All Theses and Dissertations

2010-01-28

Graph-based Global Illumination

Brian C. Ricks
Brigham Young University - Provo

www.manaraa.com

Graph-based Global Illumination

Brian C. Ricks

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Parris Egbert, Chair
Michael Jones
Eric Ringger

Department of Computer Science

Brigham Young University

April 2010

ABSTRACT

Graph-based Global Illumination

Brian C. Ricks

Department of Computer Science

Master of Science

The slow render times of global illumination algorithms make them impractical in most commercial and academic settings. We propose a novel framework for calculating the computational complexity of global illumination algorithms and show that no other recent improvements have reduced this complexity. We further show that many algorithms use a tree as their rendering paradigm. We propose a new rendering algorithm, pipe casting, which calculates light paths using a graph instead of a tree. Pipe casting significantly reduces both computational complexity and actual render time of rendering. Using an L2 pixel-wise error comparison, on average our algorithm can render a variety of scenes at the same error as traditional algorithms but in about 50% of the time.

ACKNOWLEDGMENTS

Contents

Chapter 1

Introduction

Figure 1.1: Global illumination algorithms can generate photorealistic images such as these ones generated with our new algorithm, pipe casting. Our algorithm reduces the computational complexity of such algorithms for improved render times.

All of modern society is affected by the revolution in computer graphics and rendering algorithms. In industry, design experts use computer graphics to visualize prior to production. Computer rendered simulations used in classrooms enhance comprehension. The demand for quicker and better graphics is a driving force behind entertainment sales. Architects make more informed decisions as they visualize their work before any construction begins. All of these areas benefit from better images and faster rendering algorithms.

1

The demand for improvement in computer graphics and rendering algorithms has created a large research area. This paper describes our contribution to this research area by, first, demonstrating mathematically that previous rendering algorithms have a high computational complexity that contributes to their slow execution times and, second, explaining our new algorithm that reduces rendering times.

1.1 Global Illumination

Research in the area of graphics is expansive and covers a variety of different topics. Our contribution is in the area of global illumination rendering algorithms. To better understand the contribution of this work, we first explain how global illumination fits in with the other graphics research.

The process of producing computer generated footage and images has three major steps. The first step is the creation of scenes. A scene might include meshes that define object surfaces, textures that define surface colors, an assortment of lights, and a camera. The second step is animation, in which objects, cameras, and lights move and change. The third step is rendering, in which a rendering algorithm takes the frame definitions and produces believable images. Our algorithm focuses on improving this third step of rendering by enhancing the speed of the rendering process.

There are two major divisions in rendering algorithms. The first set of algorithms is real-time rendering, which trades accuracy for speed. Real-time rendering algorithms are popular in the video game industry where an interactive frame rate is more important than the accuracy of each image. The second set of algorithms have as their goal photorealistic rendering. In this case, we are willing to sacrifice real-time frame rates to achieve a high level of realism. Photorealistic rendering, this second set of algorithms, is the focus of this work.

One of the most powerful set of photorealistic algorithms is called global illumination. The algorithm we propose is a global illumination algorithm. Global illumination algo-

rithms produce photorealistic images by using a precise understanding of how light travels through an environment and interacts with the objects in the scene. Given an accurate scene description, an image rendered with a global illumination algorithm should be indistinguishable from a photograph. Although global illumination techniques have improved significantly over the last 25 years, their effectiveness is still hampered by lengthy execution times.

1.2 Work Outline

This thesis proceeds as follows: in Chapter 2 we give an explanation of global illumination algorithms and how they simulate light. Next, we discuss the details of previous work in Chapter 3. Using this background, we introduce a measure of the computational complexity for global illumination algorithms in Chapter 4, including our proposal for a new approach for reducing this complexity. We then introduce our new global illumination algorithm, pipe casting, in Chapter 5 followed by a discussion of our results is Chapter 6. We conclude with ideas for future work in Chapter 7. The full results of our experiments can be seen in Appendix A.

Chapter 2

Light Transport

Understanding how global illumination algorithms, like the one we propose, render photorealistic scenes first requires a basic understanding of how light travels in real life. In this chapter we introduce the basic material properties that determine how light reflects and the lighting effects that these properties cause. Understanding these concepts will help unfold the complexity of simulating light transport and highlight the power of global illumination algorithms that accurately render these effects.

2.1 Light Transport

We perceive our surroundings as light leaves a light source, such as the sun or a light bulb, and arrives at our eyes or a camera. This process of light generation (from a light source) and perception (by an eye or camera) is called light transport. During light transport, we often perceive two changes in light as it collides with surfaces. First, the light appears to change color (which is actually a change in its wavelength), and second, the light changes direction (see Figure 2.1). Surfaces in the world around us—wood, carpet, leaves, plastic, etc.—each affect the color and direction of light differently. Despite this diversity of surfaces, most common materials can be described as a combination of the diffuse, reflective, and transparent properties, which we describe next.

Perfectly diffuse surfaces deflect light evenly in every direction. This even distribution means the same location on a diffuse object will have the same appearance regardless of the
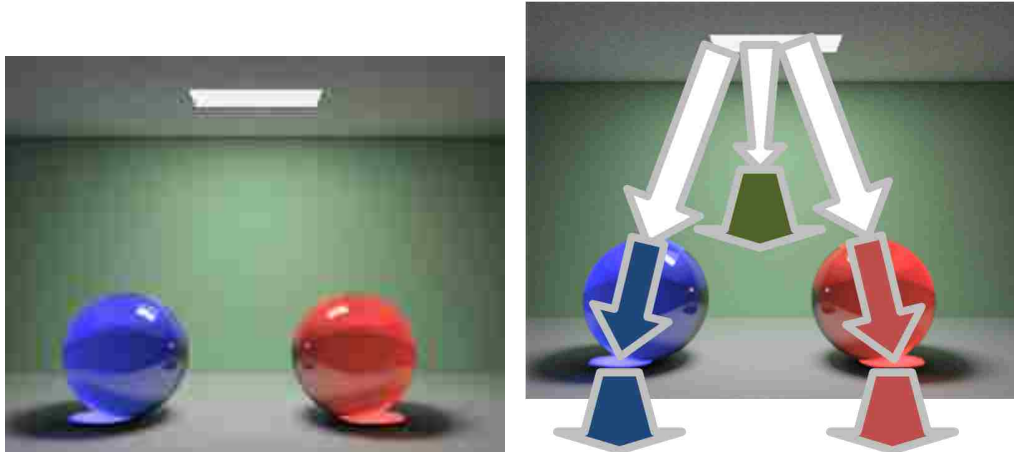
4

Figure 2.1: A diagram demonstrating the how light appears to change color and direction in a scene. On the left is a rendered image showing a room with two colored, semi-transparent spheres, an area light source on the ceiling, and a green wall. On the right is a diagram showing a few of the paths the light could take in this scene. First, notice that the white light changes to blue or red as it passes through the spheres (which results from the wavelength of other colors being absorbed). Second, the curved nature of the red and blue balls focuses light into two caustics on the floor.

viewing direction. Many common objects are diffuse in nature—carpet, wood, most fabric, and paper. The wall, ceiling, and floors in Figure 2.1 are all diffuse surfaces.

Reflective objects reflect light based on the angle the light arrived. Mirrors are the classic example of reflective objects, but many other objects are at least partially reflective, including water, polished stone, varnished wood, and smooth metals. As opposed to diffuse objects, reflective objects change appearance as the view direction changes.

Transmissive objects allow light to pass through them. Most glass objects have this property. Frequently, transmissive objects change the direction of the light passing through them. For example, a pencil in a glass of water will appear broken because the transmissive water changes the direction of the light passing through it. Like reflective objects, transmissive objects are view dependent. The spheres in Figure 2.1 are primarily transmissive.

Many objects have surfaces which can be defined using a weighted combination of the diffuse, reflective, and transmissive properties. For example, polished counters can be

5

modeled with a diffuse and reflective component and glass can be modeled using reflective and transparent components.

2.2 Lighting Effects

The appearance of objects is not determined solely by the material properties of those objects. Since each surface can change the appearance and direction of light, light can arrive at a surface directly from a light source or by first interacting with other objects. This interaction between lighting and objects can produce a wide variety of lighting effects—each of which is difficult to render. In this section we discuss several of these effects and why they are difficult to capture. Understanding this shows the power of global illumination algorithms.

2.2.1 Occlusion

The first lighting effect we discuss is light occlusion, also known as shadowing. Many objects do not let light pass through them. If these surfaces are in between a light source and another surface, the blocking surface will cast a shadow on the second surface (see Figure 2.2).

In addition to creating simple shadows, transmissive objects may block only certain wavelengths of light, which causes shadows that are not black (see Figure 2.1). Additionally, since light can reflect off surfaces and arrive at an occluded surface indirectly, the area in shadow may not be completely black. Thus, the shadows we see are the result of the position of all the objects in a scene and their respective properties. Furthermore, if the light source is large, the shadows do not have hard edges. This phenomenon, known as soft shadows, is shown in Figure 2.2.

Accurately computing soft shadows is computationally demanding. In order to calculate the color of a shadow, an algorithm needs to know more than whether or not a point is occluded; it needs to know what percent of the light source is occluded. This requires multiple samples for each point that could have a soft shadow.
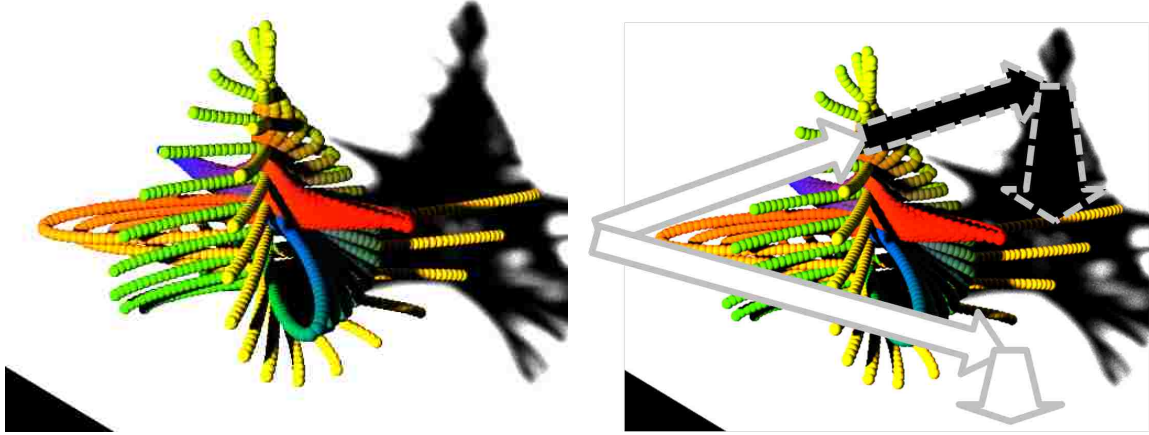
6

Figure 2.2: Figure demonstrating complicated shadow computation using global illumination. The image on the right shows two paths which light can take. In the bottom path the light collides with the white diffuse wall and bounces in the direction of the camera. On the top path, the light cannot reach the wall because a sphere deflects it first. Thus, no light is bounced towards the camera from the wall and it appears in shadow. This lack of light is denoted by a dashed arrow in this figure.

2.2.2 Caustics

The second lighting effect we discuss is caustics. Caustics occur when reflected or transmitted light is focused into one spot causing a bright highlight. For example, the curved glass on a wristwatch can focus light on a small spot on a wall (see Figure 2.1 for another example). The appearance of caustics are very sensitive to the shape of the caustic-causing objects and the surfaces around it.

Computing caustics dramatically increases the computation required to render. Since caustics mean that bright light can be coming from surfaces other than light sources, an algorithm has to sample all transparent surfaces to see if they cast a caustic.

2.2.3 Diffuse Bleeding

Perhaps the lighting effect that is most difficult to render is known as diffuse bleeding. Diffuse bleeding occurs when light arrives at a surface after first deflecting off a diffuse object (see Figure 2.3). For example, a room with lights embedded in the ceiling would have no direct

7
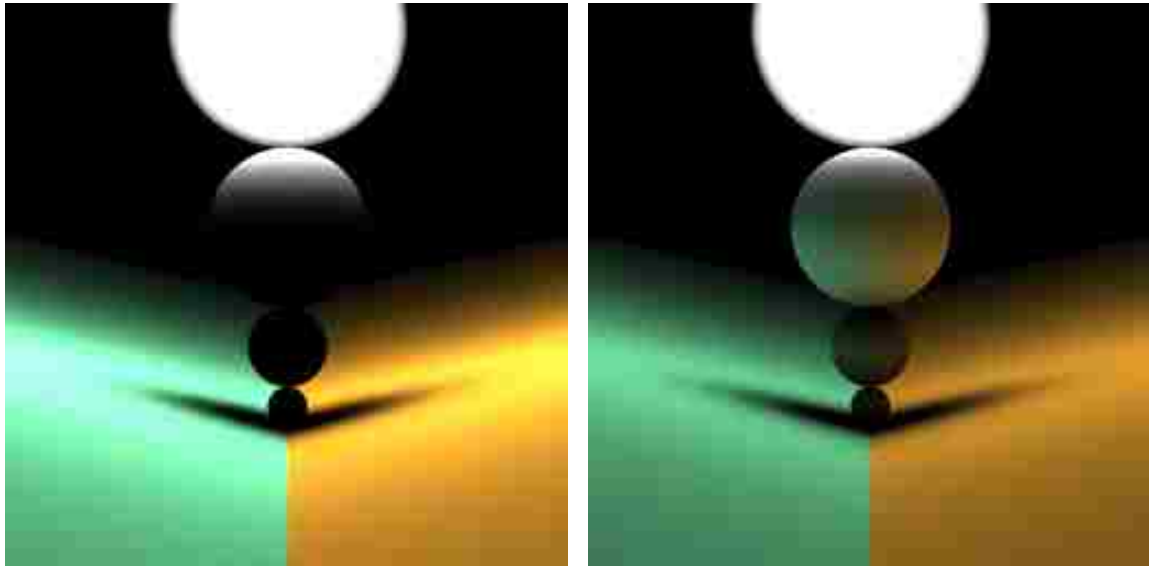
<div align="center">(a) Without Diffuse Bleeding   (b) With Diffuse Bleeding</div>

Figure 2.3: Figure demonstrating diffuse bleeding. The image on the left is rendered without diffuse bleeding—the light which deflects off the green and gold surfaces was not taken into account. The image on the right accurately calculates diffuse bleeding. This effect, which is difficult to achieve without a global illumination algorithm, makes a significant contribution to the realism of a rendered image.

path from the light to the ceiling itself. Yet the ceiling is not black because light reflects off diffuse surfaces, like the floor and walls, and arrives at the ceiling.

Calculating diffuse bleeding requires a rendering algorithm to treat every surface as if it were a light source. This mean than in addition to determining how much light arrives at a point from a light source, an algorithm that calculates diffuse bleeding has to calculate how much light arrives from every possible direction. This significantly increases the computation required to render.

2.3 Summary

In this chapter we discussed how a combination of diffuse, reflective, and transmissive properties can describe many surfaces in real life. We then discussed how the combination of these properties results in complex lighting effects. Capturing these lighting effects accurately is

difficult, but it results in a photorealistic image. In the next chapter we introduce global illumination algorithms which successfully accomplish the task of generating photorealistic images.

Chapter 3

Previous Work

In our introduction we described why photorealistic rendering is important to a variety of fields and commercial endeavors. In the previous chapter we introduced the lighting effects which make rendering photorealistic scenes a difficult task. In this chapter we introduce global illumination algorithms, which accurately render photorealistic lighting effects.

We begin this chapter with a series of mathematical concepts that lay the ground work for global illumination. Then, we introduce specific algorithms which incorporate some or all of these concepts. Our work will combine pieces from several of these systems to create a global illumination system that produces comparable or better results in significantly better time.

3.1 Previous Mathematical Work

Much of the global illumination literature resides on a foundation of key mathematical concepts and algorithmic ideas including backwards tracing, the rendering equation, and anti-aliasing. In this section we introduce these ideas.

3.1.1 Backwards Tracing

To render complex lighting effects, global illumination algorithms model how light travels from a light source to a camera. In global illumination literature, this route is referred to as a light path. Figure 3.1 highlights three different paths in a scene—one which appears
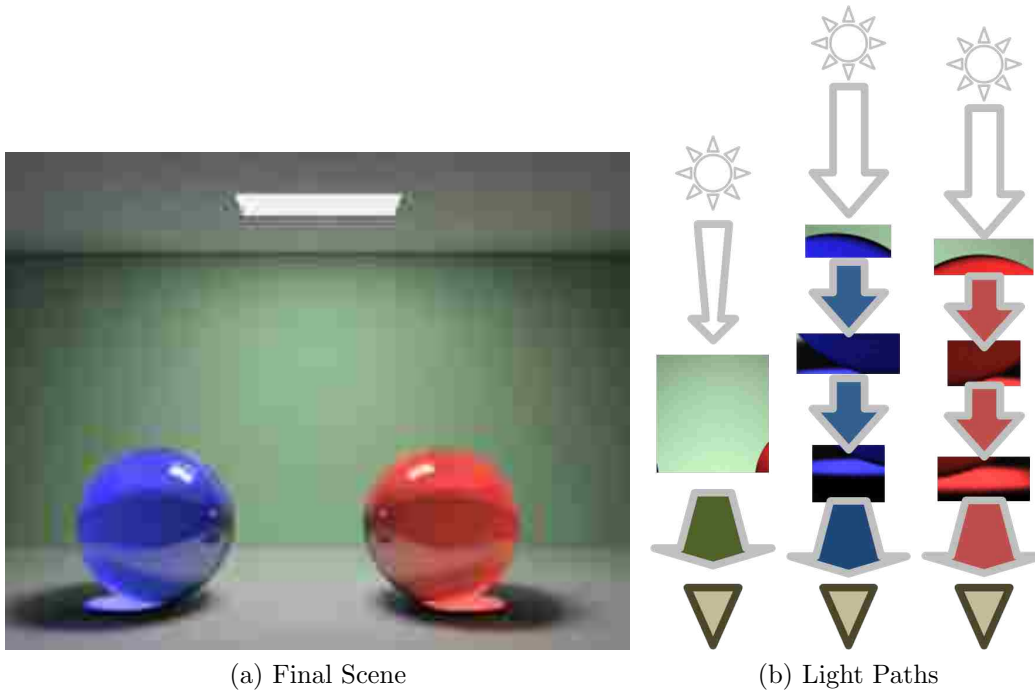
(a) Final Scene   (b) Light Paths

Figure 3.1: Figure showing three light paths in a scene (shown previously as Figure 2.1). On the left-most path is the path taken by the light which starts at the light source (sun icon), reflects off the green back wall, and bounces directly to the camera (triangle). The middle and right paths show the path taken by the light which travels from the light, through the transparent spheres, out of the spheres, and off the white floor towards the camera.

blue, one which appears green, and one which appears red. We can specify a light path by providing a starting location of the light, a list of locations where light collides with a surface, and then the ending location where the light is seen by the camera.

Using this idea of light paths, we can think of all light paths in a scene as a tree (see Figures 3.2 and 3.3). In this figure, all of the light paths start at a light source. Modeling light transport in this way is often referred to as backwards tracing.

