Optimized ray tracing for real time use without hardware accelerators

Tad DiDio

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Michael S. Hsiao, Chair

Jeffrey Scot Ransbottom

Amos Lynn Abbott

May 08, 2025

Blacksburg, Virginia

Keywords: Real time Rendering, Ray Tracing

Copyright 2025, Tad DiDio

Optimized ray tracing for real time use without hardware accelerators

Tad DiDio

ABSTRACT

Many real time applications such as games or animation engines wish to simulate realistic light transport to present the user with streams of high fidelity images. For decades rasterization was the rendering technique of choice because it is fast and produces high quality images. However, in recent years graphics hardware has begun including specialized cores to accelerate ray tracing, an alternative rendering solution to rasterization and a type of light transport simulation. Ray tracing is generally the more desirable choice due to its ability to present images with higher fidelity than rasterization, and due to advances in graphics hardware it can now be simulated in real time. However, with the price increases of graphics processing hardware outpacing their performance increases, and with many other devices still not including this specialized hardware due to power constraints, we present an optimized model which uses ray tracing to produce a higher quality output than that of a raster engine while at the same time not requiring the use of specialized hardware accelerators to achieve real time frame rates. In this thesis we present two experiments we performed in pursuit of this goal, one which simulated global illumination but could not maintain reasonable levels of accuracy, and another which simulated lighting, shadows, and reflections and produced images which closely resembled our reference model while achieving consistent real time frame rates.

Optimized ray tracing for real time use without hardware accelerators

Tad DiDio

GENERAL AUDIENCE ABSTRACT

There are many use cases for which a real time rendering engine is needed. Realistic image synthesis is a common goal of such rendering engines and this normally involves simulating how light interacts with a virtualized three dimensional scene. Most techniques which can realistically approximate light transport in a scene are far too computationally expensive to run in real time. However, recent advances in graphics processing hardware have begun commercializing the use of ray tracing, a form of light transport simulation, for real time use. While this is a significant step forward, many devices still do not contain the specialized hardware needed to accelerate ray tracing. In this thesis, we present two experiments which work towards the goal of finding a compromise between rendering speed and quality using ray tracing. Our first experiment considers caching methods to speed up runtime calculations while our second produced a model which can synthesize an image that displays characteristics of a ray tracing engine while also running in real time without the use of specialized hardware accelerators.

Dedicated to Virginia Tech.

Acknowledgments

I would like to extend my sincere gratitude to all those who assisted me during this research. A special thanks goes to Dr. Michael Hsiao, my committee chair. Without his guidance and insight this work could not have happened. In addition, I want to thank my committee members, Dr. Scot Ransbottom and Dr. Lynn A. Abbott, both of whom were instructors of classes I took at various points during my education. I also would like to thank my family and friends who supported my journey as well as the Virginia Polytechnic Institute and State University and it's faculty for providing an excellent place of learning and facilitating my personal and academic growth. Thank you to all who have helped me reach this milestone.

Contents

List of Figures ix

List of Tables xii

1 Introduction 1

2 Background 8

2.1 Ray tracing Background 8

2.1.1 Scene Definition 9

2.1.2 Ray Bounces 11

2.1.3 BRDF and the Rendering Equation 15

2.1.4 Relevant Rasterization Techniques 16

2.2 Common Problems 17

2.2.1 Noise 17

2.2.2 Other Problems 18

2.3 Lumen in Unreal Engine 5 19

2.4 Ground Truth Model 19

2.4.1 Motivation 20

2.4.2 Implementation 21

3 Irradiance Caching **22**

3.1 Motivation . . . 23

3.2 Implementation . . . 24

3.2.1 Cache Layout . . . 25

3.2.2 Cache Sampling . . . 26

3.2.3 Cache Building . . . 28

3.3 Results . . . 30

3.3.1 Cache Build Times and Memory Usage . . . 32

3.3.2 Render Results . . . 35

3.4 Insights . . . 39

4 Hybrid Approach **41**

4.1 Motivation . . . 41

4.2 Implementation . . . 42

4.2.1 Core Render Loop . . . 43

4.2.2 Post Processing . . . 53

4.3 Results . . . 57

4.3.1 Comparison to Reference . . . 57

4.3.2 Limits . . . 61

4.3.3 Post Processing . . . 63

4.4 Insights 64

5 Conclusion **65**

5.1 Bounding Volume Hierarchy 65

5.2 Adding Global Illumination to the Hybrid Approach 66

Bibliography **68**

Appendices **72**

Appendix A Appendices I **73**

A.1 A1 73

A.2 A2 73

List of Figures

1.1 Comparison of GPU performance and price. 4

1.2 A comparison between Unity's built in renderer, our solution, and ground truth. 7

2.1 A comparison of soft and hard shadows. Note the blending between penumbra and umbra when using soft shadows. 10

3.1 A simple scene rendered by our ground truth path tracer demonstrating global illumination. Note that the sphere touching the ground plane has a white albedo, but is picking up the blue and red from the walls on either side. . . . 23

3.2 Mapping between a 2D plane and a 3D hemisphere which demonstrates texture wrapping. 26

3.3 A visualization of trilinear interpolation. 27

3.4 A comparison of cached irradiance using the method described in [5] (left) and our path tracer (right) on a probe with a 20x20 uniform resolution. The probe is enlarged for this demonstration and normally is a single point. Note how the red tint spreads naturally in our version towards the region facing the white wall where as the original solution does not observe the same. This is a direct result of our path tracer bouncing light realistically around the scene rather than stopping on the first diffuse surface hit. 29

3.5 One of the test environments used for this experiment (left) and an example visualizing the cached irradiance volume (right). Note that in these images the objects in the scene are rendered using Unity's built in renderer because we cannot render the debug cache in our pipeline, though it does not affect the cache construction in any way. 31

3.6 Comparison of scene rendered using cached irradiance at different resolution caches. 35

3.7 Comparison of scene rendered using cached irradiance with different number of samples. Note that the random seed was the same in all images so the structure of the noise is identical. 36

3.8 In this image, the scene was cached, then the hovering center sphere was added and rendered using the cached irradiance. It picks up the blue and red from the walls on either side, but the resolution of the 25x25 probe hemispheres is noticeable and creates artifacts. 37

3.9 A small sphere next to a large red wall. The red light reflecting onto the sphere is smoother than the other light which is a mix of sources and intensities. . . 38

3.10 Side by side comparisons of the results and the path traced reference. 39

4.1 Direct color of each object in the scene. Starting from this baseline rather than a black screen saves massive amounts of computation while incurring some sampling bias. 45

4.2 A single frame rendered using the direct color of each object weighted by raster style scene lights which function like point lights. 46

4.3 Comparison of scene rendered using a single pseudo random sample per light versus two. 48

4.4 A comparison of hard and soft shadows in two scenes. 49

4.5 Scene rendered with reflective surfaces. 51

4.6 Recursive reflections with a depth of 2. Note that when either of the spheres renders itself reflected on its surface, it now appears as a diffuse surface. Also note that the presence of noise in the scene does not affect the purely reflective surfaces. 52

4.7 Full noise mask creation pipeline. 55

4.8 Various threshold values. 56

4.9 Reference environment 1. 58

4.10 Reference environment 2. 59

4.11 Reference environment 3 (lit by offscreen light on the right). 60

4.12 Comparison of rendering a single frame from our model versus the reference model. 61

4.13 Geometry stress test with just under 21 thousand triangles. 62

4.14 Demonstration of the effect of the post processor. 63

List of Tables

3.1 Cache build results using path tracer with 1024 samples simulating 4 bounces while varying subdivision level and samples per axis (SPA). 32

3.2 Cache build results using diffuse radiance while varying subdivision level and samples per axis (SPA). 33

3.3 Radiance sample times using while varying samples per axis (SPA) with a subdivision level of 7. The path tracer traced 1024 samples with 4 bounces each per cached value. 34

4.1 Average frame rates in each reference environment with the path tracer simulating 2 samples per pixel with 4 bounces each. 58

Chapter 1

Introduction

For a long time, performing image synthesis on dynamic, three dimensional scenes in real time has relied on rasterization techniques due to performance constraints. Realistic light transport simulation, or fast approximations of it, was simply too expensive to compute in real time. Rasterization is fast and has benefited from decades of research making it the de facto choice for many interactive or real time applications. However, the desire for more realistic image synthesis always leads one away from rasterization and towards models which are able to more accurately reflect the way that light interacts in the physical world.

Many rasterizers fall short of the modern standard for realism because they implement local lighting and shading models. Every object is processed according to itself and the lights in the scene, without regard for how others may affect it or how it may later affect others. Global illumination, reflections, refractions, and shadows are all examples which share the requirement of information outside of the object being rendered to be realistically simulated. However, due to the local shading model employed by rasterization, realistic simulations of these features cannot be expressed easily and thus many models rely on crude approximations which easily break down outside of a carefully controlled environment.

However, in recent years, ray tracing, a form of light transport simulation which aims to simulate the behavior of light rays in the real world, has become popular due to the increased fidelity of the images it produces. Since it mimics the way that light behaves in the real world, a plethora of complex effects can be achieved such as global illumination, caustics,

reflections, and refractions. Even better, many desirable behaviors follow naturally from basic simulations such as color bleeding, ambient occlusion, and bloom.

One major enabling factor for the recent popularity of ray tracing in real time graphics has been the addition of hardware accelerators in graphics processing units (GPUs). GPUs had previously been designed specifically to handle rasterization problems, however with better technology available, many manufacturers are now providing ray-tracing specific cores such as NVIDIA's RT cores and AMD's ray accelerators. This allows modern programs to get hardware level support for expensive ray tracing algorithms, making them possible to compute and converge in real time.

Ray tracing is difficult because of the nature of trying to simulate light in a physical way. We know that the number of light rays emitted from a source is very large, so it is not tractable to simulate a fully realistic scene. Instead, many approximations are used to estimate the radiance of each point in space as seen by the camera. Due to this, noise is one of the most common and difficult problems in ray tracing because of the nature of approximating and discretizing a continuous spectrum in a computationally tractable way as well as because of the random nature of light interacting with diffuse surfaces. Many ray tracing engines that do not have to run in real time, such as Blender's Cycles engine, will iteratively add illumination passes over time to refine the image. This leads to a wholly more realistic image because with the drastically increased sample size, we can simulate more realistic models without worrying about computational time. However, since we cannot afford to do this in real time, sampling techniques must use multiple strategies to cull the workload down to a reasonable level while at the same time maintaining an acceptable degree of realism.

The computational complexity of such systems is due mainly to the rendering equation, which was originally presented in Immel et al. [7] and Kajiya [9] and has been widely adopted for its ability to generalize the depiction of a scene.

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega f(\mathbf{x}, \omega_i, \omega_o)\, L_i(\mathbf{x}, \omega_i)\, (\mathbf{n} \cdot \omega_i)\, d\omega_i \tag{1.1}$$

This equation will be further explained later, but the integral is impossible to solve analytically, so many times approximations are used, such as Monte Carlo integration, which uses random samples and computes an average. However, as the authors in [16] point out, many of these techniques that involve random samples come at the price of huge amounts of noise. There are a number of noise reduction techniques, but as mentioned in the work by Wright et al. [30], the problem usually occurs before the denoiser even gets the results, normally due to a computational budget cutting off the number of samples too early to allow the image to converge well.

From this, we see that we need a high number of samples to approximate the scene well and this is why hardware level support is generally required. The core of a ray tracing algorithm involves casting rays and testing for intersections in the scene to model where light rays hit. This operation is generally expensive and is often the part which is implemented in hardware when such a GPU is available.

However, this hardware support is typically absent from lower end machines like laptops and embedded processors such as the ones found in virtual reality headsets. Many laptops and headsets choose to prioritize power saving over performance and for this reason do not ship GPUs with ray tracing hardware. Many older desktops also cannot use ray tracing hardware because they do not contain graphics cards with the capability. For this reason, ray tracing on these types of devices is far less common. Many rendering engines which target low end hardware stick to rasterization because it generally out performs ray tracing in terms of computational complexity. However, because ray tracing is able to produce more realistic images it is generally more desirable to a vast population of consumers. In recent years,

there has been a strong correlation between hardware capability and cost, that being that as ray tracing cores are added to GPUs, both the monetary price and power consumption of the device increase leading to a situation which prevents many consumers from being able to purchase or utilize hardware capable of running modern ray tracing engines.

