A Global Illumination Ray-Tracer

being a dissertation submitted in partial fulfilment of
the requirements for the Degree of Master of Science
in the University of Hull

by

Ioannis Tsiompikas
BSc (Hons) Computer Science, University of Sheffield

September 2009

Abstract

Rendering photorealistic images of 3D environments such as the ones used in computer graphics films, special effects and architectural visualization, requires algorithms capable of accurately simulating the flow of light in a virtual environment.

Global illumination is a particularly important, but problematic aspect of lighting in a 3D environment, as most conventional rendering algorithms are incapable of taking into account the contribution of light arriving at any given point from nearby surfaces, as opposed to light coming directly from the light source.

The renderer presented here uses the photon mapping algorithm in order to simulate the two most difficult to capture effects of global illumination: light being focused by specular objects such as curved mirrors and lenses (*caustics*), and diffuse interreflections, that is, light arriving to illuminate a surface after being diffusely scattered by another lambertian surface.

Contents

List of Figures

Chapter 1

Introduction

From the inception of computer graphics, back in the days of hidden line algorithms on vector displays, one of the biggest driving motives in the development of computer graphics algorithms has been the quest to reach the elusive goal of photorealism.

There are many applications which require the use of photorealistic rendering algorithms. Films and advertisments use photorealistic computer graphics to seamlessly integrate artificial characters alongside real actors, or are completely rendered by such algorithms. Products can be visualized using photorealistic graphics to see how they will look like before being put into production. Interiors can be visualized under various lighting conditions to find the best type of lighting and optimal placement of light sources.

All of the aforementioned applications have one thing in common: rendering time can be sacrificed in order to achieve the best possible rendering quality, which makes the ray tracing rendering algorithm the prime choice for such applications.

The ray tracing rendering algorithm is almost ubiquitous in off-line photorealistic renderers. Even though it's inherently much more computationally expensive than polygon filling techniques, the range of effects one can achieve with ray tracing makes it the algorithm of choice when image quality is at a premium, and execution time can be sacrificed to achieve the highest possible degree of photorealism.

The basic ray tracing algorithm is quite capable of simulating a wide range of phenomena which are hard or impossible to achieve with polygon fillers. Such effects as accurate reflections, refractions and shadows are trivial to implement with a ray tracer, but are considerably difficult to approximate with polygon filling or scanline rendering.

Moreover, the algorithm is simple to implement, more intuitive, and very elegant, requiring on the basic level only the ability to find intersections between rays and surfaces, which is then used for almost everything, throughout the whole rendering process.

However, there are some effects, which cannot be simulated by the basic ray tracing algorithm. The following are some of the phenomenae that cannot be captured by a basic ray tracer:

- Imperfect (glossy) reflection and refraction due to irregularities on the surface of specular objects.
- Shadows with penumbra regions (*soft shadows*), which is caused by partial occlusion of light sources with non-zero area.
- Motion blur, produced when fast-moving objects are photographed by a camera with finite shutter speed.
- Depth of field. That is, limited focal range of the camera resulting in out of focus areas in the rendered image.
- light arriving to a surface after being scattered by diffuse surfaces *(indirect diffuse illumination).*
- light arriving to a surface after being reflected or refracted by specular surfaces *(caustics).*

These deficiencies can be addressed by extending the ray tracing algorithm to use monte carlo integration as described by Cook et al.[1], Kajiya[2], and Jensen[3].

1.1 Project Objectives

As implied by its title, the main objective of this project, is to implement a ray tracer capable of simulating global illumination and, particularly, the rather hard to capture effects of diffuse interreflection, and caustics.

Diffuse interreflections are particularly important for photorealistic rendering. Especially in renderings of interiors, where the total illumination is commonly dominated by light arriving indirectly through windows or “hidden” lights.

Caustics are also an important element of a photorealistic image. Transparent objects would cast a solid black shadow in the absence of caustics, which looks totally unrealistic. We expect to see the light being focused by a glass of whisky on the table, or the shimmering light at the bottom of a swimming pool, and those environments would look fake in the absence of caustics.

In addition to indirect diffuse illumination and caustics, all effects previously described as falling into the domain of monte carlo ray tracing are also implemented for completeness and out of the desire to produce a useful renderer, as opposed to merely a toy program demonstrating only a narrow aspect of photorealistic rendering.

For the same reason, an effort has been made to make the renderer reasonably fast without however expending a great effort in optimizations since this wasn't a primary objective of the project. Naive algorithms have been avoided, and efficient data structures have been chosen to make the renderer run fast enough to be generally useful.

One way to speed up the rendering would be to take advantage of the highly-parallel special-purpose graphics processing units which have become a comodity in modern computers. However, a GPU implementation would have to deviate a lot from the pure CPU-based algorithms due to quirks of the GPU programming model. Such a deviation would complicate the program considerably, and distract from the stated goal which is the implementation of global illumination algorithms, and particularly "photon mapping". In fact, the application of these algorithms in the context of a GPU is an open research topic at the moment, and thus can hardly be treated as an implementation detail of a project which doesn't have that as its explicit goal.

The renderer is called $\int ray$, or "s-ray" where it is inconvenient to use the integral symbol. The name is made up by concatenation of the integral symbol, which alludes to the prominent use of monte carlo integration throughout the renderer, followed by the customary in ray tracers, "-ray" suffix.

In the following chapters, after a quick overview of the most important algorithms in the field of photorealistic rendering, which constitute the foundations onto which the s-ray renderer is built, the design and implementation of the renderer will be discussed in detail, including an in-depth explanation of the key algorithms used. Finally results will be presented, followed by a critical examination of the strengths and shortcomings of the s-ray renderer.

Chapter 2

Background

From the first computer line drawings of Ivan Sutherland to the vivid photo-realistic renderings of today, a huge amount of effort has been expended to come up with algorithms that capture various properties of the interaction of light with the environment.

2.1 Light Path Notation

When discussing the various possible interactions of light with the environment, it is convenient to have a notation which describes particular light paths, or particular classes of light paths concisely. Such a notation was proposed by Paul Heckbert[4].

Heckbert's light path notation uses a set of symbols: $\{L, S, D, E\}$ as its alphabet, which stand for *light*, *specular interaction*, *diffuse interaction*, and *eye* respectively. By combining these symbols to form strings, always starting with L and terminating in E, one is able to describe any path of light starting from a light source and reaching the viewer. For instance, light hitting a diffuse sphere, and seen through the mirror would correspond to an `LDSE` path, while a caustic formed on a diffuse surface by light being transmitted through a lens would be an `LSDE` path.

Heckbert also proposed the use of regular expressions, to describe classes of possible light paths. For instance, a simple ray tracer which can simulate at most one diffuse reflection, followed by an number ofspecular interactions along the path of light, before arriving to the observer, is able to simulate `LD?S*E` paths. In comparison, the full set of light paths simulated by global illumination algorithms fall into the `L(D|S)*E` class.

2.2 Radiance and Irradiance

Two important terms in the discussion of light which shall be needed later on, are *radiance*, and *irradiance*. Irradiance is the amount of *radiant energy*, i.e. the amount of light, that arrives at a particular point on a surface per unit time (dt). Radiance on the other hand, is the amount of light *leaving* the surface towards a particular direction.

2.3 Rendering Algorithms

Next, we shall see an overview of the most important milestones in the development of photorealistic rendering algorithms, with particular emphasis to those algorithms which are most relevant to this project.

2.3.1 Ray Casting

In 1968, the fundamendal algorithm of ray-casting was introduced by Arthur Appel, in his paper: "Some Techniques for Shading Machine Renderings of Solids" [5].

Ray-casting works by "shooting" rays from the viewer through each pixel of the image plane, and calculating where is the nearest intersection, if any, of the ray with the objects in the scene. At the point of the nearest intersection, a new ray (called a "shadow ray") is cast towards the light source to determine if there are any objects obstructing the path between that point and the light source. If no intersections are found by the shadow ray, then the point is not in shadow, and lighting calculations are carried out to compute a color for that pixel. If multiple light sources are used, then lighting is accumulated for each non-obstructed light source and the sum is used as the pixel color.

2.3.2 Whitted Ray Tracing

One of the biggest early breakthroughs in photorealistic rendering was the introduction of recursive ray tracing by Turner Whitted [6].

Whitted solved the problem of simulating reflection and refraction in a simple and elegant manner, by performing ray casting recursively when a ray intersects reflective or transparent surfaces.

When a ray hits a reflective surface, a secondary reflection ray is spawned at that point and cast recursively towards the reflection direction. Similarly,

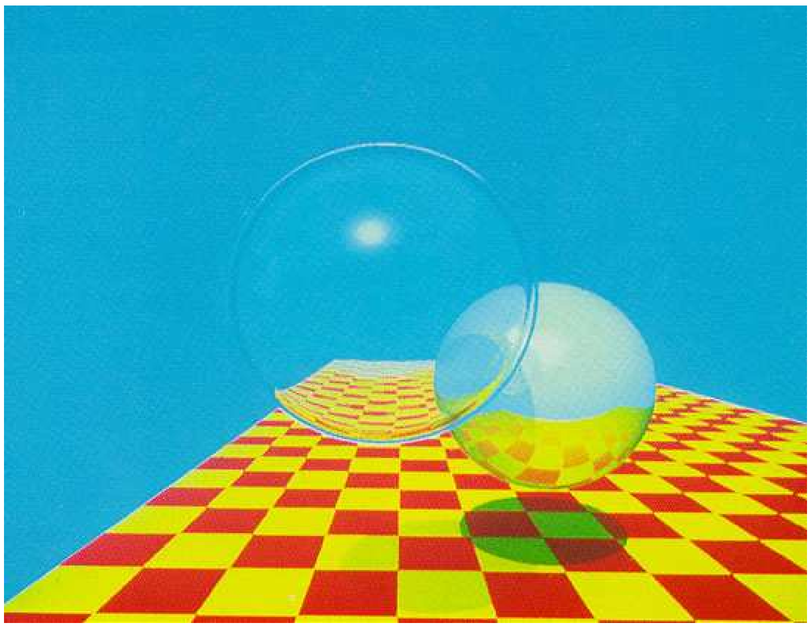

Figure 2.1: Reflection and refraction through recursive ray tracing. Turner Whitted.[6]

intersections with transparent objects result in secondary rays being transmitted through the object towards the refraction direction. Illumination collected by secondary rays are added to the direct illumination, weighted by the reflectivity or transparency of the object accordingly.

Ray tracing is in fact a global illumination algorithm, since it takes into account light arriving at perfect specular surfaces from other parts of the scene. It is not usually referred as such however, due to the very limited range of light paths it can simulate. In fact as was already mentioned, Whitted ray tracing is able to simulate `LD?S*E` paths in Heckbert's notation.

2.3.3 Distribution Ray Tracing

One failing of the simple ray tracing algorithm as presented by Whitted, is that the images it produces are artificially perfect. Every reflective surface is a perfect mirror, every transparent object is crystal-clear, etc.

The solution to that problem was given by Cook et al. by the introduction of a technique called *distribution ray tracing*,[1] which uses monte carlo integration to produce glossy reflections and refraction, penumbra shadows, motion blur, and depth of field.

