where a user compiles a shader on an Apple PowerMac computer running MacOS X, and the rendering server runs on a Silicon Graphics machine running IRIX, the shader binary cannot possibly be executed on the rendering server.

The shader implementation of libxray, however, keeps the positive aspects of compiled shaders while eliminating most of the dissadvantages. The problem is solved by delegating the compilation of shader programs, from the user to the renderer.

In essence, the user only provides the source code to the rendering system. Libxray then handles the whole process of calling a C++ compiler to compile it, create the dynamic library and load it at runtime on demand.

This means that the user doesn't need to set up a programming environment, this is the job of the administrator of the rendering server, which might or might not be the same computer, but the user/administrator roles are clearly seperate. It also means that the user doesn't need to explicitly compile the shader and determine linking flags, include paths, etc. every time he needs to test a shader, thus making the experience for the user as direct as writing in an interpreted shading language. Finally it solves the binary compatibility problem, since the server is both compiling and running the shader code. The only part which needs to be portable is the source code itself, which is trivial as it only needs to call libxray functions, and nothing more.

The only dissadvantage which is left is the perceived complexity of the C++ language by novices. However this is ameliorated by the simplicity of the API provided to the user, with the extra wrapper functions to perform common operations. Also the user in general does not have to deal explicitly with memory allocation and deallocation, or other gory details, which are handled by libxray itself. Only calls to perform calculations, manipulation of objects through their member functions, and simple loops and conditions are needed for most of the shader programs.

4.2.2.1 The Default Photorealistic Shader

It should be noted at this point, that a build-in shader program is provided by the implementation, and is called automatically for materials which do not contain a user-defined shader. It implements a variant of the phong illumination model, but also enhanced with various advanced monte carlo rendering procedures, which are used where required by material, lighting, or global rendering settings.

More precisely, this shader program implements glossy (or imperfect) reflections and refractions, as well as soft shadows produced by area lights (shadows with distinct umbra and penumbra regions). All these algorithms, which were part of the original photorealistic rendering objectives, use monte carlo integration, to achieve the desired effect.

Partial support for global illumination, was added at the last few days before the submission of this dissertation, but it's still buggy, very slow, and doesn't work correctly at this point. More on that in the results and future work chapters.

4.2.2.2 Compiling and Loading Shaders

It is very interesting to see how exactly is this automatic compilation and loading of shader programs implemented.

In order to be able to compile the shader, we need to know the path to the compiler executable. This is a configuration option provided by the administrator who installs the rendering daemon, and defaults to "`/usr/bin/gcc`". Another piece of information that we need is the location of the libxray headers, which needs to be passed to the compiler, and is also a configuration option set during the installation procedure by the administrator. The include path defaults to "`/usr/local/include/xray`".

Compiling and loading shader programs is one of these areas where the facilities of the operating system must be used. The code which performs this task is wrapped and conditionally compiled by compile-time operating system detection. The process is presented here for the UNIX case; for MS Windows, the code is very similar although more grotesque and convoluted as is everything on that system.

When the ray tracing function finds an intersection with an object with a material that refers to an external shader program, it calls the "`get_shader`" function to obtain a function pointer to the "`shade`" function of the shader, which is the entry point for shader programs.

If the requested shader does not exist in the database of the shader manager, it tries to find the requested shader source file on a pre-specified directory (or directories) in the filesystem. If the source file is found, the coresponding compiled dynamic library is located, and its modification date is checked against the modification date of the source file, by calling the "`stat`" system call. If the compiled version exists, and is up to date, compilation is skipped (the next paragraph) and the shader manager attempts to load it, and return the pointer to its entry point, as described below.

If the dynamic library does not exist, or if it is out of date, "`compile_shader`" is called automatically to compile and link it. It uses the "`popen`" UNIX system call to spawn a new process, run the compiler and get a "*read pipe*" connected to the compiler's standard output stream, in order to read the compiler messages. If the compilation fails for some reason, the output of the compiler is logged via the error message mechanism (which means that it can go to a log file or to the

user console, depending on the mode of operation). Otherwise the shader dynamic library was created, so we may proceed to the next stage.

If the shader dynamic library was created successfully, the "`dlopen`" system call is used to order the dynamic loader of the operating system to load the library and obtain a "handle" to it. If all goes well, the handle is passed to the "`dlsym`" system call to get a pointer to the "`shade`" entry function inside that shader. This function pointer is kept in the shader manager database, associated with the shader name, to be returned directly in subsequent queries, and is returned to the calling function (which is of course the ray tracing function).

4.2.3 Output Data for Each Rendering

Another important aspect of a high quality photorealistic renderer, is its output capabilities. At the very least, a renderer should be able to provide a raster image with the color of each pixel representing the sampled intensities on 3 different chanels (or wavelengths of visible light), namely the *red*, *green*, and *blue* components, calculated by means of tracing rays and calling shading functions to calculate illumination at each intersection.

This pretty much describes the standard output produced by any ray tracer. However, commercial renderers used in film production, often provide much more than that as their output (at the discretion of the user of course).

4.2.3.1 High Dynamic Range Image Colors

One very important feature that enhances the quality of the rendering by a great margin, is the ability to produce "*High Dynamic Range Images*". What this means is that, instead of just producing images in the regular 24bit (8bit per color channel) format, the renderer must be able to output images using floating point values for each color channel, thus providing a huge range of different intensities. This is exteremely important for applying exposure control functions to the rendered image, for color correction, glare reduction, and in general enhancing image clarity where there are both very dark and very bright areas in a single picture. These processes cannot be performed with the typical limited range of 256 possible intensity values in regular raster images.