In any light transport tree like the ones in Figures 3.2 and 3.3, the root is the light source and the nodes are every surface that light can collide with after leaving the light source. Each node has a child node for every surface that light arrives at after bouncing off the parent node. A leaf node is a node with no children. The bottom nodes of this tree consist of three types. First, surfaces which do not bounce or reflect light (these surfaces appear black to us) second, the camera or eye where perception occurs, and third, the background
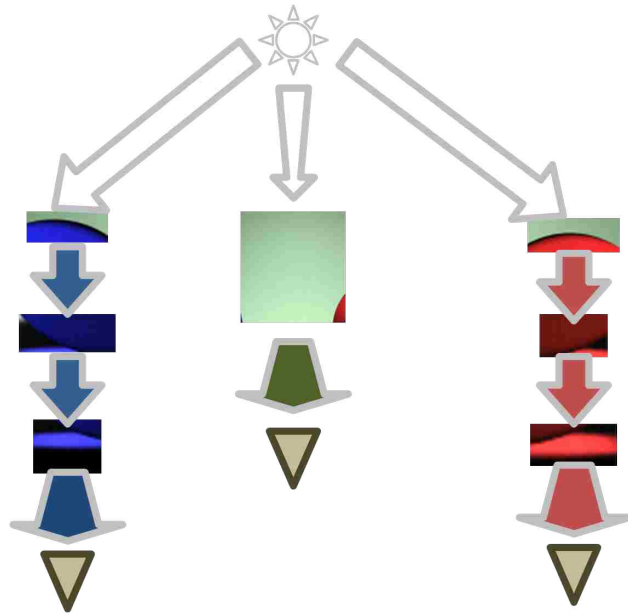
11

Figure 3.2: Figure showing how the light paths from Figure 3.1 can be thought of as a tree. With the light sources as the root of the tree, every surface where light collides becomes a node. Each node has a branch to each surface to which it can reflect light.

(such as the sky). In situations where light is generated at more than one point, the light transport is a scene can be thought of as a forest of trees with each each tree representing a different light source.

Unfortunately, since space is a continuous expanse, there are an infinite number of locations where light from a light source can arrive. As a result, the branching factor at each node is infinitely large which means perfectly representing light transport as a tree is impossible. Since computers are finite, one simplification is to assume there is a finite, though very large, number of locations where light can collide with a surface.

3.1.2 Forward Tracing

Although backwards tracing is a convenient way to think of light transport, using it directly in rendering has several pitfalls. Specifically, most of the terminal nodes in the light transport tree will either be black surfaces or arrive at the camera after the light has lost so much energy that it will appear black.
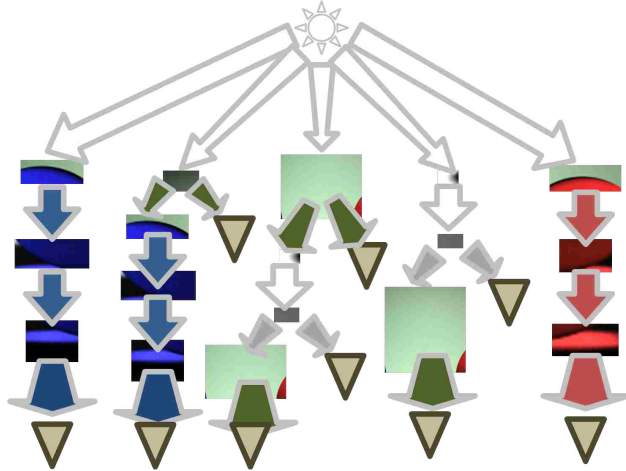
12

Figure 3.3: Figure showing a more detailed tree of the scene shown in Figure 3.1. Notice that since most surfaces are mutually visible, fully expanding this tree would result in an exponentially large tree.

To save computation time, global illumination algorithms generally reverse this tree in what is known as forward tracing (see Figure 3.4). Instead of the light source being the root of the tree, the camera or eye becomes the root of the tree. The first level nodes are surfaces which are visible to the root and could reflect light directly to the camera or eye. The second level nodes are surfaces which could reflect light to the first level nodes and so forth. In most scenes, the majority of the terminal nodes in this tree will be light sources or background. This reversal of the rendering tree means that more of the tree will contain information which contributes to the final image.

It is important to note that just like the backwards tracing tree, forward tracing is generally a forest of trees with each tree representing an individual pixel on the camera. In addition, the true branching factor is infinite since light can reflect in an infinite number of directions.

3.1.3 The Rendering Equation

The paper which laid the foundational mathematical groundwork for global illumination is *The Rendering Equation* [Kaj86]. In this paper, Kajiya presents a mathematical equation which captures the idea of forward tracing. By approximating this equation, a global illumi-

13
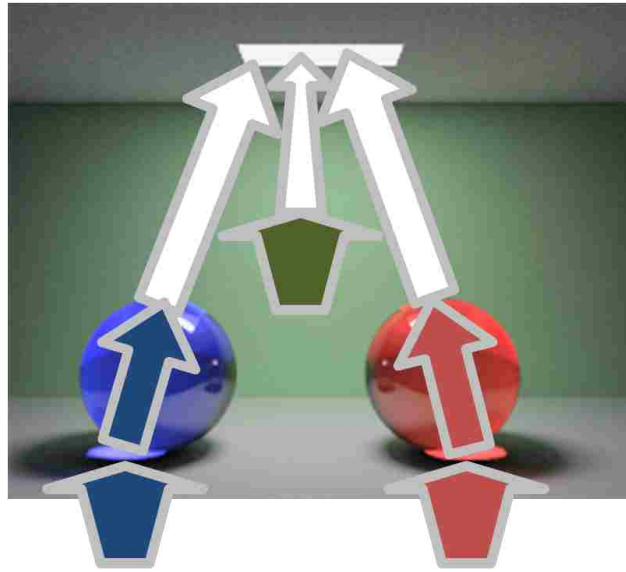
(a) Forward Tracing Example
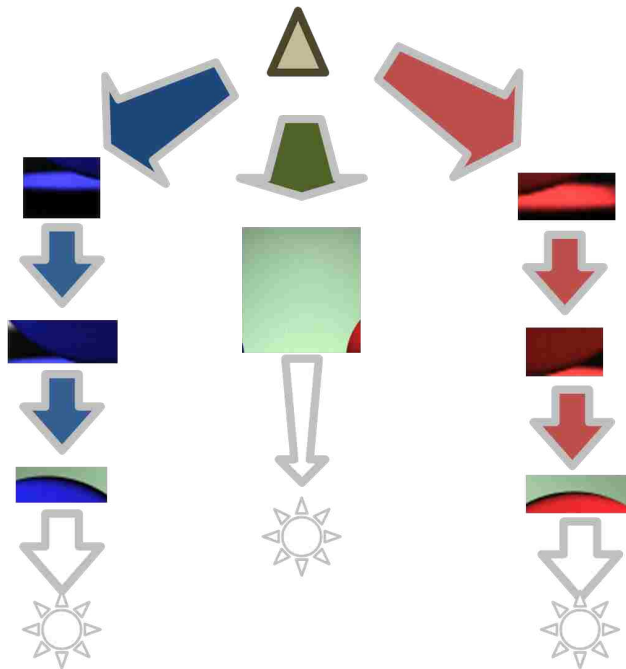
(b) Forward Tracing Tree

Figure 3.4: Diagram showing a tree representing forward tracing for Figure 2.1. By reversing the tree this way, the computation required to calculate an accurate image can be reduced.

nation algorithm can determine the value of each pixel in a rendered image. To understand this equation, we begin with an intuitive explanation of how the equation works and then present the actual equation.

Consider a tree as in Figure 3.4. If the camera in this tree represents a given pixel, we could do the following to calculate the amount of light which passes through this pixel:

- Query each node which connects to the camera node. Each node represents a surface which could reflect light directly to the camera node. If we consider all these surfaces as a set X, for each member of X called x, we check if x is a light source. If it is, we calculate how much light arrives at the camera from x.

- If a node $x \in X$ is not a light source, query the nodes below x. These nodes represent surfaces which can reflect light off x towards the camera. We will call this set of surfaces Y. For each member of Y called y, query to see if y is a light source. If it is, then calculate how much light from y reflects off x to the camera.

- If a node $y \in Y$ is not a light source, query the nodes below y. These nodes represent surfaces that can reflect light off y that can then reflect off x towards the camera. We will call this set of surfaces Z. For each member of Z called z, query to see if z is a light source. If it is, then calculate how much light from z reflects off y that reflects off x towards the camera.

- Continue this process recursively until some stopping criteria is reached.

Clearly this process is able to calculate all the light which bounces off surfaces until it reaches the pixel in question. Since each item in this algorithm is very similar, we could restate this process using structural recursion as follows: Given two nodes, a and b, define a function $lightTransport(a, b)$ that calculates the amount of light which comes to a from the direction of b. This function would first determine if b is a light source. If it is, it would calculate how much light from b goes towards a. Then for each node c that connects to b, calculate $lightTransport(b, c)$.

15

If we wanted to use the above function to calculate the amount of light arriving at a pixel, we could call the function *lightTransport* with the pixel as the first parameter and each surface visible to that pixel as the second parameter.

So far we have described how to calculate the amount of light arriving at a pixel in a scene algorithmically using a tree as our understanding of light transport. Kajiya defined the rendering equation using a mathematical function. The equation for this function is:

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

where:

- I is the light which travels from x' to x

- g is a function which determines if x and x' are mutually visible. If they are not, g evaluates to 0 which means the entire function will evaluate to 0.

- e is a function which returns the amount of light which is emitted from x' that arrives at x.

- The integral integrates over every surface s in the scene.

- $I(x', x'')$ is the recursive call which determines how much light from x'' arrives at x'.

- ρ evaluates to the intensity of light that comes from the direction of x'', bounces off x' and arrives at x. This terms dampens the value of $I(x', x'')$ to correspond with the fact that light can lose relative intensity with each bounce it takes.

By grounding rendering in a solid mathematical equation, Kajiya proposed a new rendering algorithm (which we discuss in Section 3.2.1) and opened the door for a variety of mathematical tools to be used to assist in the rendering process. In the next section we address one of the more common methods.

16

3.1.4 Monte Carlo Methods

The rendering equation is an integral which means that global illumination algorithms focus on integrating a detailed function. There are two approaches to solving this integral—analytical methods and numeric methods. Analytic methods are attractive for solving certain integrals; however, for a scene description of even minimal complexity, there is no known analytical solution to the rendering equation. To illustrate, consider the scene shown in Figure 3.5. We have approximated the two dimensional function representing the rendering equation about one point (labeled with a black dot) and graphed the result in Figure 3.6. Notice that even though some parts of this function may have known analytical solutions, the function as a whole is not well-behaved. The function would appear even more chaotic if the scene were not composed of simple geometric objects. As a result, almost all rendering algorithms use numeric solutions to solve this integral instead of analytic methods.
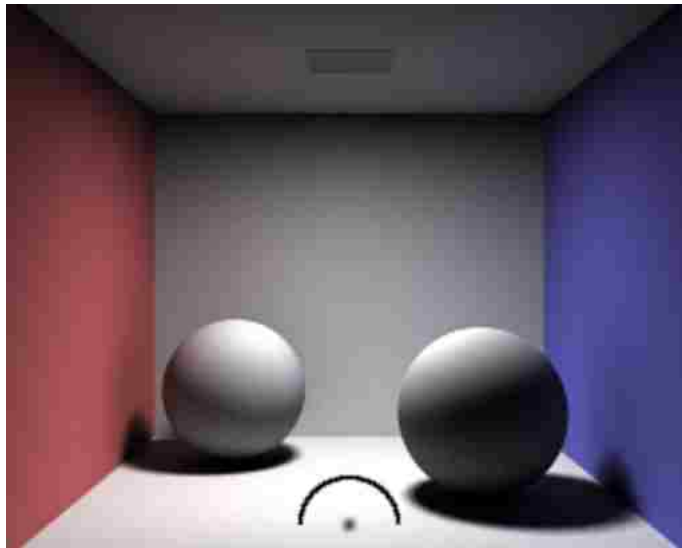

Figure 3.5: The example image in our discussion of integrals. The dot represents the surface location where the integral is being calculated. The ark represents the two-dimensional integral in question.

There are several numeric approaches to integrating the types of functions found in the rendering equation. One approach is to sample the function uniformly using the rectangle or trapezoid rule. Unfortunately, uniform sampling can lead to rough edges and sudden
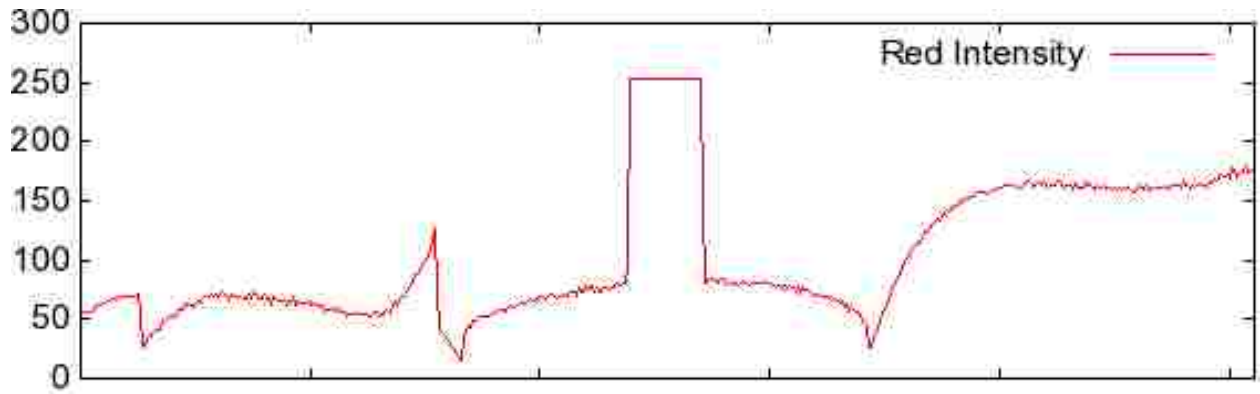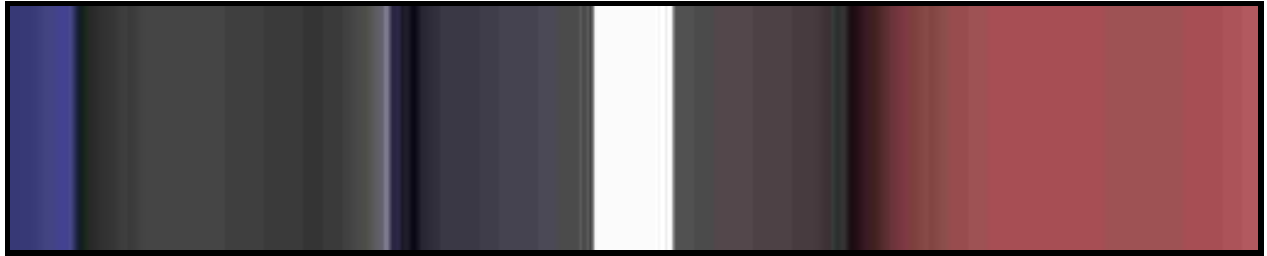
17

Figure 3.6: Example of a two-dimensional integral around a point in a scene (see Figure 3.5). Top: The incoming colors at the angle about the point change. Bottom: the intensity of the red component. Notice how poorly behaved the function is. The light on the right (coming from the red wall on the left) smoothly decreases until the ceiling is reached where there is a dramatic change in the incoming light. The incoming light changes again at the beginning and end of the light source. The blue wall changes the incoming light again with the sphere causing chaotic changes because of its curvature. There is no known way to analytically approximate this integral, hence the need for a numeric solution.

color changes that appear as aliasing artifacts (see Figure 3.7). In most global illumination literature a statistical sampling method, known as monte carlo Sampling, is used to compute the numeric integration.

In its most basic form, monte carlo sampling approximates the integral by sampling the function randomly and averaging the result. It can be shown [DBB03] that this type of integration converges with the standard deviation decreasing with $\sqrt{N}$ where N is the number of samples. Thus, when rendering with monte carlo methods, it can be shown mathematically that with time the image produced will slowly converge to the true image. The longer the algorithm runs, the closer the output image should be to the true value.

Earlier we described light transport with a tree. This tree analogy works well with monte carlo methods. Using monte carlo methods, you could approximate the amount of light which arrives at a node by sampling branches and averaging the results. The more branches sampled, the closer the result would be to the true value. Unfortunately, this sampling process converges at a rate of $\sqrt{N}$, which is very slow. As a result there are variations on the most basic monte carlo methods which decrease the convergence time. Importance sampling is one of these variations.

Importance sampling varies from basic monte carlo methods by not uniformly sampling the function. Consider the function shown in Figure 3.6. The function has very large values in the direction of the light source, and our integral would converge quicker if this area with high intensities were sampled more frequently. Importance sampling does exactly that–it samples light sources more frequently while still converging on the correct answer. A detailed look at this method can be found in [VG95].

Another set of variations on the basic monte carlo methods are known as biased algorithms (as opposed to unbiased algorithms like path tracing). A biased algorithm is one which does not converge on the correct answer, but converges on a value which is close. These algorithms trade a small amount of accuracy for a speed increase. For example, an algorithm may not accurately converge to the right light intensity at the corners of walls or at the edge

19

of objects [Sch03]. Some of the specific algorithms considered later in this chapter, including photon mapping and light cuts, use this method to increase their rendering efficiency.

3.1.5 Anti-aliasing

One problem evident in early rendering algorithms in known as aliasing. Aliasing is evidenced when the edges of objects do not appear smooth in an image (see Figure 3.8). The human eye does not have this problem [Coo86] because the photo receptors in our eyes are not evenly distributed. The aliasing in early rendering algorithms was a result of uniformly sampling in pixels–a problem with similar consequences as uniform numeric integration.

A robust solution to this aliasing problem is to sample each pixel in a non-uniform way, specifically by replicating a Poisson distribution (for a more in depth approach see [DW85]). The simplest method for approximating a Poisson distribution is to select a random direction within a subsection of the pixel. This simple method for randomizing sample direction has a significant affect on the resulting image (see figure 3.8).

3.2 Previous Algorithms

In the previous section we introduced some of the groundwork of rendering including the rendering equation which captures all the lighting effects discussed in Chapter 2. With this background we can now discuss some of the details of previous implementations of global illumination algorithms.