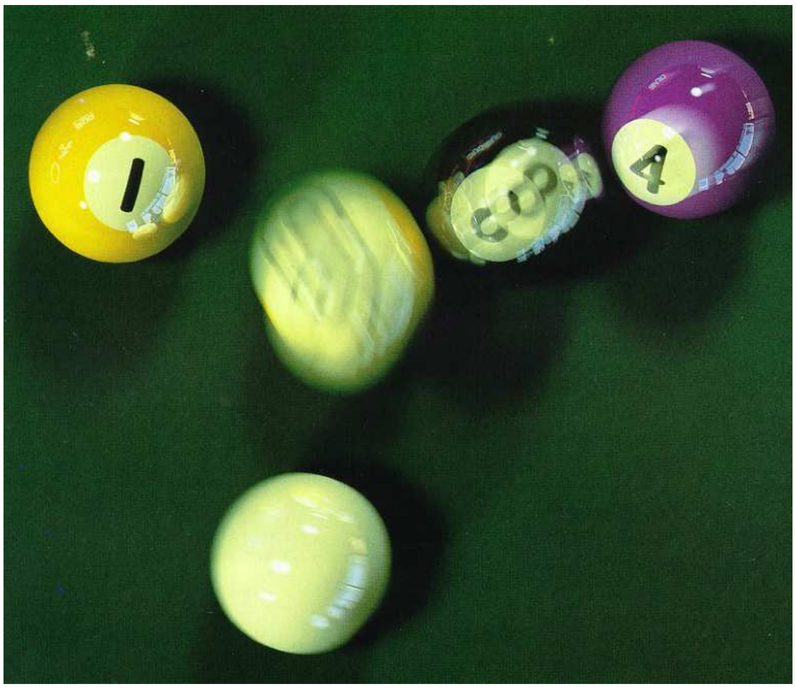

Figure 2.2: Motion blur through distribution ray tracing. Cook et al.[1]

All these effects are implemented by random (monte carlo) sampling at various rendering stages. For instance, glossy reflections are simulated by spawning a bunch of secondary reflection rays, randomly distributed around the perfect reflection direction, whilst motion blur is simulated by temporal distribution of rays around the frame time, in the interval dictated by the shutter speed.

2.3.4 Radiosity

The radiosity algorithm [7, 8] takes photorealistic rendering one step further by being able to simulate diffuse interreflections between objects, which is generally what is referred to by the term “global illumination”.

Radiosity is an adaptation to graphics, of a finite element technique used in calculating thermal transport [9]. It works by breaking up the scene into a number of “patches” each of which acts as a lambertian reflector, gathering energy from all patches in the hemisphere above its surface, and reflecting a part of it back out towards all directions. Running a lot of iterations of going arround each patch and gathering energy from visible patches, eventually converges to a good approximation of the distribution of light in the scene.

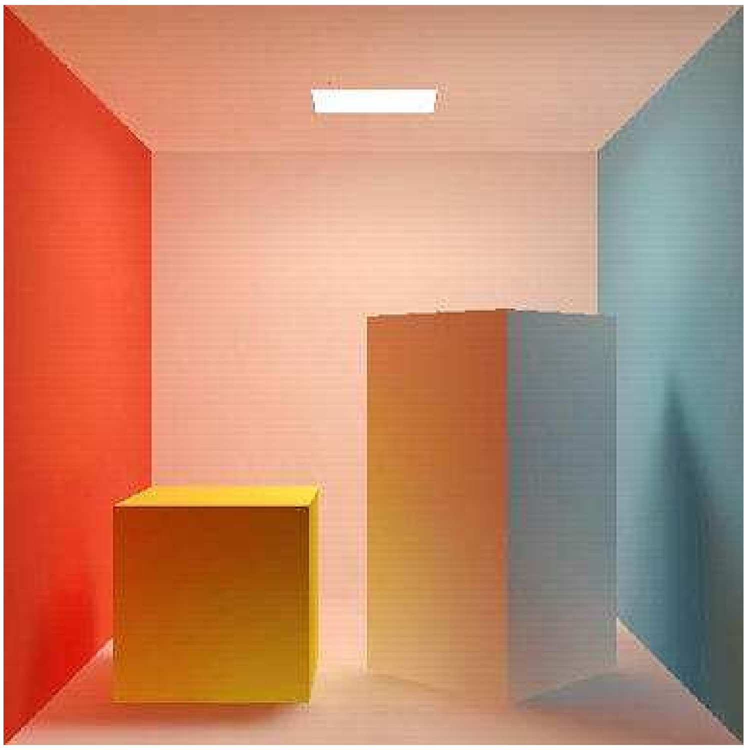

Figure 2.3: Diffuse interreflections simulated with the radiosity algorithm. Cohen et al.[8]

In light path notation, a radiosity renderer is capable of simulating only `LD*E` paths.

2.3.5 The Rendering Equation

James Kajiya published in 1986 his seminal paper "The Rendering Equation", which unified all rendering algorithms by showing that they are all in fact approximations to the solution of a single rendering equation. [2]

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

He also introduced *path tracing*; a ray tracing algorithm which uses monte carlo integration to completely solve the rendering equation. Although quite inefficient, path tracing is indeed capable of simulating all possible light paths in a 3D environment (`L(D|S)*E` paths), including difficult to capture effects such as diffuse interreflection and caustics.

Path tracing works by shooting a lot of rays for each pixel of the image, and then following each of those rays as it interacts with the environment. Each ray follows just a single path, so for instance if a ray hits a reflective and refractive surface, only one of those directions will be followed randomly using the reflectivity and transmission coefficents of the material as a probability density function. In case a ray hits a diffuse surface, a random direction on the hemisphere above the intersection point is picked and followed.

It is natural that all this randomness results in a lot of noise in the generated picture, however by averaging the results of many paths per pixel the image slowly converges to an accurate solution of the rendering equation.

2.3.6 Photon Mapping

Finally, in 1995, Henrik Wann Jensen described a technique called *photon mapping*, an efficent and accurate algorithm for computing global illumination and caustics[3]. A considerable advantage of photon mapping over the radiosity algorithm is that illumination calculations are doucoupled from the underlying geometry of the scene, and also it can easily be added on top of an existing monte carlo ray tracer without much added complexity.

The photon mapping algorithm is split into two distinct phases. First photons are traced from the light sources as they interact with the various objects in the scene, and are finally stored in a spatial data structure called a *photon map* when they are absorbed. Then, during ray tracing, the photon map can be used to provide radiance estimates and importance sampling information, needed for computing global illumination and caustics effects.

The photon mapping algorithm will be explained in greater detail in subsequent chapters, as it forms the basis for this project.

2.4 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.

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 incident light, and properties of the surface.

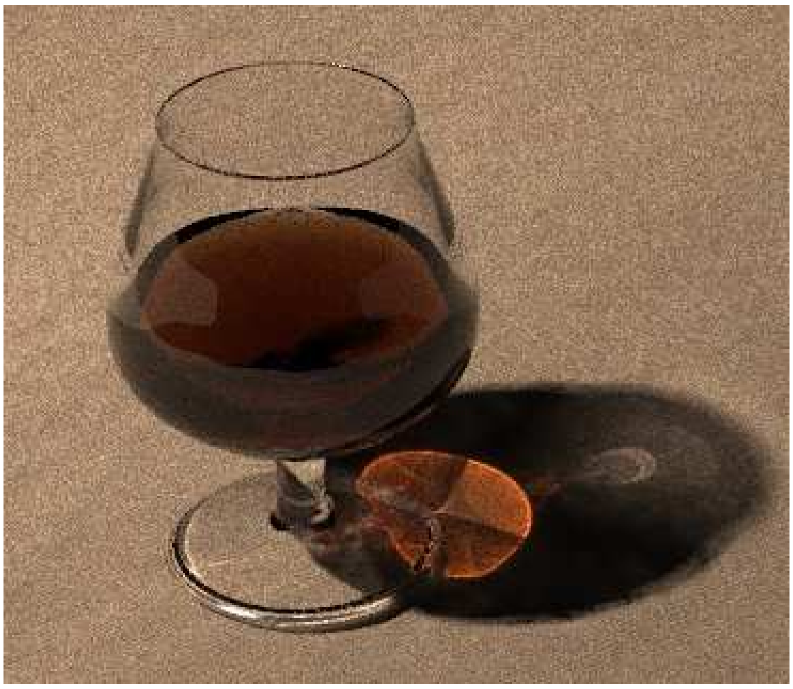

Figure 2.4: Caustics can be simulated easily by the photon mapping algorithm. Henrik Wann Jensen[3]

2.4.1 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.

Lafortune et al. presented a way to generate random samples on the phong specular lobe[12] which can be used to importance sample the phong BRDF in monte carlo ray tracers.

2.4.2 Cook and Torrance

Torrance and Cook [13] in 1981, presented a physically accurate reflectance model which is particularly suited for metallic surfaces, which the phong

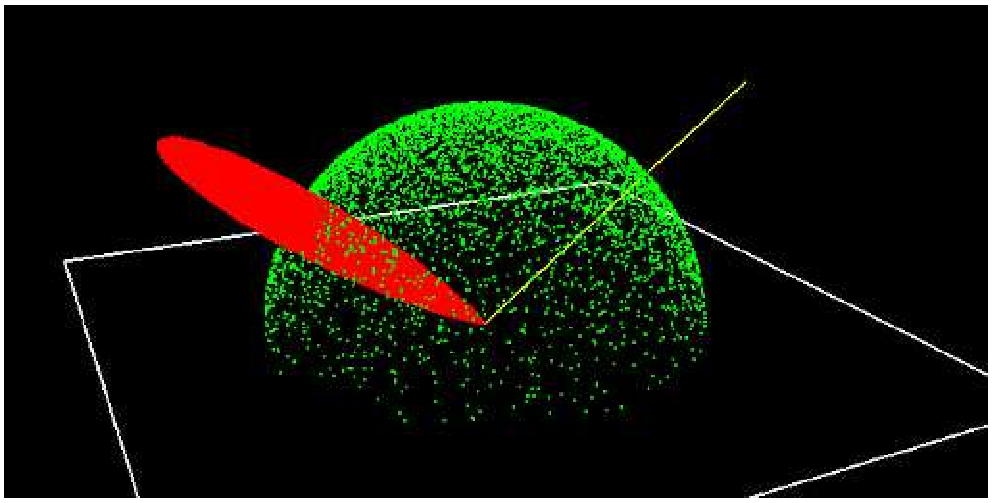

Figure 2.5: Importance sampling the Phong BRDF using Lafortune's algorithm. Output of a test program written during the course of this project.

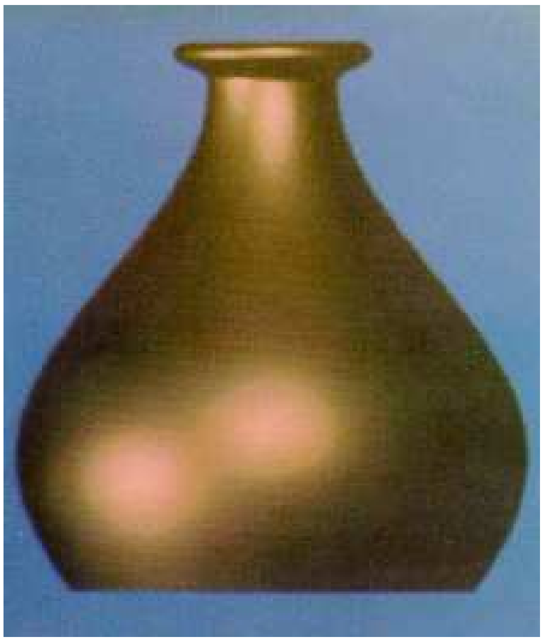

Figure 2.6: Copper vase rendered using the Cook-Torrance BRDF. Cook and Torrance[13]

model does not handle very well. The Cook-Torrance 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.

2.4.3 Oren and Nayar

In their paper "Generalization of Lambert's Reflectance Model"[14], Oren and Nayar introduced an accurate model for lambertian reflectors, which takes into account retro-reflection in rough clay-like diffuse surfaces.

2.4.4 Schlick's Model

A relatively simple but physically plausible, empirical reflectance model was introduced in 1993 by Christophe Schilck [15]. Its advantages are the high degree of parametrization with an intuitive set of parameters, and the fact that it's computationally cheaper, than full physically based reflectance models, while still obeying physical laws such as conservation of energy. Additionally most terms of the reflectance function can be importance sampled.

Chapter 3

Design

In this chapter we shall discuss the design of the s-ray renderer, and present the rationale behind the most important design decisions taken during the course of its development.