4.2.3.2 Surface Attributes, Depth and Normals

Another important feature of advanced rendering systems used in film production environments, is the ability to output enough meta-data about the rendered image, to enable the artists to perform various operations such as compositing and relighting in the final image.

Compositing is the act of adding together multiple renderings from the same camera of different objects into a single image, in a way that produces a sensible

result. In order to be able to do this, the renderer must output a depth value for each pixel of the raster image, which records its distance from the camera in some arbitrary, but consistent between renderings, unit. Then for any given pixel (x, y) of every image, their distances from the camera will be compared, and the nearest one will be kept for the output, composited image. Of course this is based on the assumption that the closest object obscures the object which is further away, which is only true for non-transparent objects. This process can speed up the production of an animation sequence considerably and is used where appropriate by all major animation studios, so it is very important to be able to support it.

Relighting, on the other hand, is the ability to add, or tweak lights interactively, in the rendered image. This may seem impossible at first, however it's very easy to do for direct illumination, if we have for each pixel of the rendered image, the surface normal of the corresponding object at that point, along with some of the material parameters. This information is certainly available to the renderer for shading calculations, so it is trivial to be able to output it as part of the rendering. Again this can only work for special cases, but is widely used to perform minor tweaks to the illumination of a rendered image, without having to rerender it from scratch, which might be very time-consuming.

4.2.3.3 Motion Vectors

Finally, some rendering programs are able to output a motion vector for each pixel of the final image. In essence a 2-dimensional vector pointing to the direction of movement relative to the previous frame in an animation sequence. This is useful to the artist for adding "fake" motion blur to the output image, without having to calculate real motion blur through temporal distribution of rays and monte carlo integration.

This last one is the only one of the output data discussed in this section that is not supported by this rendering system. However this is not a big issue, as the renderer is able to distribute rays temporally and integrate the result, to produce real motion blur.

Chapter 5

Rendering Daemon, Client, and Other Tools

After describing libxray in detail, the rest of the system's components will be examined in turn. The essential parts are the rendering daemon (server), the various client programs, and additional tools that were developed to support the use of the system, but also support debugging and testing.

5.1 The xrayd Daemon

The xrayd rendering server (or daemon in UNIX terminology), is the program that performs the actual rendering. The advantages of seperating the renderer from the user progam are the following:

- The rendering server could run on a powerful, dedicated computer, to take advantage of its number crunching capabilities, while the users interact with the renderer to send rendering requests and receive results, by using inexpensive workstations or personal computers.
- Transparent distributed rendering can be achieved easily, by writing a program that poses as a rendering server to clients, but instead of performing the actual rendering, it breaks up the picture, or sequence of pictures, and distributes the work to multiple real servers in the network.
- This seperation, makes client programs extremely simple to write, and multiple different clients can be written in various forms to suit different users. For example apart from the simple command line client implemented in this project (see next section), it should be easy for someone to write a GUI program that provides a more user-friendly way to render pictures. Or maybe one could implement a client for xrayd as a plug-in for a 3D authoring program like Maya. Such a program would just need to generate the XML scene

description from the "current" scene state of the host program, and connect to xrayd to request a rendering. It is even possible, to write a web-based renderer where the user can upload a scene description through a form, and get the rendered image back.

In reality, xrayd is also a realtively simple program. It is linked with libxray, which performs all the actual work, it only has to deal with implementing the specified communication protocol, keep a state for each client, and call libxray functions wherever appropriate to perform the rendering.

Currently, the xrayd only works on UNIX systems (which includes the majority of target systems), it would be however, a simple matter to also produce a windows version as well with small modifications, mainly involving differences in process spawning and making the program a proper daemon ("service", on windows).

The communication between clients and server is performed through TCP/IP. The rendering daemon listens for connections on port 2048 (which is configurable of course), and spawns a seperate process to handle each incoming connection in order to be able to deal with multiple connections simultaneously.

5.1.1 The Communication Protocol

The protocol designed for the communication between client and server, is text based in order to make it as open and easy to use, even with primitive tools, as possible. For example, at an early point, before the client was written, all testing of xrayd was done interactively through a telnet client.

The original idea was for the protocol to be stateless (like HTTP/1.0), in order to simplify the server. This however would be rather wasteful, due to the fact that there is a considerable amount of data to be transmitted by the client for each rendering request, such as the scene definition, rendering options, and any external resources (textures for example). In a stateless protocol all this data would have to be retransmitted for every rendering request, even if multiple images need to be generated from a single scene, which is often the case with 3D animation.

So, a stateful protocol was defined instead, which implies that a client may send multiple requests before disconnecting. After connection is established. the client may issue one of the following commands:

1. *Options*, followed by any number of linefeed-seperated lines, containing key-value pairs for various rendering options. The option stream must be terminated by the *end-of-options* string, on a line by itself.

2. *Scene*, followed by the XML scene description, as defined previously, terminated by the </*scene*> tag.

3. *File*, followed by a filename, a count of bytes, and the contents of the file itself (which might be text or binary), transmits a specific file to the server,

which can be a texture image used in the 3D scene, or a shader source file. The server stores this file in a standard location for later retrieval during rendering. All the files provided by a connection are removed when the connection is closed.

4. *Render*, followed by a linefeed, tells the daemon to begin rendering in the background. The socket is *not* blocked while rendering is performed, instead two more commands are provided for status checking and syncronization.

5. *Retrieve*, followed by a linefeed, makes the daemon transmit first the size in bytes if the rendered image in text format, followed by a linefeed, followed by the binary data of the image itself. If the rendering is not completed, the equivalent of a *wait* command is executed prior to the retrieval.