3.2.1 Path Tracing

In his rendering equation paper, Kajiya presented a global illumination algorithm for approximating the rendering equation called path tracing. Path tracing renders by sampling the forward tracing tree as described in this chapter. For each pixel, samples are taken down the tree with the results of each sample being averaged. This algorithm was revolutionary with its capacity to capture many complicated lighting effects. It also was easy to extend with

(a) Sampling without Jittering   (b) Sampling with Jittering

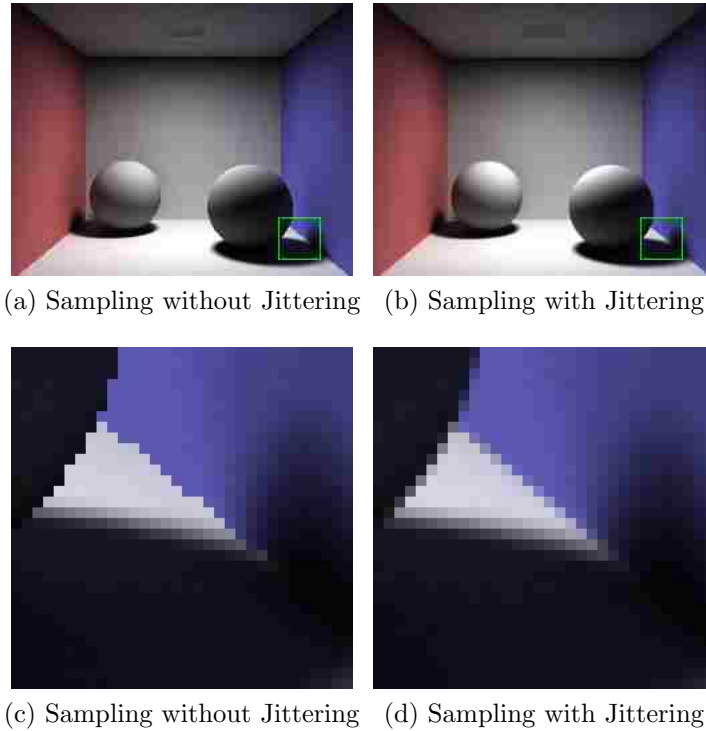(c) Sampling without Jittering   (d) Sampling with Jittering

Figure 3.7: Top: Output from a path tracing algorithm without jittering (left) and with jittering (right). Bottom: A zoomed look at the lower right (highlighted in green) of the two top images.

importance sampling and anti-aliasing capacity. Since path tracing is the foundation from which we derive our new global illumination algorithm, we here present the basic pseudo code for this algorithm:

for each pixel **do**

  for sample = 1 to N **do**

    p = new path

    while we haven't found a light or the path is not too long **do**

      sample in one direction

      add the resulting surface to our path

    end while

    calculate the color contribution of p

  end for

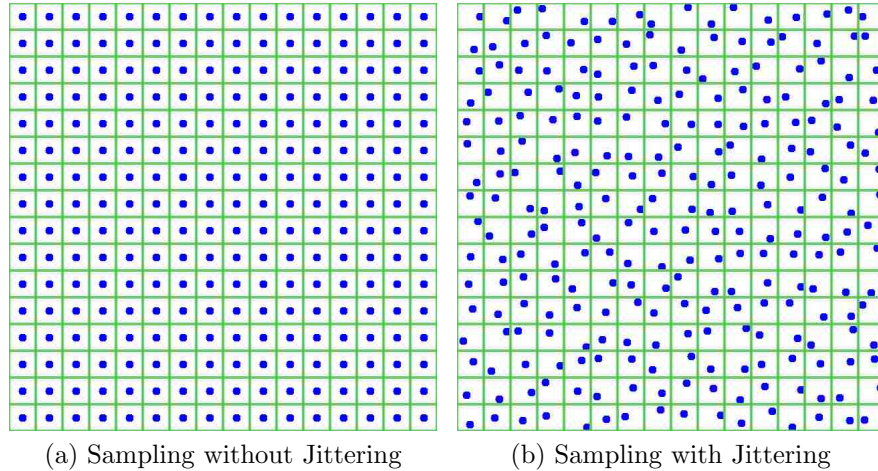(a) Sampling without Jittering (b) Sampling with Jittering

Figure 3.8: Diagram showing the distribution of samples without jittering (left) and with jittering (right). Since the receptors in our eyes are not evenly distributed, sampling with Jittering more closely approximates what the human eye perceives.

end for

for each pixel **do**

    color = average of all samples

end for

Unfortunately, this method of rendering is very slow for reasons that will be discussed in Chapter 4 in detail. Many of the algorithms which followed path tracing tried to address these speed issues.

3.2.2 Photon Mapping

Photon mapping [Jen01] as presented by Jensen is a global illumination algorithm which tries to address the speed issues of path tracing by storing some lighting information and breaking the rendering process into two passes.

In the first pass "photons" are shot from light sources. The photons travel through the scene by reflecting off surfaces until a stopping criteria is reached. When a photon is stopped, the algorithm remembers the photon's location and the light intensity at that location in space.

22

In the second pass, the algorithm proceeds in a manner similar to path tracing with one difference–when a light ray is generated, the algorithm checks if any photons are stored near the location of the end of the light path. If there are enough photons near this point, the algorithm uses the light information of the photons instead of continuing to sample. If there are not enough stored photons, the path tracing-like algorithm continues.

Photon mapping has proven to be an excellent algorithm for rendering scenes with complicated caustics (especially those cast by water). However, although this method can converge quickly, in Jensen's form the algorithm is biased–it converges on an inaccurate image [Sch03] with under-lit corners and edges. Also, some paths may not be able to take advantage of all the photon preprocessing if no photons land near a path. Lastly, the more photons cast the more accurate the image will be. Unfortunately, the more photons cast, the more memory is needed to store them and many of the photons generated may never contribute to the calculation of the rendered image. As a result, large amounts of storage can be required for information which may not be used.

3.2.3   Progressive Photon Mapping

One of the more recent extensions to photon mapping is progressive photon mapping [HOJ08]. The goal of this extension is two fold. First, it focuses on scenes where the light is enclosed by a transmissive surface (for example a light enclosed by a glass). Second, it alters photon mapping so it can be run for long periods of time without consuming large amounts of memory. To achieve these two goals, progressive photon mapping reverses the two passes in photon mapping.

In the first pass, a path tracing-like step is followed with paths being terminated when they collide with a diffuse surface. When a collision occurs, the location of the collision is remembered, as well as the pixel which started the path and the bounces taken.

In the second pass, another path tracing-like step is followed, except this time the paths start at the light source. When a path collides with a diffuse surface, the algorithm

23

searches for nearby collision points from the first pass. If there are close collision points from the first pass, then the algorithm knows that there is a complete path from the light to a pixel and updates the images based on that path.

Progressive photon mapping improves on photon mapping by reversing the order of the generated paths and only storing a limited amount of information. However, the algorithm is built with the assumption that there will only be one diffuse surface in each path. Unfortunately, this property does not hold in most scenes.

3.2.4 Path Reuse

Bekaert et al. [BSH02] presented a variation on path tracing which followed the theme of trying to increase the speed of global illumination. One of the common sources of error in path tracing is that smooth surfaces do not appear smooth. This "noise" on the image is an artifact from the way each pixel is converging. In Bekaert's path reuse algorithm, several paths are calculated at once in an attempt to smooth out this noise across neighboring pixels. When a new surface is found in one path, the other paths being generated sample in that direction. The idea is these groups of paths would be similar enough that their resulting color values would be similar, thus reducing the noise across neighboring pixels. However, their algorithm also produces incorrect bright spots where pixels are sampled together.

3.2.5 Radiosity

Radiosity [GTGB84] is an area of photorealistic rendering which approaches the problem of generating images in a different way than path tracing. Radiosity discretizes a scene into small patches and builds a system of equations which represents the transfer of light in a scene.

Radiosity differs from path tracing-type algorithms in three ways. First, calculations are usually done without shooting rays into a scene. Instead, form factors are used to approximate how much light travels from one patch to another. Calculating these form factors

24

is usually the most expensive part of the radiosity computation. Second, with radiosity it is relatively easy to recalculate the position of the camera without recalculating all light interaction. Third, radiosity generally focuses on diffuse light calculations which limits the types of material properties in a scene. As a result, images generated with radiosity often cannot simulate scenes which have specular reflections or refractions.

Two variations on the basic radiosity algorithm have bearing on this work since they focus on precomputing light segments and paths to reduce rendering time. They are stochastic jacobi radiosity and precomputed radiance transfer.

Stochastic Jacobi Radiosity

A variation on radiosity [CCWG88] is an algorithm called stochastic jacobi radiosity [DBB03]. The goal of this algorithm is to avoid the expensive time spent calculating form factors and the space requirements for stored visibility relationships. Instead of calculating these relationships, at each step the algorithm calculates light segments for each surface patch. If enough segments are generated per patch, they approximate the form factors computation, thus reducing the total computation time by introducing monte carlo methods.

Precomputed Radiance Transfer

Another variation of radiosity is called precomputed radiance transfer [SKS02]. Algorithms of this type compute an equation representing the amount of light that would arrive at each object from distant light sources prior to actually generating the image. Once these equations are stored, an image can be quickly rendered by solving the equations for a specific set of distant lights. Storing these precomputed equations allows for the rapid rendering of images in a variety of lighting environments and at different camera angles.

Although precomputed radiance transfer has proven effective in rendering certain scenes, the constraints prevent precomputed radiance transfer algorithms from rendering scenes with light sources close to objects.

3.2.6 Metropolis Light Transport

Metropolis light transport (MLT) [VG97] and similar algorithms (for example [CTE05]) propose a technique which improves the speed of global illumination by using Metropolis-Hastings sampling methods.

In path tracing, a path is constructed by generating new rays at the end of the current path until a stopping criteria is reached. In the Metropolis-Hasting sampling, new paths are generated by taking a known path and altering it. For example, MLT might take a path and change the light source it samples at the end of the path.

By pertubating paths instead of calculating each path independently, MLT is able to leverage previously known information to quickly generate new paths. Also, since these paths are similar, the resulting image should have less noise since samples were taken in the same direction.

3.2.7 Lightcuts

Lightcuts (see [WFA*05] and [WABG06]) is a new rendering technique which works particularly well with scenes with multiple light sources. These types of scenes are particularly time consuming to render because the number of light sources to sample is very high. Scenes with this characteristic include scenes with a TV or computer screen (with each point on the screen showing a different color) or large fluorescent lights. Lightcuts handles the rendering of these scenes by clustering lights together based on how similar they are. When lights are clustered, the algorithm only samples representative lights from a cluster. This technique can exponentially reduce the number of lights to be sampled, thus reducing the number of paths which need to be generated. However, clustering lights simplifies the scene definition. The result is a biased algorithm—the resulting image will not converge to the true image.

Chapter 4

Computational Complexity of Rendering

In chapter 3 we discussed a series of global illumination algorithms, each of which was designed to improve the speed or quality of global illumination. In this chapter we introduce methods that have been used in the past to compare algorithms and then discuss significant issues with these methods. We then propose a novel method of comparing algorithms and use this method to show where significant improvement in global illumination algorithms can be made. We then discuss the theory behind our new algorithm that uses these improvements. Our implementation of this algorithm will be discussed in Chapter 5.

4.1 Previous Algorithm Comparison Methods

A variety of methods have been used to compare and contrast global illumination algorithms. One fundamental way of comparing algorithms is to contrast their capacities. In Chapter 3 we compared algorithms based on what type of lighting effects they could render (for example some algorithms cannot natively handle reflections) and others were compared based on whether or not they were biased or unbiased (as discussed in Section 3.1.4). More subjective capacity comparisons focus on where certain algorithms tend to do better. For example, photon mapping is considered better at rendering detailed caustics than path tracing. Many algorithms try to fill a niche by doing one particular type of rendering better. For example, light cuts [WFA*05] and [WABG06] focuses on scenes with a large number of lights and progressive photon mapping [HOJ08] focuses on scenes with a light enclosed by a specular

27

surface. Unfortunately, neither of these two methods help when trying to objectively compare two algorithms that focus on rendering identical types of scenes.

Another approach to determining a metric for comparing global illumination algorithms is to compare the speed at which they converge on an ideal image. This approach is useful when comparing algorithms that are designed to render the same types of scenes. There are a variety of methods for comparing rendering speeds, each with its strengths and weaknesses.

4.1.1 Pixel-wise Error Comparison

We discussed in Chapter 3 how unbiased monte carlo global illumination algorithms converge on the correct image with time. Each pixel can be thought of as slowly arriving at the "true" color value. One way of comparing two global illumination algorithms is by comparing the rate at which their pixels converge on the true value.

In general the steps for this process are as follows:

1. Render an image for a long time. This image is referred to as the ideal image and it is assumed that the pixels have converged to their "true" values.

2. Using the first rendering algorithm, render for a shorter amount of time than that used in rendering the ideal image. This image, which will probably have some noticeable flaws, is called candidate one.

3. Using the second rendering algorithm, render candidate two.

4. Using some error metric determine the difference between the ideal image and candidate one.

5. Using the same error metric, determine the difference between the ideal image and candidate two.

6. Compare and contrast the results of these two comparisons.

www.manaraa.com

The result of this process is two error numbers that represent how each candidate image compares to the ideal image. This method is popular because it produces a numeric comparison that is preferred over a subjective classification. Also, since the output of rendering algorithms are the same format (images) this process can be used to compare algorithms no matter how different the implementations are. For an example of the results of this error calculation method consider Figure 4.1.
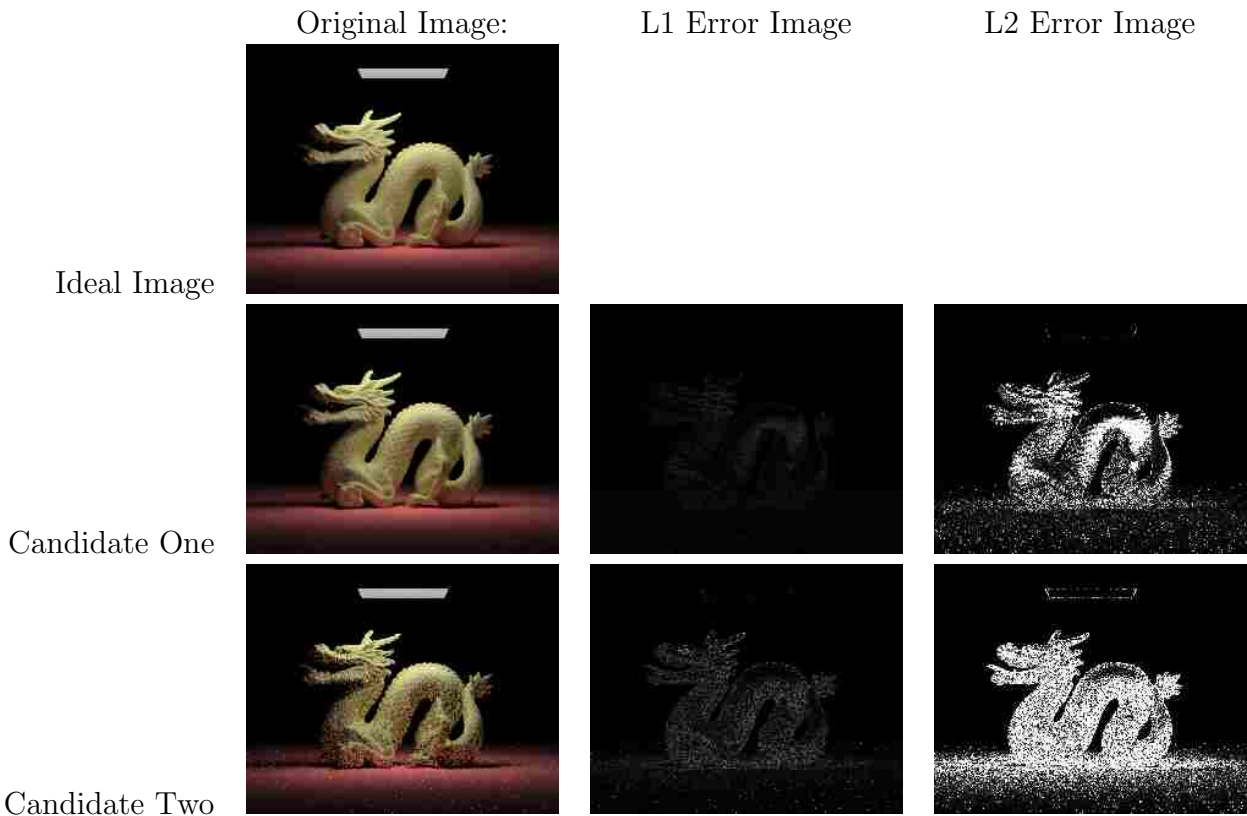

Figure 4.1: Example of pixel-wise error metrics. The ideal image (top row) is compared to the candidates (bottom two rows). Candidate one was rendered with 16 rays per pixel and has converged significantly more than candidate two. As a result, the error in candidate one is lower than that of candidate two. A visual way of depicting this error can be seen in the second and third columns. The whiter parts of these "error images" show where there is large error, i.e. where the candidate image is very different from the ideal image. The numeric errors for these candidates are listed in Table 4.1

Like any method for determining error, there are a variety of functions for calculating the error between two numbers. This is done by comparing each pixel in the ideal image

29

Image:	L1 Error	L2 Error
Image One	158,394	244,0546
Image Two	417,811	18,926,240

Table 4.1: Pixel-wise error results from the images shown in Table 4.1. Notice that the difference between the L2 errors is greater than the difference between the L1 errors. This is because L2 error exaggerates large errors more than it does small errors.

and the pixels in the candidate images. The error calculated is often done using an L1 or L2 metric as follows:

$$L1\ error = \sum_{p\in Image} |Intensity(idealImage(p)) - Intensity(candidate(p))|$$

$$L2\ error = \sum_{p\in Image} (Intensity(idealImage(p)) - Intensity(candidate(p)))^2$$

Where $Intensity(p)$ can be defined any number of ways. One method is to compare gray-scale values. Another way is to consider each red, green, and blue channel separately. We use both of these techniques in our results section.

Despite the empirical nature of this error metric, [BSH02] points out that this error metric does not necessarily conform to the error perceived by the eye. Consider Figure 4.2 with three very different candidates, each with the same numeric error result. This has to do with two problems with the pixel-wise error metric. First, as seen in candidate three, using gray scale values can mask texture flaws that are highly noticeable. Second, and more importantly, the human eye is tuned to notice texture. The pixel-wise error metric considers each pixel independently and cannot detect if there are strong changes in texture where the ideal image is smooth. Thus, candidate two has visible striping, yet it still results in the same error metric as the candidates that are as smooth as the ideal image.

Although using pixel-wise error metrics are convenient and simple, the metric does not help us design new algorithms. In order to make significant improvements in the area of global illumination, a better comparison must be used to judge rendering algorithms.

| (a) Ideal Image | (b) Candidate One | (c) Candidate Two | (d) Candidate Three |
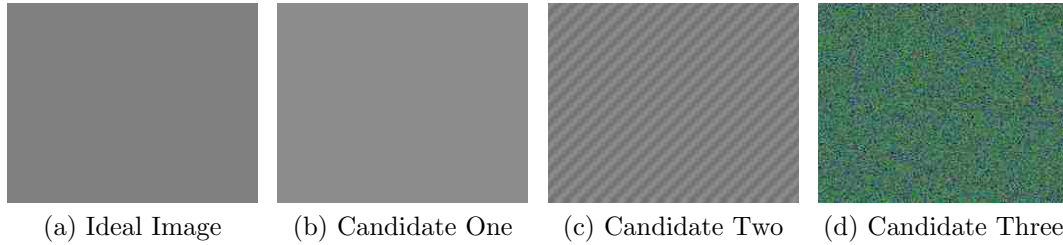
Figure 4.2: Example of three candidate images with the same error. Even though candidate two is striped and candidate three is very noisy, each pixel has the same error as candidate one. Thus, even though the numerical error calculated is identical, there is huge visual discrepancy visible to the human eye.

4.2 The Computational Complexity of Rendering

We propose using computational complexity to compare global illumination algorithms because computational complexity addresses fundamental strengths and weaknesses of an algorithm. The concept of computational complexity is not new to the field of computer science, but in the area of photorealistic rendering we know of no attempt to perform an in-depth comparison of the computational complexity of various global illumination algorithms. We believed that when used in conjunction with other metrics, this technique will provide a more accurate comparison of algorithms by mathematically comparing fundamental convergence rates.

In order to define computational complexity for rendering algorithms, we need to define the input variables to our function and determine the stopping condition for the function. With these two parts we can construct a function that approximates the expected amount of ray casts required to generate an ideal image. With this complexity we can more accurately compare and contrast global illumination algorithms that cast rays.

4.2.1 Input Variables

The input variables into a computational complexity measure are used to indicate the complexity of the problem being solved. For example, in the Traveling Salesman problem the input represents the number of cities—a number that directly reflects the complexity of

31

the problem. For rendering, we propose two input variables that represent the difficulty of rendering as follows:

1. Computers have a limited level of granularity when calculating numbers. Since the rendering equation requires an infinite integral but computers are inherently finite and discrete, there is a limited number of locations where light can collide with the objects in a scene. In radiosity, this discretization is done explicitly as part of the "patch" definition. In path tracing-based algorithms this discretization is much smaller—usually down to the resolution of floating point numbers. Whatever the method, the number of distinct locations where light can collide with a surface in a scene is a strong indicator of the complexity of the scene in question. We will refer to this number as n. We use this number as opposed to the polygon count or other primitives for several reasons. First, it allows our measure to be independent of the actual primitives used since all primitives eventually are rendered using collisions with their surface. Second, it allows our measure to be independent of how primitives are stitched together. For example, a flat wall could be composed of one rectangular polygon, two triangular polygons, or up to an infinite combination of the two.