3.1 Choice of Algorithm

The photon mapping algorithm was chosen to implement global illumination.

The only practical alternative that could have been chosen instead is the radiosity algorithm. Radiosity, as was explained previously in the background chapter, is perfectly capable of calculating diffuse interreflections, but it lacks support for specular interactions, such as caustics.

It is possible to use ray tracing in conjunction with radiosity to add limited support for specular interactions to the radiosity algorithm, which is otherwise incapable to take them into account. First a radiosity solution is computed for the scene, storing direct and indirect diffuse illumination in the surface patches. Then a ray tracer can be used to render the scene, using the stored radiosity values as the diffuse component of illumination, and calculating only the specular part (including reflection and refraction).

The main drawback of radiosity, even paired with a raytracer, is the inability to render caustics, which is an important and very visually significant part of global illumination.

Another inherent drawback of the radiosity algorithm is that illumination sampling depends on the tesselation of the underlying geometry. Since lighting is computed per polygon essentially, low-tesselation areas will miss high-frequency lighting variations such as those at shadow edges. An adaptive subdivision technique[16] can be used to ensure proper mesh resolution wherever it is needed, which complicates the algorithm a lot, and increases the number of polygons considerably, which in turn increases rendering time.

In comparison, the photon mapping algorithm can easily simulate caustics, and it works independently of the resolution of the underlying geometry. In fact it doesn't event dictate a certain surface representation, it works equally well on polygon meshes, implicit surfaces, voxels, point sets, and really anything for which a ray-intersection routine can be defined.

Another alternative would be to use path tracing. Path tracing has the advantage that it's arguably the simplest global illumination algorithm and, like photon mapping, it calculates the full rendering equation. It does however converge very slowly, making it impractical.

3.2 Target Platform

The renderer should be able to run on any general purpose computer running a variant of UNIX, irrespective of endianess or word size. The choice of UNIX as the target operating system offers some considerable advantages.

UNIX is highly portable, with variants of UNIX running on everything from handheld devices to supercomputers. A well written program can in principle be recompiled on any variant of UNIX without, or with minimal, modifications.

Also most UNIX APIs are well designed, and more powerful than those provided by other operating systems. So for instance since the renderer uses the X window system for it's output window, and since X is network-transparent, it's possible to run the renderer on a remote computer and still be able to monitor the output image on the local display as it's being rendered.

Moreover, UNIX is widely emulated by compatibility layers on non-UNIX systems. Although no testing was done using that configuration, theoretically the renderer should be able to run under microsoft windows using the cygwin UNIX compatibility layer or microsoft's "UNIX services for windows" package.

In the end however, the choice of operating system was mostly arbitrary, based on personal preference and familiarity with the system.

3.3 Code Organization

The code structure has been designed with simplicity in mind. Object-oriented constructs where used wherever made sense, without abusing them to implement simple procedural concepts with classes.

For instance the 3D math library is a collection of classes for vectors, matrices, quaternions, etc. The renderable objects (spheres, meshes, cylinders) are all subclasses of the `Object` class, which is in turn a subclass of `XFormNode` which handles the hierarchical keyframe animation capabilities.

On the other hand, the image loader is a simple C module, exposing a `load_image` function. The thread pool implementation, XML parser, and window-system glue code also follow the same purely procedural design.

3.3.1 Dependencies

Three important parts of the code are not included in the project per-se, rather they are reusable libraries written by the author and maintained separately, all of them pre-dating this project by 3 to 9 years. These are the math library, the image file loading/saving library, and the kd-tree implementation. They are all available on public free software project hosting sites such as *sourceforge* and *googlecode*.

The renderer also uses the *expat* XML parser, *POSIX threads*, and *Xlib*.

3.4 Flexible Materials

There are two notable points about the design of materials in s-ray. First of all, material attributes are pairs of a value, and a texture map. The value part is a 3d vector, which can be used as color where convenient, or as a scalar by discarding its last three components. The map which is optional, if specified, modulates the value over the surface of the object.

This value–map pairing is performed internally and the texture modulation is performed auotmatically, such that for instance, when a shader function asks for the diffuse color, it automatically gets the diffuse color specified in the value part, modulated by the texture map if one is specified. This design provides the ability to render effects such as a partially frosted glass due to condensation, by just varying the specularity, reflectivity, transparency, and glossiness with a texture map over the surface of the object, without any special-cases in the shader.

The second interesting bit is the flexibility afforded by named material attributes. All attributes are looked up by name, so that each material will be able to provide all the parameters needed by the shader which uses it. So, for instance a material using a phong shader function, needs to define specularity and shininess, while a cook-torrance material will have a roughness attribute.

3.5 Parallelism

Ray tracing is one of those algorithms that can be characterized as "embarrassingly parallel". Each primary ray can be traced independently of its neighbors, making it trivial to split the rendering job among multiple processes or threads. For that reason, and since symmetric multiprocessors have become a comodity in recent years, with even the cheapest personal computers having at least two execution cores, it was decided from the start that it would be inexcusable to utilize only a single processor.

There are many ways to parallelize a ray tracer. The simplest would be to just split the image in n equal parts, where n is the number of processors, and spawn one thread for each part. Such a scheme would work perfectly if every part of the image required the same amount of processing, which sadly is far from true in a ray tracer. Consider for instance a landscape scene, being viewed roughly parallel to the ground, being rendered by a ray tracer that splits the image in half, spawning one thread for the top half and one for the bottom part. In that pathological case, almost all the rays on the top half of the image will escape to infinity without hitting anything at all, so the thread will finish its work in milliseconds. The bottom half of the image however contains essentially the whole scene, which means that all the rendering will be done by the bottom thread, utilizing only a single processor.

S-ray follows a slightly more complicated approach to parallelize rendering, which ensures an even distribution of work among processors, even when all of the geometry is concentrated in small areas of the image. The image is broken up into blocks of 64x64 pixels[1] and the blocks are added to a work queue. A fixed number of threads, equal to the number of processing units available are spawned from the beginning, and as soon as there is work in the queue they wake up, remove a block from the queue and start rendering it. This means that even if some of the blocks finish very quickly, the threads will keep busy as long as there are more blocks to be rendered.

This block-by-block rendering also helps cache coherency as most of the rays in the same block are liable to follow similar paths, accessing the same parts of the scene database.

3.6 Ray Intersection Acceleration

The most important operation in a ray tracer is undoubtedly the ray intersection routine. It is used continuously during rendering and for that reason it must be as efficient as posible. So, a most important question during

[1]That's actually the default block size, which can be overriden by the user.

the design of a ray tracer is how to organize the scene database in order to minimize intersection costs.

The naive algorithm would test every ray against every object of the scene, which is obviously very slow for anything but the simplest scenes, especially if we consider the fact that, for polygon meshes, all polygons of every mesh must be tested against each ray.

As is often the case with optimizations, the best way to accelerate ray intersections is to choose a better data structure. A linear list of objects simply does not provide enough information to test for intersections efficiently.

S-ray uses an octree to accelerate ray intersections. Each node of the octree has an axis-aligned bounding box and pointers to eight child nodes.

To build the scene octree, initially the bounding box of the whole scene is computed, which is assigned to the root of the tree. That box is then subdivided into 8 parts which become the root's child nodes. The process continues, splitting each node recursively, until the bounding box of each leaf node contains at most a fixed number of objects, or the maximum tree depth is reached.

Then, in order to find intersections, we start at the root of the tree and test if the ray intersects it. If it does, then we recursively test its eight children and so on until we reach a leaf node at which point all the objects contained in its bounding box are tested using the regular ray-object intersection routines. This way, if a ray does not intersect the bounding box of a subtree, the whole subtree is automatically discarded and all objects that are located only in that part of the scene are skipped.

Quickly finding which object is intersected by a ray however, is not enough for raytracing polygon meshes. To find the actual intersection point with the surface of the object, we must still test all the polygons for intersections. For that reason, s-ray builds a separate octree for the polygons of each object, computed in the object's local coordinate system. Using the local coordinate system of the object for its polygon octree means that it can be constructed once at the beginning, and doesn't have to be rebuilt for subsequent frames if the object moves. Of course this means that rays must be transformed with the inverse of the object's transformation matrix, to bring them into the object's local coordinate system for polygon intersection testing. That's not an issue however, since that is exactly how all the ray-object intersection tests are implemented anyway, in order to simplify the math and support arbitrary transformations for all types of objects.

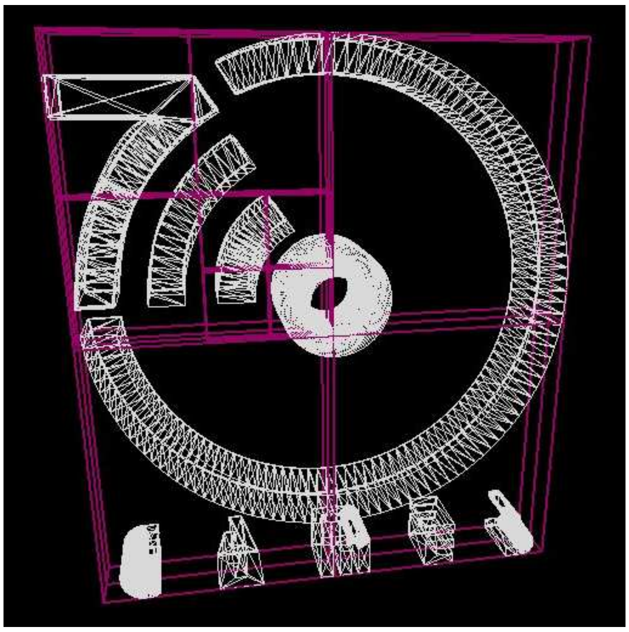

Figure 3.1: Scene octree visualization. The letters at the bottom are all part of a single mesh

.

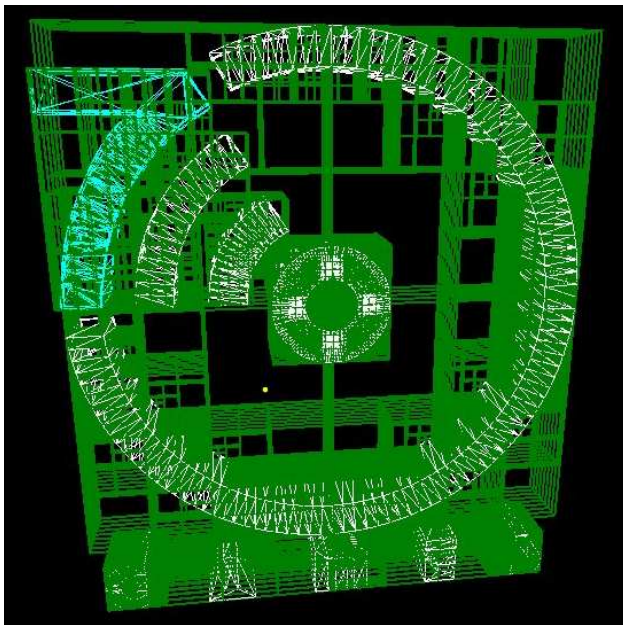

Figure 3.2: Visualization of the octrees of all mesh objects.

3.7 Scene Description Format

It was decided early on to use a custom XML file format for the input. XML is well suited for this task, because of the following reasons:

- It's easy to edit an XML file, even with a simple text editor. Indeed the most simple test scenes, such as the cornell box, used throughout the development of the s-ray renderer where typed in manually in a text editor.
- It's very simple to write converters from other 3D file formats, or exporters from 3D modelling programs that output an XML file. Indeed there are two such converters written as part of this project, included in the source distribution of s-ray: *obj2sray* and *3ds2sray*. They both act as command line filter programs, reading Alias—Wavefront OBJ files and Autodesk 3D studio files respectively from `stdin` and writing the corresponding s-ray XML scene description to `stdout`.
- It's easily processed by simple programs. One can easily envision a pipeline of programs generating geometry and doing various transformation on the input XML file, before it is fed to the renderer. The renderer is designed to accept scene descriptions from `stdin` to facilitate this usage. As a trivial example, counting the number of polygons in a scene can be done using the following simple pipeline:

  ```
  $ cat scene.xml | grep '<face' | wc -l
  ```

- It's self-documenting, making it very easy for someone to understand the file format by just glancing at a scene file.