6. *Retrieve-full*, is similar to the *retrieve* command, with the only difference that it requests a full floating point image, with additional normals, and depth information per pixel, suitable for post-processing and compositing.

7. The *status* command can be issued to check the status of the server. The server, subsequently proceeds to send back a status string containing either the string *BUSY*, if it is currently rendering something for this client, or *OK* otherwise, followed by a number indicating the server load (how many renderings are performed currently in parallel). This last piece of information, can be queried periodically by load-balancing distributed rendering managers, to assign jobs to idle servers in a network with multiple servers.

8. Finally *wait*, causes the server to stop sending data or waiting to further commands until the rendering is finished, and then send a response to the client, thus providing a mechanism for the client to block itself (by reading from the socket) waiting for the response without having to check the status continuously in a busy loop, which wastes both processor and network bandwidth resources.

5.1.2 Security Considerations

It should be obvious, that providing the ability to compile and execute any shader specified by the user can be hazardous, if we allow anyone to connect and render images to our rendering server. Security is *not* an objective, nor a requirement of this system. This rendering system is supposed to be used inside protected local area networks, accessible only to trusted people within an organization. And that is why, there isn't even any provision for authentication in the xrayd communication protocol.

5.2 The Client Program

The client is, by design, a minor part of this project. An extremely simple command-line client, able to send rendering requests to the xrayd server and retrieving the rendered pictures was written for the needs of this project. Something more elaborate, with extensive user interaction or user-friendly interface is completely out of the scope of this project, and left as possible future work.

The client program accepts as a command line argument the filename of the scene description file, and any of the following options:

`-o` followed by a filename for writing the rendered image file
`-c` followed by the name of a rendering settings configuration file
`-a` followed by the address of the rendering server (hostname or IP)

If the address is not specified, it defaults to localhost. Also, if the scene file is not specified, the scene description is read from the standard input. Finally, if the output filename is not specified, the result is written to standard output.

5.3 Additional Tools

A number of additional tools were created during the development of this project. Some of them would be useful to the user of the finished system, while others were written to help with the development, debugging, and testing of the system.

5.3.1 The Stand-Alone Renderer

A very simple stand-alone renderer was created during the development of this project, which does not use the rendering server, but rather links libxray directly and calls the appropriate functions to render a scene itself.

This program was more of a useful debugging shortcut, rather than an actual part of the complete system, since it's much easier to debug the renderer without having to shut down, recompile, and restart the rendering server all the time. The usage of this stand-alone renderer is similar to the simple client program described above.

5.3.2 The 3ds2xray File Format Converter

A very useful tool to provide some more interesting test scenes for the renderer, is the "*3ds2xray*" scene file format converter. This program uses the "*lib3ds*" library to load a scene from a "3ds" file, and writes the equivalent XML scene description, which can be loaded by the rendering system.

The 3ds file format, does not support all the features supported by libxray, which means that a scene generated in this manner will not exercise the full potential of the rendering system, without further editing. However it provides a con-

vinient way to create a complex scene description easily, by using any 3D authoring program that supports exporting in the 3ds file format (such as 3D Studio MAX).

It is desirable to eventually produce a number of such converters for many different standard scene file formats, in order to further enhance interoperability between this renderer and industry-standard 3D authoring tools.

5.3.3 The Simple Image Viewer

As mentioned earlier, libxray does not provide built-in presentation capabilities, to show an output image to user directly. For this reason, in order to remove the burden even from the client programs to implement such functionality, a seperate image viewing program is provided which can easily be called from the client program, or even from a shell script that first calls the client to render the picture, and subsequently calls this image viewer to present it to the user.

The image viewer accepts an image filename as a command line argument, loads it, and presents it on a window waiting for the user to press escape in order to exit. The image viewer uses the "X Window System" (X11) to present the image and handle user input.

5.3.4 OpenGL Real-Time Scene Previewer

Rendering a complicated scene may take a lot of time, especially due to the rudimentary nature of the implemented optimizations. Thus a real-time OpenGL, rendering application was created, able to read most of the XML file format used by the actual rendering system, and provide a very fast, but crude, preview of the geometry and the positioning of the objects in the scene.

5.3.5 The Fractal Generator

A very simple scheme[1] program (`genfract.scm`) was written, to generate a complex 3D fractal object comprised of thousands of spheres, for early performance testing (before the 3ds2xray converter was written). More on this in the testing chapter.

Also this program proves that the objective of a simple, easy to manupulate, scene description was attained, since a simple program such as this is able to generate a complex scene description for the renderer.

[1] Scheme is a purely functional LISP dialect, developed at the MIT by Guy Steele Jr. and Gerald Sussman

Chapter 6

Testing and Results

This chapter will discuss the testing methods, and talk about the results achieved by this project. Objectives met, features left unimplemented, and lessons learned during the development of this humongous final project.

6.1 Testing

There are many different aspects to the testing performed during the development of this system. The most important are, testing of the rendering algorithms, as they were implemented, cross-platform compatibility testing, and performance testing.

6.1.1 Rendering Algorithm Testing

Throughout the development of the system, the rendering algorithms involved were continuously tested as they were implemented, to verify that the expected output is generated. Due to the nature of the output, the only way to test it is of course by looking at the picture, having in mind a preconception (from experience) of what the picture should look like, and trying to guess what did go wrong if a discrepancy pops up.