2. The second input variable has to do with the number of light bounces. As light bounces through a scene, it can lose intensity as it is absorbed and re-emitted by surfaces. Thus, the longer light rays are calculated, the less information they contribute to the final image. As a result, many algorithms specify a cap on the length of light rays as shown in Figure 4.3. The depth of light paths that are calculated directly reflects on the difficulty of the rendering. We call the maximum number of light bounces ld in our computational complexity measure.

In our computational complexity measure we have chosen not to include the number of polygons and objects in our scene for two reasons. First, the use of trees to partition space can be used to dramatically reduce the slow-down caused by a large number of individually

defined surfaces. Second, our first variable, n, encapsulates the amount of surface area to be considered in the scene.

4.2.2   Finishing Criteria

Unlike other computational complexity measures, the monte carlo methods used in global illumination algorithms make it difficult to exactly determine when an algorithm has completed. In practice, a renderer is finished when either the output "looks good enough" or the pixel-wise error has dropped beneath some arbitrary threshold. Clearly, neither of these measures can be used objectively and consistently in a computational complexity measure. Instead, we define the algorithm as having finished when it has run long enough to have calculated each possible path of length ld in a scene. In other words, even though in practice the same path may be calculated more than once, we say the algorithm is done when it could have calculated each path. We call the result of this computation the "ideal image". For example, for the scene in Figure 4.3, the render would be done when an algorithm has run long enough to have explored the entire tree.

4.2.3   Computational Complexity of Tree-Based Algorithms

In the previous chapter we demonstrated that most current global illumination algorithms compute light transport as a recursive integral without storing path information. We thought of this process as having a rendering tree. Having now defined the variables n and ld we can derive the computational complexity of an algorithm that renders in this tree-based way.

We defined a tree where the branching factor of each node represented the number of locations where light could bounce. Using recursive integrals, the number of directions where light could be sent was infinite. However, as discussed above, the number of locations on a computer is finite. The number of collision points will be on the order n, the number of distinct collision locations in the scene. As a result, we can say that this rendering tree will have a branching factor on the order of n. Likewise, we defined the point at which light

paths are terminated as ld. Thus, we know the depth of this rendering tree will be ld. There will actually be a tree for each pixel in the image as discussed earlier, but since this will be a multiplicative factor in every algorithm, it does not affect our final Big-O result.

Clearly the amount of ray casts required to generate this tree without any optimization is on the order of the number of nodes in the tree, n^{ld}. This is an enormous number that grows rapidly with an increase in n and even faster with an increase in ld.

With this computational complexity measure we can assign a computational complexity to the unbiased algorithms introduced in Chapter 3:

- Path Tracing: In order for path tracing to calculate every path in the ideal image, ld rays would have to be calculated for each of the n^{ld} paths in the scene. Since every path is being calculated without using information from other paths and ray casts, the entire tree must be traversed. Thus, the computational complexity of path tracing is $O(n^{ld})$.

- Path Reuse: Fundamentally, Bekaert's path reuse algorithm reorders the sequence in which paths in a scene are calculated. As a result, ld rays would have to be calculated for each of the n^{ld} paths in the scene–the exact same number as in path tracing. Thus, the computational complexity of path reuse would remain $O(n^{ld})$.

- Metropolis Light Transport: As MLT mutates to generate new paths, no new path is generated without at least one new ray being cast. Previously cast rays cannot be used to remove this step. Thus, the computational complexity of MLT remains on the order $O(n^{ld})$.

It is clear from looking at these algorithms that despite the specific variation, a tree-based algorithm that requires at least one ray cast to generate a new path will have a computational complexity of at least $O(n^{ld})$.

34

4.2.4   Computational Complexity of Graph-Based Rendering

One of the main contributions of this work is to propose an approach to rendering that fundamentally reduces the computational complexity of rendering below $O(n^{ld})$. The foundation of this approach involves changing the tree-based algorithms into a graph-based one.

To understand the advantage of a graph-based approach, consider Figure 4.3 that shows the tree required to render a trivial scene. In the tree of light-depth 4, notice that the connection between nodes 4 and 7 is calculated six times. Storing this information would reduce the time required for this calculation by 5/6, or more than 83%.

A graph is a structure that can store the connection between nodes. Since a graph is only of size n^2, it would only take n^2 ray casts to have all the connections between nodes, regardless of the light depth. Such an algorithm could significantly increase the speed of global illumination since there is no longer a requirement that a new ray be cast for every new path. As we will show later, this type of algorithm could render an exponentially higher number of paths than ray casts.

The implementation of such an algorithm requires two passes. In the first pass, visibility information is determined and stored. In the second, this information is used to calculate all light paths that exist.

Graph-based First Pass

In the first pass of a graph-based algorithm the goal is to generate the binary relationship between pairs of surface points in the scene. This is done by casting rays into to scene and recording the result. This relationship indicates whether the surface points are mutually visible. If there are n surface points, then there are on the order of n^2 relationships between these points.

Graph-based Second Pass

In order to render a final image, a rendering algorithm needs to know more than just which surfaces are mutually visible. It also needs to know how much light travels between surfaces and how much of that light arrives at individual pixels. This can be done by calculating how much light travels between each set of points for a given light depth.

Storing this information can be done with a three-dimensional table where each entry represents the amount of light traveling between two surfaces for a given light depth. For example, the amount of light traveling from surface A to surface B at a light depth of 3 would be stored in table entry (A, B, 3). Since there are n values for A and B and at most ld values for L, calculating this information would be on the order of $n \cdot n \cdot ld$ or $ld \cdot n^2$. Again, this number is significantly smaller than the n^{ld} minimum in tree-based algorithms.

A Simple Example

To illustrate the difference between tree-based methods and graph-based methods consider the scene presented in Figure 4.3. For illustrative purposes we consider node 1 to be a pixel in the image and nodes 6 and 7 to be light sources. Figure 4.4a shows all the ray casts that would be necessary to render this scene at a light depth of five. There are 24 different paths with over twice that many ray casts. As mentioned before, most of those ray casts are redundant. The graph-based representation shown in Figure 4.4b has 14 ray casts and all 24 paths worth of information can be quickly retrieved. Notice that the graph-based approach has done fewer ray calculations than there are paths, thus dramatically decreasing the time required to render this scene.

One draw-back of the graph-based approach is that rendering with a tree-based algorithm can be done recursively with little memory usage. In a graph-based approach the memory required to store the connections is on the order of $O(ld \cdot n^2)$. One way to think about a graph-based algorithm is that is trades memory for time. In most rendering applica-

tions, memory is significantly less expensive than time, so this type of trade-off is desirable. In Chapter 6 we give the details of our memory usage.

In the next chapter we discuss our specific implementation of a graph-based algorithm.
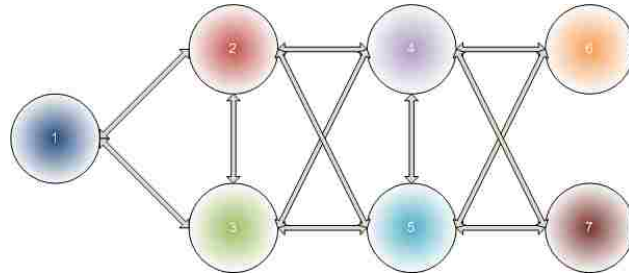
(a) An example scene
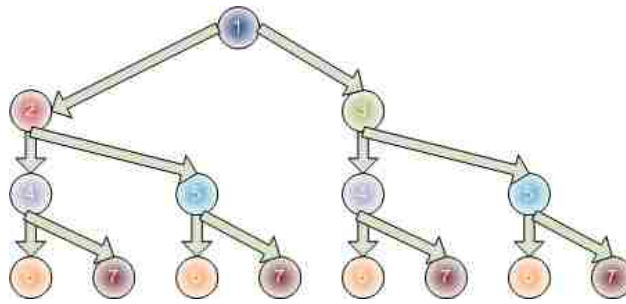
(b) Light Depth 3
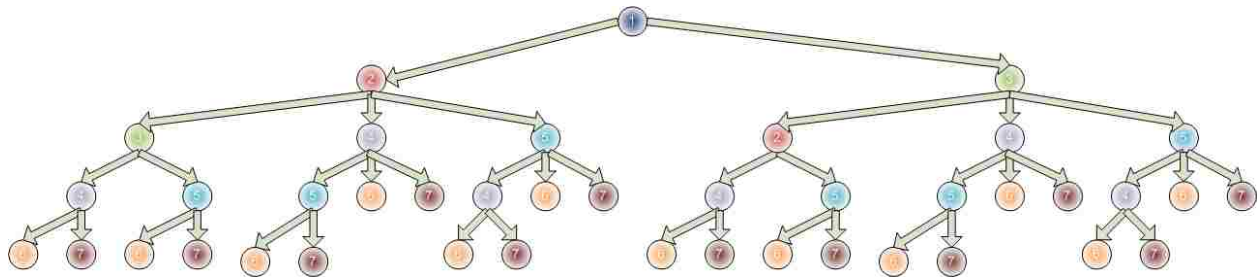
(c) Light Depth 4

Figure 4.3: Figure showing an example scene with 7 nodes. If nodes 6 and 7 are light sources, the middle and bottom figures show the tree-based rendering for a light depth of 3 and 4. Notice the significant increase in the amount of information when the light depth increases by one. This is why we include light depth in our complexity measure.
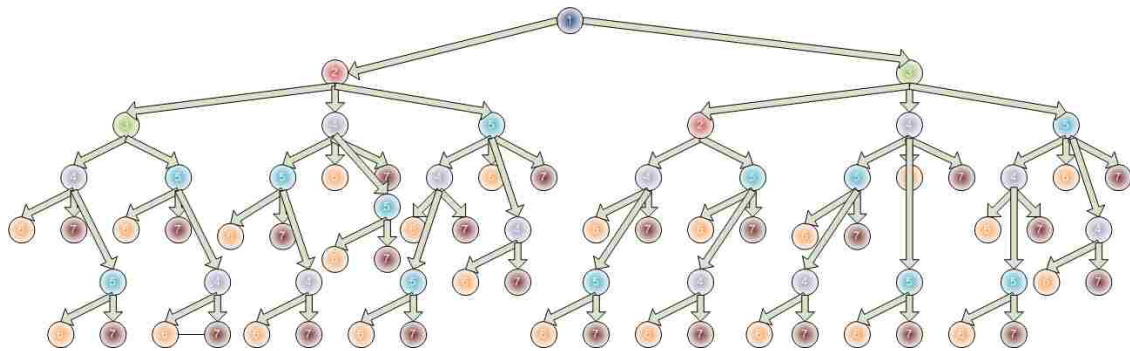
(a) A Tree-Based Representation
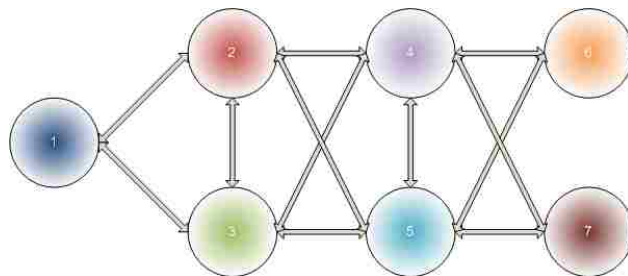
(b) A Graph-Based Representation

Figure 4.4: Comparison of how a tree-based and a graph-based algorithm would calculate light rays for the scene in Figure 4.3. Notice that there are significantly fewer rays in the graph-based approach even though both approaches represent the same information.

39

Chapter 5

Pipe Casting Implementation

In Chapter 3 we discussed several previous global illumination algorithms. In Chapter 4 we demonstrated that these algorithms are tree-based and have a computational complexity of $O(n^{ld})$. We then explained how a graph-based algorithm could render in significantly lower time. In this chapter we explain our implementation of a graph-based algorithm which we call pipe casting. We use the name pipe casting since the algorithm keeps track of how light flows between locations in a way similar to how pipes control the flow of water. In Chapter 6 we discuss in detail the results of pipe casting and how it can render in about 50% of the time used by path tracing.

5.1 Implementation of Path Tracing

To construct our new algorithm, we modified a path tracing implementation and split it into two passes as introduced in the previous chapter. To begin, we describe the details of our path tracing implementation. To describe our path tracing algorithm we cover the inputs to our algorithm, the volume subdivision scheme used, and the ray casting algorithm. All of the comparable components are the same in our path tracing and pipe casting algorithms.

5.1.1 Algorithm Inputs

Both path tracing and pipe casting share a significant amount of code. As such, the input to both algorithms is very similar. The following user-specified inputs are shared between both algorithms:

- Both algorithms read a modified RIB format with support for polygons and spheres, the three main surface properties described in Chapter 2, and area lights. The RIB specification is a file format designed by Pixar. Major 3D modeling programs such as Maya can export this format.

- The user-specified light depth represents the maximum number of light rays that can exist in a light path. If this number is too low, then the algorithm cannot capture all lighting effects. If it is too high, time can be wasted searching long light paths that do not contribute to the final image. Our results (shown in Chapter 6) were all generated with a high light depth to capture significant light interactions in each input scene.

- Since light intensities vary across RIB files, our algorithms take a toning parameter as input. This toning parameter is similar to the exposure setting on a a camera—it changes how bright the lights appear in the final image.

- Global illumination algorithms are stochastic as described in 3. As a result, one of the inputs to our algorithms is the number of samples to take at each pixel. A higher number of samples means the output of each pixel will be closer to the true value. The lower the number the faster the algorithm will run.

5.1.2 Volume Subdivision

To increase render speeds, many photorealistic algorithms divide the scene geometry using trees to accelerate collision detection. In an unoptimized form, ray calculations are done by calculating where a ray collides with every surface and choosing the closest collision.

41

This process is $O(n)$ where n is the number of surfaces since each ray is compared to every surface. With volume subdivision algorithms, the ray collision algorithm can be reduced to $O(log(n))$.

We used a volume subdivision technique called median split. Median split begins by building a tree that holds all the objects in the scene as shown in Algorithm 1. Each pass of the recursion splits a bounding volume in half until each bounding box holds only a few objects.

Algorithm 1 Pseudo code for the the first step of building a median split volume subdivision data structure. This recursive function builds a tree of bounding volumes that can be used later to reduce light ray calculation time from $O(n)$ to $O(log(n))$.

Given input objects O and parent node P:
if there are less than 2 objects in O **then**
    return
end if
Build a bounding box, B, for all objects in O
Set P as the parent of B
Find the dominant axis of B
Find the median value of all objects along that axis
Create two empty lists of objects, O1 and O2
for each object o in O **do**
    if o is in the the first half of B **then**
        Add o to O1
    end if
    if o is in the second half of B **then**
        Add o to O2
    end if
end for
Add O1 and O2 as children of O
Recursively call this function on O1 and O2

Once the tree is built, ray collisions can be calculated by recursing down the tree as shown in Algorithm 2. The tree algorithm speeds up this process since volumes are ignored if a ray cannot intersect them.

Algorithm 2 Pseudo code for using the volume subdivision data structure described in Algorithm 1.

Given bounding volume B and ray direction and origin R:
initialize closest collision, C, to infinity.
if B has no children **then**
  for each object O in B **do**
    if R collides with O nearer than C **then**
      C = collision of R at O
    end if
  end for
  return C
else
  if R collides with child O1 before O2 **then**
    Call this function on O1
    C = function return value
    if C = infinity **then**
      Call this function on O2
      C = function return value
    end if
  else
    Call this function on O2
    C = function return value
    if C = infinity **then**
      Call this function on O1
      C = function return value
    end if
  end if
  return C
end if

5.1.3   Calculating Light Paths

Once the scene is read in and split for optimization, the algorithm begins to calculate light rays to determine light paths. To do this, the algorithm loops across all the pixels in the image. Rays are generated starting at the camera and are jittered (see Section 3.1.5) to allow for anti-aliasing. When a ray collides with a surface, the material properties of the surface are used to determine where the light travels next. High level pseudo code for our path tracing algorithm can be seen in Algorithm 3.

In our implementation of path tracing, several checks are done when a light ray collides with a surface. First, if the light depth limit is reached, the path is assumed to contribute no light. If not, there is a check to see if the surface is light emitting. If it emits light, then the path is stopped and the amount of light coming from the light source in the direction of the ray is calculated and returned. If neither of these checks stop the light path, then other rays are generated to extend the light path.

Algorithm 3 High level Pseudo code for our path tracing algorithm. The code for the Shoot() function is found in Algorithm 4

Load RIB file
Build median split volume subdivisions
for each pixel p **do**
  for $i = 0$ to $numberOfRaysPerPixel$ **do**
    Calculate ray direction
    Jitter ray direction
    Shoot(ray direction)
  end for
  Set final pixel color to the average of every shot
end for

When a collision occurs, the object that stopped the light ray is queried for its fundamental properties (i.e. diffuse, reflective, or transmissive) and a new random direction for the next ray is determined based on these properties. Also, since our algorithm uses importance sampling, direct illumination is also calculated. For each light source a random spot on its surface is chosen to determine the ray direction. If a ray shot in that direction

44

collides with the light, then the original surface is considered illuminated. Choosing random locations on the lights allows the image to converge on soft shadows.

Once the indirect and direct rays are returned, the final value is determined by averaging their values using importance sampling. The importance sampling equation is shown in Equation 5.1 and 5.2.

$$Light\ Value = \frac{1}{numberOfSamples} \cdot \sum_{samples} light(sample) \cdot w(sample). \qquad (5.1)$$

$$w(sample) = sizeOfSamplingArea(sample)/probabilityOfChoosing(sample). \qquad (5.2)$$

For direct sampling of a light source, $sizeOfSamplingArea(sample)$ is the percentage of the unit hemisphere around the point that is covered by the light. The closer the point is to the light source, the higher this number will be. For indirect lighting, $sizeOfSamplingArea(sample)$ is 1 since we sample across the entire unit hemisphere.

The pseudo code for this entire process can be seen in Algorithm 4.

Before moving on to pipe casting there are several things to note. First, the most time intensive operation is calculating ray collisions. Even with the spatial subdivision described earlier, our tests show that approximately 75% of the proccessing time is spent calculating collisions. Thus, a decrease in the amount of ray casts required would result in a significant time improvement. Second, no information is stored by path tracing, so many of the recursive calls in path tracing are redundant. Our implementation of pipe casting addresses both of these issues.

Algorithm 4 Pseudo code for the function which implements the shoot function in Equation 3.

shoot(rayDirection)
Calculate ray collision
Get Material Type
if material type = light **then**
  return light value
end if
if light path length ≥ max light depth **then**
  return black
end if
if there was no collision **then**
  return background
end if
if material type = reflective **then**
  Calculate reflective ray direction
  return shoot(ray direction)
else
  if material type = transmissive **then**
    Calculate transmissive ray direction
    return shoot(ray direction)
  else
    if material type = diffuse **then**
      Calculate indirect lighting
      Calculate direct lighting
      return average using importance sampling (Equation 5.1)
    end if
  end if
end if

5.2 Implementation of Pipe Casting

The goal of our pipe casting implementation was to demonstrate that the theoretical ideas proposed in Chapter 4 would indeed produce a faster algorithm by taking advantage of available memory. To do this, we designed a new algorithm that reduces the computational complexity of rendering by storing which points are mutually visible. We implemented this idea by modified our path tracing algorithm keeping as much identical code as possible. For the rest of this chapter we describe the changes required. Then in Chapter 6 we discuss the results of our implementation.

5.2.1 Pipe Casting Data Structures

The first main change in our pipe casting implementation is the introduction of new data structures, surface points, joints, and pixels. These structures hold the graph-based data that allows for a reduction in the computational complexity of rendering.