The alternatives would be to use an existing 3D file format such as OBJ, or a scene description language such as the ones used by renderman or povray. These ideas where considered briefly, but where abandoned in favor of the custom XML format.

The problem with most existing 3D file formats, such as OBJ, 3DS, milkshape, ply, etc. is that they are not really suitable, as they lack features needed by a photorealistic ray tracer. None of them supports complex materials beyond the default phong model, and most of them describe only geometry whereas the renderer needs a file format describing lights and viewing parameters as well.

There's one existing format which could suit the needs of the renderer, called *collada*. It's a standardized XML scene description format, which includes all information needed by a ray tracer, and more. The problem with collada is that in an effort to be extremely generic and flexible, in order to support everything under the sun, it's quite bloated and cumbersome. It's hard to write a conformant collada loader and only a subset of the format would be useful for the needs of the s-ray renderer anyway. So, it was decided that it doesn't make sense to keep all that buggage in the renderer, and if collada support is desired in the future to increase interoperability with 3D modelling programs, a converter which reads collada XML and outputs the s-ray XML scene description would be simple to write.

Using a scene description language such as the renderman language, and to a lesser extend the pov-ray language sounds rather interesting. That idea however was discarded, since it would take a considerable amount of time to implement a complete programming language for the renderer, it would complicate the implementation considerably, and it would distract from the main objective.

3.7.1 Overview of the Scene Input Format

Let's take a quick look at the XML file format used by the s-ray renderer. Listing 3.1 shows a simple scene containing a textured floor, which is a mesh with two triangles, and a refractive sphere, lit by a point light.

The root XML element of the s-ray scene format is "scene", which may contain five different element types: "env", "material", "object", "light", and "camera".

The env element specified environmental parameters which are constant throughout the scene. Currently it can be used to specify an ambient color, background color (which is used when a ray escapes to infinity), and environmental index of refraction.

Material elements are always named, which is how objects refer to them through the "matref" element, and they contain a series of material attribute ("mattr") elements. Material attribute elements in turn have XML attributes stating their name, value, and optional texture map, as mentioned previously.

Object elements must specify the object type in the "type" attribute, and the parser then expects to find an appropriate sub-element, so for instance a sphere object will always have a sphere child element under the object element. Object sub-elements shared by all kinds of objects are: "xform" which defines a keyframe, and "matref" which binds a material to that object.

Mesh elements which are found inside object parent elements of type "mesh" contain essentially lists of vertices, normals, texture coordinates, tangent vectors, and faces, which refer to all of the above by their id. See listing 3.1 for more details.

Listing 3.1: Sample XML scene file format

```
<scene name="optional scene name">
    <env ambient="0.05 0.05 0.05" bgcolor="0 0 0" ior="1"/>
    <material name="sphmat" shader="phong">
        <mattr name="diffuse" value="0 0 0"/>
        <mattr name="specular" value="0.6 0.85 1"/>
        <mattr name="shininess" value="80"/>
        <mattr name="reflect" value="0.9"/>
        <mattr name="refract" value="1"/>
        <mattr name="ior" value="1.56"/>
    </material>
    <material name="floormat" shader="phong">
        <mattr name="diffuse" value="1 1 1" map="tiles.jpg"/>
    </material>
    <object name="sph1" type="sphere">
        <matref name="sphmat"/>
        <xform pos="0 1 0"/>
        <sphere rad="1"/>
    </object>
    <object name="ground" type="mesh">
        <matref name="floormat"/>
        <mesh>
            <vertex id="0" val="-30 0 -30"/>
            <vertex id="1" val="30 0 -30"/>
            <vertex id="2" val="30 0 30"/>
            <vertex id="3" val="-30 0 30"/>
            <normal id="0" val="0 1 0"/>
            <texcoord id="0" val="0 0"/>
            <texcoord id="1" val="4 0"/>
            <texcoord id="2" val="4 4"/>
            <texcoord id="3" val="0 4"/>
            <face id="0">
                <vref vertex="0" normal="0" texcoord="0"/>
                <vref vertex="1" normal="0" texcoord="1"/>
                <vref vertex="2" normal="0" texcoord="2"/>
            </face>
            <face id="1">
                <vref vertex="0" normal="0" texcoord="0"/>
                <vref vertex="2" normal="0" texcoord="2"/>
                <vref vertex="3" normal="0" texcoord="3"/>
            </face>
        </mesh>
    </object>
    <light type="point" color="1 1 1">
        <xform pos="-12 13 -10"/>
    </light>
    <camera type="target" shutter="0" fov="45">
        <xform pos="3 3 -5"/>
        <target><xform pos="0 0.5 0"/></target>
    </camera>
</scene>
```

The same principles apply to camera and light elements, so again refer to the example (listing 3.1) for details. Note that the xform tag is reused there, and can be used to specify multiple keyframes, essentially animating the light, camera or its target, just like it does for objects.

3.8 Scene Previewer

Due to the non-interactive nature of the ray tracer, it's hard to quickly verify the correctness of the scene, make adjustments to viewing parameters, verify that enough photons will be shot to achieve the required photon density in various parts of the scene, etc. For all those reasons, it was deemed necessary to provide an interactive scene preview tool along with the renderer.

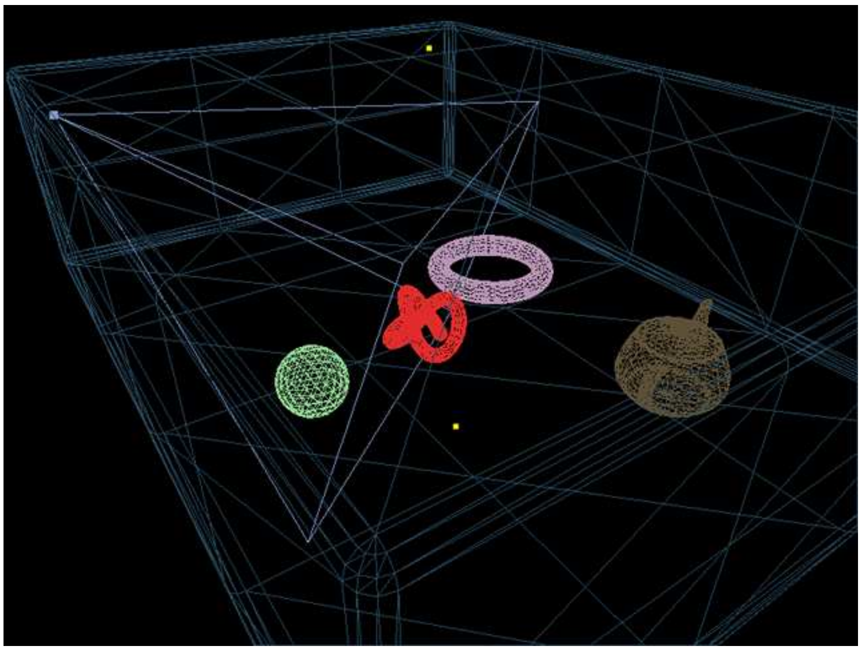

Figure 3.3: Interactive OpenGL scene previewer.

The scene previewer takes an XML scene file as input, and presents the user with a wireframe visualization of all objects, lights, and cameras. The user is allowed to move through the scene, and if desired a camera XML element can be written to stdout corresponding to the current view.

Photon visualization is also available. Photons are shot from all light sources in exactly the same way as the renderer does it, and they are visualized at the positions where they are stored in the photon map data structure.

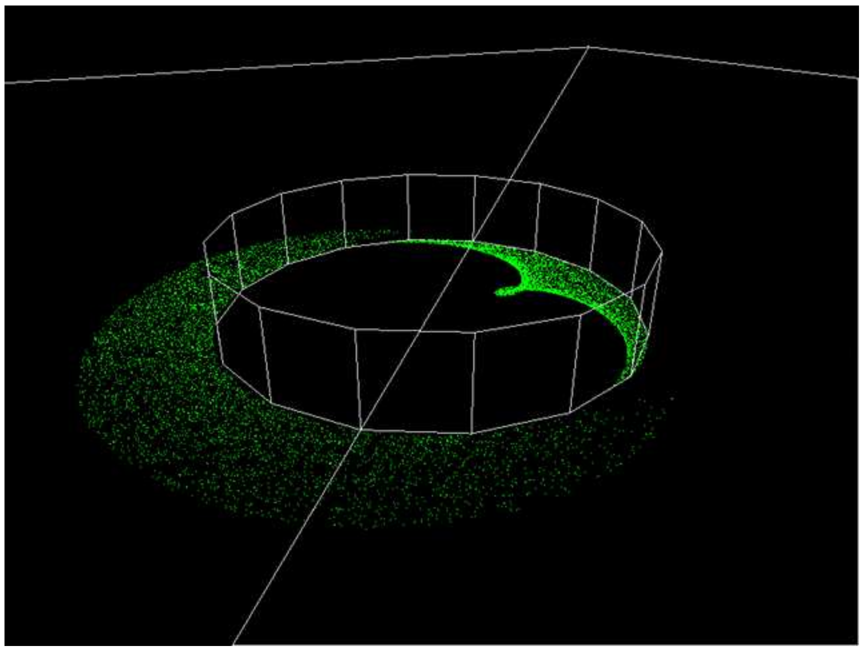

Figure 3.4: Photon visualization in the scene previewer.

The interactive previewer was also an invaluable debugging tool during the development of the renderer, as it was easily extended to visualize any aspect of the scene, such as octrees[2], photon projection maps, surface normals, etc.

3.9 User Interface

The s-ray renderer is a command line program, taking a scene description file as input and producing the rendered image, or a sequence of images when rendering an animation, as output.

The default mode of operation, is to present a window to the user, showing the image as it's been rendered, although the renderer can also be instructed to run non-interactively so that it may easily be used without requiring a display, say on a remote host through ssh.

The rationale behind making the program interactive is two-fold. First of all by showing the output bit by bit as it's being rendered, enables the user

[2]The octrees in figures 3.1 and 3.2 where visualized by the scene previewer

to quickly see if something is wrong with the rendering and cancel it, without having to wait for the whole image to finish. To further facilitate this usage, the renderer prioritizes blocks of the image near the center, and renders them first, since this is usually the most interesting area of the picture, progressing outwards towards blocks at the edge of the image, which are rendered last. Finally, the output of the renderer is a deep framebuffer with floating point high dynamic range color. The interactive window can be used to provide the user with tone mapping controls. Tone mapping is not implemented at the moment so no such user interface is currently exposed. Instead a fixed linear mapping is performed from floating point color values to the 24bit per pixel image shown to the user.

Chapter 4

Implementation

After discussing the major design issues and decisions, we shall continue with details about the implementation of the most notable algorithms used by the s-ray renderer.

4.1 Rendering Process Overview

Let's start with an overview of how the renderer operates in order to take the input scene file and transform it into a raster image. More details will follow on the most interesting parts.

During initialization, the command line options are parsed allowing the user to override any default setting. Such parameters as image size, number of photons tp use, rays per pixel, animation time interval to render, among many others can be specified by the user.

If the renderer is running in interactive mode, a connection to the X server is established, the rendering window is created, and event handlers are set up. See section 4.2 for more details.

The operating system is queried for the number of available processors, and the thread pool is initialized, creating an appropriate amount of worker threads.

Next, the renderer creates the scene database by reading the scene description XML from a file specified by the user through the command line, or from the standard input stream if no file was specified.

Before rendering the image, the scene and mesh octrees are constructed by the recursive space subdivision algorithm described in the previous chapter.