This problem is especially important because of the prevalent use of real time rendering engines in common applications such as video games and virtual or augmented reality programs. The inflation seen in the monetary value of graphics processing hardware in recent years makes ray tracing inaccessible to many people. As an example, Figure 1.1 shows performance benchmarks as reported by an online GPU benchmark and the current prices of each of the base 40 and 50 series GPUs released by NVIDIA. The price increase between the best GPUs of the previous and current generation releases, the 4090 and 5090, was 10% despite performance only increasing by slightly over 4%. We see this trend throughout Figure 1.1.

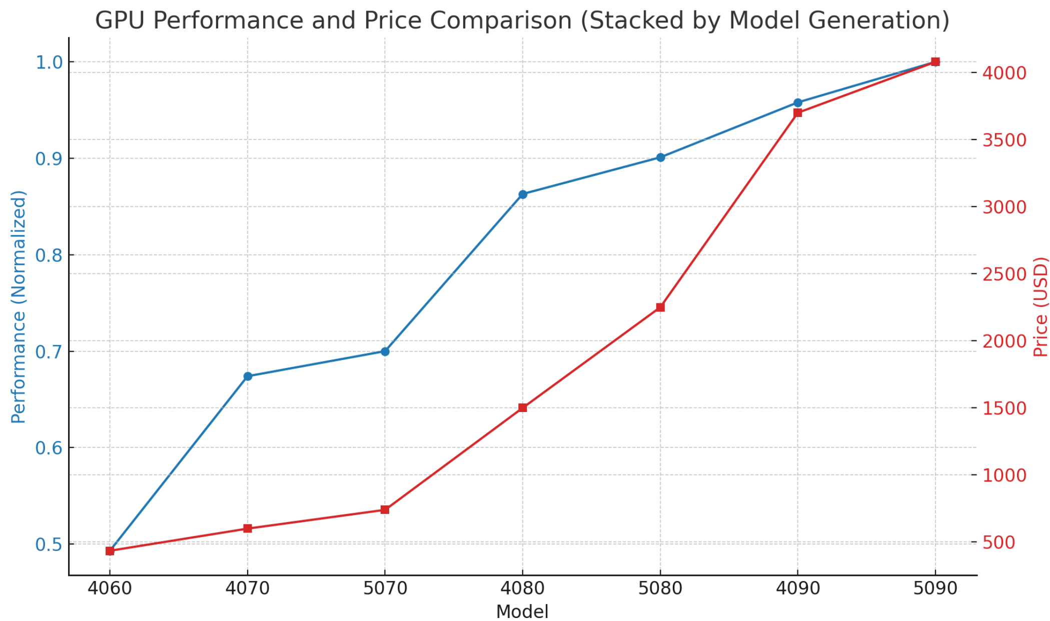


Figure 1.1: Comparison of GPU performance and price.

Because of these limitations, in this thesis we propose a custom approach which combines

elements from both rasterization and ray tracing to produce an image with increased visual fidelity but which converges in real time without relying on hardware accelerators. Our goal is to render three dimensional scenes at a higher quality than typical rasterization can achieve while at the same time keeping the computational complexity in the range of generic graphics hardware. Many of our techniques that will be presented favor speed over accuracy, but all aim to maintain an acceptable level of realism. The main contributions of the thesis will be described in Chapters 3 and 4. Our solution in Chapter 4 works to mitigate nearly all noise by using a combination of many techniques. It should be noted that our solution still relies heavily on the existence of a GPU due to the massively parallel nature of rendering and that performance will scale depending on the quality of the GPU being used.

At a high level, the process and application of our solution is as follows. First, we sample the color of a pixel directly to provide a lit basis from which to derive an approximation of the ray traced scene. Doing this allows us to remove a huge portion of noise at the expense of some inaccuracy because instead of relying on rays to fully light the scene, we start with an estimation and refine it rather than construct it. Next, we cast various lighting rays using importance sampling to reduce the number of rays which are wasted by missing light sources entirely. However, these lighting rays reintroduce some noise, so to combat that, we use a custom method to further refine the noisiest areas of the render. First, we generate a mask to determine where the noisiest areas are, then we send additional rays from those areas to attempt to converge the estimation of the irradiance faster. We dynamically decide how many rays should be sent to do further refinement based on the current frame rate so that we can ensure our system does not slow past the real time frame rate.

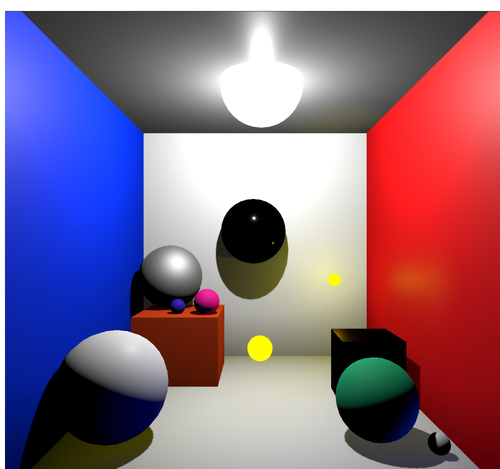
This thesis will be organized into five chapters, the first of which is this introduction. Chapter 2 will present all background information which is necessary to understand the problem and proposed solution in full. It will cover theory pertaining to ray tracing, including various

equations and approximations and will define terms which will be used throughout the thesis. It will also go into common issues which make ray tracing difficult and which motivated the specific solution to be presented, as well as explore various current techniques which are employed by commercial ray tracing engines to accelerate the computations. Finally, it will look at the implementation details and justifications of a custom model which we implemented and treated as ground truth throughout the research process. The custom model which will be covered is a path tracer and is not intended to run in real time, but can provide images which are close to complete physical accuracy.

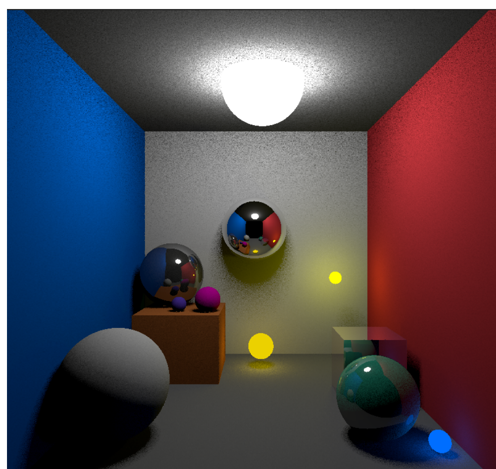
Chapter 3 will be dedicated to exploring the first solution which was attempted. The key idea of this solution was to spend time caching the results of the physically accurate path tracer, then to quickly estimate the incident light on a point based on the precomputed values. This solution ultimately failed to sythesize images with an acceptable level of quality, but it is technically interesting and contains elements which could inspire future work.

Chapter 4 will detail the final solution to result from this research. This solution leverages both rasterization and ray tracing techniques to present an image which has a higher fidelity than rasterization normally achieves. The images produced specifically achieve higher degrees of realism in the lighting, shadowing, and reflections because of the global context of the simulation while at the same time significantly reducing the amount of noise typical in a ray tracer at the cost of some accuracy. The main contribution of this research is the unique combination of various techniques from both ray tracing and rasterization algorithms to produce a high quality image in real time as well as our custom post processor which works to dynamically reduce noise in an image.

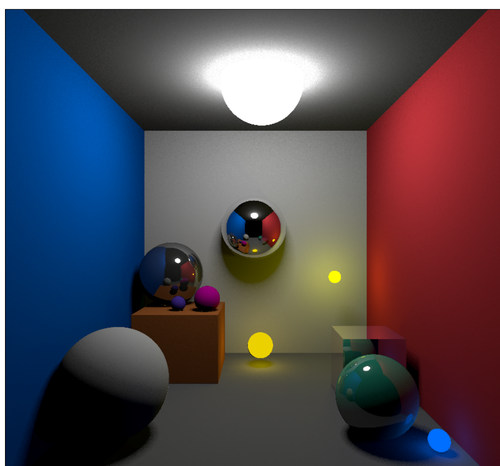
Finally, Chapter 5 will provide a conclusion and ideas for future work that could expand this topic. Figure 1.2 shows a comparison of static frames rendered by various different renderers, including Unity's built in solution, our solution, and our ground truth model.



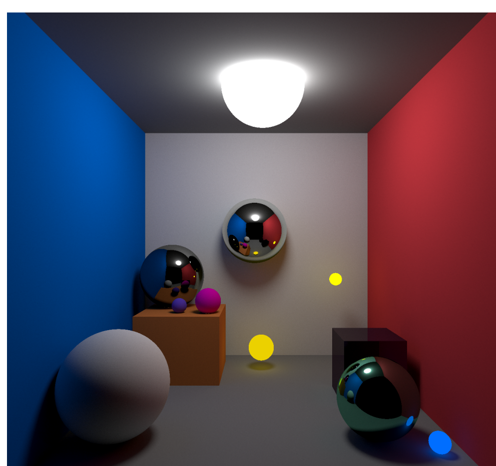
(a) Built in Unity renderer: 1.4(ms)



(b) Our solution after 1 frame: 4.3(ms)



(c) Our solution after 10 frames: 43(ms)



(d) Fully converged ground truth: 120(s)

Figure 1.2: A comparison between Unity's built in renderer, our solution, and ground truth.

Chapter 2

Background

Ray tracing is a rendering model that is heavily based on physical phenomena. Generally, it aims to simulate the way that light bounces around a scene, and as we will see, there are many variations on this core concept. In this chapter, we will explore all the necessary theory underlying modern ray tracing as well as look at current methods and commercial solutions. However, due to the strong increase in GPU abilities in the past decade, nearly all modern solutions will rely on this computing hardware. Since our goal is to synthesize a solution which does not rely on hardware accelerators, we will also introduce relevant rasterization techniques which we will make use of. In recent years, research on ray tracing that does not utilize hardware accelerators has become much less common; however, we will look at Unreal Engine 5's Lumen system which offers both a hardware and a software ray tracing pipeline to facilitate customers who do not have ray tracing hardware [30].

2.1 Ray tracing Background

Ray tracing is a general term for casting rays in a scene. In our context, these rays are normally querying material, surface, and lighting properties from a simulated three-dimensional environment. This section will describe how ray tracing works by giving information on the required elements and then giving background on some of the specific techniques which our solutions utilize.

2.1.1 Scene Definition

A scene is a three-dimensional (3D) representation which can be rendered by a computer. To define a scene in a language that the computer can understand, we must define geometry, lights, and an eye or camera (eye and camera will be used interchangeably). Geometry refers to all the things in the scene, such as boxes, spheres, cars, hills, or buildings. Lights are sources of illumination which will light up the scene by casting light on the geometry present. This is technically not how lights will work, but conceptually a fine model to understand a scene. Finally, the camera defines the perspective by which the user will view the scene.

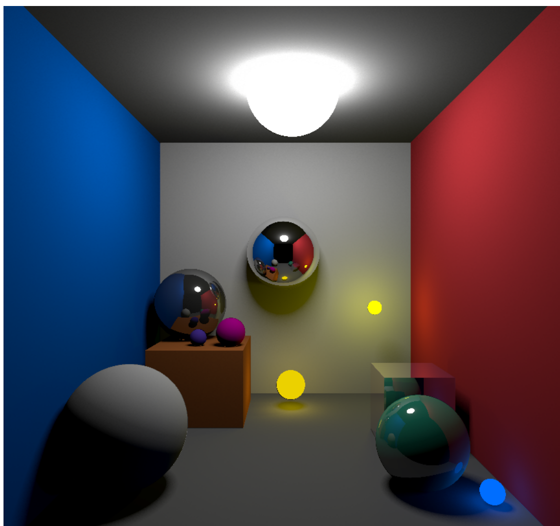
Geometry

The most common representation of geometry, and the one which this thesis uses, are triangle meshes. Triangles are the most basic unit of rendering because they cannot bend or warp in non-planar ways as quads can. This is useful because we can unambiguously define and interpolate data across the surface of a 3D object using barycentric coordinates which are a local coordinate system based on the vertices of a triangle [32]. By combining many triangles, we can create arbitrary shapes which with high enough resolution can achieve excellent approximations of smooth geometry.