Whenever a ray collides with a surface, a surface point is generated to store the information about that collision. Each surface point contains the following information:

- The location of the surface in 3D space.

- The normal of the surface at the given location.

- The material properties of the surface at that point.

- A pointer to the joint data structure wherein the location resides.

Each surface point structure corresponds to a node in the graph-based analogy as discussed in Chapter 4.

Surface points are connected using a second structure called joints. A joint is a data structure with a list of surface points. This list represents all the surface points that are visible to the joint. Consider Figure 5.1. In this figure there are two surface points on the left, a joint in the middle, and three surface points on the right. The surface points on the

47

left point to the joint. This means it is simple to determine the the joints are the left are mutually visible to the joints on the right. In this way joints correspond to a row of edges in the graph since they store which surface points are mutually visible.
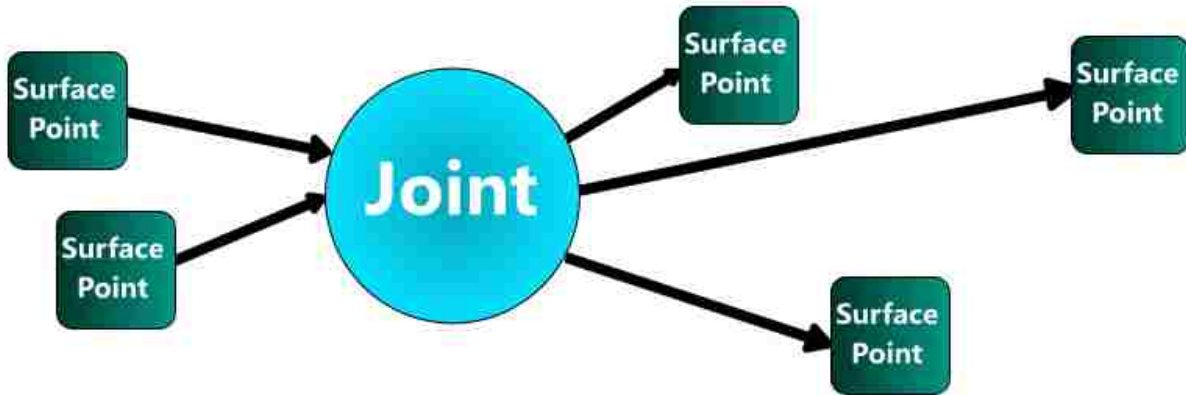
Figure 5.1: An image of a joint and surface points. The surface points on the left share the joint that points to other surface points on the right. Once the joint is made, it is easy to determine that both surface points on the left are mutually visible to the surface points on the right.

A pixel is a special version of a surface point that represents a pixel in the final image. Before pipe casting runs, an two-dimensional array of pixel data structures are created with each entry x, y corresponding to pixel x, y of the final image. In this way, when pipe casting finishes calculating lighting information, the algorithm simply queries each pixel object for its final value. That value is assigned to the corresponding pixel.

5.2.2   Basic Pipe Casting Algorithm

Our implementation changes path tracing in two main ways. First, it modifies how paths are generated in path tracing. Second, it adds a second pass to the algorithm that generates the final image. In this section we describe the basic algorithm. In the next section we go into detail about specific optimizations for additional speedups. The high level pseudo code for pipe casting is shown in Algorithm 5.

48

Algorithm 5 High level pseudo code for the first pass of pipe casting.
___
Load RIB file
Calculate median split volume subdivisions
for each pixel in the image **do**
   Build a pixel data structure
end for
for each pixel P **do**
   for $i = 0$ to $numberOfRaysPerPixel$ **do**
      Calculate ray direction
      Jitter ray direction R
      Shoot in direction R from P (see Algorithm 6).
   end for
end for
Calculate values of all Ps (see Algorithm 7).
___

First Pass

The first pass of pipe casting is shown in Algorithm 6. The majority of this first pass is done using a recursive algorithm similar to the main path tracing algorithm (shown in Algorithm 4). This function starts at a joint, J, and shoots a ray. If there is no collision, then J is connected to the background and the function returns. If the ray hits a light, a surface point is generated for that light, J is connected to a new surface point at the light, and the function returns. If the light path has exceeded the light depth limitation, then the light path is terminated and the function returns.

If the collision occurs at a location where the light path can continue, then the incoming joint is connected to the respective surface point and the function recurses. The direction of the next ray is determined by the surface properties. If the material is diffuse then two rays are generated, one for indirect lighting and another for direct lighting. For a detailed look at this first pass, see Figure 5.2 and Algorithm 6.

The result of this first pass is a graph of surface points and joints that represents all the light rays in the scene. The second pass uses this information to generate all light paths.

Algorithm 6 A detailed look at the recursive function in the first pass of pipe casting.

Given a ray direction R and joint J
Calculate ray collision
Calculate a new surface point, SP, for this collision
Get Material Type
if material type = light **then**
   Connect J to SP
   return
end if
if light path length ≥ max light depth **then**
   Connect J to null
   return
end if
if there was no collision **then**
   Connect J to the background
   return
end if
if material type = reflective **then**
   Calculate reflective ray direction, R2
   Call this function given R2 and J2
else
   Generate a new joint, J2, for SP
   if material type = transmissive **then**
     Calculate transmissive ray direction, R2
     Call this function given R2 and J2
   else
     if material type = diffuse **then**
       Calculate ray direction R3 for indirect lighting
       Call this function given R3 and J2
       Calculate ray direction R4 for direct lighting
       Call this function given R4 and J2
     end if
   end if
end if

Second Pass

In the second pass of pipe casting, the algorithm uses the surface points and joints to calculate all light paths and the resulting image. This pass starts by iterating over each pixel. Since pixels are extensions of joints, they have a list of surface points to which they are connected. The second pass iterates over all of these surface points (see Algorithm 7) and averages the color contribution of each. The resulting color value is toned and stored in the final image.

Algorithm 7 High level pseudo code for the second pass of pipe casting. This pass relies on Algorithm 8, the function that calculates the color contribution of a surface point.

```
Given a pixel P
SumColors = 0
for each surface point S in P do
    Calculate color contribution from S (see Algorithm 8)
    SumColors += color
end for
Average color values
Tone average color
Set pixel in final image to toned color
```

The majority of the second pass relies on a recursive function that calculates how much light a surface point reflects in a given direction (see Algorithm 8). This function takes a surface point S and direction R. If S is background or a light, the corresponding color is calculated and the function returns. If it is not, then the joint J that corresponds to S is determined. J stores all the surface points visible to S. The function then recurses over all those surface points. The results of these recursive calls are altered by the material properties of S and the resulting color contribution of S is returned.

5.2.3 Specific Optimizations

As noted in the previous chapter, calculating the full graph for a scene is unreasonable. Instead, pipe casting approximates the full graph just as other global illumination algorithms approximate the full tree. In this section we discuss several design decisions and optimizations used to further speed up pipe casting.

(a) Step 1 (b) Step 2 (c) Step 3

(d) Step 4 (e) Step 5 (f) Step 6
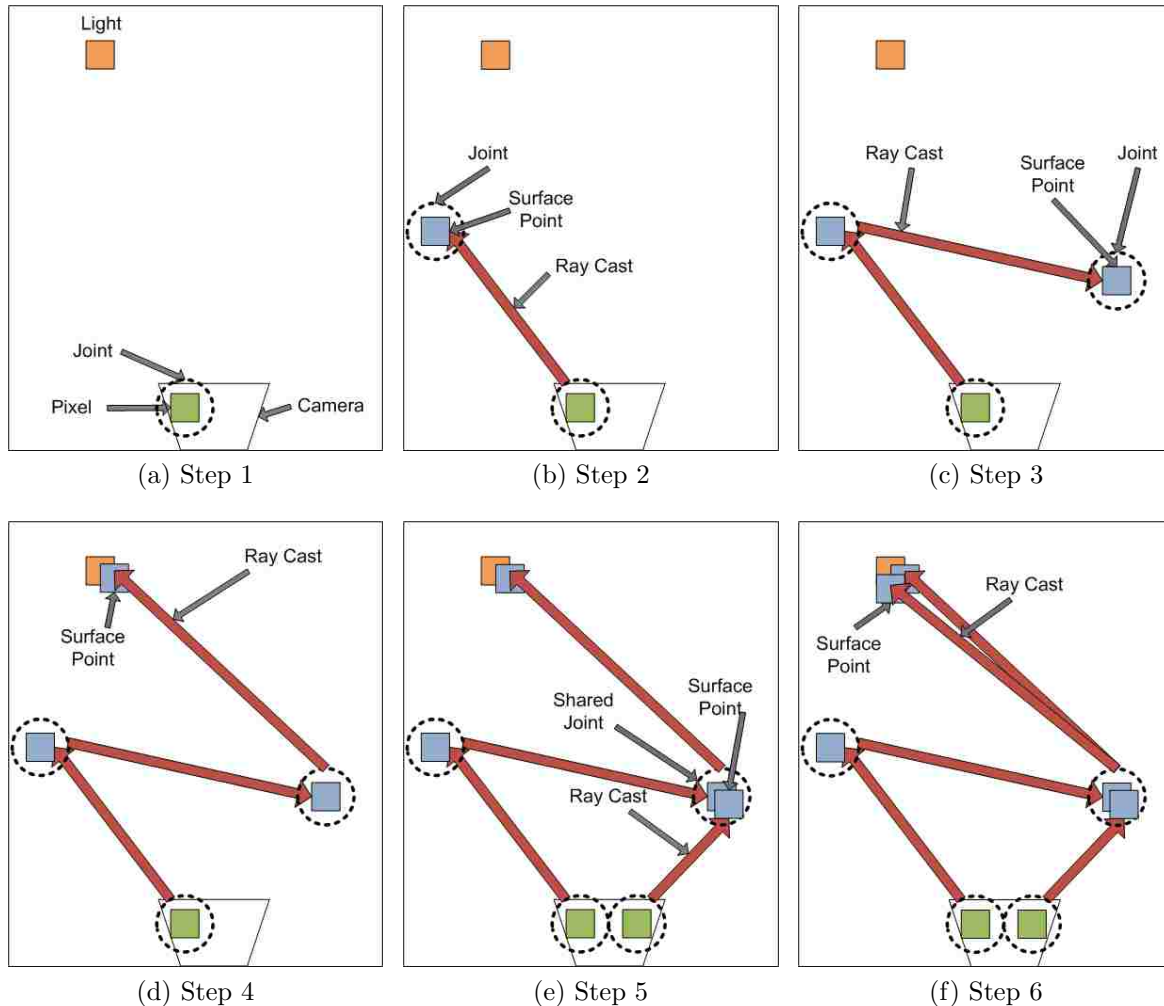
Figure 5.2: Basic pipe casting steps for the first pass. (a): A pixel object is generated with a new joint object. (b): A ray is cast. When it collides with a surface, a surface point is created. Since there is no joint in the vicinity, a new joint is created. (c): Another ray is cast. Again a surface point and joint are created where the ray collides with a surface. (d): A ray is cast towards the light source. Since the path is terminated at the light, just a surface point is created at this collision point. (e): Starting at a different pixel a new ray is cast. It collides within the radius of an existing joint, so the existing joint is used. (f): Another ray is cast towards a different point on the same light. The originating joint now has two rays towards a light. This means any light path through this joint will have access to both of these light rays when computing lighting information.

Algorithm 8 A detailed look at the recursive function in the second pass of pipe casting.

Given a surface point S, direction R, and depth D
if S is background **then**
    return background color
else if we have reached our light depth **then**
    return black
else if S is a light source **then**
    return the value of the light based on R
else
    Get joint J corresponding to S
    SumColor = 0
    for each surface point S2 in J **do**
        Calculate direction R2 to S2 at depth D - 1
        Recursively calculate the color contribution from S2
        Add color contribution of S2 to SumColor
    end for
    return average of SumColor
end if

Finding Nearby Joints

During the first pass of pipe casting whenever a surface point is generated it is assigned to the joint in that area. If left unoptimized, the search to find this joint would have a high computational cost. Our first optimization is to use a hash table to find joints. Since our hash table is large, this process has a time complexity of $O(1)$.

When pipe casting is initialized, a hash table is generated to store joints as they are created. When a surface point is generated, the algorithm hashes the collision location. This hash is used to index into the hash table. If a joint exists in the area of the surface point, it will be at that index. This significantly reduces the search time for existing joints and reduces the complexity. We tried a variety of hashing functions in our algorithm including one suggested in a real-time collision detection book [Eri05]. Our arbitrary hashing function significantly outperforms the other techniques we tried. The formula used is found in Equation 5.3.

$$x * 2 + y/2 + z/128 \; BINARY\_AND \; 1048575 \qquad (5.3)$$

Since there were 1048576 bins in the hash, the $BINARY\_AND$ 1048575 operation is the same as the MOD 1048576 operation. We used a $BINARY\_AND$ as opposed to a MOD since it is a faster operation.

The location of a surface point is rounded before its value is hashed. Rounding proved to be a beneficial technique since we want each joint to cover a small area of the scene. By rounding the value of each surface point before hashing, a joint represents the volume of points that all round to the same value.

Setting an arbitrary rounding constant is difficult since scene descriptions come in a variety of scales; some scenes define the surfaces on a scale of 0 to 1, others from 0 to 100, and others across an even larger area. To compensate, our algorithm found the bounding volume for the scene. The bounding volume was set to be a unit cube and each collision was translated into this space before rounding. To do the rounding, the x, y, and z components of each collision were rounded to the nearest 1/256 place (since using a denominator that is not a power of two led to unpredictable results).

The net result of this rounding inside a bounding volume is that there is space for exactly 256^3 joints in any given scene. The trade off with this constant was that a smaller constant meant that there would be more shared information but more bias from the rounding. A larger constant would mean less shared information but less bias from the rounding. Figure 5.3 shows the same image rendered at different rounding levels.

Cached Shared Information

In the course of developing pipe casting we experimented with a variety of ways of reusing information. The most successful technique involved caching additional information in the second pass by modifying the recursive function shown in Algorithm 8. Whenever the color contribution of a surface point is calculated, the surface point caches that value. When the function starts, it checks to see if it has already been queried at the same light depth. If the surface point has been queried for the same value before, it simply returns the cached value.

54

This decision means our second pass has a low complexity. Since each diffuse surface point will only calculate its contribution once for each light depth, the number of uncached queries will be at most $O(ld \cdot n)$. A uncached query will take at most $O(n)$ time to complete. Thus, with this caching scheme, the time complexity of the second pass is $O(ld \cdot n^2)$, a significant reduction from previous algorithm's $O(n^{ld})$ complexity.

The optimized function of Algorithm 8 is shown in Algorithm 9.

Algorithm 9 The final version of Algorithm 8 that uses cached information.

Given a surface point S, direction R, and depth D
if S is background **then**
   return background color
else if we have reached our light depth **then**
   return black
else if S is a light source **then**
   return the value of the light based on R
else
  if there is a cached value for S at D **then**
    return the cached value.
  else
    Get joint J corresponding to S
    SumColor = 0
    for each surface point S2 in J **do**
      Calculate direction R2 to S2 at depth D - 1
      Recursively calculate the color contribution from S2
      Add color contribution of S2 to SumColor
    end for
    Cache average of SumColor
    return average of SumColor
  end if
end if

5.3 Conclusion

We began this chapter by describing our implementation of path tracing. We then explained how we modified it to implement pipe casting. We also discussed several optimizations which further increased the speed of pipe casting. In Chapter 6 we give the results of our implementation.

55
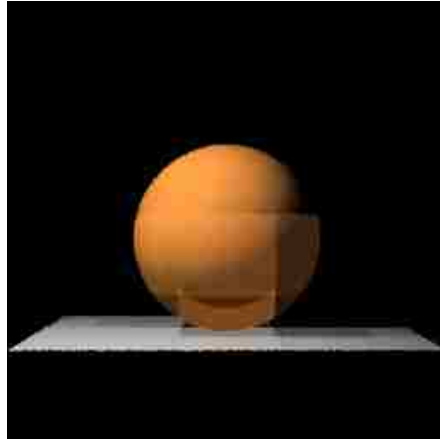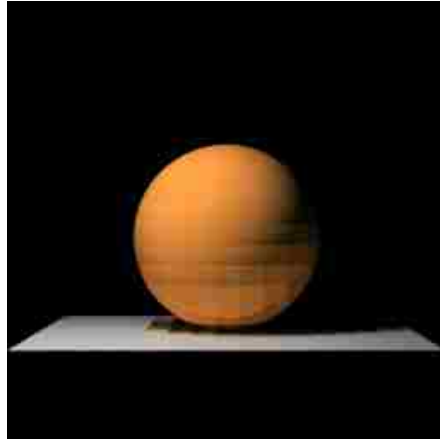
(a) 4^3 (b) 16^3
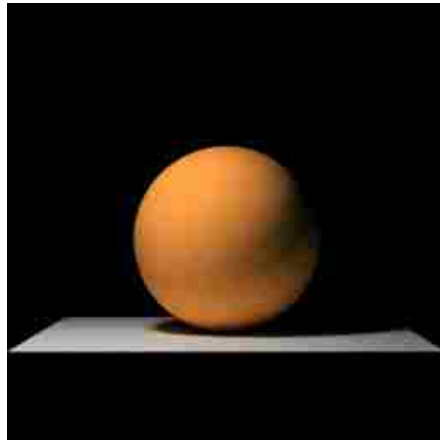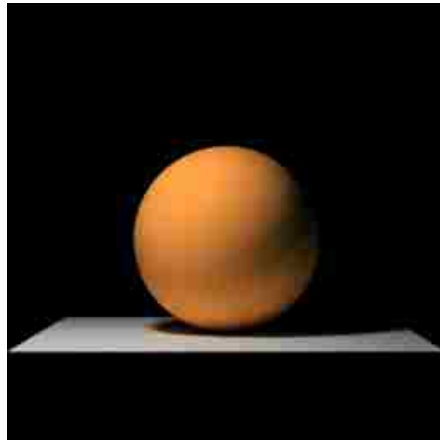
(c) 64^3 (d) 256^3
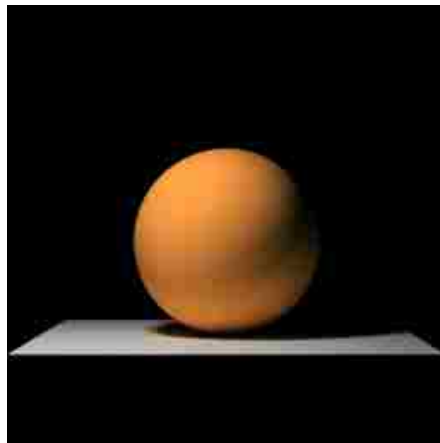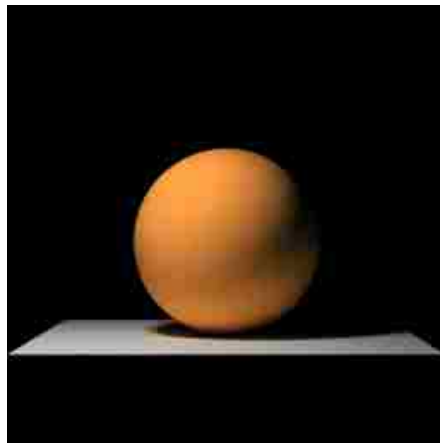
(e) 1024^3 (f) 4096^3

Figure 5.3: Comparison of rounding constants. Each image is labeled with the number of joints in the bounding volume. As the number of joints increases the amount of sharing decreases, but the accuracy increases. Notice that with a large rounding constant there are no visible lines on the boundaries between joints, but they take longer to converge. Also, there is no visible boundaries on the top half of the sphere since none of these points receive diffuse light. If there is no diffuse light, the constant does not affect the image.