If photon mapping is to be used for global illumination or caustics, the photon maps are then poppulated by shooting photons from all light sources as described in section 4.5.2.

The frame is broken up into blocks, assigned a priority based on their distance from the center of the image, and passed to the worker threads for rendering, as described in detail in section 4.3.

After rendering finishes, if an animation is being rendered, the frame is saved as a PNG file in the current directory, and we proceed to render the next frame. Otherwise, the program just waits for the user to press escape to exit. At this point the user may also save the image, or dump the photon maps to a file so that they might be reused.

4.2 Rendering Window

As discussed previously, the X window system is used to present the user with a window showing the output image as it's been rendered, and to allow the user to interact with the renderer.

Initially, a connection with the X server is established, and availability of the X shared memory extension is determined. The standard `DISPLAY` environment variable is can be set by the user to instruct the program to connect to any local or remote X display.

When the renderer is connected to an X display on the local host, the X shared memory extension is used for efficient updates of the rendering window. This basically works by mapping a memory segment to the address space of both the renderer and the X server through the SystemV IPC API. Then updating the rendering window when a block has finished rendering is a simple matter of copying the pixels into the shared memory and sending an `XshmPutImage` request to let the X server know that a part of the window must be repainted by blitting parts of the shared image into it.

When connected to a remote X display, the X server will respond that Xshm is not supported, and the renderer will fall back to updating the window using the slower traditional `XPutImage` method, which sends the image data through the socket to the X server.

4.3 Thread Pool

Durign startup, the renderer creates a number of worker threads, as many as there are processors in the system.

When there's no work to be done, all the worker threads are blocked waiting on a *condvar*, consuming no resources.

Rendering is initiated by calling `start_frame`, which creates an, optionally sorted by priority, linked list of blocks that have to be rendered, and appends them on the work queue, at the same time broadcasting a wakeup signal to all threads waiting on the condvar.

As soon as a worker thread is woken up, it enters a loop where it tries to grab a block from the work queue and render it, until there are no more blocks left in the work queue, at which point the thread goes back to sleep.

When rendering of any block is completed, the main thread must be notified in order to update the image. A typical self-pipe trick is used to let the rendering thread asynchronously notify the main thread that a block is finished. A pipe is created during initialization, the write end of which is used by the rendering threads to send a single byte when a block is finished. The main thread is blocked in a call to select, waiting for data from the file descriptor of the read end of the pipe, and for events from the X server socket. When a byte is read from the pipe, the main loop wakes up and updates the rendering window.

4.4 Caching and Lazy Evaluation

A few parts of the ray tracer use lazy evaluation or perform expensive calculations only when and if they are needed, caching the results and using them as long as they remain “valid”.

4.4.1 Animation

Specifically, the animation system exposes a set of functions in the transformation node class, to retrieve the aggregate node transformation matrix, and its inverse, including hierarchical transformation inheritance and interpolation of all keyframe tracks, for a specific time value. The resulting matrices are calculated on demand and cached, as long as the same time value is used in further requests, and if there haven't been any modifications to the keyframe tracks, the cached matrices are returned immediately without recalculation.

4.4.2 Bounding Box Determination

A similar approach is employed in the calculation of bounding boxes for the various objects of the scene. A `get_bounds` function is exposed by the object class, which lazily calculates the axis-aligned bounding box of the object and caches the result. Note that the bounding box of an object is also time-dependent since it must be defined in world space.

4.4.3 DataCache and Multithreading

The multithreading nature of the renderer intoduces concurrency issues in the management of the aforementioned cached data. It was deemed necessary to avoid the overhead incurred by using locking for every access to the cached data, for which reason a slightly more complicated cache management scheme was introduced.

The `DataCache` template class is responsible for maintaining an arbitrary data item, along with a validity verification key, *per thread*. The thread id is used to lookup into a hash table and retrieve the cached data item, using the key value to verify if the cached data are valid or stale.

For example in the matrix caching case described above, a data cache holding a pair of matrices (regular and inverse) along with a timestamp, is kept per transformation node. When a matrix is requested, the requested time value is compared with the timestamp in the cache, and the decision is made to use the cached data or recalculate them. All of this can be done without any locking as it operates on thread-specific data.

4.4.4 Adaptive Multisampling

The user is free to specify a fixed number of rays per pixel, or a range with a minimum an maximum value. In the second case the renderer determines how many rays to use per pixel by calculating the variance between the rays already shot, and comparing it with a user-defined variance threshold. When the variance drops below the threshold, the process stops.

A histogram of the number of samples actually used per pixel can be written to the disk by pressing the ‘h’ key at any time during rendering.

Figure 4.1 shows the sampling histogram image produced during the rendering of the image in figure 5.1. The number of samples are visualized using the red channel of the image.

By examining this histogram visualization it is readily apparent that a lot of rays are used to sample parts of the picture with sharp color variations, such as object silhouettes and shadow boundaries, and few are used for other more smoothly varying parts of the image.

4.5 Photon Mapping

The photon mapping is used by the s-ray renderer to simulate two different aspects of global illumination: diffuse indirect illumination, and caustics. Now we shall discuss in detail how both of those are implemented in s-ray.

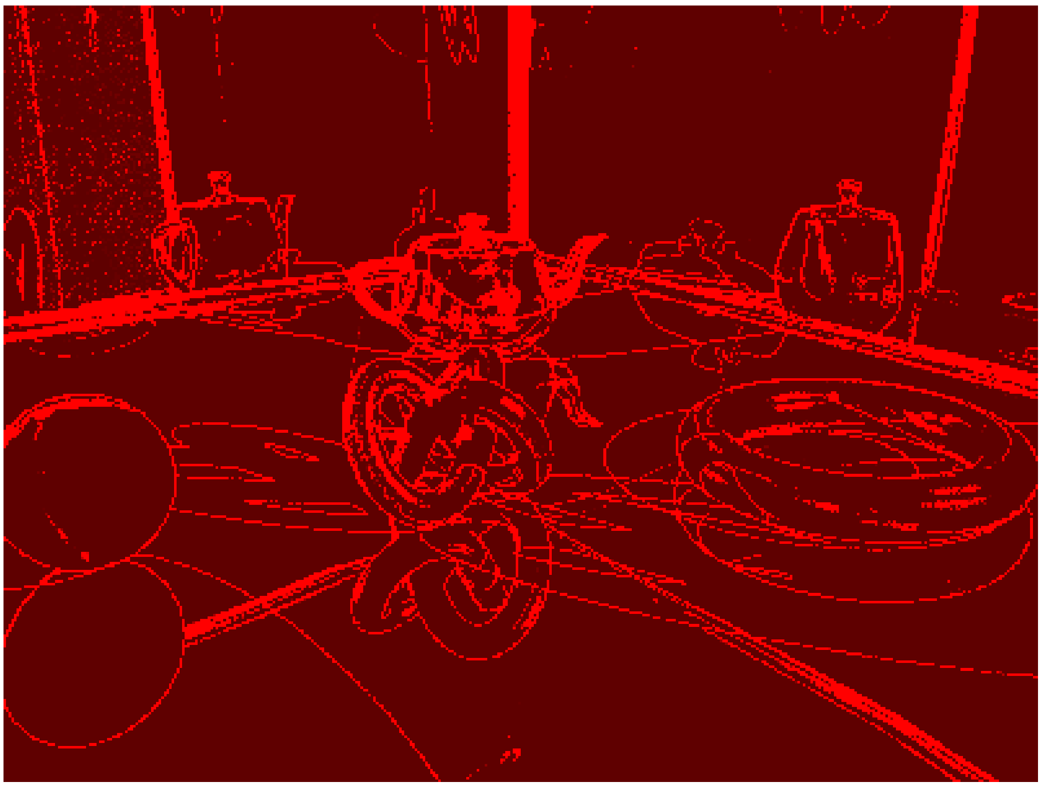

Figure 4.1: Sampling histogram visualization. More rays are used to sample parts of the image with sharp color variations.

Photon mapping works by first shooting photons from all light sources, and tracing their paths through the environment, until they are absorbed at which point they are stored in the photon map. The stored photons are then used during the ray tracing stage to evaluate the global illumination terms of the rendering equation.

Two separate photon maps are used for each scene: the caustics photon map and the global photon map. In the caustics photon map we only store photons that arrive after a specular interaction such as reflection or refraction, while in the global photon map we store all photons.

4.5.1 Photon Map Data Structure

It is obvious then, that the photon map data structure plays a very important role in the whole process. The renderer needs to be able to locate stored photons by their proximity to ray-object intersection points, and so the data structure used by the photon map must support fast nearest-neighbor queries.

S-ray uses a kd-tree for the storage of photons, as suggested by Jensen. A kd-tree is essentially a multidimensional binary search tree [17] where each level of the tree uses a different axis to split the space. So for instance the root node might use the x coordinates to split nodes into its left and right subtrees, the next level down might use y, then z, then x again, and so on.

Nearest neighbor queries in a kd-tree exhibit logarithmic average complexity, making the kd-tree a good choice for the photon map data structure. Jensen proposes the use of balanced kd-trees, to guarantee logarithmic complexity and avoid pathological worst cases of linear complexity in completely skewed trees. However, such worst case scenarios aren't likely due to the random nature of photon storage, so kd-tree balancing isn't implemented in s-ray at the moment.

4.5.2 Photon Shooting

The number of photons to be shot by each light source is calculated by dividing the total number of photons among all the light sources, weighted by their relative intensities. It is important to note that since brighter lights cast more photons according to their intensity, the individual photon power doesn't have to be modified, and all photons carry the same amount of energy.

Projection Map

Shooting photons towards empty space is wasteful, especially in sparse environments. Similarly shooting caustics photons towards parts of the scene that do not contain specular objects is also wasteful.

In order to concentrate photons towards interesting parts of the scene, a projection map is used. The possible directions around each light source is discretized into a number of cells. Each cell is marked as full if it corresponds to directions that can reach geometry, or empty otherwise. Then, during photon shooting, random directions are picked from within the full cells, thus avoiding shooting a lot of photons towards empty space.

The energy of each photon must be scaled by fraction of the full sphere of directions used for picking photon directions, to account for the energy that would otherwise be lost, carried by photons that fly out to empty space.

Two projection maps are calculated for each light source, one for shooting caustics photons towards specular objects, and one for shooting global photons towards all kinds of objects.

Figure 4.2 shows a visualization of the caustics projection map, used to concentrate caustic photons towards directions where they might reach the

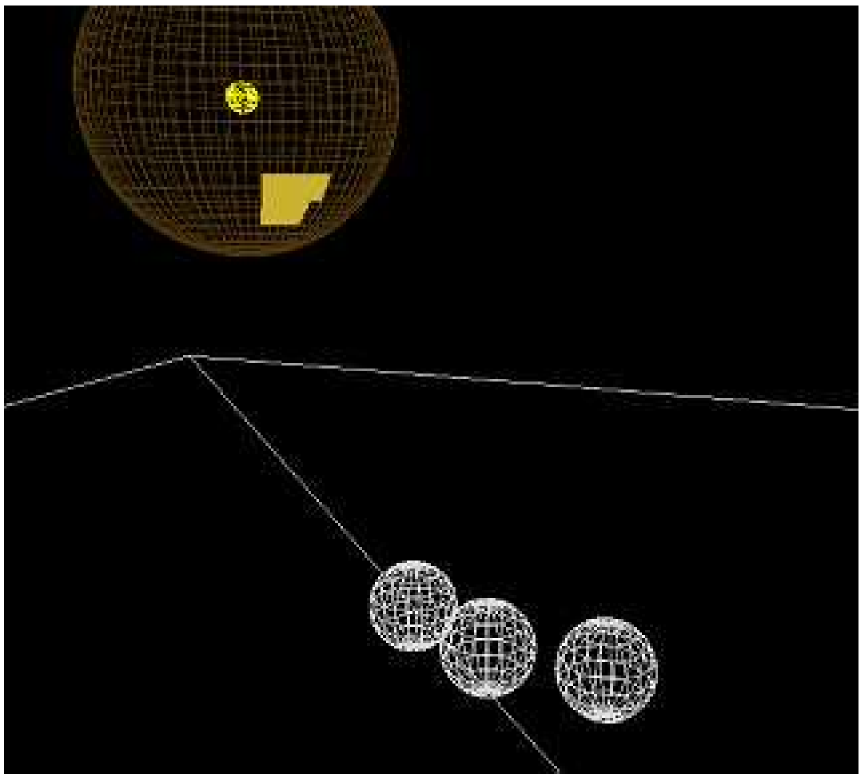

Figure 4.2: Projection map visualized as a sphere made out of full and empty direction cells around the light source

three refractive balls. The filled yellow polygons on the surface of the big sphere denote full cells, while the rest of the sphere is made up of empty cells, shown in wireframe. It is apparent from the visualization that for this scene, only a very small percentage of the total number of photons would have a chance to reach the refractive balls, if the projection map wasn't used. The visualization was produced by the scene previewer.