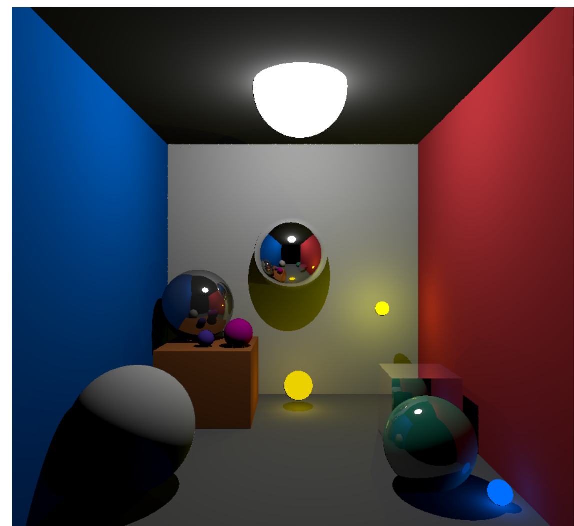
Lights

Simulated lights come in many forms. The most common are point lights, directional lights, spot lights, and area lights. Point lights are infinitely small points that illuminate geometry in a radius, while directional lights give constant-strength illumination to a scene in a given direction. These are approximations of an infinitely far away point light with an infinite radius. Spot lights are similar to point lights, but illuminate the scene in a specified cone

with the apex at its location. Area lights are the most important for us as they can be any shape or mesh and provide illumination only to points which have direct line of sight to part of the area light. Of the lights mentioned, only area lights can provide soft shadows since the illumination of a point depends on the number of rays which hit the area light rather than the a single ray binary decision. This allows for natural transitions between the umbra and penumbra due to light occlusions [21]; the umbra being a region fully in shadow and the penumbra being only partially in shadow.



(a) Scene rendered considering the surface area of lights



(b) Scene rendered considering lights as point lights

Figure 2.1: A comparison of soft and hard shadows. Note the blending between penumbra and umbra when using soft shadows.

Pinhole Camera

To view the scene, we need to define a camera which specifies the perspective, and we need to be able to project the 3D representation of the scene onto a two dimensional (2D) screen for display. A pinhole camera is a simple but effective way to do this. The pinhole model of a camera assumes that all light which is visible to the camera comes through a single point in

space and is projected onto an image plane [22]. This model of camera does not incorporate distortion or noise terms without modification, but this is generally a desirable trait as our goal is to render scenes as they would be perceived by a human in the real world.

Because of this projection, the image plane is upside down and backwards; however, if we consider the plane which is an equal distance to the focal length in front of the pinhole (the virtual image plane) we see a corrected image. We will use this plane for our calculations.

2.1.2 Ray Bounces

When light collides with objects in the real world, there are many ways that it can respond, and this is how we get a multitude of different effects. For our use case of rendering opaque objects, we will consider two main interactions, diffuse reflections and specular reflections. With just these two lighting interactions, we can simulate a believable scene where light scatters randomly and transfers energy to various surfaces.

In the physical world, light emits from sources and is registered by cameras when it bounces into the lens. Similarly, in our simulation, a surface will only be lit and shaded if we happen to trace a ray which travels from a light source then bounces off said surface and into the camera. With each bounce, the ray will lose energy according to the per-channel attenuation constant, or albedo, of the surface which is hit. A ray's energy is recorded as a color in red-green-blue (RGB) color space so that each channel can react independently to the simulated pigments of surfaces. If the ray finally hits the camera, we render the energy as a color to the screen to perform the projection from a 3D virtual scene to a 2D monitor.

One important optimization that is universally adopted is to generate rays from the camera, rather than from light sources. Though this is backwards to the physical way that light behaves, it drastically culls the amount of wasted computation. Rather than trying to land

a ray on the camera, we now test if it ends on a light source. If not, there are generally two cases. In case one, the ray exceeds it's bounce limit and collides with another surface. In this case, we either discard the ray, or, if there is light energy cached at this location, we interpolate what energy it might have picked up to gain some benefit from an other wise wasted computational path. In case two, the ray hits nothing and extends infinitely into space. Typically this is considered to be the sky and an ambient term is added to again salvage the computation. It also allows us to affect the render based on a simulated sky which can increase the fidelity as well.

Diffuse Reflectance

Lambertian reflectance is a powerful model of diffuse scattering which simulates rough surfaces that cause light rays to bounce randomly in a hemisphere around the normal direction upon collisions [12]. This reflectance model assumes ideal diffuse surfaces in which the light scatter is uniformly distributed and is independent of the viewing angle. Derived from Lamberts' cosine law, Equation 2.1 models Lambertian reflectance

$$L_o = \rho L_i(\omega_i) \cos(\theta_i) \tag{2.1}$$

where L_o is the uniform radiance in all directions given a collision in which the angle between the surface normal and the incident lighting ray is θ_i, the incoming radiance from direction ω_i is L_i, and the surface reflectance coefficient, or albedo, is ρ. Note that the $cos(\theta_i)$ term reduces the contribution of light rays that are at progressively more grazing angles. This is due to foreshortening as the angle gets steeper; the same amount of light energy is being distributed over a larger surface area, thus reducing its density and reducing the irradiance of each point on the surface. Lambertian shading is a convenient choice because it linearly

relates incoming light energy per area to an outgoing irradiance that is constant over all viewing directions.

Another popular and more sophisticated model is the Blinn-Phong model, which extends Lambertian shading to simulate glossy surfaces with highlights [23], [10]. This model is represented by Equation 2.2

$$I = I_l(\mathbf{N} \cdot \frac{\mathbf{L} + \mathbf{V}}{\|\mathbf{L} + \mathbf{V}\|})^n, \tag{2.2}$$

where I_l is the incoming light, $\mathbf{N}$ is the surface normal, $\mathbf{L}$ is the incoming light direction, $\mathbf{V}$ is the viewing direction, and n is a constant representing the shininess of the surface. Typically, you would see this term summed with a Lambertian term to apply full shading to an object.

Specular Reflectance

Some surfaces will reflect more directly than diffuse surfaces and are known as specular surfaces. These surfaces tend to reflect incident light rays over the surface normal keeping the angle of reflection close or equal to the angle of incidence. The law of reflection is shown in Equation 2.3 and models how an incident ray is reflected specularly on a surface

$$\mathbf{D_r} = \mathbf{D_i} - 2(\mathbf{D_i} \cdot \mathbf{N})\mathbf{N} \tag{2.3}$$

where the normalized reflected direction $\mathbf{D_r}$ is expressed in terms of the normalized incident direction, $\mathbf{D_i}$, and the surface normal $\mathbf{N}$.

A Whitted-style ray tracer [29], [28], [13] is one which models perfect reflections by using

Equation 2.3 when light rays intersect with surfaces. It uses recursive rays to model accurate reflections by querying the radiance of surface points both in and out of the camera's viewport.

Combining shading models

Believable image synthesis can be achieved by combining Lambertian or Blinn-Phong shading for diffuse modeling, and Whitted style reflections for reflective modeling into a single kernel which can interpolate between both. Most objects in the real world exhibit characteristics of both, so to create a believable approximation of light transport, we must consider both terms. A good way to approximate this is to define a material which has properties to control the albedo, specularity, roughness (a term analogous to the intuitive concept of how matte a surface is), and emission of a surface. There are many more terms that can be simulated, especially in more complex cases which consider more realistic phenomena like refraction, transmission, and subsurface scattering, but for our purposes these are enough. For a given point, the albedo, or color, is an attenuation rate which simulates pigmented surfaces in the real world and diminishes the energy a ray carries with each bounce while carrying a per-channel granularity, meaning the red, green, and blue color channels can be affected independently to allow for a full hue spectrum. The specularity property governs how likely a surface is to reflect a ray specularly as opposed to diffusely. This is treated as binary operator in the kernel and is analogous to a probability of choosing a specular bounce over a diffuse bounce. The roughness controls how random a diffuse bounce should be, with a fully rough surface scatterly light uniformly in the hemispherical domain defined by the surface normal and a fully smooth surface being analogous to a specular surface. Finally, the emission property of our materials will declare how much light energy is generated by this surface. For simplicity, we will assume in our models that surfaces which emit light do

not also reflect it, because we can save performance by calculations who's contributions are likely to be overshadowed by the dominant light emitter.

2.1.3 BRDF and the Rendering Equation

To discuss the bidirectional reflectance distribution function and the rendering equation, we must formally define radiance and irradiance as they relate to radiometry. Radiance is the density of light energy which flows in a given direction measured at a given point [5]. Irradiance is moment of radiance which is the total energy per area on a surface. The BRDF is a general function that relates the incoming radiance from a direction to the outgoing radiance in another direction at a point in space. The function generally describes the ratio of light which a surface will reflect, Lambert's cosine law being an example of a BRDF already discussed.

The rendering equation is the most fundamental equation in the field of realistic light transport and is shown here

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_{\Omega} f(\mathbf{x}, \omega_i, \omega_o)\, L_i(\mathbf{x}, \omega_i)\, (\mathbf{n} \cdot \omega_i)\, d\omega_i \tag{2.4}$$

where $L_o(\mathbf{x}, \omega_o)$ is the outgoing radiance at point $\mathbf{x}$ in direction ω_o, $L_e(\mathbf{x}, \omega_o)$ is the emitted radiance, $f(\mathbf{x}, \omega_i, \omega_o)$ is the bidirectional reflectance distribution function (BRDF) describing how light is reflected, $L_i(\mathbf{x}, \omega_i)$ is the incoming radiance from direction ω_i, and $\mathbf{n} \cdot \omega_i$ represents the cosine of the angle between the surface normal $\mathbf{n}$ and the incoming direction ω_i, accounting for Lambert's cosine law.

This integral is intractable because it is a summation of incoming radiance $L_i(\mathbf{x}, \omega_i)$ over all directions, and the emitters in those directions must themselves recursively evaluate the

rendering equation. Because of this, an approximation model must be used, and regardless of the BRDF we will assume some error.

Monte Carlo integration is a numerical technique that approximates integrals by averaging random samples between the limits of integration [1]. This method is highly versatile because it can approximate the rendering equation, which is recursive and in many cases depends on incoherent radiance distribution functions. The radiance of a scene is discontinuous as occluders frequently disrupt light flow.

One particular downside of Monte Carlo integration is that the approximation often depends on high amounts of samples to minimize error. In the case of light accumulation, low sample counts often appear as noise in the output render as we fail to correctly estimate the contribution of all light sources and reflections in the scene. These problems will be discussed more below, and also provide a motivation for the use of raster-style local color shading which is used in our solution in Chapter 4.

2.1.4 Relevant Rasterization Techniques

We employ several rasterization techniques in our hybrid pipeline to achieve a faster simulation while still maintaining better visual quality than pure rasterization. For decades, raster-based rendering was typically done with a forward rendering pipeline in which each object was lit and shaded locally and independently of the rest. This can be inefficient for large numbers of lights because each object but search through each light leading to a computational complexity of $O(objects * lights)$ [25]. Deferred rendering addresses this problem by introducing a geometry buffer and handling geometry separately from lighting. The idea of a geometry buffer which caches information that will be repeatedly accessed during the current frame is useful to use because in Chapter 4 we will apply many effects which wish to

read this information and querying and caching the scene data once far outperforms querying it many times. The geometry buffer contains information to describe the scene, such as the albedo, distance, and normal direction of the surface that is hit by a ray which travels from the camera through the pixel at the corresponding location on the virtual image plane.

Another useful idea for us is the light attenuation model that rasterization normally uses. Convergence through large sample counts is one reliable way to denoise a render and accurately estimate the irradiance of a point, but this is not feasible in real time. We can instead use importance sampling (defined later) along with an exponential falloff to weight the illumination that a surface receives. This costs accuracy but is computationally feasible.

2.2 Common Problems

Ray tracing rendering engines face a plethora of common problems which require careful attention to reduce. In most cases, the goal will be minimizing error rather than removing the problem entirely which follows from the nature of simulating realistic light transport in real time.

2.2.1 Noise

Noise appears everywhere in ray tracing simulations due to the numerical nature of approximating integration. By nature, rendering requires a recursive integration of the contributions of every possible direction in a scene at every visible point, but this is clearly impossible to compute, let alone in real time. Monte Carlo and other numerical approximations can achieve believable results at best, and noisy, uninterpretable results at worst. Due to the limited computational power that all processors, even GPUs with hardware support, must

operate in, noise reduction is a very challenging problem. Clever sampling techniques, computational batching, heuristics, and many other optimizations can be used to reduce noise, however, each normally has a tradeoff, computational cost, or both. Denoisers are normally post process effects which take as input the noisy render and attempt to extrapolate the converged result from it. Traditionally, denoisers have used classical methods like temporal accumulation (averaging frames across time to effectively achieve greater sampling counts), real time adaptive filters, or stochastic measures [18], but more recently artificial intelligence has been experimented with as in Reeze et al. [17] and Yen et al. [31].