Chapter 6

Results

A film studio, professional architect, or educator who uses photorealist algorithms would be benefited by an algorithm which renders images faster than current approaches. In Chapter 4 we described a graph-based algorithm which should accomplish this goal in theory. In Chapter 5 we described our implementation of a graph-based algorithm called pipe casting. In this chapter we give the results of our experiments that demonstrate empirically the speed increases realized by pipe casting. Rendering across a variety of scene descriptions, we have found that pipe casting renders scenes in about half the time required to render those same scenes with path tracing.

6.1   Our experiments

The goal of our experimentation was to determine whether pipe casting could render images at the same error level as path tracing in a significantly reduced amount of time. As an example of what this means, consider the two example results from our experiments shown in Figure 6.1. The figure shows two images, one rendered with path tracing and one rendering with pipe casting. Both images were rendered for approximately the same amount of time. By comparing the quality of these two images against an ideal image we can quantitatively determine which algorithm produces better images in the same amount of time. An alternative approach is shown in Figure 6.2. This shows the images from path tracing and pipe

casting which have arrived at a similar error level. Comparing times shows that pipe casting produced an image of the same quality in significantly less time.

To run these experiments, we gathered or developed 12 scene descriptions. In order to make these scenes representative of our problem domain, the scenes included all the material properties and lighting effects described in Chapter 2. Furthermore, some had low surface counts (as low as 5 in one scene) while others had a large number of surfaces (with more than 800,000 in the Stanford Dragon). The scenes also had varying light depths, going as low as 2 and as high as 15. We rendered each of these scene for upwards of fifteen hours to create ideal images.

All tests were run on Windows Vista with a 64-bit Intel Core2 Quad 2.34GHz CPU. The images rendered required approximately 7 to 8 gigabytes of RAM to render with Pipe Casting. We would expect Path Tracing to be in the hundreds of megabytes range or less.

As described in Chapter 4, we generated error metrics for path tracing and pipe casting by comparing their output against the ideal images. To make our experiment more detailed, we rendered each scene at a variety of different rays per pixel. This resulted in 240 final images for us to analyze.

On average, pipe casting rendered the images at the same level of error as path tracing in about 50% of the computation time. This matches our theoretical expectations of a significant speed-up from Chapter 4.

In this chapter we begin with a summary of our results. Following that, we show the details of each of our scene descriptions including graphs of all four major error metrics. We end the chapter by discussing the specifics of certain scenes which seemed to affect pipe casting's performance.

6.1.1   Summary of our Results

In this section we give a summary of the results of our experiments. In the next section we show the result for each scene description individually.

We designed our experiments to determine the speedup of pipe casting over path tracing by comparing speed and error. For each run of both algorithm we obtained a data point with two features—run time and L2 RGB error. For each data point generated by path tracing, we linearly interpolated between the two nearest data points from pipe casting to estimate how long it would take for pipe casting to generate exactly the same amount of error (for an example image, see Figure 6.3). For each image we averaged these results, which indicated how much faster pipe casting was on that image. In every case pipe casting out performed path tracing. We then averaged the results of these tests to give an overall speed-up for pipe casting.

Table 6.1 gives the results for our experiments. The table shows the 12 images used in our test suite. Two numbers are next to each image. The first number gives the ratio of rendering with pipe casting over path tracing. For example, the first image has a a value of 34%. This means that on average, the time to render with pipe casting was only 34% of the time required by path tracing. A value of 100% would indicated that pipe casting and path tracing ran at the same speed. Values below 100% indicate that pipe casting went faster.

We use this table to arrive at our average speed increase. By averaging the results of our experiments across all the images, we conclude that our algorithm runs in 52% of the time required by path tracing, which represents a significant improvement.

In Appendix A we give the details of the 240 renders done to collect our results.

(a) Path Tracing, 491 seconds, 128 rays/pixel, 1083.68 RGB L2 error

(b) Pipe Casting, 481 seconds, 76 rays/pixel, 570.275 RGB L2 error
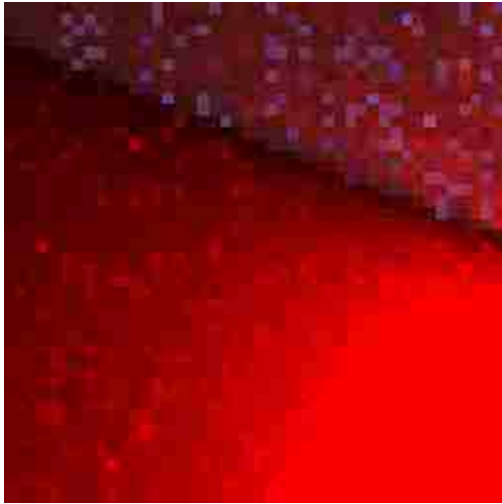
(c) Blow-up of Path Tracing
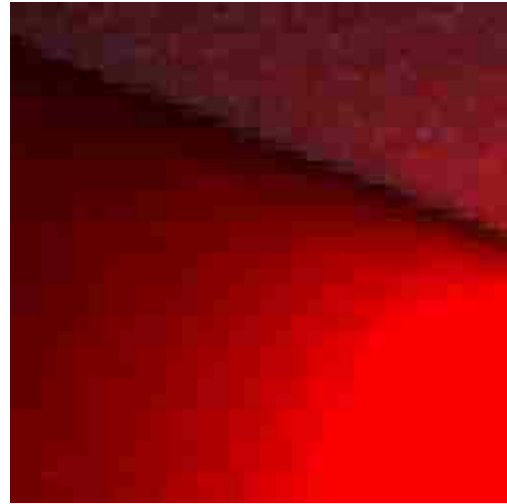
(d) Blow-up of Pipe Casting

Figure 6.1: Top: Results of path tracing and pipe casting after the same amount of time. Notice that the image produced by pipe casting has significantly less speckling, especially on the ceiling and floor. Bottom: Blow-up of the upper-left corner of both images. Notice that the pipe casting image is noticeably smoother.
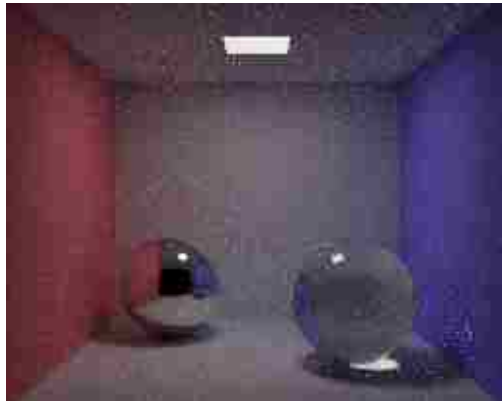
(a) Path Tracing, 435 seconds, 128 rays/pixel, 711.0028 RGB L2 error

(b) Pipe Casting, 198 seconds, 36 rays/pixel, 695.331 RGB L2 error

Figure 6.2: Results of path tracing and pipe casting with similar error results. Notice that the image produced by pipe casting was produced in significantly less than 50% of the time that path tracing took.
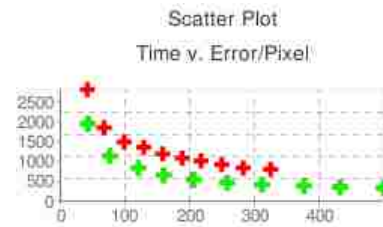
Figure 6.3: A figure showing example results for one scene. The final image shown on the left was rendered using path tracing and pipe casting. The line graph in the middle plots the number of rays cast against the L2 RGB error. Path tracing is in red, pipe casting is in green. Notice that in every case pipe casting has significantly less error for a given number of ray casts. The scatter plot on the right plots time against L2 RGB error. The better-preforming will run faster and have less error, i.e. it will plot lower and to the left. Notice that pipe casting, plotted in green, clearly outperforms path tracing, plotted in red.
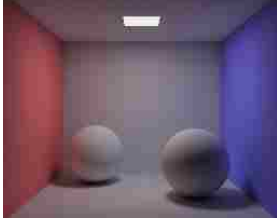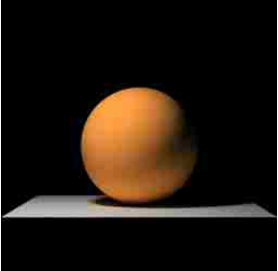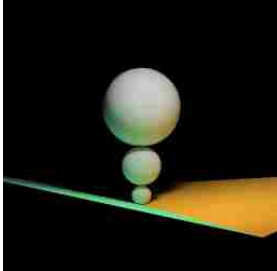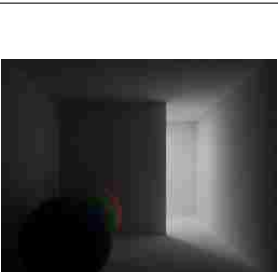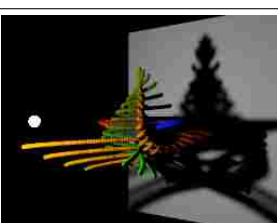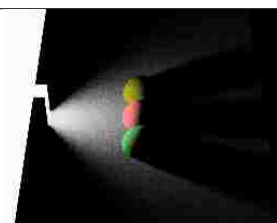
Image	Pipe Casting Time	Image	Pipe Casting Time
	34%		**50%**
	39%		**49%**
	52%		**53%**
	34%		**48%**
	56%		**72%**
	77%		**61%**
Average :	**52%**	Median:	**52%**

Table 6.1: Table showing thumbnails of the images in our experiments. The number next to each image shows the speed up relative to path tracing. If the number were .4, then pipe casting requires only 40% of the time required by path tracing to arrive at the same error level.

6.2   Strengths and Weaknesses

These results show that pipe casting is a superior algorithm across a wide variety of scene descriptions and light depths (see Figures 6.4, 7.1, 7.2). On average, pipe casting produces an image at the same level of error in about half the time required by path tracing. This significant improvement validates the theoretical use of a graph-based system. It also shows that pipe casting has real value as a global illumination algorithm. In this section we point out several of the specific advantages of pipe casting as well as its weaknesses.

Pipe casting performed particularly well on scenes which had deep diffuse bleeding. These scenes often included walls which reflected diffuse light back and forth several times. Pipe casting handled these well since it excels in reusing diffuse information. Since pipe casting can generate an exponentially larger number of rays from paths, diffuse patches looked particularly smooth since there was so much information to reuse.

Pipe casting also did well on images with reflections and transmissions which reflected or transmitted objects that appeared elsewhere in the scene. Since pipe casting reuses information, when an object appears in a reflection, all the information used originally to render the object is available. Thus, with pipe casting, reflections and transmission were noticeably smoother than in path tracing.

Although pipe casting outperformed path tracing in every scene description, this speed increase depended on the scene.

The slowest result was in the scene of "Spheres casting a complicated shadow". In this scene the light depth was only 2. In Chapter 4 we pointed out that at this low light depth, the computational complexity of a graph and tree-based algorithm would be on the same order. Thus, pipe casting's performance matches the theoretical expectation—it is still faster than path tracing which validates it's usefulness across a variety of scenes.

We would expect the performance of pipe casting to be worse than path tracing if there is no spatial coherence in the scene description. Since pipe casting is designed to speed up rendering by storing rays, the extra memory dedicated to storing those rays would not be

63
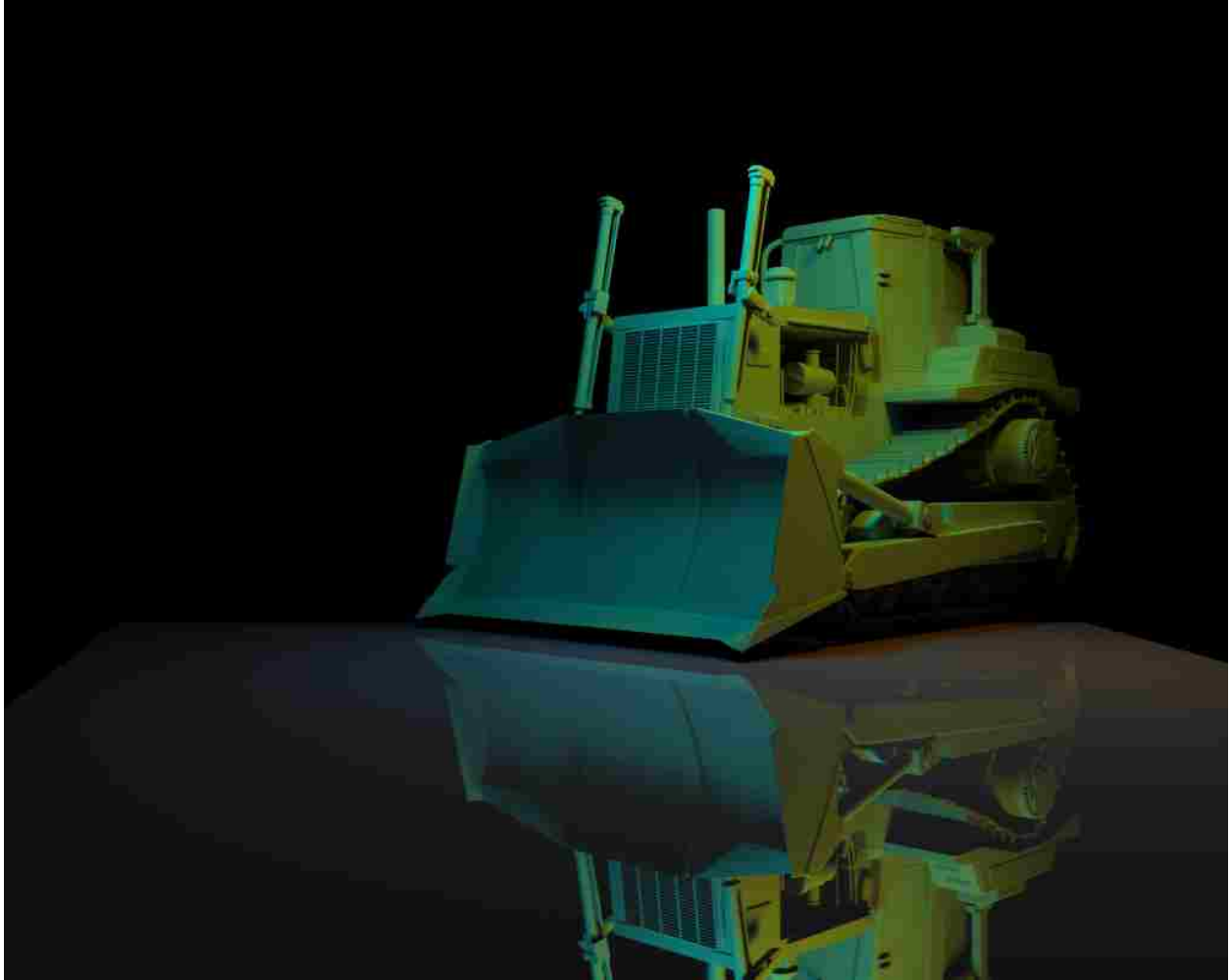
Figure 6.4: A large example of an image generated with pipe casting

used in scenes there is no valued in stored ray casts. Such scenes would need to have frequent and large changes in geometry such that a minute change in the angle of a ray coming from the camera would dramatically change where the ray collides. Such a degenerate case would be difficult to design and it is highly unlikely that such a scene would have any value other than for theoretical reasons.

Overall, the results of our experiments demonstrate that our implementation of a graph-based global illumination algorithm is superior to path tracing.

Chapter 7

Conclusion

We conclude this work by discussing future improvements to our pipe casting algorithm and then by exploring research areas which are affected by our findings.

7.1 Future Improvements

Pipe casting has proven to be an effective algorithm for improving the results of global illumination, but there are still areas where improvement can be made. The following sections address some of these issues.

7.1.1 Computational Complexity

The computational complexity measure presented in this work is able to judge the computational complexity of many modern global illumination algorithms. However, there are variations to these algorithms for which we have not been able to calculate a meaningful computational complexity. In a variation known as Russian Roulette algorithms, the maximum length of paths are not specified as part of the scene description. Instead, a probability is given. Whenever a ray collides with a surface, the probability is used to determine whether the path should continue or terminate. Calculating the computational complexity of this type of rendering variation would improve the usefulness of our computational complexity measure.

65
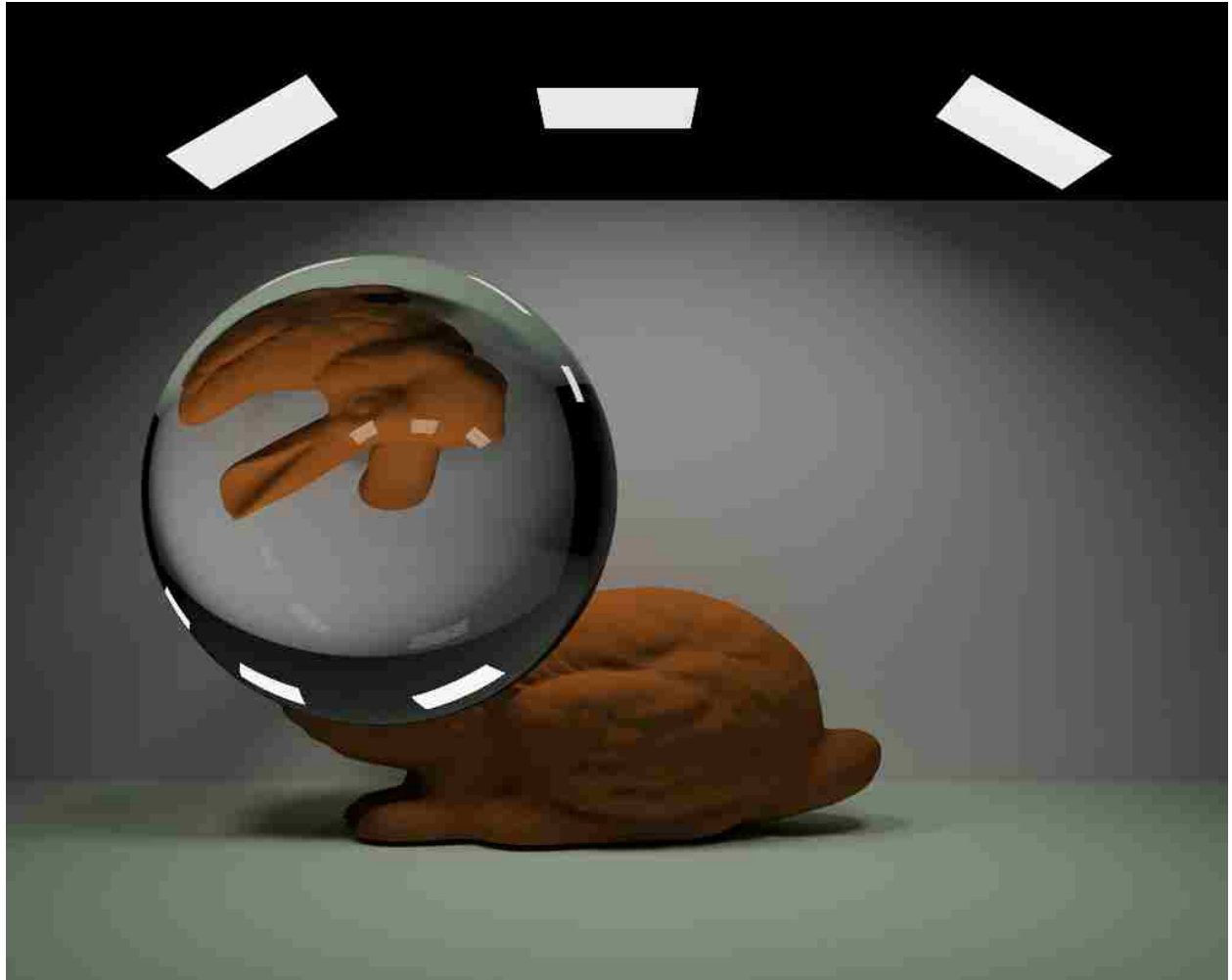
Figure 7.1: A large example of an image generated with pipe casting

7.1.2 Pipe Casting Implementation

In Chapter 6 we demonstrated that pipe casting is faster than path tracing in all of our scene descriptions. There are still areas in which we think our implementation could be improved. In this section we describe these areas.