4.5.3 Photon Tracing

After leaving the light source, photons must be traced as they interact with the environment. At each intersection of a photon with the geometry, a decision must be taken on what to do next. The process is slightly different between caustics photons and global photons so let's examine them separately.

Tracing Caustics Photons

When a caustics photon hits a non-specular object, its path is terminated and it is stored in the photon map. For specular objects, the photon could either be reflected, refracted or absorbed (stored in the photon map).

One possibility would by to spawn new photons for each case, with energy scaled by the reflection or refraction factors of the material. But that would be wasteful, as we would end up tracing photons carrying very little energy, and thus visually insignificant.

A much better way proposed by Jensen is to use russian roulette to pick one of the possible outcomes. Essentially what this means is that we use the material factors as probabilities for each outcome. So for instance a photon hitting an object with reflectivity of 0.5, has 50% chance of being reflected, with its full energy.

The advantage of russian roulette is that we concentrate on important photons, and also that all photons in the photon map end up with similar energy levels, something that helps the accuracy of the density estimates calculated during rendering.

Tracing Global Photons

Global photon tracing follows the same general idea, with the difference that the possible outcomes after an intersection are different.

First of all russian roulette is employed to choose between the possibilities of diffuse interaction, specular interaction, or absorption. The probabilities of diffuse and specular interaction are calculated by the average diffuse and specular colors respectively.

Selecting absorption terminates the recursion and the photon is stored in the photon map. Otherwise, if diffuse interaction is picked, a random direction for diffuse scattering is chosen on the hemisphere above the surface, in which case the power of the photon is multiplied by the lambertian BRDF (the dot product of the incident photon direction with the normal). Finally if specular interaction is picked, another russian roulette is employed to determine whether the photon will be reflected or refracted.

4.5.4 Caustics

In order to render caustics, we must be able to calculate the irradiance due to reflected or refracted light at every point of the scene, tha is, the amount of photon energy that reaches any given point after a specular interaction.

Using the caustics photon map, it's easy to estimate the irradiance at any given point by calculating the density of the photon energy in the neighborhood of that point. Nearby photons inside a given radius are gathered by searching through the kd-tree, and a the density is calculated by dividing the accumulated photon energy of all the gathered photons by the area of the gather disc.

The value from the irradiance estimate is added directly to the rest of the illumination computed by the renderer at that point, which results in bright areas wherever a lot of light is arriving due to specular interactions.

4.5.5 Diffuse Global Illumination

A similar approach could be taken, for diffuse global illumination by directly estimating radiance due to diffuse scattering at any given point from the global photon map, which in that case should only contain photons coming from diffuse scattering only. That doesn't work very well however, because in order to get a good radiance estimate from the photon map, a huge amount of photons would be required, otherwise the result looks splotchy.

Figure 4.3 shows the first attempt to implement diffuse global illumination by direct radiance estimation from the global photon map.

Final Gather

A much better technique, which is used by s-ray, is called "final gather". Two methods are defined for calculating radiance at any given point: an accurate calculation, and a rough estimation.

The accurate method is used when we are calculating illumination for points directly visible, or seen through reflection and refraction. It works by calculating direct illumination as usual, by sampling light sources with shadow rays, then casting reflection or refraction secondary rays if applicable, estimating irradiance from the caustics photon map as described previously, and finally spawning a number of rays randomly distributed on the hemisphere above the surface, to sample incoming indirect illumination from other objects.

The inaccurate method is used only when have gone through a diffuse indirect illumination ray, and it consists of estimating radiance directly from the global photon map.

So essentially we calculate illumination as usual, with the addition of gathering light from nearby surfaces by spawning a number of rays which doesn't recurse further but use the photon map to estimate radiance wherever they hit.

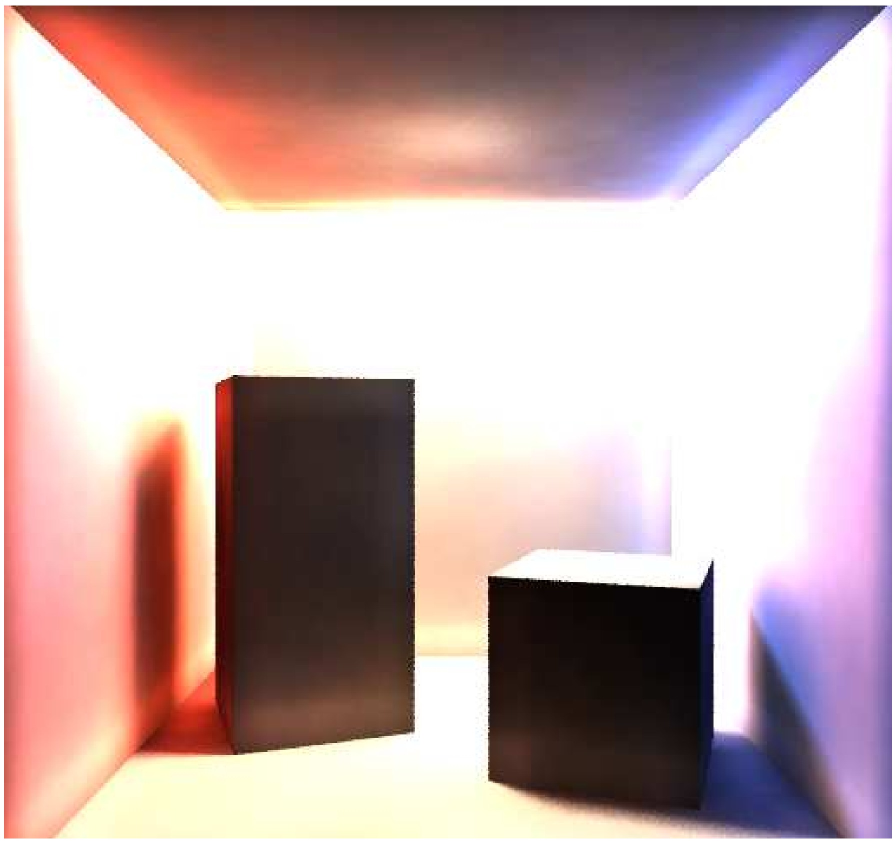
Figure 4.3: Initial attempt at global diffuse illumination by directly estimating radiance from the global photon map

Chapter 5

Results

The best way to communicate the results of a 3D rendering project, is naturally through a collection of output images produced by the renderer.

Before going into the most interesting part of the results, which is the global illumination renderings, let's first take a look at the less important features.

5.1 Whitted Ray Tracing

During the early stages of development, while implementing and optimizing mesh rendering, a simple test scene containing various reflective polygon objects was created in 3D studio max. To verify the correctness of the renderer, the scene was also rendered by the internal ray tracer of 3D studio max in order to compare it with the output of s-ray. Figure 5.1 shows the two images next to each other. Slight differences in specular highlights are due to the fact that s-ray uses the original Phong model, while 3D studio uses the Blinn-Phong variant.

5.2 Animation and Motion Blur

As previously indicated, the s-ray renderer implements distribution ray tracing as described by Cook et al.[1]. Figure 5.2 shows off the use of distribution ray tracing to render motion blur. Rays are distributed during the time interval that the virtual shutter remains open, to produce the characteristic streaking of fast-moving objects, like the red sphere in this picture which is rotating fast around the blue cylinder. The renderer's keyframe animation capabilities where used to define the motion of the red ball.

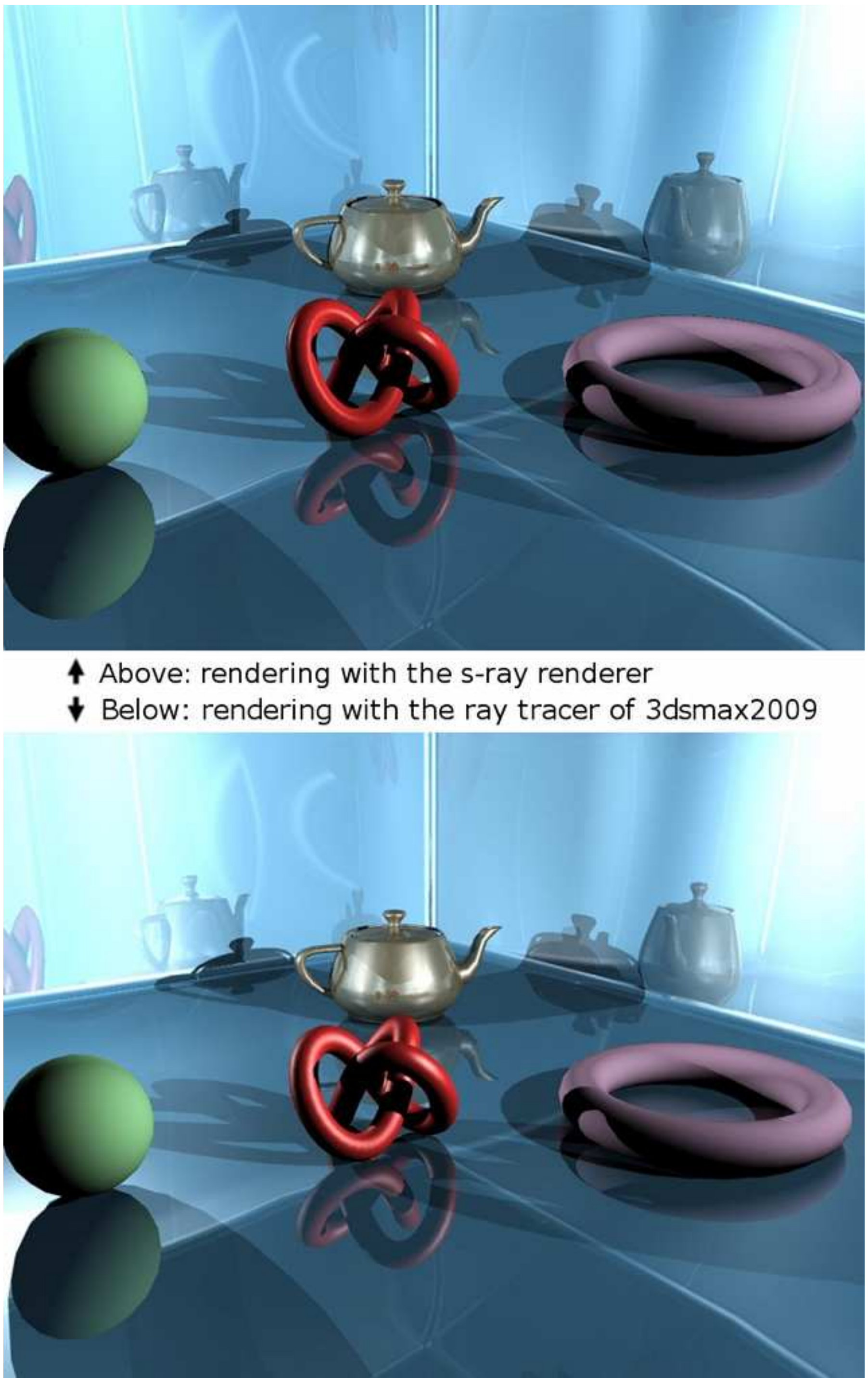
↑ Above: rendering with the s-ray renderer
↓ Below: rendering with the ray tracer of 3dsmax2009

Figure 5.1: Comparison between s-ray and the 3D studio ray tracer. Specular highlights differ slightly due to different reflectance model used.