2.2.2 Other Problems

There are many common problems, but as noise is the most common and most apparent, we will just mention and define a few others without giving them as detailed a look. Leaking can occur, especially when using caching techniques, when elements in the foreground begin to bleed illumination to the background. This typically appears in the form of artificially bright or dark patches due to the lighting information being shared in a local screen space region without properly considering the or L2 norm. Wasting computational power on rays which will not contribute majorly to the final render is another common problem. To address this, importance sampling is used to direct rays to spots that will contribute more to the overall convergence. Conceptually, many of these techniques are attempting to sample with larger values of the incoming light function evaluated in the rendering equations with higher resolution. However, if this is overdone, sampling bias can become a problem which normally manifests itself as artificially brighter scenes.

2.3 Lumen in Unreal Engine 5

While most ray tracers today rely on hardware, Unreal Engine 5's Lumen technology provides a notable exception. As a commercial game engine, Unreal found it important to provide a software approximation of their hardware ray tracing pipeline. The results presented by Wright et al. [30] are very impressive and are state of the art for commercial rendering. However, they use a slightly different technique from ray tracing. They utilize a technique called ray marching which requires all geometry in the scene to be converted to signed distance fields rather than triangle based meshes and requires additional tooling for developers and artists. It also relies heavily on another Unreal Engine 5 subsystem called Nanite which virtually removes polygon limits in real time rendering. Due to its Nanite dependencies, it does not handle skinned mesh rendering, which are meshes whose vertices can move dynamically at runtime. While Lumen is an impressive system, our research focuses on different techniques due to time constraints and the presented dependencies on the Unreal Engine ecosystem. Our model is general purpose and can be translated to any environment without special dependencies.

2.4 Ground Truth Model

In order to guide our work, we created a Monte Carlo path tracer to compare our solutions against. The path tracer traces the paths of rays in the scene in a physically accurate but computationally naive way. It takes a relatively long time to converge due to the large number of samples required for noise reduction and thus cannot be considered a real time renderer, but it produces images that are very close to physical accuracy in the domain of problems we care about (opaque objects with reflections). The goal of the work presented

in both Chapter 3 and Chapter 4 is to appear as visually similar to the output of the path tracer as possible, but to render frames at real time. We will see that the work in Chapter 3 displays levels of accuracy and quality that are not acceptable due to the low fidelity of the output. However, the work in Chapter 4 looks very similar to the output from the path tracer, with few noticeable differences visually, but a convergence rate that is nearly 3000x faster than that of the path tracer. We considered a refresh rate of 60 frames per second (FPS) to be the real time standard as most games pick this value for their minimum acceptable frame rate.

2.4.1 Motivation

Creating our own path tracer to use as a reference is greatly beneficial because we can validate our methods of comparison by controlling the entire experiment. Firstly, we created it without using hardware accelerators. This gives us a reasonable baseline to compare our new solutions to in order to calculate a speed up because they make the same assumptions. Secondly, the same development and runtime environments were used for both, once again increasing the validity of comparisons. Also, something we have not mentioned yet is that our solutions do not use any ray-triangle intersection acceleration structures. Commonly, a ray tracing engine will compute a bounding volume hierarchy or some other search accelerator to cull large amounts of work, and though this would be a natural step for future work, it was not implemented in our solutions yet [6], [4]. We created the path tracer without one as well which once again allows us to make comparisons in an unbiased way. Generally, by creating a reference, we are able to make unbiased comparisons and note the speed ups of our methods without side effects.

2.4.2 Implementation

Using ideas presented in [19] and [2], as well as examples from [14], we implemented a Monte Carlo path tracer. It gets its name from the numerical style it utilizes to solve the rendering equation. By taking and average many samples, we can progressively improve our estimation of the lighting in the scene until the image converges. Our solution sends multiple samples per pixel and each sample bounces multiple times in the scene before recording the final contribution. Direct lighting is simulated when rays hit a single surface before bouncing into a light source. Indirect lighting is simulated when two or more bounces occur before hitting a light source and terminating. Any rays which do not ultimately hit a light source are discarded as they contribute nothing to the final output. We use a Lambertian BRDF for diffuse surfaces and interpolate between diffuse and specular bounces based on a surface's roughness property.

For each sample in a single pixel, we add a small jitter to the ray configuration so as to sample a slightly different spot on the first surface hit. This decreases sampling bias and helps to converge a more realistic image. We vary the direction the ray casts in randomly within the square representing the area for a given pixel on the virtual image plane. In addition to this we also jitter the ray origin slightly. This is essentially a small defocusing constant which acts as a blurring mechanism. With a small amount of defocusing we can build anti-aliasing into our model which helps to reverse the aliasing cause by discretizing the continuous representation of the scene for display on a monitor.

Chapter 3

Irradiance Caching

In a static scene, irradiance, the summation of all radiance incident upon a surface, is constant and can be precomputed to save computational cost at run time. Various methods exist to do this, including photon mapping [8], [11], and the irradiance volume proposed by Greger et al. [5] which most heavily inspired this work. In the original paper, Greger et al. [5] propose a system which stores irradiance in a volume rather than only on surfaces. Then, they use this cached lighting information to light dynamic objects in a semi-static scene where only dynamic objects are rendered using the cached lighting. In our first experimental approach, we looked to implement the solution described here but to extend it to fit rendering an entire scene including the static objects considered in the caching process as well as any dynamic object added afterwards. In addition, we use a more physically accurate method to collect samples for the cache than the original paper and will discuss its merits. It could be useful to implement a system like this because we could avoid having multiple rendering systems active at once in a scene and reduce the amount of redundant data which would likely mean eliminating baked lighting maps. In this chapter we will look at the motivation, implementation, results, and reasons for failure of this experiment.

3.1 Motivation

Global illumination refers to the portion of a scene's illumination gathered by indirect lighting, or the sum of the radiance which bounces at least once before contributing to the output of a particular surface. It is considered one of the most important features to synthesize realistic renders, however, it is also one of the most expensive to compute which makes it a desirable but difficult-to-implement feature of a real time ray tracer.

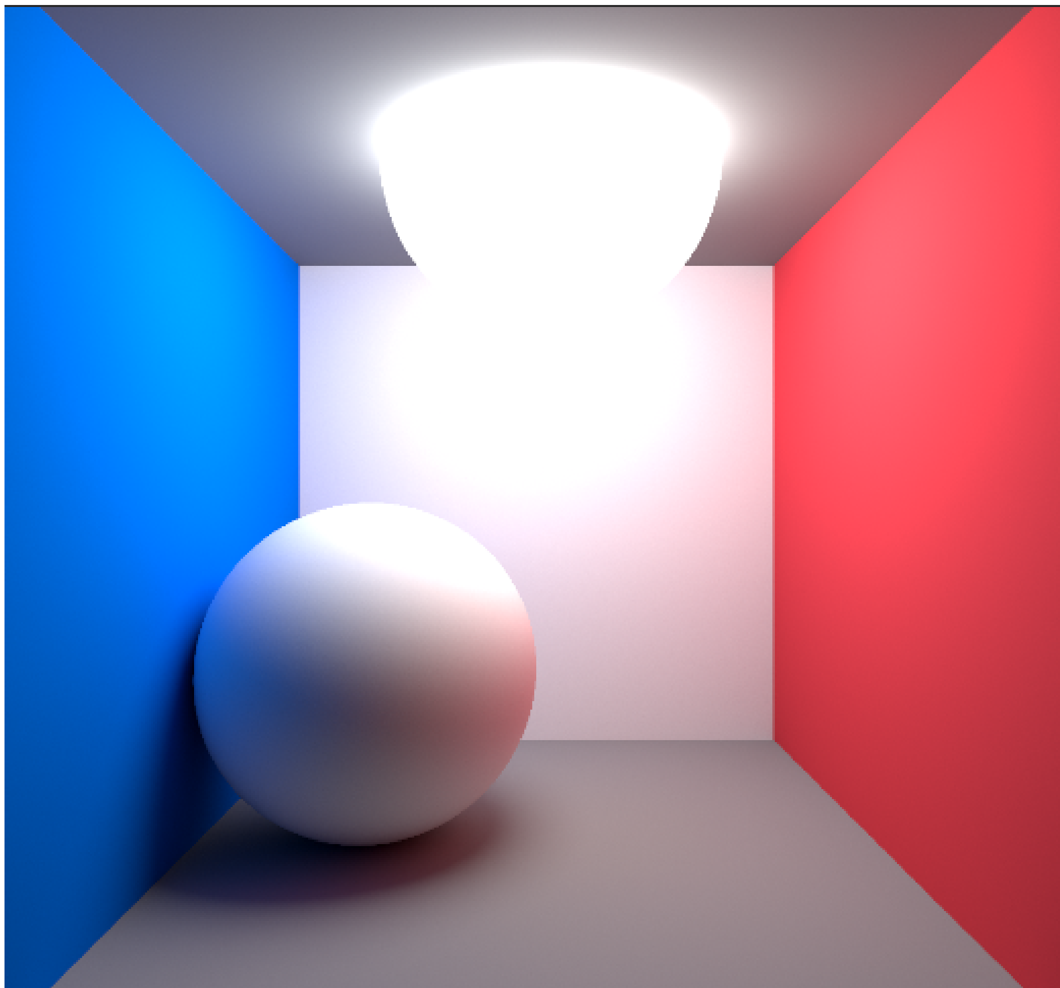


Figure 3.1: A simple scene rendered by our ground truth path tracer demonstrating global illumination. Note that the sphere touching the ground plane has a white albedo, but is picking up the blue and red from the walls on either side.

Global illumination is responsible for lighting a scene more realistically because it scatters light more evenly, softens shadows, and provides color bleeding, an effect which can be seen in Figure 3.1 and causes the colors of objects to reflect onto other nearby surfaces. However, global illumination is one of the more expensive simulations that can be targeted by a ray tracer due to the number of samples required to converge the lighting. This computational increase follows directly from the fact that the more bounces we consider, the more times we are evaluating Equation 2.4 and thus the more Monte Carlo samples we need. Computing this in real time with a path tracer is normally infeasible, so we need to save computations wherever possible.

One idea that we may consider to implement global illumination is to accept a large up-front caching time to build a reference volume which we can then easily query in real time. This volume will cache the irradiance it sees from various uniformly distributed directions around a point.

This method has various trade-offs. The benefits of this method are that the time to query the volume is very small, it can provide a good approximation of the global illumination of a scene, and it is generally noise-free. However, significant down sides include the time needed to build the cache, that lights must be static or the cache will be invalidated, and that the entire scene must be cached and thus finite. It should be noted that this system only approximates global illumination and so reflections and shadows would need to be handled separately.

3.2 Implementation

The original paper by Greger et al. [5] describes the way in which we will store the irradiance of the scene as well as how to sample it. However, because we want to use the volume to

approximate the lighting of an entire scene, we will modify the way in which the irradiance is collected while building the cache. In this section, we will describe our implementation of the modified irradiance volume with a custom irradiance collection method.

3.2.1 Cache Layout

In the original paper, the ray tracing algorithm is implemented on a central processing unit (CPU) and thus a bi-level grid was used to gain resolution in areas which needed finer sampling while optimizing the cache access. Since we are making use of the GPU, we discarded the concept of the bi-level grid and instead cache the radiance in a uniform way throughout the entire scene since we can access the cache in parallel. The grid defines cells which each have eight vertices, and each vertex consists of a spherical probe which queries and stores the irradiance of the scene around it. The probes are laid out in a grid so that trilinear interpolation can be used to determine the global illumination affecting an arbitrary point in space.

Each spherical probe is stored as two texture maps which map to the upper and lower hemispheres. Based on the work in [20], this is a convenient way to achieve uniform spherical sampling simply by defining the resolution on a 2D grid. The grid defines the number of buckets we will store and is conceptually mapped to a sphere when being queried.

Figure 3.2 shows examples of how a 2D plane maps to a 3D hemisphere without changing the relative area of the buckets in either container. Each bucket will store the irradiance that it observes, computed as the quadrature of the radiance samples. It is useful to define the probes as spheres because when we sample them, a direction vector is synonymous with a point on the surface of a unit sphere.