Hashing

In the course of our implementation we used a variety of three dimensional hashing algorithms including one proposed in [Eri05], but no hashing algorithm was able to out-perform the one we originally used. The speed of pipe casting could be improved by even further investigation into hashing functions since our collision rate was higher than we expected.

Grid Choice

Part of pipe casting's speed increase is due to the use of the joint object which partitions the scene into a grid. Improvement in the design of this grid may also increase the speed of pipe casting. In our implementation the grid is uniform regardless of the distance a surface is from the view point. This can be a problem since the areas close to the view point are seen in the most detail and require the densest grid in order for features not to be blurry. Similarly, areas far away from the point of view can have a sparser grid without losing detail. If the grid adapted its size based on the distance from the viewpoint, perhaps more grid cells could be shifted forward, thus lowering the number of grid cells required. Also, the overall size of the scene affects the size of the grid cells since the grid is placed on the axis-aligned bounding box for the scene. As a result, all of our scene descriptions did not contain surfaces which extended in directions not visible to the virtual camera. A more robust grid algorithm would adjust the areas which have grid cells assigned based on which surfaces are visible to the camera.
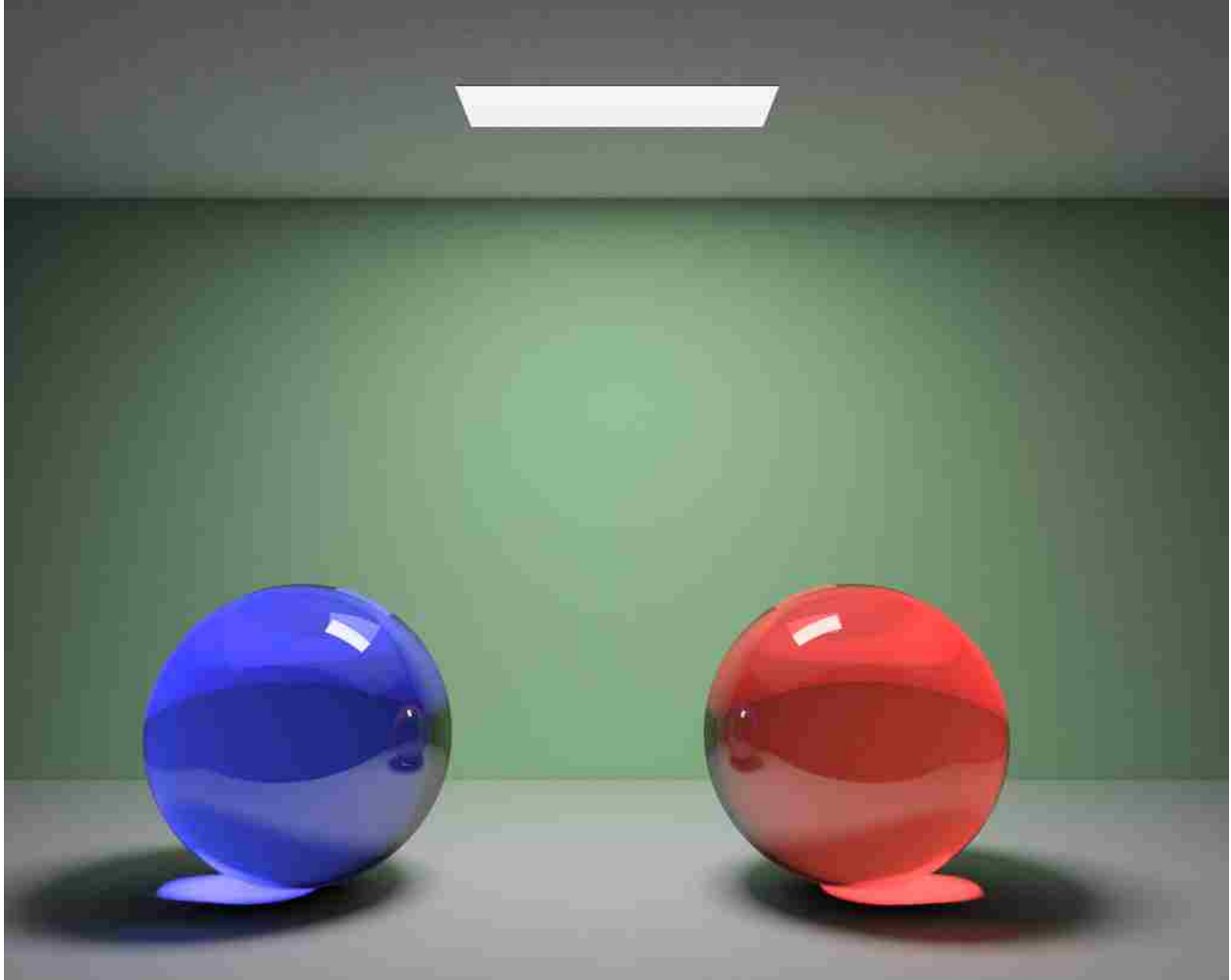
Figure 7.2: A large example of an image generated with pipe casting

7.2 Future Extensions

The research presented in this work is beneficial to applications other than just path tracing. As stated earlier, graph-based rendering presents a paradigm shift in how global illumination can be done. As such, it can have far reaching implications.

7.2.1 Animation

One particularly intriguing area of extension is in the area of animation. Our algorithm is designed to render individual images. Many applications, like animation, require the rendering of multiple images where the scene has not changed significantly. In most global

illumination algorithms the rendering process would have to start completely over with each new frame. However, pipe casting stores information about each scene as it renders. We believe that if there are only minor changes to each frame in an animation sequence, much of the information calculated by pipe casting could be reused, thus removing the need to restart the rendering process with each frame.

7.2.2   Other Global Illumination Algorithms

In the previous work section we discussed a variety of global illumination algorithms which build on the tree-based paradigm introduced with path tracing. We believe many of these algorithms, including modern algorithms like light cuts and metropolis light transport, can be improved by basing them on the graph-based approach introduced in this work.

7.2.3   Interactive Rendering

Another area where pipe casting could be useful is in interactive rendering. In this type of scenario, when a render is finished the animator using the image may find areas where more rays need to be sent to improve convergence. Since pipe casting can store all the information used to render, instead of starting the render over, the render could be continued with a high concentration of rays in certain areas.

Appendix A

Appendix

In this appendix we show the results for each individual scene description.

For each scene we ran path tracing and pipe casting at an increasing number of rays per pixel. For each image we ran four error metrics against the ideal image. For each of the images we show an example result, the average increase in performance of pipe casting over path tracing, and two sets of graphs. The first group of graphs shows how much error was present in each image as the number of rays per pixel increased. The second set of graphs are scatter plots showing how the error changed with total run time. We consider the scatter plots the most meaningful comparison since they address exactly what we are interested in: time vs. accuracy. In these graphs the superior algorithm will have lower time and error; graphically it will trend lower and to the left. As the results in the graphs show, pipe casting consistently out performed path tracing in a noticeable way.

A.1 The corner of a red room with a wall mirror

Here is the comparison of pipe casting and path tracing for the scene description of "The corner of a red room with a wall mirror". This is a scene of a red wall next to a blue-tinged mirror. Pipe casting does particularly well on this image since the wall in the reflection is already rendered. This means there is a large amount of information to reuse which pipe casting takes advantage of.

Figure A.1: The scene, "The corner of a red room with a wall mirror" generated by pipe casting

| Percent time to render the same quality as path tracing:	**0.3419384**
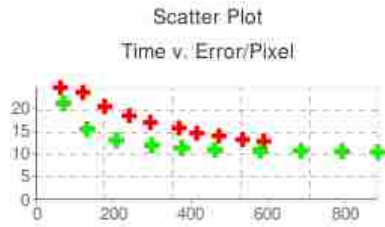
(a) Scatter Plot of Grayscale Error L1 v. Total Render Time
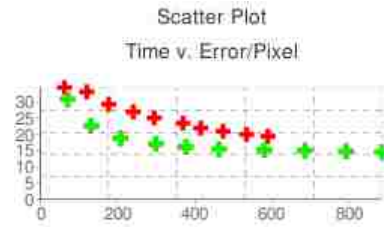
(b) Scatter Plot of RGB Error L1 v. Total Render Time
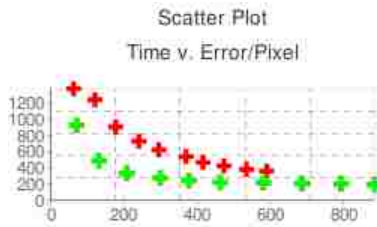
(c) Scatter Plot of Grayscale Error L2 v. Total Render Time
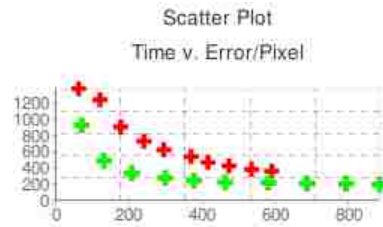
(d) Scatter Plot of RGB Error L2 v. Total Render Time

Figure A.2: Scatter Plot of different error metrics versus total render time. The green line represents the pipe casting results and the red line represents the path tracing results.

A.2 Diffuse spheres with colored walls

Here is the comparison of pipe casting and path tracing for the scene description of "Diffuse spheres with colored walls". This scene is an excellent example of diffuse bleeding. Notice how the sides of the spheres are either red or blue. This is a result of the light reflecting off the red and blue diffuse walls. Notice also that the shadows are either slightly red or blue as a result of this same effect.
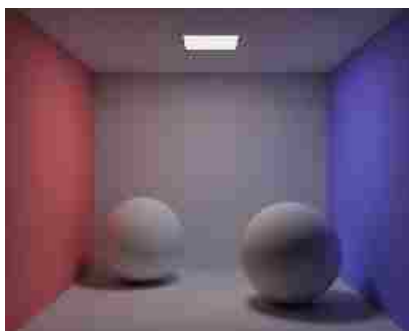
Figure A.3: The scene, "Diffuse spheres with colored walls" generated by pipe casting

| Percent time to render the same quality as path tracing:	**0.4981459**
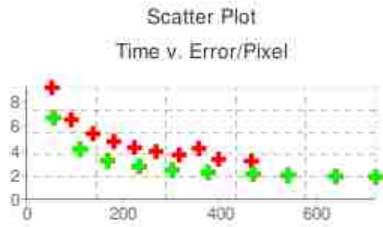
(a) Scatter Plot of Grayscale Error L1 v. Total Render Time
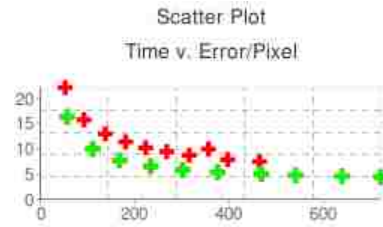
(b) Scatter Plot of RGB Error L1 v. Total Render Time
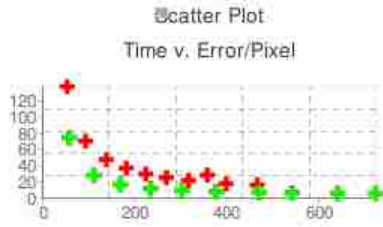
(c) Scatter Plot of Grayscale Error L2 v. Total Render Time
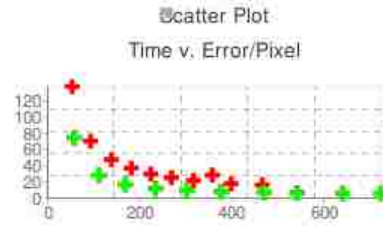
(d) Scatter Plot of RGB Error L2 v. Total Render Time

Figure A.4: Scatter Plot of different error metrics versus total render time. The green line represents the pipe casting results and the red line represents the path tracing results.

A.3 A reflective sphere shining on a transmissive sphere

Here is the comparison of pipe casting and path tracing for the scene description of "A reflective sphere shining on a transmissive sphere". This scene represents a particularly chaotic scene. There is diffuse bleeding as in previous images and also reflection and transmission. The sphere on the right is both reflective and transmissive. There are two caustics in this scene. One is on the floor cast by the light and one is on the wall cast by lighting reflecting off the reflective sphere through the transmissive sphere.
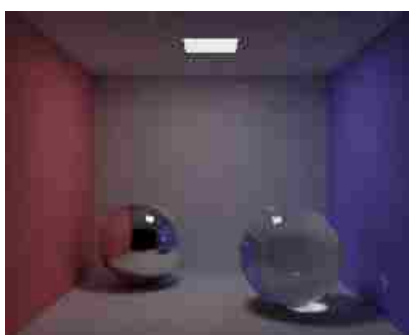
Figure A.5: The scene, "A reflective sphere shining on a transmissive sphere" generated by pipe casting

Percent time to render the same quality as path tracing:	**0.3880686**
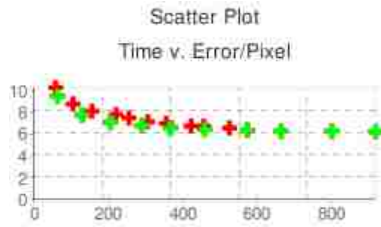
(a) Scatter Plot of Grayscale Error L1 v. Total Render Time
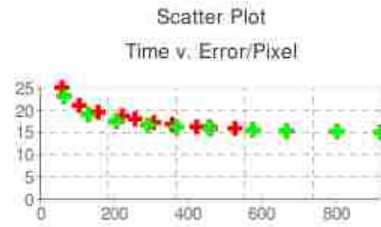
(b) Scatter Plot of RGB Error L1 v. Total Render Time
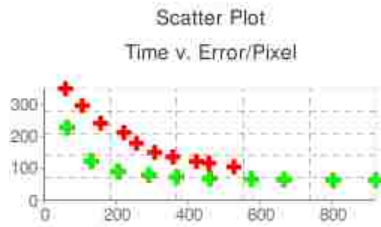
(c) Scatter Plot of Grayscale Error L2 v. Total Render Time
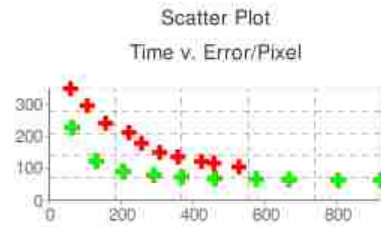
(d) Scatter Plot of RGB Error L2 v. Total Render Time

Figure A.6: Scatter Plot of different error metrics versus total render time. The green line represents the pipe casting results and the red line represents the path tracing results.

A.4 Colored transmissive spheres causing caustics

Here is the comparison of pipe casting and path tracing for the scene description of "Colored transmissive spheres causing caustics". This scene highlights diffuse bleeding and transmission. The angle between the spheres and the light causes strong caustics on the floor.
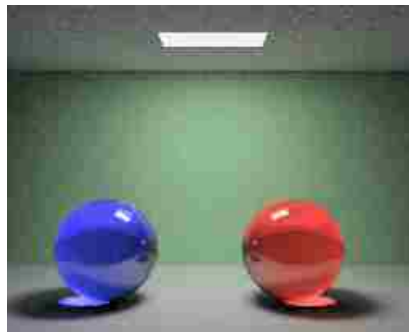
Figure A.7: The scene, "Colored transmissive spheres causing caustics" generated by pipe casting

| Percent time to render the same quality as path tracing:	**0.4945189**
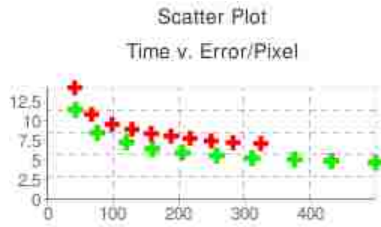
(a) Scatter Plot of Grayscale Error L1 v. Total Render Time
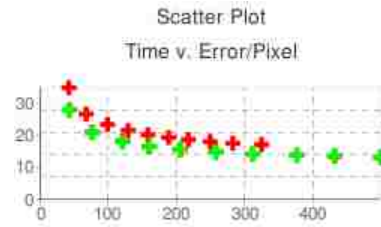
(b) Scatter Plot of RGB Error L1 v. Total Render Time
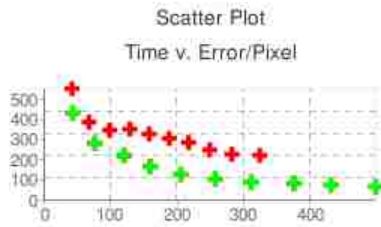
(c) Scatter Plot of Grayscale Error L2 v. Total Render Time
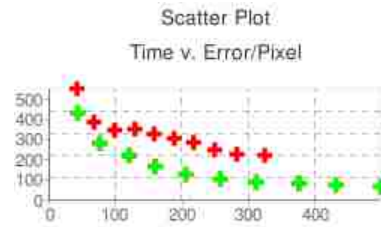
(d) Scatter Plot of RGB Error L2 v. Total Render Time

Figure A.8: Scatter Plot of different error metrics versus total render time. The green line represents the pipe casting results and the red line represents the path tracing results.
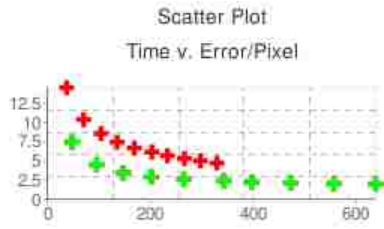
A.5 Still Life

Here is the comparison of pipe casting and path tracing for the scene description of "Still Life". This simple scene is an exercise in diffuse bleeding. Notice that the right side of the sphere is not directly visible to the light. It is not totally in shadow because light reflects off the shelf.
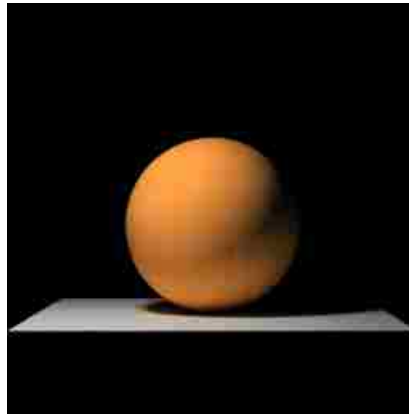
Figure A.9: The scene, "Still Life" generated by pipe casting

| Percent time to render the same quality as path tracing:	**0.5181965**
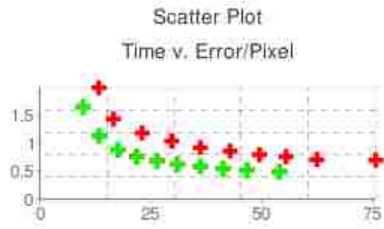
(a) Scatter Plot of Grayscale Error
L1 v. Total Render Time
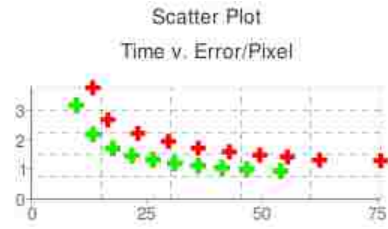
(b) Scatter Plot of RGB Error L1
v. Total Render Time
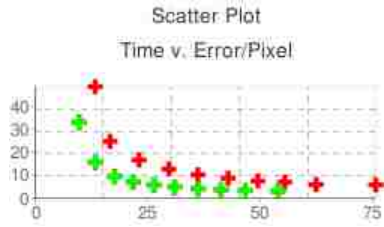
(c) Scatter Plot of Grayscale Error
L2 v. Total Render Time
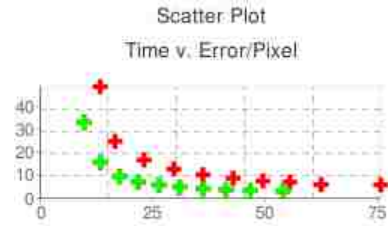
(d) Scatter Plot of RGB Error L2
v. Total Render Time

Figure A.10: Scatter Plot of different error metrics versus total render time. The green line represents the pipe casting results and the red line represents the path tracing results.