This technique works well most of the time, due to the tendency of rendering bugs to leave visible artifacts in the rendering. For example, during the initial implementation of the basic ray tracing algorithm, there was an obvious mystery bug in rendering transparent objects (where refraction rays are spawned after hitting the surface), that instead of producing the expected refraction through the transparent object, made the object become mostly grey-colored, with multiple specular highlights being visible as if by many reflections. After a great deal of intrumentation and debugging, the culprit was found to be a single missing line of code, that was needed to invert the surface normal if a ray is exiting, as oposed to entering the object. This omission would result in rays entering the object to

become trapped inside, being reflected around to the recursion limit, instead of exiting from the oposite side of the object.

6.1.2 Cross-Platform Compatibility Testing

Since one of the major objectives of the project is cross-platform compatibility, it was very important to verify that the program is able to work on many different architectures and operating systems. This is where cross-platform compatibility testing is useful, since although the code was written with such compatibility goals in mind, it is sometimes hard to predict some tricky compatibility bugs that may creep in the program.

For this sort of testing, of course the procedure is to try to compile, and run, the renderer on different architectures, and see if the results are the same. Compatibility was continuously verified by testing the whole system during implementation, on three different target machines, namely:

- a 32bit x86 computer (little endian), running GNU/Linux;
- a 64bit x86-64 computer (little endian again) computer, running GNU/Linux;
- and a 64bit MIPS-based (big endian) Silicon Graphics workstation, running the IRIX operating system.

As a result of the extensive testing, the whole rendering system is verified to run correctly on all of these architectures.

Furthermore, partial testing on various parts of the rendering system was performed on an Apple ibook running MacOS X, a SUN UltraSPARC workstation running Solaris, and some last-minute tests were performed on an MS Windows based PC.

As an extreme experiment in cross-platform compatibility, a small subset of the rendering code implementing only the most basic ray tracing algorithm, was modified slightly to run on a very specialized and resource-limited embedded platform, the Nintendo GameBoy Advance, with an ARM7 processor, and no operating system.

The modifications necessary for that experiment, apart from adding code to handle graphics output for the gameboy hardware, involved merely reducing the memory footprint of the program to fit into the 288 kilobytes of RAM provided by the system. Figure 6.1 shows this experiment in action.

6.1.3 Performance Testing

Performance testing became a secondary concern, since the original optimizations planned for the rendering system were not implemented due to time constraints. However, an early performance test was conducted using the previously described

Figure 6.1: GameBoy Advance Experiment

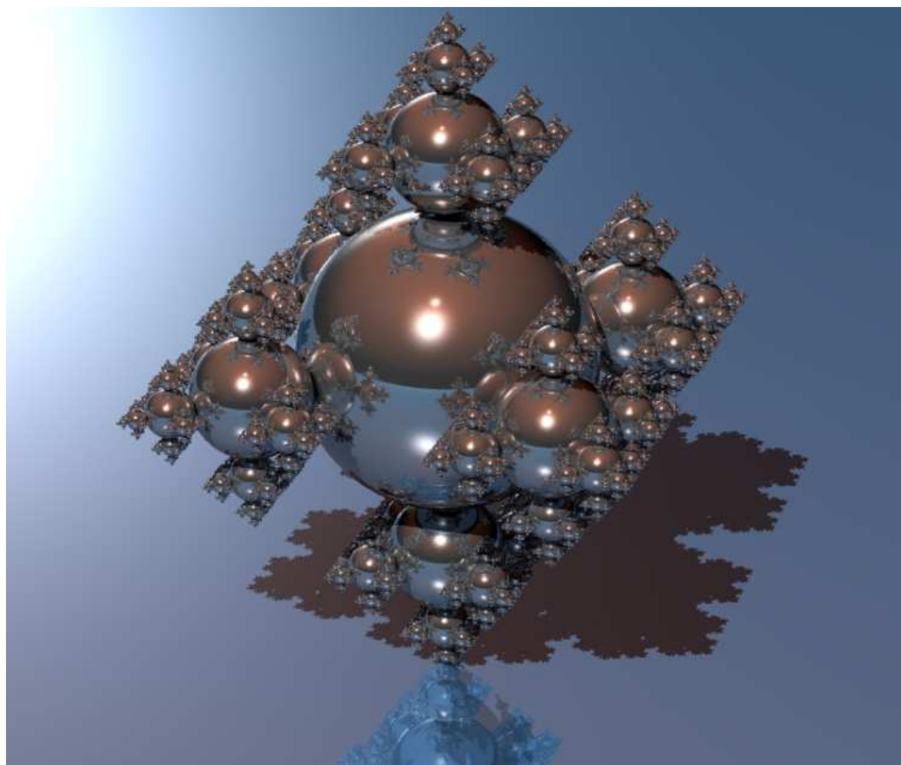

Figure 6.2: Sphere Fractal Rendering

fractal generator program, to generate a 3D fractal containing many thousands spheres, with full shading and inter-reflections (see figure 6.2).

The result of the performance test, was as expected. The time spent rendering increases exponentially with the amount of objects in the scene. Note that, the implementation of a simpler bounding volume optimization is still significant for mesh rendering, by providing the chance to cull whole objects without having to test each of their polygons, otherwise the computational time would explode with multiple polugon meshes in a single scene.

6.2 Results

Even though the project proved to be very big in terms of raw lines of code, for the time available for its completion, and some parts of the initial design did not make it into the final system due to time constrains, it seems to be a success. Most of the aims and objectives of the project are fully realized, and most of the important design elements were implemented sucessfully.

A detailed discussion of the points of success and failure of the project follows. But before continuing with the technical points, I'd like to add another success of this project; this paper was accepted for presentation at the "5th International Student Spring Symposium on the Internet and Web Technologies" in Thesaloniki in a special section about "software development and computer graphics".