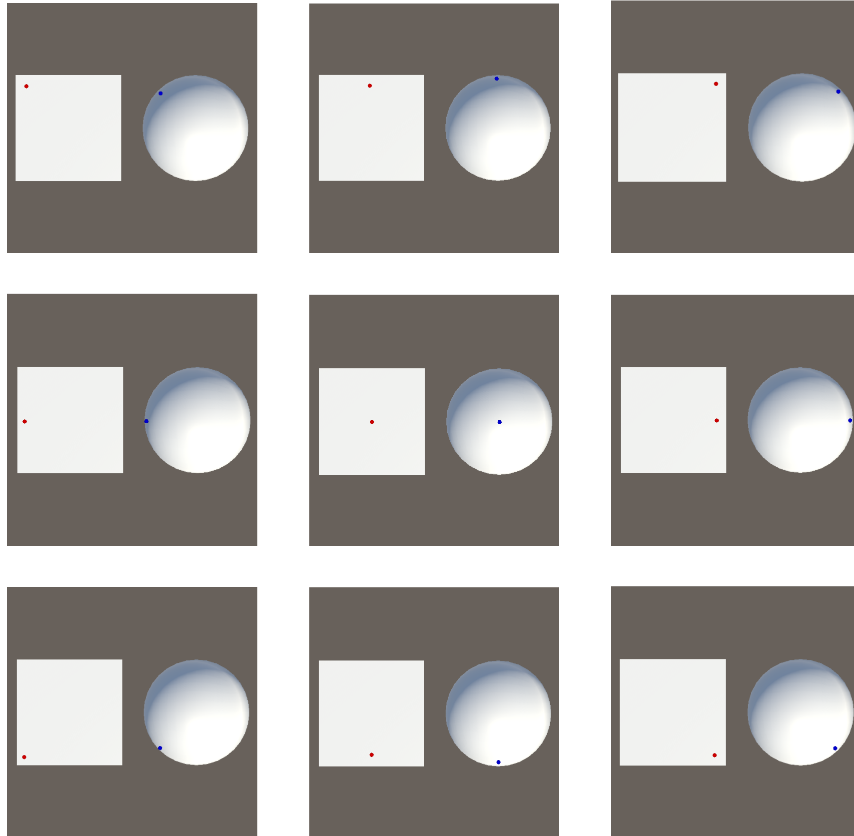
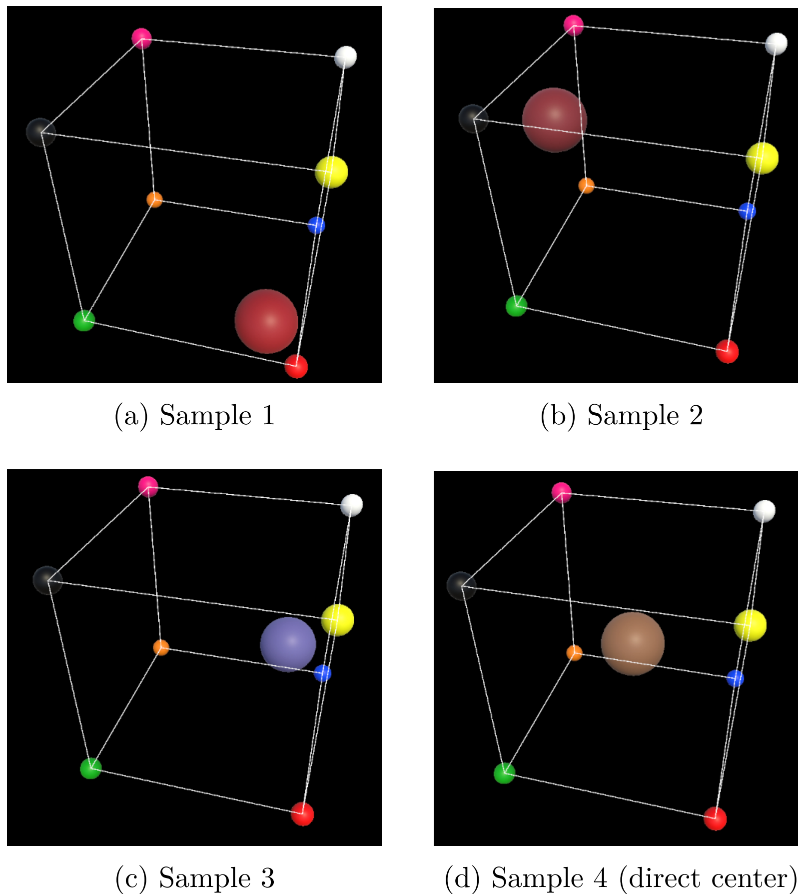


Figure 3.2: Mapping between a 2D plane and a 3D hemisphere which demonstrates texture wrapping.

3.2.2 Cache Sampling

Sampling the cache for an arbitrary point in the domain of the volume involves first determining which grid cell the point lies in, then interpolating between the eight vertices by applying the inverse mapping from hemisphere back to 2D texture. In order to find the correct grid cell, we can use the bounds of the volume and the number of cells on each axis to determine which cell a position maps to. Since we are lighting even static objects, we needed to expand on the original paper by handling cases outside the volume. This is important because it allows us to define a small padding between the volume and any containing geometry, such as a room, so that we can put space between the probes and the outer walls

while still having a reasonable approximation for the lighting of the walls. This small space is desirable to avoid floating-point imprecision when casting rays from the probes at the walls which ensures that hits are properly registered, but this will be discussed more in the next section. To validate locations outside of the volume, we chose to extend the border in a way analogous to extending the border of an image by copying the pixel data in that direction. We simply consider any position outside of the volume to have the same lighting data as the nearest cell to its position, which becomes a progressively worse approximation as the distance increases, but in a well defined volume, there should rarely be points far outside the volume. A room is one common case where we may want to allow this.



(a) Sample 1

(b) Sample 2

(c) Sample 3

(d) Sample 4 (direct center)

Figure 3.3: A visualization of trilinear interpolation.

Once the cell is found or extrapolated, we then query the eight associated probes and in-

terpolate between the values using trilinear interpolation. Figure 3.3 visualizes trilinear interpolation, which, as the name suggests, linearly interpolates between values at vertices on three independent axes.

From this we can see that trilinear interpolation acts as a smoothing function over the values, irradiance in our case, at the vertices. This tells us that the finer our grid size, the better the approximation of the global illumination will be. A finer grid comes at the cost of more memory usage and larger up-front caching times. To shade a surface, we first send an eye ray into the scene and find the point visible through a given pixel. We then query the cache by passing in the direction of the normal of the surface. The original paper does not specify how to choose the direction, so we chose to use the normal of the surface being shaded because the irradiance values stored for each direction represent an integral over the radiance seen by the hemisphere around it, which is precisely what we are trying to approximate. We then multiply this cached irradiance with the albedo of the surface to further attenuate the lighting and factor the pigment of the object being simulated into the lighting estimation. This product is the final color rendered to the screen.

3.2.3 Cache Building

Our solution differs greatly from the original work in [5] in the way that we build the cache. In their paper, they make the assumption that each static surface being queried is fully lit, meaning that the radiance is then simply the albedo of the diffuse surface attenuated by any previous specular bounce. For our use case, we cannot make the same assumption since we want to illuminate the entire scene by the cached irradiance values. We have an opportunity to improve upon the estimation provide in the original paper by utilizing our path tracer.

Instead of counting the first diffuse surface as the radiance coming in from a direction, we

can consider each probe to be a camera and perform path tracing for each direction being cached. The benefit of this is two-fold. First, we get a more physically accurate simulation of global illumination because we can simulate diffuse, specular, and hybrid bounces rather than just specular. This gives us a more realistic simulation of the way that the light rays would have bounced around the scene. Secondly, we also consider direct light sources instead of relying on a generic ambient term. Since our path tracer already calculates the delivered energy based on the lights the ray has hit, we naturally obtain darker sections where less paths hit a direct light source and vice versa.

(a) Irradiance from diffuse radiance - original method

(b) Irradiance from path tracer - our method

Figure 3.4: A comparison of cached irradiance using the method described in [5] (left) and our path tracer (right) on a probe with a 20x20 uniform resolution. The probe is enlarged for this demonstration and normally is a single point. Note how the red tint spreads naturally in our version towards the region facing the white wall where as the original solution does not observe the same. This is a direct result of our path tracer bouncing light realistically around the scene rather than stopping on the first diffuse surface hit.

There is one other benefit that we hoped to gain by using this method. The original paper implicitly culls the contributions of indirect lighting because their algorithm caches the albedo of the first diffuse surface hit after attenuating based on any previous specular bounces

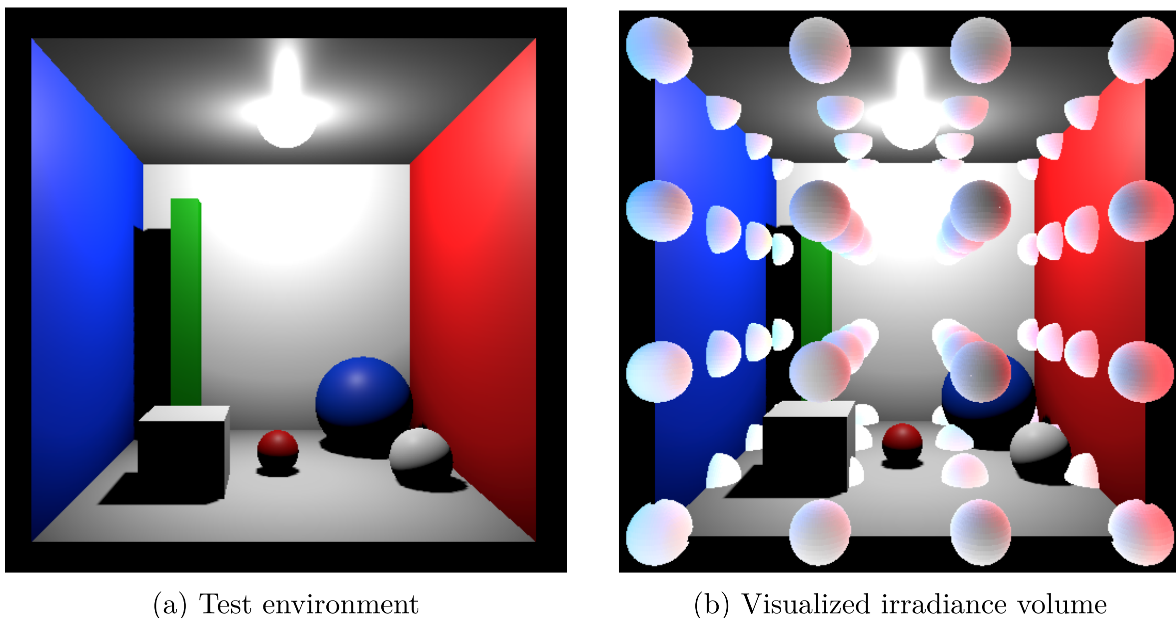
incurred. While this gives a simple approximation, we can model the physical behavior of light more accurately by path tracing because even rays which miss the surface still have a chance to diffusely bounce and contribute majorly. Consider the example scene shown in Figure 3.4 in which two walls form a corner and are white and red respectively. Then consider a probe that queries the scene and hits the white wall first. The original solution would count the radiance in this direction as purely white despite the rendering equation requiring recursive analysis at the collision point. Using our Monte Carlo path tracer, as long as the bounce limit has not been exceeded, we will perform this recursive analysis and potentially see that the red wall was near by and realize it should have been reflecting light onto the white wall, thus reflecting red light back onto the probe. This method also factors in light occluders which block direct light rays and cast shadows in the environment thus dropping the irradiance levels. However, despite path tracing sounding good in theory and being a large motivation for attempting this experiment, as we will see in the results section, the reality is that path tracing is generally too unstable in the way we used it to generate a well formed cache.

3.3 Results

In this section we will discuss the results and findings of our first experiment. All tests were rendered at 1080x1920 resolution on an AMD Ryzen 7 7800X3D 8-Core processor, 32GB of DDR5 RAM, and an NVIDIA GeForce RTX 4060 TI with 12GB of VRAM. While this GPU is capable of doing hardware ray tracing, none of the RTX cores were utilized during any part of this experiment.

We conducted our tests in a room similar to a Cornell box [15] in which we set up two walls with different colors in an otherwise white room. Most, but not all, of the objects in

the scene are white to test more easily the global illumination of the system. We can vary three cache parameters, the first being the number of cells which will be defined in terms of a subdivision level where level 1 is the coarsest level possible and places a single probe at each corner of the test environment and each subdivision after that breaks each cell into eight uniform volume child cells. We can also vary the number of samples per probe which is defined in terms of a samples per axis parameter that defines the resolution of the square texture mapped to each hemisphere. For example, 10 samples per axis would result in 200 samples per probe because we square the samples per axis then double that result to account for each hemisphere. Finally, we can vary the padding of the volume which defines the space between the outer cells and the edge of the test environment. As previously mentioned, a small distance is useful here for floating-point precision reasons, and generally a value of 0.1 should suffice for all our testing.