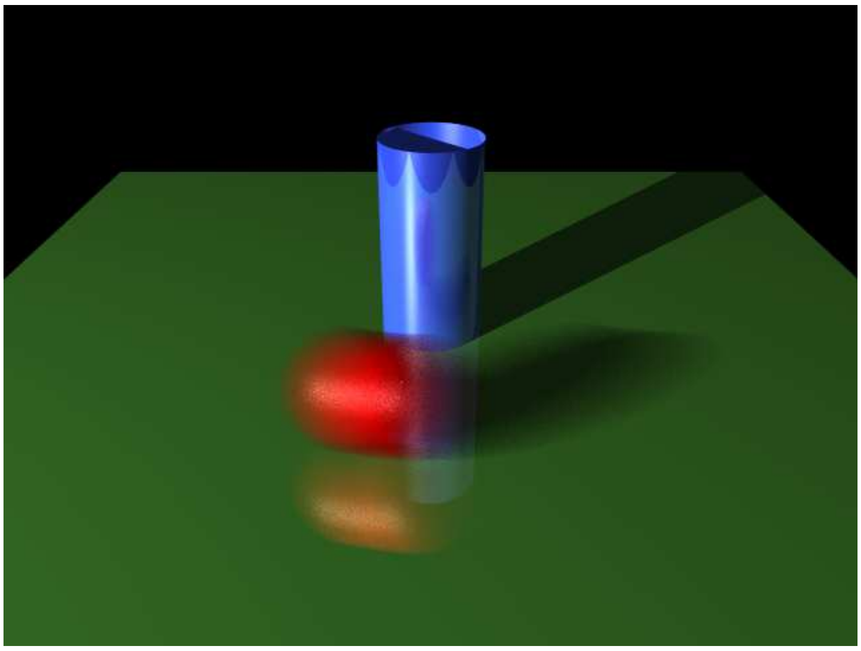

Figure 5.2: Motion blur

5.3 Caustics

Caustics rendering was a major goal of this project, as they are a visually significant part of global illumination. The following images demonstrate the capability of the s-ray renderer to produce caustics.

Figure 5.3 shows the caustics produced due to light being focused onto the floor by refraction through three colored glass spheres. Not very easily discernible in this low-resolution version is the fresnel effect, which increases reflectivity near the edges of the glass balls. Finally, also noticable is the penumbra regions in the shadows, due to monte carlo sampling of the large spherical area light used to illuminate the scene.

Figure 5.4 demonstrates reflective caustics formed by light being reflected from a shiny copper ring onto a wooden table. Notice the characteristic cardioid shape produced by many cylindrical reflective surfaces, such as the inside surface of a coffee cup.

Figure 5.5 shows caustics being formed by light transmitted through a complex glass object, such as the stanford bunny.

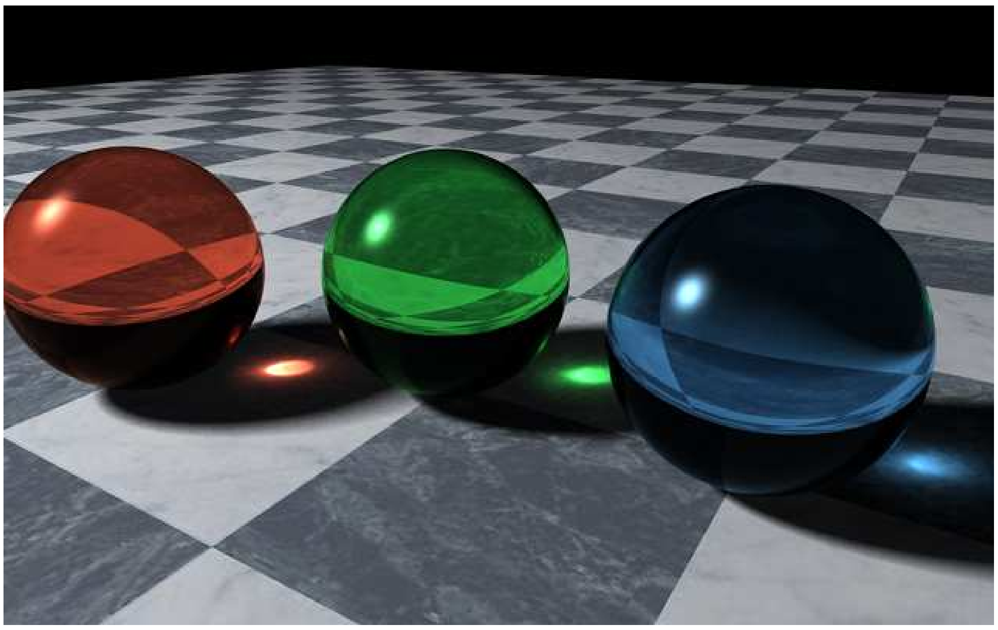

Figure 5.3: Refractive Caustics

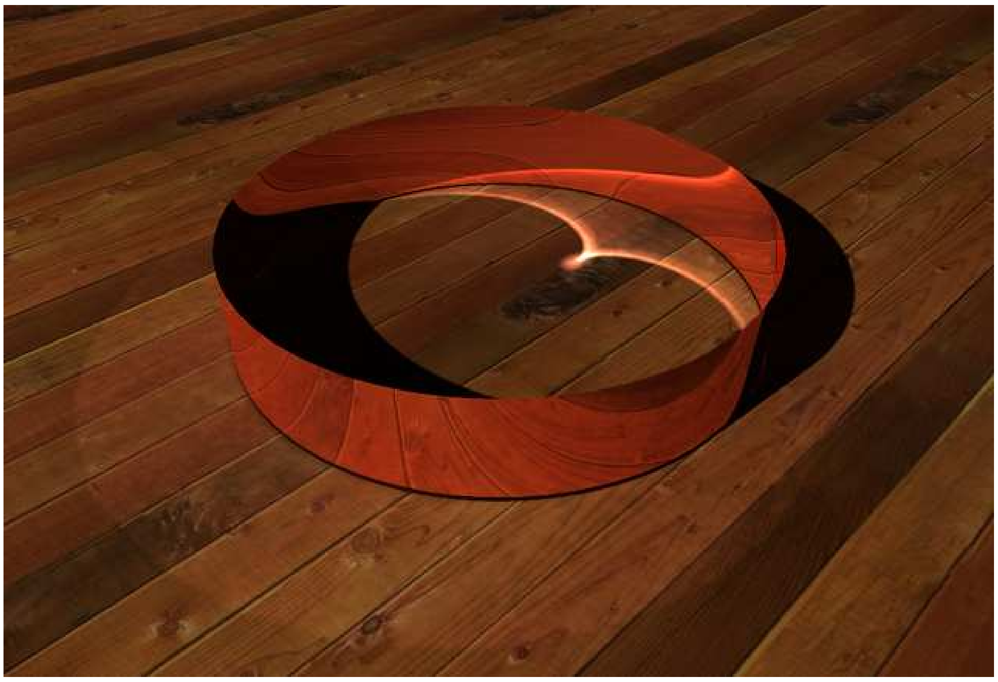

Figure 5.4: Caustics formed by light being reflected from a copper ring onto a wooden table.

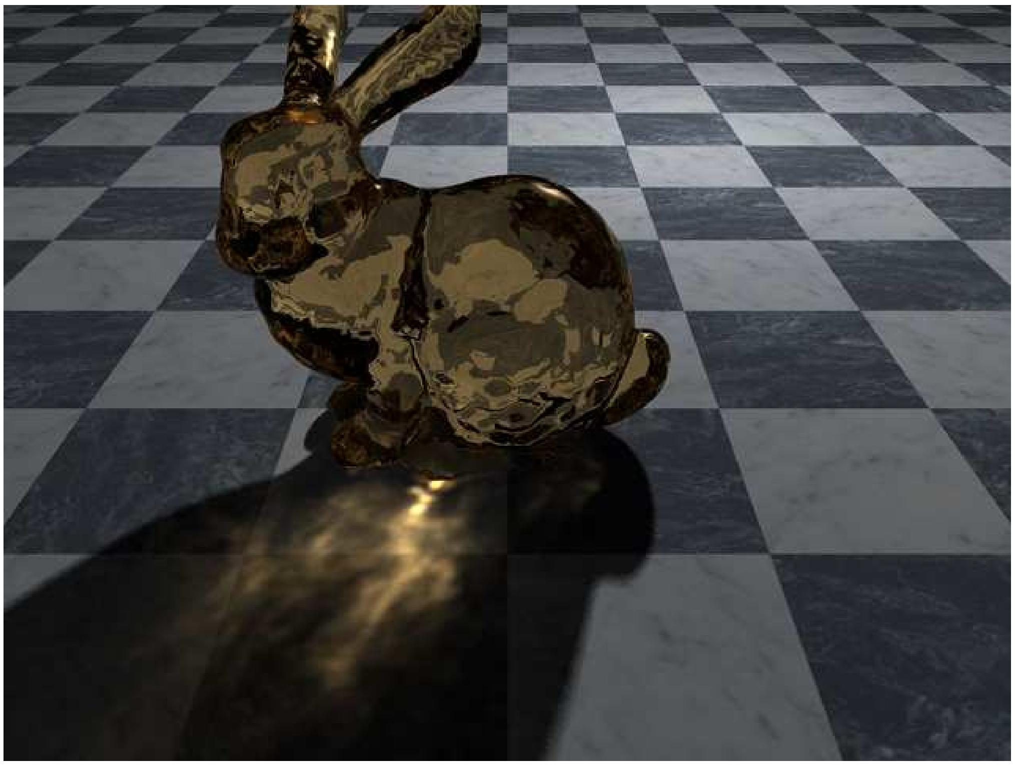

Figure 5.5: Caustics due to light transmitted through a glass stanford bunny. Model source: Stanford University Computer Graphics Laboratory.

5.4 Diffuse Global Illumination

The ability to simulating diffuse indirect lighting was also a major goal of this project. The following images demonstrate the diffuse global illumination capabilities of the s-ray renderer.

The cornell box is a good test scene to demonstrate the color bleeding effect occuring by light arriving to a surface after being diffusely scattered by another colored surface. Figure 5.6 is a global illumination rendering of the cornell box, using final gather. Notice the intense color bleeding from the walls to the ceiling and the, and the sides of the two boxes facing the walls. The light intensity was turned up in this picture to make color bleeding clearly visible.