6.2.1 Sucesses

The major objective of programmability and extensibility through programmable shaders, was attained as described in the design and implementation of the libxray core library. The user is able to specify shaders for each material, to control the shading calculations and further rendering process. The shaders are efficient (compiled), easy to use (automatic server-side compilation), and easy to write (intuitive shader API).

Another major objective, cross-platform compatibility, is also achieved, as described previously. The system is able to compile and run on any of the target architectures without any changes, and even extremely usual architectures can be supported with some small modifications.

The next major objective, of photorealism is definitely achieved, by simulating a wide range of optical effects, and light interactions in a 3D environment, including perfect and imperfect (glossy) reflections, refractions, shadows from point or area light sources (soft shadows), temporal and spatial antialiasing, high dynamic range output, etc. Also the ability to define custom shaders for each surface aids photorealism in a big way, by providing the tools to simulate the interaction of light with any kind of surface, capturing the subtleties of various materials, or to implement new more realistic rendering techniques.

Figure 6.3: Soft shadows, glossy reflections, and texture modulation of any material attribute, like the typical diffuse color, or the unusual reflectivity coefficient on the shiny sphere

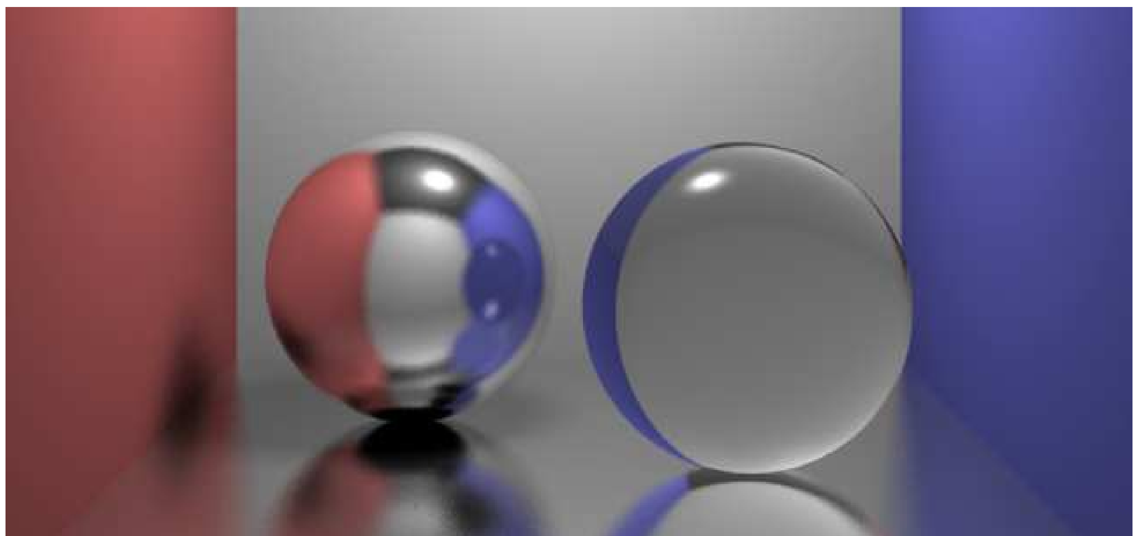

Figure 6.4: Soft shadows, glossy reflections and refraction in a cornell box

Figure 6.5: A scene converted from a 3ds file

Figure 6.6: A more complex scene converted from a 3ds file

Interoperability is achieved by using a simple to create, modify, and parse XML based file format for scene representation, which makes it easy to write converters from many different available file formats (see 3ds2xray discussion above).

The implementation of the final system is according to the original elegant design, with clean seperation between components and networked rendering support, providing maximum reusability, and flexibility. Despite the time constrains, the code is kept clean, readable, and modular. Without resorting to hacks and quick & dirty coding.

6.3 Between Success and Failure

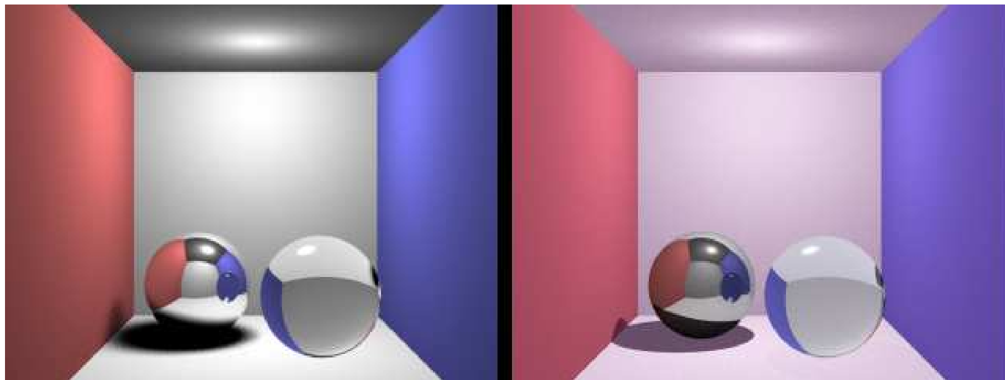

Figure 6.7: Left: the cornell box with direct illumination only. Right: mixed success in the attempt at simulating global illumination

There was not enough time to simulate global illumination and caustics completely which would add another level of photorealism to the renderings of this system. Figure 6.7 shows a last minute attempt to compute diffuse-diffuse global illumination interactions, shown side-to-side with the same view using only direct rendering. It is aparent, that some amount of light, did travel to places where the direct illumination failed to reach, which means that there is indeed light traveling indirectly through other surfaces to these areas. However there is no color bleeding from the blue wall, only from the red one, this hints to a sampling error, or wrong distribution of sampling rays for the diffuse bounce. So, some aspects of the global illumination were captured, but it's buggy, slow, and doesn't exactly work correctly. Also another aspect of global illumination, caustics were not implemented at all.