(a) Test environment (b) Visualized irradiance volume

Figure 3.5: One of the test environments used for this experiment (left) and an example visualizing the cached irradiance volume (right). Note that in these images the objects in the scene are rendered using Unity's built in renderer because we cannot render the debug cache in our pipeline, though it does not affect the cache construction in any way.

Figure 3.5 shows one of our example test environments rendered with Unity's built in renderer and also visualizes an example cached volume with 5 subdivisions and 21 samples per axis by expanding the probes from single points to spheres to display the irradiance they captured.

3.3.1 Cache Build Times and Memory Usage

Table 3.1: Cache build results using path tracer with 1024 samples simulating 4 bounces while varying subdivision level and samples per axis (SPA).

	5 SPA	**10 SPA**	**15 SPA**	**20 SPA**	**25 SPA**
L3 Subdivision	0.46s 0.05mB	1.03s 0.20mB	3.61s 0.44mB	10.80s 0.78mB	25.23s 1.22mB
L4 Subdivision	0.46s 0.10mB	1.70s 0.38mB	6.70s 0.86mB	20.56s 1.53mB	49.26s 2.39mB
L5 Subdivision	0.50s 0.17mB	2.60s 0.66mB	11.41s 1.48mB	35.45s 2.64mB	84.96s 4.12mB
L6 Subdivision	0.70s 0.26mB	4.15s 1.05mB	17.97s 2.36mB	56.71s 4.19mB	134.02s 6.54mB
L7 Subdivision	0.91s 0.39mB	5.86s 1.56mB	26.74s 3.52mB	83.66s 6.25mB	200.24s 9.77mB

Table 3.1 shows the cache build times and memory usages across different subdivision levels and numbers of samples per axis. For each direction in each probe, the path tracer traced 1024 samples with 4 bounces each. To compare against, Table 3.2 was generated using the method described in [5]. The tables show that the build times grew exponentially when increasing either the subdivision level or samples per axis, though this is more easily apparent with the samples per axis due to the larger jumps. This makes sense given the computational complexities of increasing the subdivision level $O((L_s+1)^3)$ and the samples per axis $O(2S^2)$

Table 3.2: Cache build results using diffuse radiance while varying subdivision level and samples per axis (SPA).

	5 SPA	**10 SPA**	**15 SPA**	**20 SPA**	**25 SPA**
L3 Subdivision	0.06s 0.05mB	0.67s 0.20mB	3.28s 0.44mB	10.26s 0.78mB	24.72s 1.22mB
L4 Subdivision	0.10s 0.10mB	1.29s 0.38mB	6.28s 0.86mB	19.88s 1.53mB	49.30s 2.39mB
L5 Subdivision	0.16s 0.17mB	2.21s 0.66mB	10.96s 1.48mB	34.51s 2.64mB	83.43s 4.12mB
L6 Subdivision	0.25s 0.26mB	3.49s 1.05mB	17.24s 2.36mB	54.88s 4.19mB	132.43s 6.54mB
L7 Subdivision	0.37s 0.39mB	5.20s 1.56mB	25.87s 3.52mB	81.11s 6.25mB	196.95s 9.77mB

where L_s is the subdivision level and S is the number of samples per axis. However, this scaling would quickly become a problem in larger scenes and implementing a multi-level grid would likely be a feasible approach to gain resolution where it is needed without letting the caching times and memory sizes grow out of control. Note that the memory usage in both tables is exactly the same since only the radiance querying method is changing, not the storage method.

The results seem to indicate that the Monte Carlo path tracing and the diffuse radiance methods both took about the same amount of time to gather information, despite the huge amount of samples in the former. However, this is skewed heavily by the method in which we are calculating the irradiance. While we use the GPU device to gain massive parallelization while collecting the radiance samples, we perform a read back to the CPU to solve the integrals and calculate the estimated irradiance from all the samples. This problem would

ideally be done fully on the GPU, but due to time constraints only the sample collection was implemented in shader code.

However, if we ignore the costs of the CPU portion, we can still extrapolate the time spent collecting samples to gain a reasonable measure of how much slower the Monte Carlo path tracer is than the original method. Since calculating irradiance works on the radiance samples, it does not depend on the collection method and is a constant time across algorithms. This is why the build times appear nearly identical in Table 3.1 and Table 3.2.

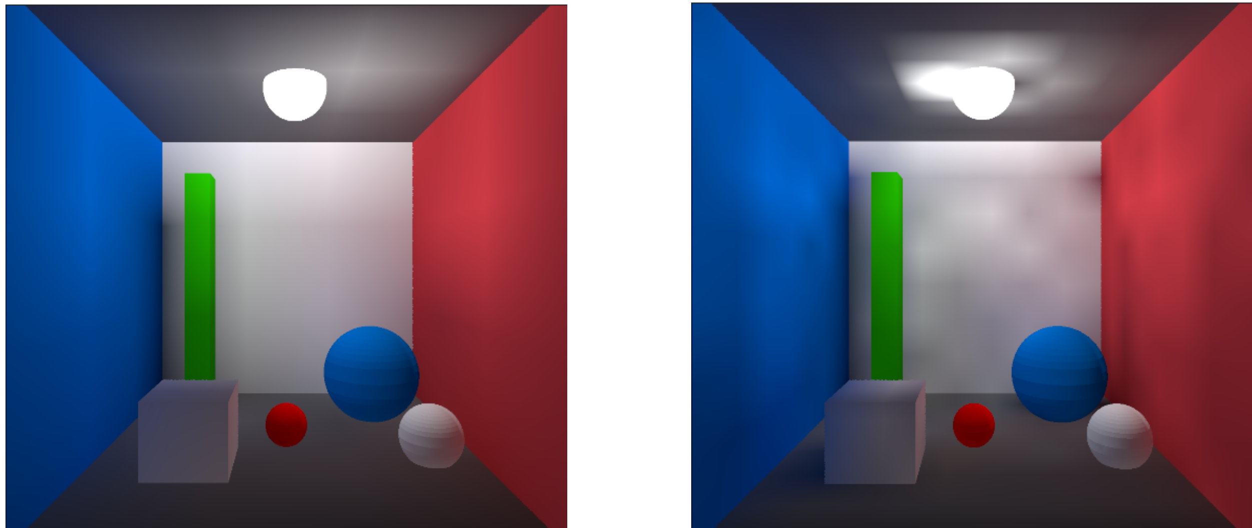
Table 3.3: Radiance sample times using while varying samples per axis (SPA) with a subdivision level of 7. The path tracer traced 1024 samples with 4 bounces each per cached value.

Method	**5 SPA**	**10 SPA**	**15 SPA**	**20 SPA**	**25 SPA**
Path Tracer	3.64ms	9.60ms	13.42ms	27.90ms	42.10ms
Diffuse Radiance	0.02ms	0.02ms	0.03ms	0.03ms	0.03ms

Table 3.3 shows a comparison of the radiance sampling times between both methods. These times were gathered by averaging 100 samples of the sampling shader. It is likely that some sampling bias prevented accurate timings from being collected for the diffuse radiance method since it executes so quickly; however, the results indicate that the path tracer, while still well under a second, performed much slower than the original method. Despite this slower execution, it is still faster than the time that a human might expect to wait to load a scene meaning it is a feasible method of sampling. Now we will take a look at the results when using the cache to render a scene.

3.3.2 Render Results

The results of this experiment revealed several major problems with this design. First, while using Monte Carlo path tracing to collect radiance is good in theory, in practice it is very difficult to obtain enough samples to converge the lighting which leads to pseudo-random artifacts appearing in the cache. This manifests itself in the scene as artificial dark patches which are most apparent on large, flat surfaces.



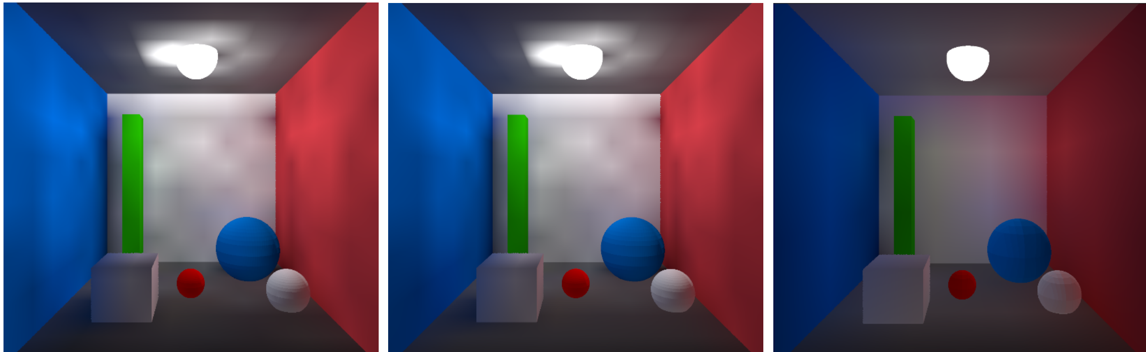
(a) Scene rendered using L3 subdivision and 25 samples per axis

(b) Scene rendered using L7 subdivision and 25 samples per axis

Figure 3.6: Comparison of scene rendered using cached irradiance at different resolution caches.

Figure 3.6 shows a scene being rendered using the cached irradiance at two resolutions. With a lower resolution, the result actually appears better because the cache structure has lower frequency noise whereas the higher resolution cache sees more frequent changes. As discussed, this is due to the Monte Carlo samples failing to converge in the number of samples given, 32 in this case.

From Figure 3.7, we can see that even 8192 paths per radiance sample is not enough to converge the image, though the image is slightly less noisy than the version with 32 samples.



(a) Scene rendered using 32 paths per radiance sample

(b) Scene rendered using 8192 paths per radiance sample

(c) Scene rendered using 1M paths per radiance sample and L3 subdivision and 10 samples per axis

Figure 3.7: Comparison of scene rendered using cached irradiance with different number of samples. Note that the random seed was the same in all images so the structure of the noise is identical.

After one million samples, the image had made notable improvement, but note that due to the time it took we could not create the same resolution cache as the others. The reason for this slow convergence rate goes back to the way the path tracer was designed. It traces paths in a naive but more physically accurate way, meaning many more samples are required before convergence is achieved. The next section will briefly discuss how this could be optimized further to keep the number of samples in a realistic range because our previous results showed that the time needed to build the cache grew exponentially with number of samples.

Another problem is that the resolution of the individual probes does not scale well in terms of memory or time, yet a very high resolution is needed to remove artifacts. Greger et al. [5] originally developed this method to work in semi-dynamic environments where the objects being illuminated using the cache were small compared to the environment. In attempting to extend this to full environments we found that the resolution of the probes was a much larger factor.