There is some high-frequency variance visible in this image. This is due to the per-pixel monte carlo diffuse sampling of the hemisphere above each primary ray intersection point, in order to gather indirect illumination from the global photon map. The effect is aggravated due to the high light intensity

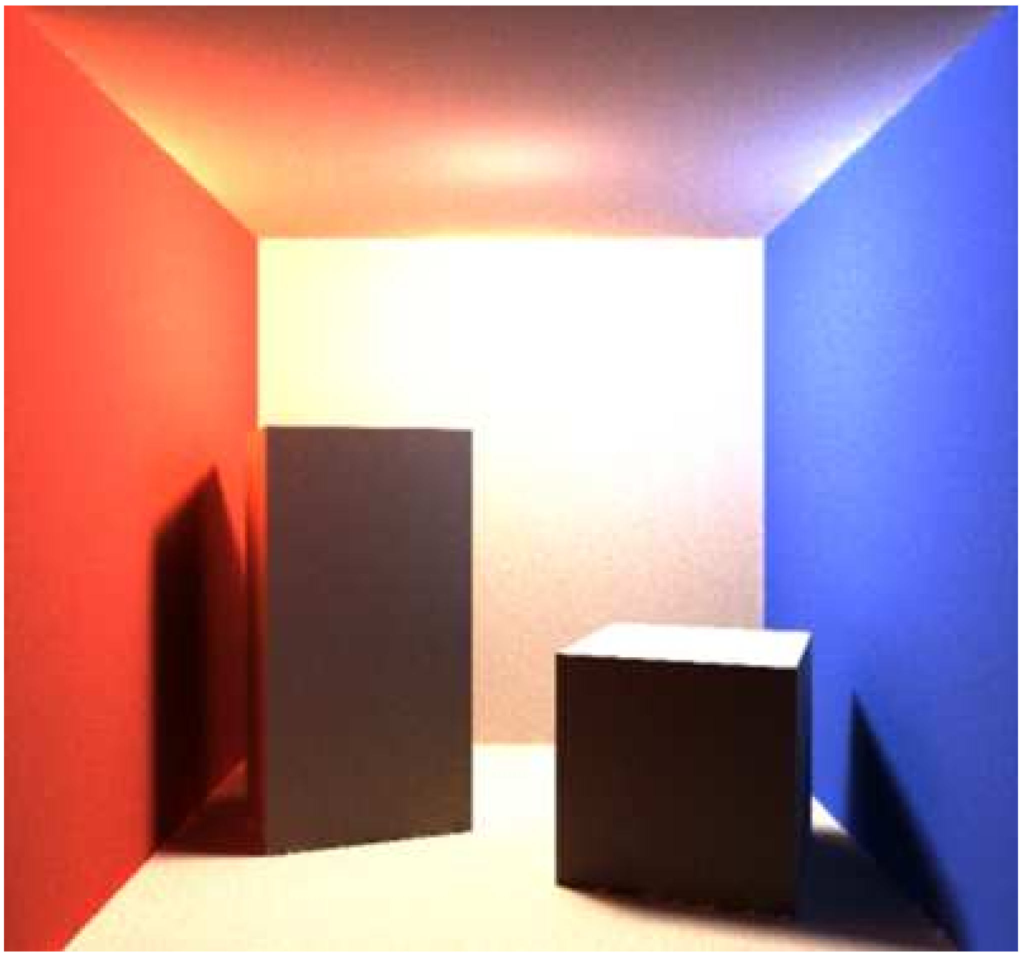

Figure 5.6: Color bleeding due to diffuse interreflections in the Cornell box.

used to to make color bleeding more apparent.

Global illumination in a more complex environment is demonstrated in figure 5.7. In this rendering of the Sponza atrium model, illumination is provided by a huge rectangular area light at the opening at the top of the model. Notice how light arrives and indirectly illuminates the ceiling and the walls visible through the pillars, as well as under the pillar heads and arches.

A rendering of the same environment using direct illumination only is provided for comparison (figure 5.8). Notice how dark and underlit the scene looks without the global illumination component.

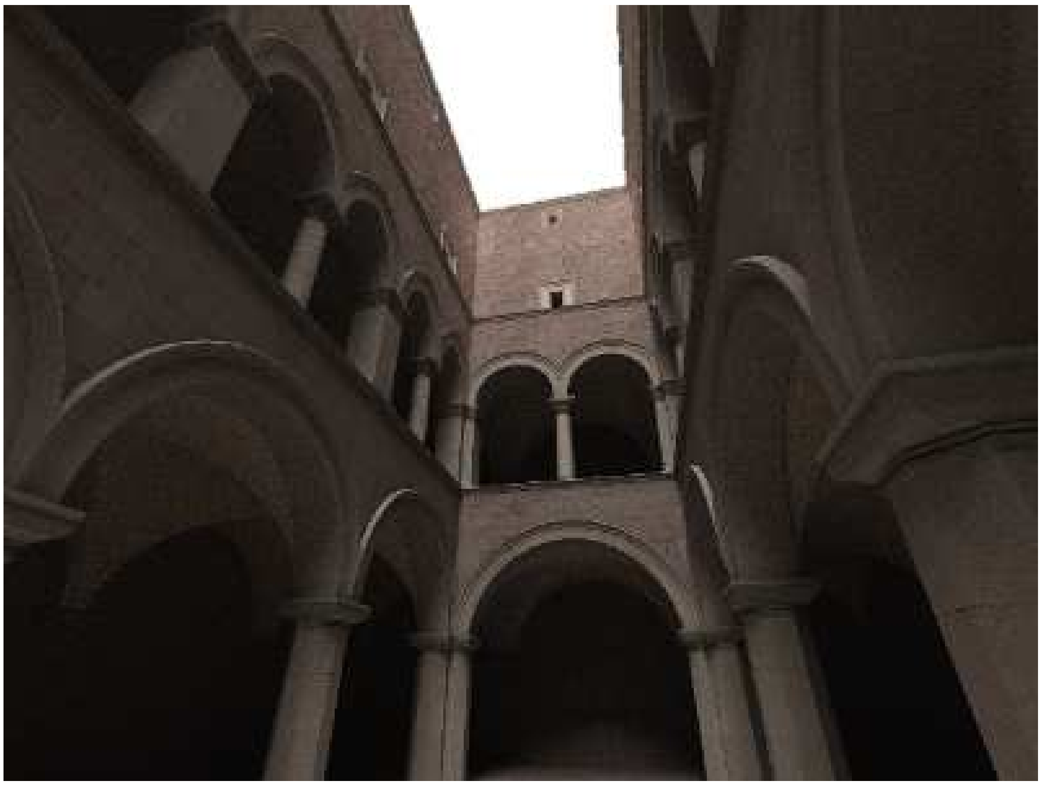

Figure 5.7: Global illumination rendering of a complex environment. Sponza atrium modelled by Marko Dabrovic.

.

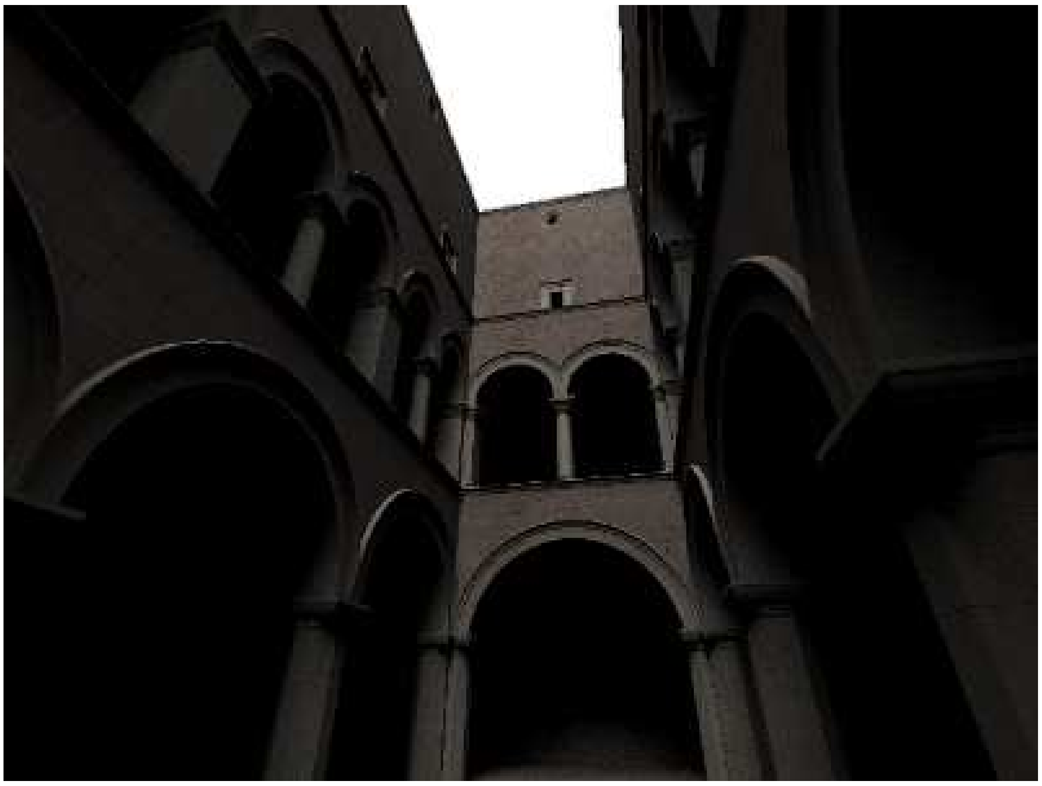

Figure 5.8: Direct illumination only rendering of the same model for comparison.

Chapter 6

Conclusions and Critical Analysis

The s-ray renderer is a reasonably efficient photorealistic renderer, capable of simulating all `L(D|S)*E` light paths, including indirect diffuse illumination and caustics, which was the original goal of the project.

Furthermore, the renderer supports a good range of features such as: full use of symmetric multiprocessing systems, efficient rendering of arbitrary polygonal meshes, adaptive anti-aliasing, extended animation capabilities, motion blur, soft shadows, etc.

6.1 Limitations

Still, it doesn't mean that the renderer is perfect. Performance is one thing that could be improved a lot, especially during final gather.

6.1.1 Irradiance Caching

A big performance benefit can be obtained by not using final gather to calculate indirect diffuse illumination everywhere, but instead caching irradiance values on diffuse surfaces and re-using them by interpolating between cached values wherever possible.

The interpolation between cached irradiance values would also help reduce the high-frequency noise due to per-pixel monte carlo evaluation of indirect diffuse illumination, thus improving image quality.

6.1.2 Importance Sampling using the Photon Map

Another important optimization would be to use the photon map, during final gather, to find the directions where most photons are coming from, and concentrate the diffuse samples towards those directions.

6.1.3 Photon Filtering

Caustic quality can be improved by weighing the contribution of each photon in the irradiance estimate by its proximity to the point of evaluation. This would reduce blurriness, and make it possible to get crisp caustics with even fewer photons in the photon map, and larger gather distance.

6.2 Future Work

Future work on the s-ray renderer, should initially concentrate on performance optimizations, such as the ones mentioned in the previous section. Especially implementing irradiance caching will probably bring considerable performance gains to the renderer.

Furthermore, it would be beneficial to explore alternative space subdivision techniques for ray intersection acceleration. Especially using a kd-tree for space subdivision, with the surface area heuristic[18] for deciding which splitting plane to use for each node, seem very promising for improving the overall performance of the ray tracer.

On the features front, the first thing to add would be an implementation of tone mapping, for proper conversion of the internal floating point high-dynamic range framebuffer to the low range 8bit per color component image presented to the user.

Also, it would be a good idea to get rid of arbitrary light intensity and photon power settings, and use standard CIE illuminants instead. That would remove the trial & error tuning of light parameters, and allow the user to just select the appropriate illuminant and let the renderer figure out the rest.

Finally it more interesting features could be added such as more complex reflectance models, or even a general shader plugin system for letting the user write custom shader programs in the form of dynamically loaded libraries.

Bibliography

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

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

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

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

[7] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, "Modeling the interaction of light between diffuse surfaces," in *Proceedings of SIGGRAPH 1984*, pp. 213–222, ACM, 1984.

[8] M. F. Cohen and D. P. Greenberg, "The hemi-cube: a radiosity solution for complex environments," in *Proceedings of SIGGRAPH 1985*, pp. 31–40, ACM, 1985.

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

[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] E. P. Lafortune and Y. D. Willems, “Using the Modified Phong BRDF for Physically Based Rendering,” Technical Report CW197, Department of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium, Nov. 1994.

[13] 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.

[14] 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.

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

[16] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg, “A progressive refinement approach to fast radiosity image generation,” *SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 75–84, 1988.

[17] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[18] V. Havran, *Heuristic Ray Shooting Algorithms.* PhD thesis, Czech Technical University, 2000.