6.4 Failures

The original intention to use a spatial data structure (a regular grid) with a 3D version of bresenham to accelerate ray testing was abadoned due to time contrains. Instead a simpler, but not insignificant, optimization was implemented, using bounding volumes to cull away complicated objects (such as meshes) without having to test every polygon for intersection. Still, rendering times for complicated scenes are increasing exponentially, and a better acceleration scheme definitely needs to be implemented sooner or later (see future work).

This isn't really a failure as such, but it would be better to have time to implement more illumination models in the default shaders, such as "Torrance & Cook" or "Oren & Nayar". While not essential, it would provide a wider selection of ready to use shaders, instead of having to implement them himself if they are needed.

Chapter 7

Conclusion & Future Work

7.1 Future Work

This section will provide ideas on how to enhance the rendering system further and how it can be taken into new directions not explored in this project.

7.1.1 Global Illumination

One of the few features partially implemented due to time constrains, is the simulation of global illumination fenomena. This will definitely be completed later on, as it is a very interesting algorithmically, to implement and optimize in order to make it useful, for taking photorealism to the maximum level. A combination of Kajiya's path tracing along with the photon mapping importance sampling algorithm is the prefered way of attacking this problem.

7.1.2 Optimization

Another important possible improvement to the system which was left out due to time constrains is more elaborate optimization of the rendering process. A first and effective step to achieve much faster rendering rates would be to use a regular grid with each cell containing pointers to objects that intersect it. This way we can use a simple discrete cell line tracing algorithm like bresenham to find the cells from which each ray passes through, and only test the relevant objects for intersection.

Another interesting technique, is to use a fast rendering algorithm to calculate and store which object is the first one to be intersected by any *primary* ray, thus accelerating the first stage of ray tracing by a great margin. Of course this may only work for objects that can be handled by fast rendering algorithms, such as triangular meshes, but not pure mathematical surfaces.

7.1.3 Distributed Rendering

This has been hinted at many points throught this document. The flexibility that we get from having a seperate rendering server, can be exploited to distribute the rendering process to multiple computers in a network. This can be done simply by writing a program that responds to client programs in exactly the same manner as the rendering server, but instead of doing the actual work, it splits it up and distributes it among the real rendering servers. This is a great way to attack the problem of distributed rendering, as it needs no modifications to the actual server or client programs, and the user does not need to even know that it's happening.

7.1.4 Converters and Plug-Ins

Again this has been discussed in the previous sections of this document. It would be very useful to have converters from various different scene file formats to the XML format used by this renderer. Also creating plug-ins that integrate the functionality of this renderer with widely used 3D authoring suits would make the system very easy to use for artists, thus boosting the popularity of this project, and attracting a user community, and developers.

7.1.5 Add Animation Capabilities to the Scene Format

The fact that the scene format does not support the keyframe animation capabilities available by the rendering system, hampers its use for creating animated sequences. Thus, it would be a very useful addition to add such capabilities to the file format.

7.2 Conclusion

By looking back at the whole development of this project, it is aparent in retrospect, that it was a really large and ambitious endeavor. To create such an advanced, photorealistic, and programmable rendering system needs a lot of effort, time, and experimentation. Not to mention, a lot of knowledge, and prior experience in the field of computer graphics.

Fortunately, I had the necessary background to tackle this problem effectively, but unfortunately not enough to do so "single-handedly". This, coupled with poor time management led to significant time constraints during the last part of the project which in turn led to the partial failure of one of the elements of advanced photorealistic rendering. And complete abadonment of anything more than rudimentary performance optimization.

Still, these where but small issues in the overall aims and objectives of the project, which managed to provide a surprisingly big amount of functionality com-

parable to huge comercial renderers and previously unavailable to the free software community.

I'm convinced that if the enhancments suggested into the "future work" section are performed, and the limitations are eliminated, this may be a very useful tool for the graphics community, to use, as both a tool to cover most 3D visualization needs, or as a development platform for further research in the field of computer graphics and photorealistic rendering.

Bibliography

[1] "Pov-ray web site." [Online document], [cited 2005 Dec 14], Available HTTP: http://www.povray.org.

[2] "Mental ray web site." [Online document], [cited 2005 Dec 14], Available HTTP: http://www.mentalimages.com.

[3] W. Turner, "An improved illumination model for shaded display," *CACM, 1980*, vol. 23, no. 6, pp. 343–349, 1980.

[4] P. S. Heckbert, "Adaptive radiosity textures for bidirectional ray tracing," in *Proceedings of SIGGRAPH 1990*, vol. 24, pp. 145–154, Aug. 1990.

[5] A. Appel, "Some techniques for shading machine renderings of solids," in *AFIPS 1968 Spring Joint Computer Conf.*, vol. 32, pp. 37–45, 1968.

[6] R. Siegel and J. R. Howel, *Thermal Radiation Heat Transfer.* Hemisphere Publishing Corp., 1981.

[7] J. T. Kajiya, "The rendering equation," in *Proceedings of SIGGRAPH 1986*, pp. 143–150, ACM Press / ACM SIGGRAPH, 1986.

[8] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing," in *Proceedings of SIGGRAPH 1984*, pp. 137–145, July 1984.

[9] H. W. Jensen, *Realistic Image Synthesis Using Photon Mapping.* A. K. Peters, 2001.