Figure 3.8 shows that even at the finest resolution tested, the images rendered still showed

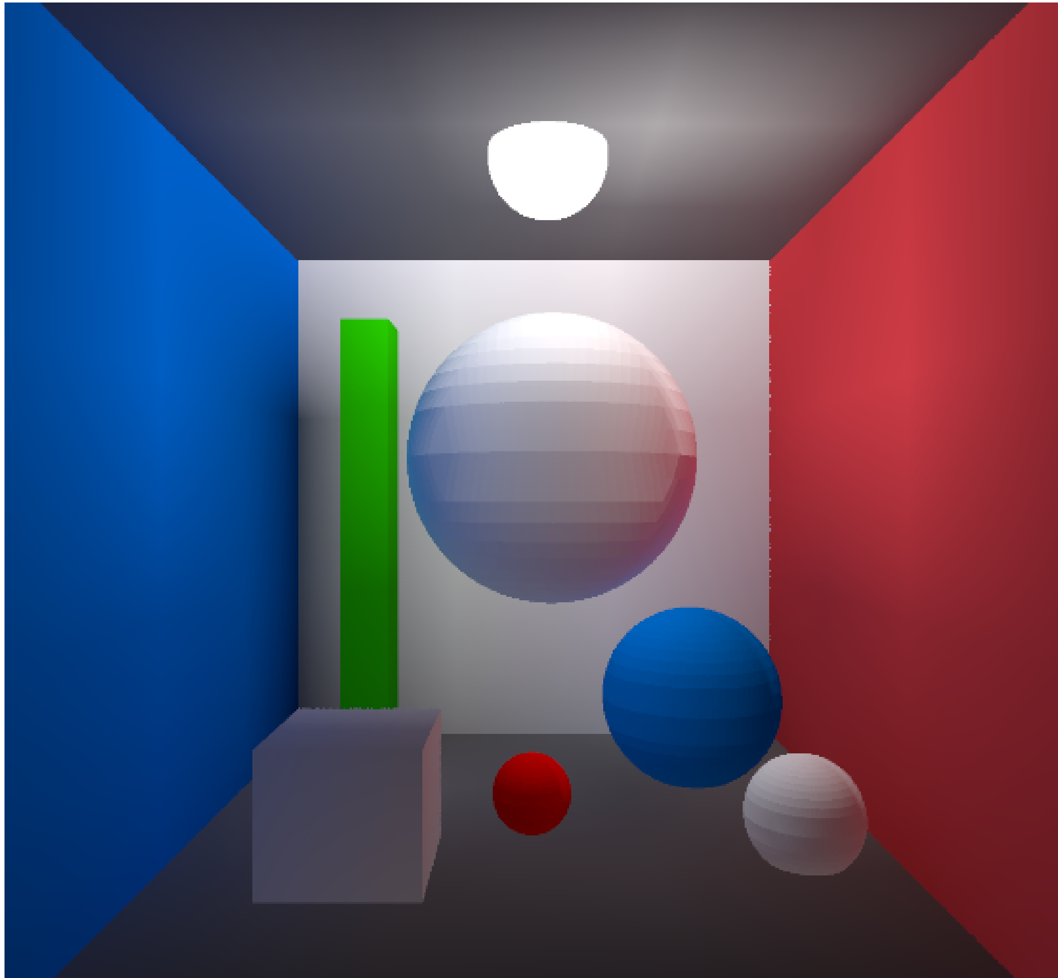


Figure 3.8: In this image, the scene was cached, then the hovering center sphere was added and rendered using the cached irradiance. It picks up the blue and red from the walls on either side, but the resolution of the 25x25 probe hemispheres is noticeable and creates artifacts.

clear artifacts at the boundaries of the buckets being sampled from the probes. This is evident in both the scene items, and the hovering sphere which was added after the caching was complete. This problem is exacerbated by the smooth surfaces generated by the spheres which are rendered using signed distance fields rather than polygonal meshes because the normal vectors on the surface of the sphere change continuously while the stored irradiance is discrete, leading to significant aliasing when sampling. This is due to the discrete nature of the stored irradiance. In scenarios that closely resemble the original intent set out by the

paper, such as small objects next to large flat surfaces with near constant irradiance, our solution appears to work better than in other cases.

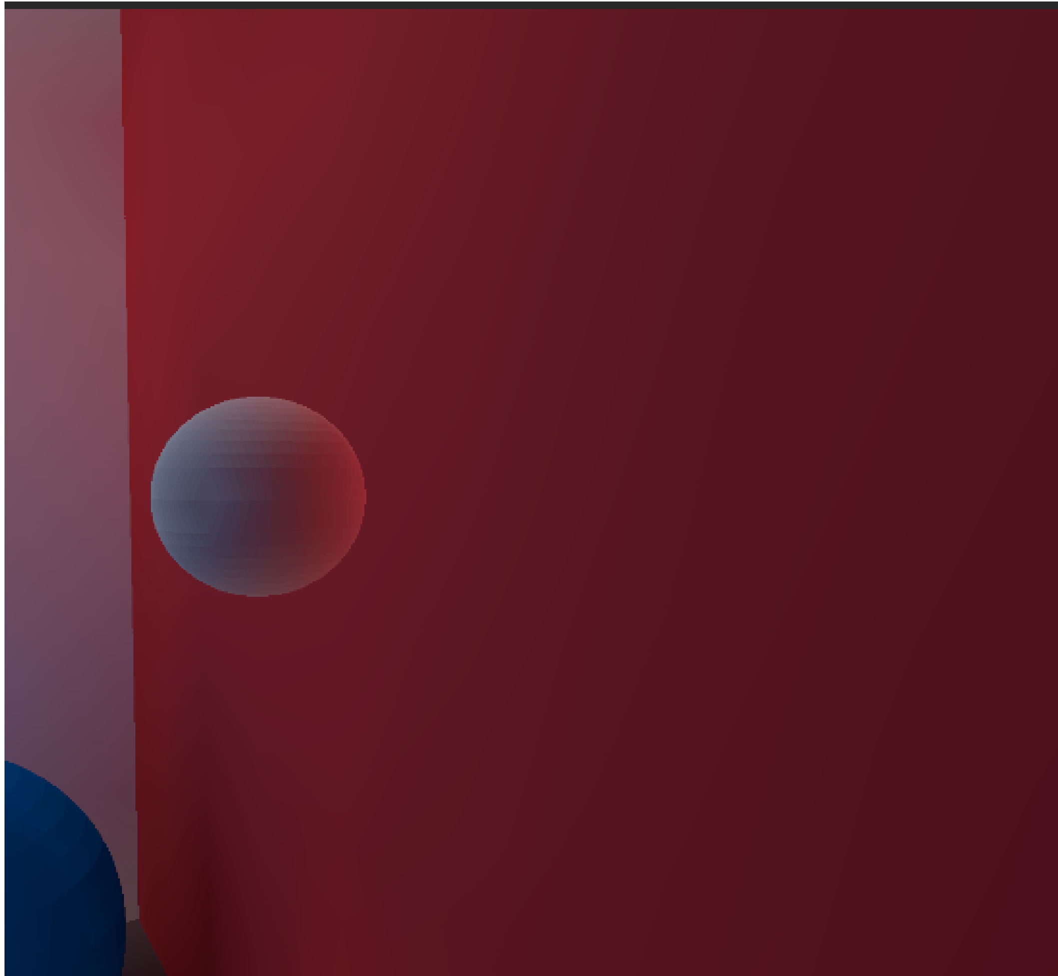
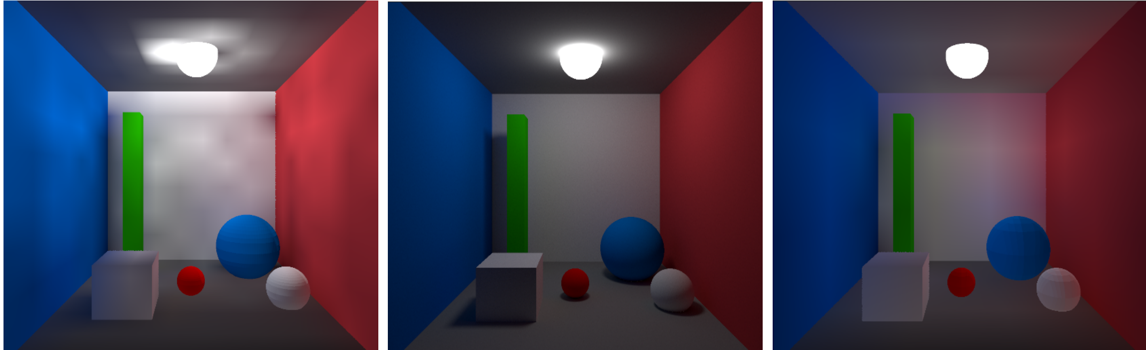


Figure 3.9: A small sphere next to a large red wall. The red light reflecting onto the sphere is smoother than the other light which is a mix of sources and intensities.

Figure 3.9 shows an example of this using a sphere next to a large red wall. We can see that the light reflecting onto the sphere from the wall appears nearly constant due to the proximity of the probes to the wall.

Despite the many problems revealed by this experiment, there were some aspects which worked well. Notably, in this scene, we achieved an average frame rate of 738 frams per second (FPS) as opposed to the path tracer which took 120 seconds to render a single frame



(a) L7 subdivision, 25 samples per axis, 8196 paths per sample

(b) Reference image accumulated for 2 minutes.

(c) L3 subdivision, 10 samples per axis, 1M paths per sample

Figure 3.10: Side by side comparisons of the results and the path traced reference.

noise free. This is encouraging because it means that the cache lookups are more than fast enough to achieve real time frame rates. In addition to the fast render times, we also observed that the color bleeding due to the global illumination was easily noticeable despite being noisy. When adding objects to the scene that did not exist during the caching process, they were able to pick up the irradiance from nearby surfaces as shown in Figure 3.8.

3.4 Insights

From this experiment it became clear that to approximate realistic scenes without compromising heavily on quality, we would need a technique which had a much finer resolution. While the caching is interesting because it is very fast, the problem is that radiance is not a continuous function so details are missed frequently when discretizing it. The need for higher resolution is what motivated the next solution; however, there are some interesting things about the approach taken in this chapter which could still be useful if we had more time to develop them.

As shown in Figure 3.10, when using a value approaching the proper number of samples,

we got a result similar to the reference image. That render used one million paths per radiance sample which is extremely high. Instead, we could drastically improve this by using importance sampling to bounce more samples into the light sources. We could also look further at adaptive resolution caching and rather than bringing back the idea of the bi-level grid, we could build out an octree to gain massive resolution boosts in the areas with the most intricate geometry in a memory efficient way. However, the most likely use of this method will be discussed more in Chapter 5 which would be to combine this with the solution in Chapter 4.

Chapter 4

Hybrid Approach

While the experiment in Chapter 3 attempted to light a scene using cached irradiance, the solution presented here will not only light the scene, but also render shadows and reflections. This approach borrows concepts from rasterization to support the ray casting core while trading some accuracy for a drastic computational speedup. In addition, it does not rely on caching or pre-computing data which allows it to load immediately on arbitrary scenes and avoids the scaling issues discussed in the previous chapter. One thing that this solution does not include is global illumination, though as the end of this chapter and Chapter 5 will both discuss possible further work to include it. Despite the accuracy tradeoffs and the absence of global illumination, our final result looks much closer to the reference than the previous solution due to a variety of reasons which will be discussed throughout this chapter.

4.1 Motivation

In addition to adding support for shadows and reflections, this system alleviates another limitation of the experiment in Chapter 3. Since we were caching irradiance, any objects which were in the scene while building the cache would be forced to remain static due to artifacts left behind. The Monte Carlo path tracer we used to gather the radiance samples would bake shadows and color bleeding information into the cache which were not removed when the object which sourced them was. In other words, an object's effects were seen

regardless of whether the object was still present or not if it were included in the caching phase. This solution completely avoids that problem because it does not cache any data except the previous frame which updates in real time.

This also means that the solution supports fully dynamic scenes, including moving light sources. It is fast enough to recalculate the full lighting of the scene each frame while querying multiple light sources and simulating shadows and reflections. Better still, our solution demonstrates nearly a 3000X convergence rate speed up over the Monte Carlo path tracer. It is able to achieve well over real time results in simple scenes and could be drastically sped up by implementing an acceleration structure like a bounding volume hierarchy. This is something every ray tracing engine does because the performance impact is significant, but due to time constraints, we did not implement one. This will be discussed in more detail in Chapter 5.

4.2 Implementation

The implementation for this solution will be broken into two parts, the core render loop and the post processing. The core loop is responsible for producing a full image with lighting, shadows, and reflections, then various post-processing tools will help to denoise the result. In the core render loop, our overall strategy will be to provide the post processing stack with the least noisy image possible while maintaining good quality. Image quality will be defined as how close the result visually appears to the reference image.

This is where our hybrid approach will be crucial to speed up the lighting estimations. In this section we will discuss our techniques for lighting, shadows, and reflections as well as the various optimizations made to achieve real time frame rates. We will present the various settings which can be tuned on a per-machine basis to trade progressively higher

levels of quality for performance while attempting to keep the output at a reasonable level or believability. We will order the presentation of this material so that the system becomes a better approximation of our reference image at the cost of higher levels of computation with each added feature.

During post processing, we employ both temporal accumulation and a custom PID (proportional, integral, derivative) control loop to target the noisiest areas in the image with additional and variable levels of processing based on the current frame rate. The post processing stack is responsible for preparing the image for display as best as possible given the current frame produced by the core loop. As we will see, in the worst cases, there is a small amount of visible noise when an object is moving, while in the best cases the image appears virtually noise free while operating at a fraction of the cost of the Monte Carlo path tracer.

4.2.1 Core Render Loop

To render a frame we perform the following steps in order; first, we cache a geometry buffer, then we shade every surface with its albedo, then we sample lights to add shadows and illumination data, then finally we calculate reflections in a second pass. The order is purposeful for several reasons. First, we immediately cache a geometry buffer so that we only suffer the expensive scene querying cost once for data that is needed repeatedly throughout the pipeline. The most expensive part of the ray tracing algorithm is the ray-geometry intersection testing so we can save a lot computationally by reducing the number of tests we perform. Despite it's name, the geometry buffer will hold more than just geometry and will act as a server of various information such as the material properties, the surface normal, the distance to the camera, and more. Secondly, we calculate reflections last in the pipeline. This is crucial because often times we can reuse previously calculated lighting information

in a reflection if the reflected surface also appears in the scene. This is not always the case, but our system optimistically assumes that it will have that data and will fallback on more ray samples if it cannot find it.