A.6 Diffuse Light on stacked Spheres

Here is the comparison of pipe casting and path tracing for the scene description of "Diffuse Light on stacked Spheres". This scene is another exercise in diffuse bleeding. Down the middle of the spheres the gold and green combine to create a white band.
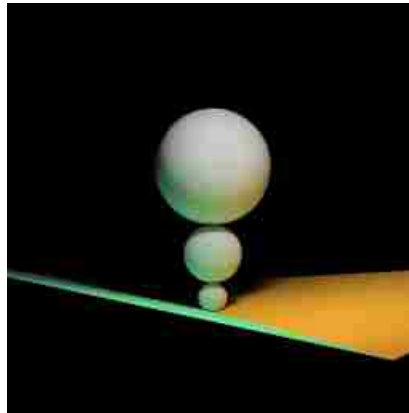
Figure A.11: The scene, "Diffuse Light on stacked Spheres" generated by pipe casting

Percent time to render the same quality as path tracing:	**0.5264266**
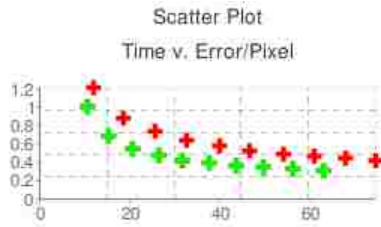
(a) Scatter Plot of Grayscale Error
L1 v. Total Render Time
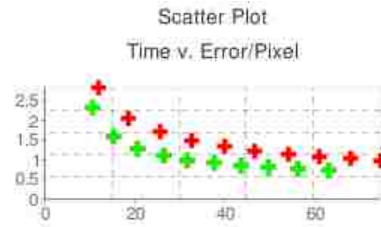
(b) Scatter Plot of RGB Error L1
v. Total Render Time
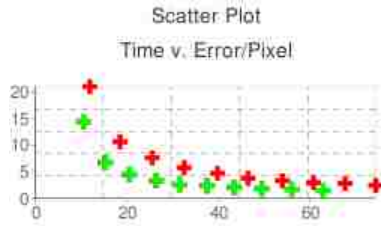
(c) Scatter Plot of Grayscale Error
L2 v. Total Render Time
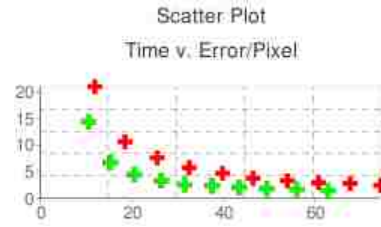
(d) Scatter Plot of RGB Error L2
v. Total Render Time

Figure A.12: Scatter Plot of different error metrics versus total render time. The green line represents the pipe casting results and the red line represents the path tracing results.

A.7 Colored spheres being lit indirectly

Here is the comparison of pipe casting and path tracing for the scene description of "Colored spheres being lit indirectly". This scene highlights the power of global illumination algorithms. Notice that none of the spheres are visible to the light source. However, they are illuminated as the light reflects off the walls, sometimes after taking several bounces.

Figure A.13: The scene, "Colored spheres being lit indirectly" generated by pipe casting

| Percent time to render the same quality as path tracing:	**0.3369418**

(a) Scatter Plot of Grayscale Error
L1 v. Total Render Time
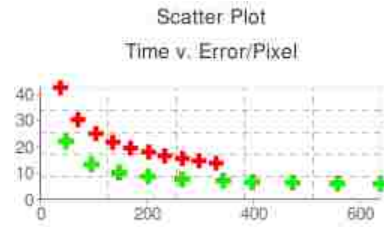
(b) Scatter Plot of RGB Error L1
v. Total Render Time
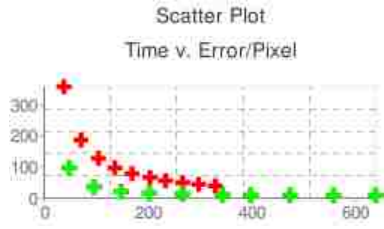
(c) Scatter Plot of Grayscale Error
L2 v. Total Render Time
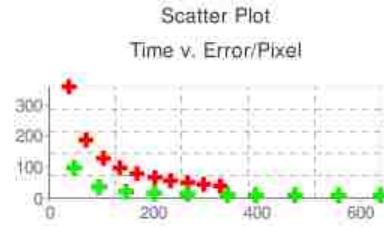
(d) Scatter Plot of RGB Error L2
v. Total Render Time

Figure A.14: Scatter Plot of different error metrics versus total render time. The green line represents the pipe casting results and the red line represents the path tracing results.

A.8   Boxes in a room with colored walls

Here is the comparison of pipe casting and path tracing for the scene description of "Boxes in a room with colored walls". This classic scene demonstrates the realism of diffuse bleeding. Notice that the shadows are far from gray. Notice also that the front of the right box has a complex lighting pattern due to its relative angle to the light and edge of the floor.

Figure A.15: The scene, "Boxes in a room with colored walls" generated by pipe casting

| Percent time to render the same quality as path tracing:	**0.4836108**
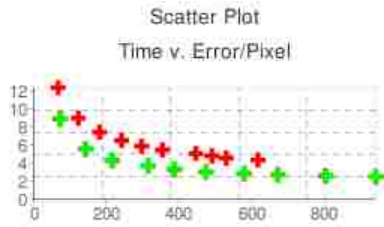
(a) Scatter Plot of Grayscale Error
L1 v. Total Render Time
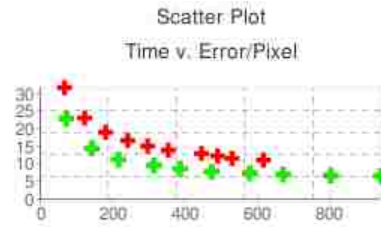
(b) Scatter Plot of RGB Error L1
v. Total Render Time
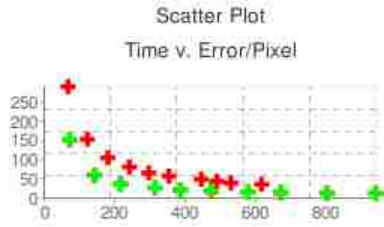
(c) Scatter Plot of Grayscale Error
L2 v. Total Render Time
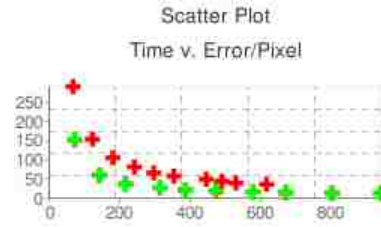
(d) Scatter Plot of RGB Error L2
v. Total Render Time

Figure A.16: Scatter Plot of different error metrics versus total render time. The green line represents the pipe casting results and the red line represents the path tracing results.

A.9 The Stanford Dragon

Here is the comparison of pipe casting and path tracing for the scene description of "The Stanford Dragon". This model of the Stanford Dragon has over 800,000 faces. Each scale on the dragon is represented by several hundred polygons. In a scene like this a reduction in the number of ray casts is significantly noticeable.

Figure A.17: The scene, "The Stanford Dragon" generated by pipe casting

Percent time to render the same quality as path tracing:	**0.5633115**
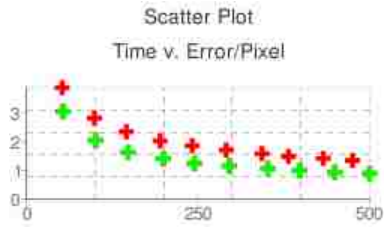
(a) Scatter Plot of Grayscale Error
L1 v. Total Render Time
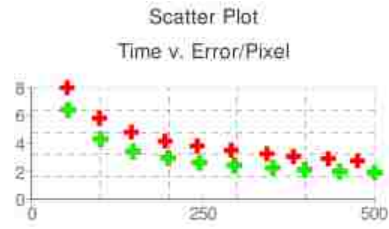
(b) Scatter Plot of RGB Error L1
v. Total Render Time
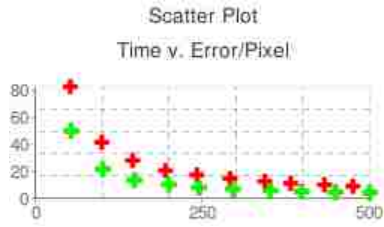
(c) Scatter Plot of Grayscale Error
L2 v. Total Render Time
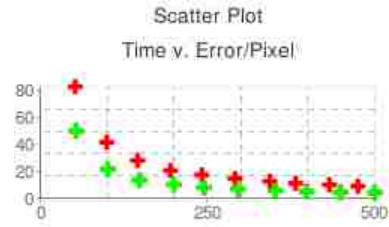
(d) Scatter Plot of RGB Error L2
v. Total Render Time

Figure A.18: Scatter Plot of different error metrics versus total render time. The green line represents the pipe casting results and the red line represents the path tracing results.

A.10 A bunny whose head is flipped in the sphere

Here is the comparison of pipe casting and path tracing for the scene description of "A bunny whose head is flipped in the sphere". This scene highlights the capacity of curved transparent surfaces to flip the objects behind them. Since the sphere is also reflective, there is a repeating light pattern visible in the sphere.
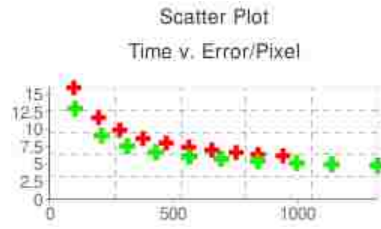

Figure A.19: The scene, "A bunny whose head is flipped in the sphere" generated by pipe casting
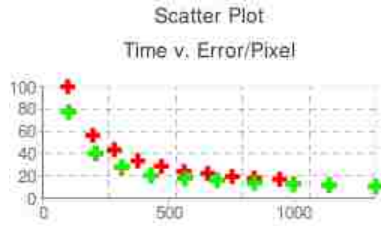
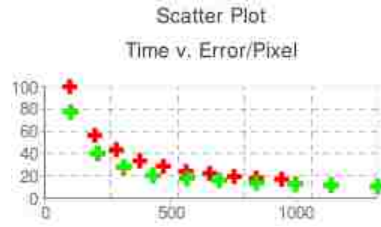| Percent time to render the same quality as path tracing:	**0.7176586**
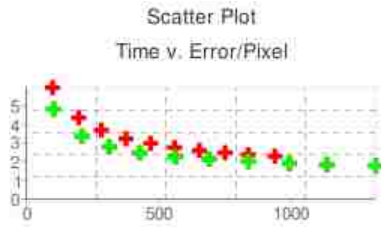
(a) Scatter Plot of Grayscale Error L1 v. Total Render Time

(b) Scatter Plot of RGB Error L1 v. Total Render Time

(c) Scatter Plot of Grayscale Error L2 v. Total Render Time

(d) Scatter Plot of RGB Error L2 v. Total Render Time

Figure A.20: Scatter Plot of different error metrics versus total render time. The green line represents the pipe casting results and the red line represents the path tracing results.

A.11 Spheres casting a complicated shadow

Here is the comparison of pipe casting and path tracing for the scene description of "Spheres casting a complicated shadow". The shadow cast by the spheres in this scene is extremely complicated. It is made more complicated by the fact that the light has large area. In order to converge on the right shadow, numerous samples must be taken for each pixel.
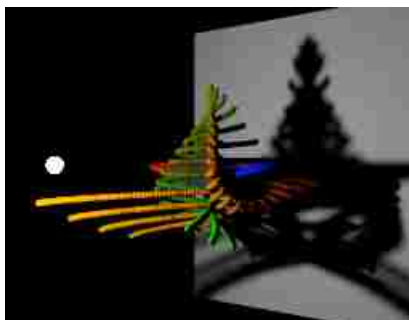

Figure A.21: The scene, "Spheres casting a complicated shadow" generated by pipe casting

Percent time to render the same quality as path tracing:	**0.7731773**

(a) Scatter Plot of Grayscale Error L1 v. Total Render Time

(b) Scatter Plot of RGB Error L1 v. Total Render Time

(c) Scatter Plot of Grayscale Error L2 v. Total Render Time

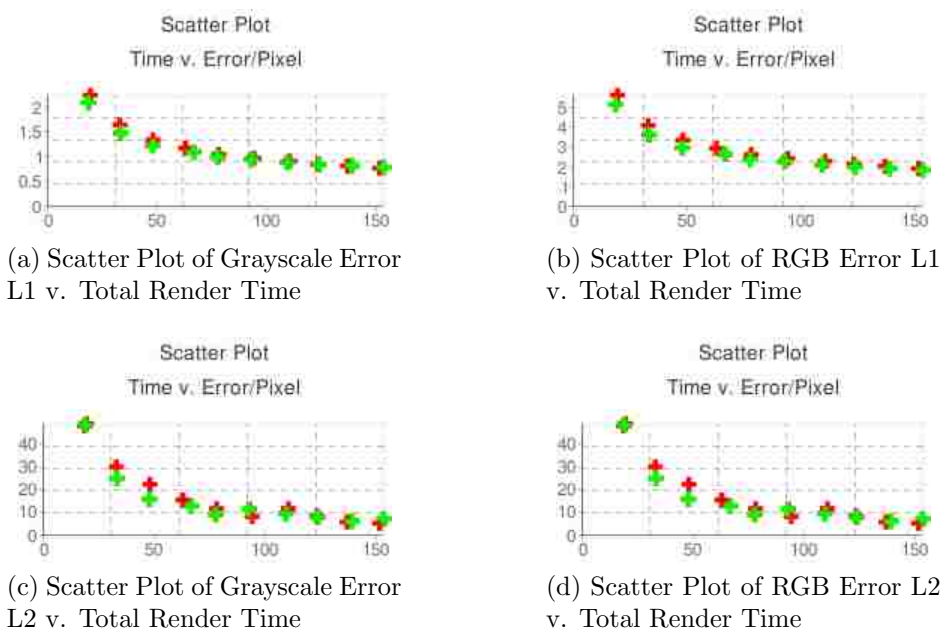
(d) Scatter Plot of RGB Error L2 v. Total Render Time

Figure A.22: Scatter Plot of different error metrics versus total render time. The green line represents the pipe casting results and the red line represents the path tracing results.

A.12 Spheres lit by an area light visible through a slit.

Here is the comparison of pipe casting and path tracing for the scene description of "Spheres lit by an area light visible through a slit.". In this scene an area light is blocked by a slit. During the direct lighting step in rendering the algorithm has no idea that the light will only be visible through the slit. Again, this requires numerous samples to converge. Pipe casting's ability to reuse information makes it highly effective at images of this type.
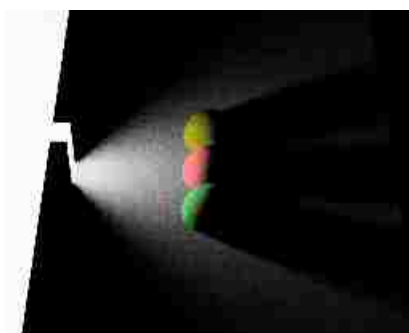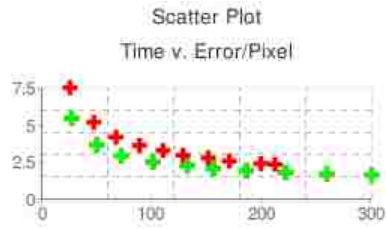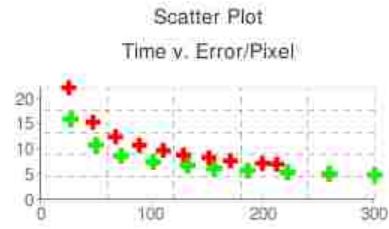

Figure A.23: The scene, "Spheres lit by an area light visible through a slit." generated by pipe casting
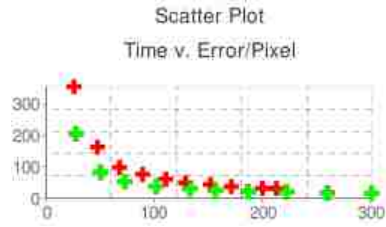
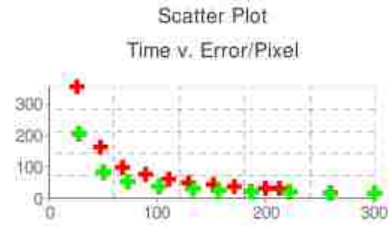| Percent time to render the same quality as path tracing:	**0.614825**

(a) Scatter Plot of Grayscale Error L1 v. Total Render Time

(b) Scatter Plot of RGB Error L1 v. Total Render Time

(c) Scatter Plot of Grayscale Error L2 v. Total Render Time

(d) Scatter Plot of RGB Error L2 v. Total Render Time

Figure A.24: Scatter Plot of different error metrics versus total render time. The green line represents the pipe casting results and the red line represents the path tracing results.

References

[BSH02]   BEKAERT P., SBERT M., HALTON J.: Accelerating path tracing by re-using paths. *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering* (2002), pp. 125–134.

[CCWG88]   COHEN M. F., CHEN S. E., WALLACE J. R., GREENBERG D. P.: A progressive refinement approach to fast radiosity image generation. *SIGGRAPH 22*, 4 (1988), pp. 75–84.

[Coo86]   COOK R. L.: Stochastic sampling in computer graphics. *ACM Transactions on Graphics 5*, 1 (1986), pp. 51–72.

[CTE05]   CLINE D., TALBOT J., EGBERT P.: Energy redistribution path tracing. *ACM Transactions on Graphics* (2005), pp. 1186–1195.

[DBB03]   DUTRÉ P., BEKAERT P., BALA K.: *Advanced Global Illumination*. AK Peters, Ltd., 2003.

[DW85]   DIPPÉ M. A. Z., WOLD E. H.: Antialiasing through stochastic sampling. *SIGGRAPH 19*, 3 (1985), pp. 69–78.

[Eri05]   ERICSON C.: *Real-Time Collision Detection*. Morgan Kaufmann Publishers, 2005.

[GTGB84]   GORAL C. M., TORRANCE K. E., GREENBERG D. P., BATTAILE B.: Modeling the interaction of light between diffuse surfaces. *SIGGRAPH 18*, 3 (1984), pp. 213–222.

[HOJ08]   HACHISUKA T., OGAKI S., JENSEN H. W.: Progressive photon mapping. *ACM Transactions on Graphics 27*, 5 (2008), pp. 1–8.

[Jen01]   JENSEN H. W.: *Realistic Image Synthesis Using Photon Mapping*. A K Peters, Ltd., 2001.

[Kaj86]   KAJIYA J. T.: The rendering equation. *SIGGRAPH* (1986), pp. 143–150.

[Sch03]   SCHREGLE R.: Bias compensation for photon maps. *Computer Graphics Forum* (2003), vol. 22, pp. 729–742.

[SKS02]   SLOAN P.-P., KAUTZ J., SNYDER J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *SIGGRAPH* (2002), pp. 527–536.

[VG95]   VEACH E., GUIBAS L. J.: Optimally combining sampling techniques for monte carlo rendering. *SIGGRAPH* (1995), pp. 419–428.

[VG97]   VEACH E., GUIBAS L. J.: Metropolis light transport. *SIGGRAPH* (1997), pp. 65–76.

[WABG06]   WALTER B., ARBREE A., BALA K., GREENBERG D. P.: Multidimensional lightcuts. *SIGGRAPH* (2006), pp. 1081–1088.

[WFA*05]   WALTER B., FERNANDEZ S., ARBREE A., BALA K., DONIKIAN M., GREEN-BERG D. P.: Lightcuts: a scalable approach to illumination. *ACM Transactions on Graphics 24*, 3 (2005), pp. 1098–1107.