[10] B. T. Phong, *Illumination for computer-generated images.* PhD thesis, Dept. of Electrical Engineering, University of Utah, 1973.

[11] J. F. Blinn, "Models of light reflection for computer synthesized pictures," in *Proceedings of SIGGRAPH 1977*, pp. 192–198, July 1977.

[12] R. L. Cook and K. E. Torrance, "A reflectance model for computer graphics," in *Proceedings of SIGGRAPH 1981*, Computer Graphics Proceedings, Annual Conference Series, ACM, ACM Press / ACM SIGGRAPH, 1981.

[13] M. Oren and S. K. Nayar, "Generalization of lambert's reflectance model," in *Proceedings of SIGGRAPH 1994*, Computer Graphics Proceedings, Annual Conference Series, ACM, ACM Press / ACM SIGGRAPH, 1994.

[14] C. Schlick, "A customizable reflectance model for everyday rendering," in *Fourt Eurographics Workshop on Rendering*, pp. 73–84, Eurographics, June 1993.

[15] P. E. Debevec, "Rendering synthetic objects into real scenes," in *Proceedings of SIGGRAPH 1998*, pp. 189–198, 1998.

[16] T. Apodaca, "The RenderMan interface," *j-BYTE*, vol. 14, pp. 167–176, Apr. 1989.

[17] "Final render web site." [Online document], [cited 2005 Dec 14], Available HTTP: http://www.finalrender.com.

[18] "Yafray web site." [Online document], [cited 2005 Dec 14], Available HTTP: http://www.yafray.org.

[19] A. Watt, *3D Computer Graphics.* Addison-Wesley, third ed., 2000.

[20] A. Watt and M. Watt, *Advanced Animation and Rendering Techniques: Theory and Practice.* Addison-Wesley, 1992.

[21] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer graphics: Principles and practice in C.* Addison-Wesley, 2nd ed., 1996.

[22] A. S. Glassner, ed., *An Introduction to Ray Tracing.* Academic Press, 1989.

[23] R. Parent, *Computer Animation, Algorithms and Techniques.* Morgan Kaufmann, 2002.

[24] K. Perlin, "An image synthesizer," in *Proceedings of SIGGRAPH 1985*, Computer Graphics Proceedings, Annual Conference Series, pp. 287–296, ACM, ACM Press / ACM SIGGRAPH, 1985.

[25] W. T. Reeves, "Particle systems: A technique for modeling a class of fuzzy objects," in *Proceedings of SIGGRAPH 1983*, Computer Graphics Proceedings, Annual Conference Series, ACM, ACM Press / ACM SIGGRAPH, 1983.

[26] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface reconstruction algorithm," in *Proceedings of SIGGRAPH 1987*, Computer Graphics Proceedings, Annual Conference Series, pp. 167–169, ACM, ACM Press / ACM SIGGRAPH, 1987.

[27] J. F. Blinn and M. E. Newell, "Texture and reflection in computer generated images," in *Proceedings of SIGGRAPH 1976*, Computer Graphics Proceedings, Annual Conference Series, ACM, ACM Press / ACM SIGGRAPH, 1976.

[28] W. R. Stevens, *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[29] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming: The Sockets Networking API*, vol. 1. Addison-Wesley, third ed., 2004.

Appendix A

Student – Supervisor Meeting Logs

Student: Ioannis Tsiombikas
Supervisor: Panagiotis Bamidis

Topics Discussed		
The first meeting's discussion was about an overview of the project, various goals and objectives of the project, and my initial ideas for various implementation details I had in mind.		
Deliverables & Drafts brought by student		
N/A		
No.	**Date**	**Signature**
1	27 September 2005	

Topics Discussed		
During this meeting, the overview of the system, and the various modules or subsystems were discussed. Also, we went over the organization of the literature review, and discussed the various sources.		
Deliverables & Drafts brought by student		
System overview notes, and a literature review skeleton		
No.	**Date**	**Signature**
2	12 December 2005	

Topics Discussed		
Second literature review feedback, and short discussion about the progress so far.		
Deliverables & Drafts brought by student		
None.		
No.	**Date**	**Signature**
3	13 January 2006	

Topics Discussed		
Demonstration of the system so far, including the experimental xrayd/protocol, and initial scene file input code, and scene file format. Also, the short term schedule was discussed, and various possibilities for testing the system on various platforms.		
Deliverables & Drafts brought by student		
Code and binaries of the running system, and a testing scene XML file, for the demonstration.		
No.	**Date**	**Signature**
4	8 February 2006	

Topics Discussed		
The final choice of using C++ shaders was discussed along with some early performance testing results.		
Deliverables & Drafts brought by student		
None.		
No.	**Date**	**Signature**
5	28 March 2006	

Topics Discussed		
Tight schedule constraints, and the possibility of dropping some of the initial optimization ideas while trying to complete the rest of the stuff. The 3ds2xray converter tool was demonstrated using a crude real-time OpenGL based renderer.		
Deliverables & Drafts brought by student		
3ds2xray + OpenGL previewer		
No.	**Date**	**Signature**
6	15 April 2006	

Topics Discussed		
Some ideas on the writeup were discussed, in the absence of an actual draft, due to delays in trying to implement and test some last-minute features.		
Deliverables & Drafts brought by student		
None.		
No.	**Date**	**Signature**
7	25 May 2006	

Appendix B

X-Ray Rendering System Installation Manual

Welcome to the installation manual of the X-Ray Rendering System. The following manual, describes the process of installing and configuring the rendering server. These instructions are applicable to the installation of the UNIX version of the rendering system, which runs in a multitude of different UNIX systems and computer architectures, and is the only version that is officially supported through these manuals.

B.1 Requirements & Dependencies

These are the requirements for compiling and installing the rendering system:

- Any UNIX system should do, for example: GNU/Linux, Any BSD, IRIX, Solaris, or MacOS X. Using cygwin for compiling and installing on windows, should also work.
- The GNU C and C++ Compilers (aka GCC) needs to be installed in the system. These compilers are used both for compiling the rendering system itself, and for during the operation of the renderer, for compiling user-supplied shader programs. Using another compiler should work, but is unsupported at this point.
- A standard bourne-compatible shell (like bash) is needed for running the configuration script, and GNU Make must be installed for compiling the renderer.

B.2 Installation Instructions

1. From the top-level xray directory, run "`./configure`" to start the custom auto-configuration script, which will detect your compiler, and create the appropriate makefile. If you wish to change the default installation prefix (`/usr/local`), you may also pass a "`--prefix=dir`" argument to the configuration script. For more options, run the script with "`--help`".

2. After the auto-configuration completes, type "`make`" (or "`gmake`" if your default make utility is not the GNU make), to compile the rendering library and the rendering daemon. If you also wish to compile the client program, change into the "`frayclient`" subdirectory and type "`make`".

3. When the compilation is complete, log in as root, and type "`make install`" to install everything automatically. If you also wish to install the client program, change into the "`frayclient`" subdirectory and also type "`make install`".

That's it, the rendering server (and/or client) is installed and ready to be used. At any point, type "`xrayd`" in a terminal to start the rendering daemon.

In the top-level xray directory, there are also other subdirectories with various utilities related to the rendering system. If you wish to use any of these, change into their respective directory and type "`make`" and "`make install`" to compile and install them as well.

Enjoy creating amazing renderings with the X-Ray Rendering System.

Appendix C

X-Ray Rendering System User Manual

This short manual describes how to use the "frayclient" program for rendering images with the X-Ray Rendering System by providing a few simple usage examples.

If you have a scene file called "foo.xml" and you wish to render it on a rendering server running on the host "rsrv", just type:

```
$ frayclient -a rsrv -o foo.ppm foo.xml
```

Also, instead of the host name of the rendering server, you may use its IP address. If you don't specify an address or host name, the client will try to connect to the localhost. Also if you don't specify an input or output file, it defaults to reading from the standard input and writing to the standard output.

So, let's say you rather wish to generate a scene file by running the program genfract.scm, through a scheme interpreter, like GUILE, and pass it directly to the renderer, you may use a command like this:

```
$ guile genfract.scm | frayclient -a rsrv >foo.ppm
```

Thus using the rendering client as a filter program in a pipeline.

Finally you can convert a 3ds scene file to an XML representation readable by the X-Ray Rendering System, with the command:

```
$ 3ds2xray foo.3ds >foo.xml
```

Appendix D

The DTD of the XML Scene Description Format

```
<!ELEMENT scene (environment?,mat-list?,(object|camera|light)*)>
<!ATTLIST scene name CDATA #IMPLIED>

<!ELEMENT environment ((background|env-ior)?)>
<!ELEMENT background EMPTY>
<!ATTLIST background rgb CDATA "0, 0, 0">

<!ELEMENT mat-list (material*)>

<!ELEMENT material (attr*)>
<!ATTLIST material name CDATA #REQUIRED>
<!ATTLIST material shader CDATA #IMPLIED>

<!ELEMENT attr EMPTY>
<!ATTLIST attr name CDATA #REQUIRED>
<!ATTLIST attr val CDATA #IMPLIED>
<!ATTLIST attr tex CDATA #IMPLIED>

<!ELEMENT object (matref?, xform?, (mesh|mesh3ds|sphere)?)>
<!ATTLIST object name CDATA "unnamed">

<!ELEMENT matref EMPTY>
<!ATTLIST matref ref CDATA #REQUIRED>

<!ELEMENT xform EMPTY>
<!ATTLIST xform pos CDATA #IMPLIED>
<!ATTLIST xform rotquat CDATA #IMPLIED>
<!ATTLIST xform scale CDATA #IMPLIED>
<!ATTLIST xform pivot CDATA #IMPLIED>

<!ELEMENT mesh (vert-list,tri-list)>
<!ELEMENT vert-list (vertex*)>
<!ELEMENT tri-list (tri*)>
```

```
<!ELEMENT vertex EMPTY>
<!ATTLIST vertex pos CDATA #REQUIRED>
<!ATTLIST vertex tex CDATA #IMPLIED>

<!ELEMENT tri EMPTY>
<!ATTLIST tri idx CDATA #REQUIRED>

<!ELEMENT mesh3ds EMPTY>
<!ATTLIST mesh3ds file CDATA #REQUIRED>
<!ATTLIST mesh3ds name CDATA #IMPLIED>

<!ELEMENT sphere EMPTY>
<!ATTLIST sphere rad CDATA "1">

<!ELEMENT light EMPTY>
<!ATTLIST light name CDATA #IMPLIED>
<!ATTLIST light type CDATA #IMPLIED>
<!ATTLIST light pos CDATA #IMPLIED>
<!ATTLIST light size CDATA #IMPLIED>
<!ATTLIST light color CDATA #IMPLIED>
<!ATTLIST light int CDATA #IMPLIED>

<!ELEMENT camera EMPTY>
<!ATTLIST camera name CDATA #IMPLIED>
<!ATTLIST camera pos CDATA #IMPLIED>
<!ATTLIST camera target CDATA #IMPLIED>
<!ATTLIST camera fov CDATA #IMPLIED>
```