Lighting and Shadows

The lighting section is where we borrow our key idea from rasterization. The Monte Carlo path tracer does not assume any baseline for illumination in a scene and this is the biggest reason there is so much noise. Instead of waiting for enough samples to estimate the contribution of various light sources, we will invert that and directly shade the surfaces in the scene with their albedo, effectively assuming that they are fully lit with white light.

Figure 4.1 shows an example scene with only this first step rendered. It looks flat and washed out because there are no shadows or lighting information, but this is a much better baseline to start from when accumulating illumination than a black screen because we don't need as many samples to converge the image. The path tracer takes a long time to converge because it incrementally adds to the irradiance of a surface based on random samples, but we circumvent that by initializing each surface to its albedo. The downside of this approach is that we suffer some sampling bias and it will manifest itself further down the pipeline as lighting error. However, the computational speed up and the relatively minor visual difference justifies the approach.

To then add believable lighting to the scene, we loop over every light in the scene and calculate its contribution for each surface. Our solution offers two options for this, one which is faster and generates no noise, but is significantly less accurate, and another which considers the surface area of lights for significant visual improvements at the cost of speed. Both techniques will apply their results to the final pixel color by multiplying the calculated

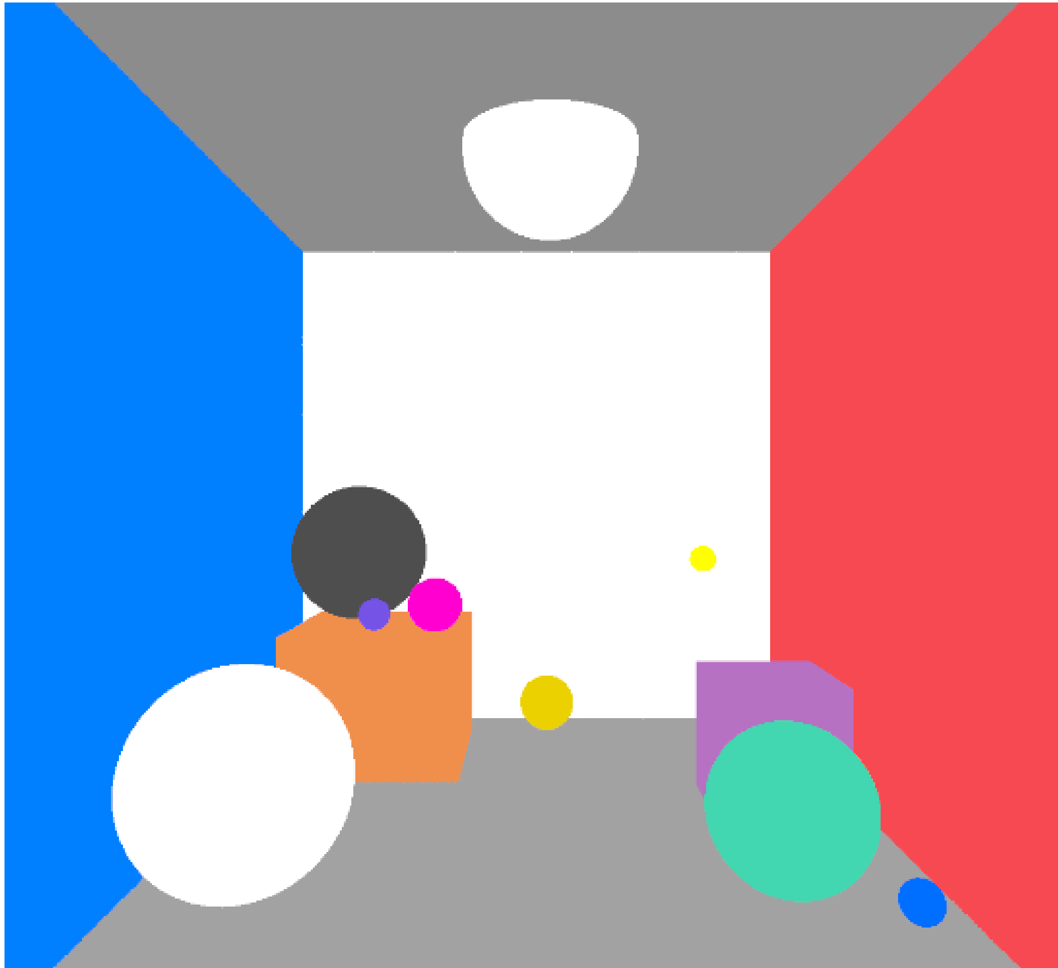


Figure 4.1: Direct color of each object in the scene. Starting from this baseline rather than a black screen saves massive amounts of computation while incurring some sampling bias.

lighting into the albedo which was cached in the previous step. The distance of the light will factor into an exponential attenuation function which helps us estimate the number of rays which would hit based on distance if we were path tracing. All this together allows us to utilize the albedo as an approximation of the surface color and incorporate the scene lights on top of it.

Our first option for doing this is to use a purely raster style technique. Rather than ray casting against the surfaces of lights, we can simply query the center of each light with a single ray to tell if the light has line of sight to the surface at that point. If it does, we will

apply the lighting by factoring in the distance and using a Lambertian BRDF as well as the light's color and power.

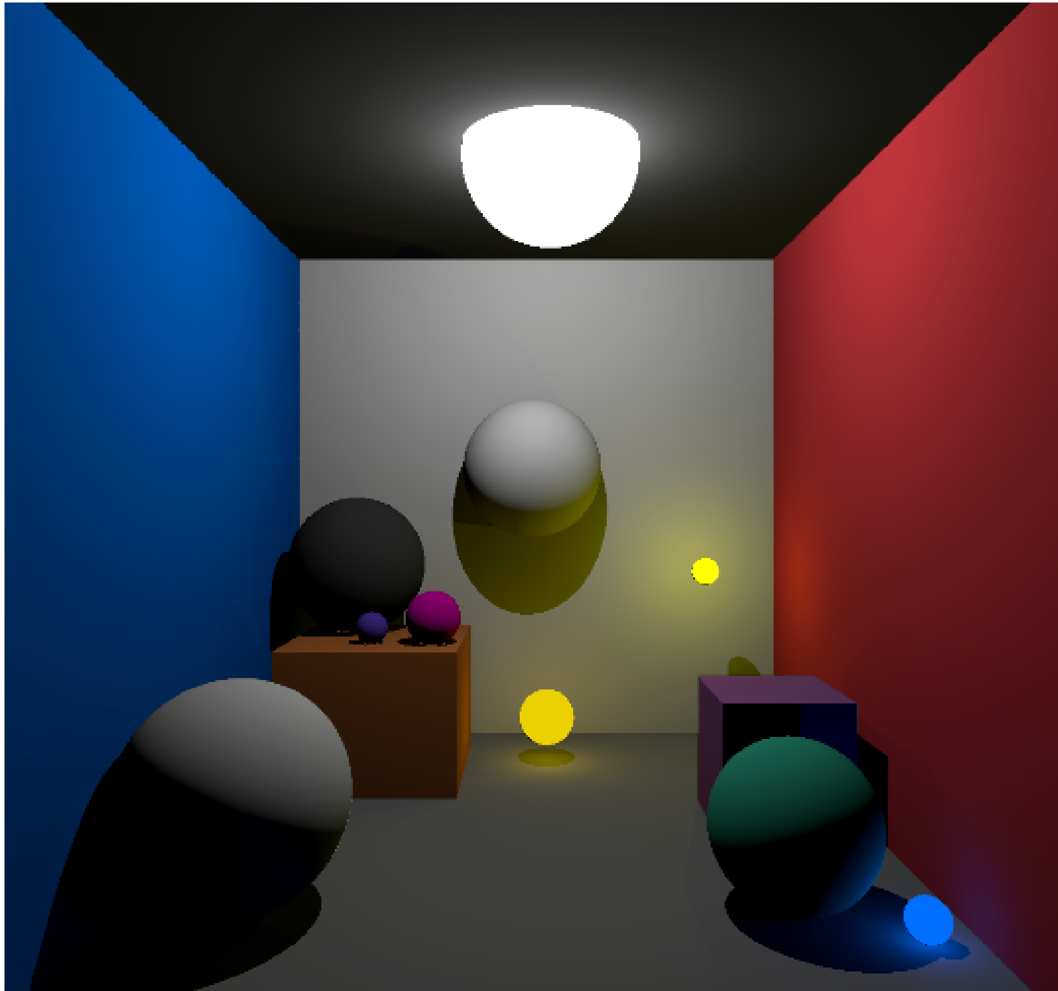


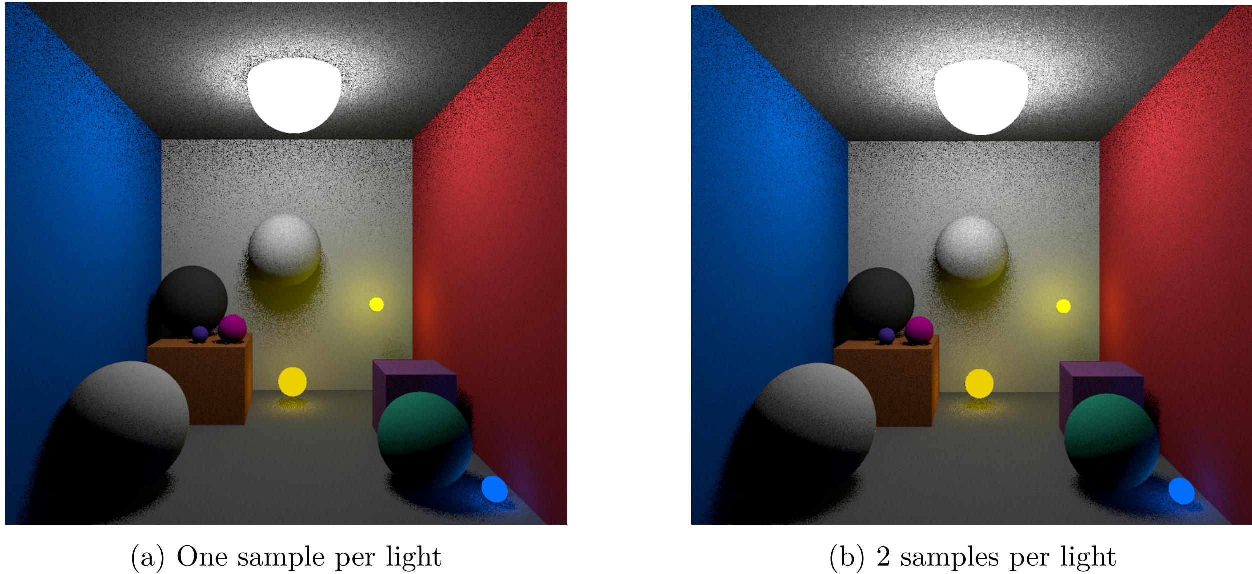
Figure 4.2: A single frame rendered using the direct color of each object weighted by raster style scene lights which function like point lights.

Figure 4.2 shows the same scene but with the lighting data factored in using a raster style approach. From the result, we can see that the light appears to falloff naturally as the distance from the sources increases, particularly noticeable on the blue and red walls. We also note that shadows are a natural byproduct of our lighting method. If no light is within view of a point on a surface, that point is in shadow and rendered black. This binary method looks better than it otherwise would because we are using a Lambertian BRDF which

simulates the light falloff as we transition to shadow, though a global illumination solution would be needed to achieve realism in this scene which this solution does not implement. However, we can add more fidelity to this render, and specifically to the shadows, by using our second lighting method. Note that the edges of these shadows are hard, meaning there is no apparent penumbra, and instead the fully illuminated regions abruptly transition to regions of shadow, or umbra. This is a consequence of raster style lights where each surface point in the scene only samples the middle of the light and either fully incorporates or fully ignores it's effects. We can use our second method to simulate areas of penumbra, although this technique introduces noise which is currently absent from our simulation. The entire post processing stack aims solely to remove this noise because soft shadows are important and go a long way towards increasing the believability of a render.

Our second option considers the surface area of each light and sends multiple pseudo-random samples per surface point being lit. This allows us to approximate which regions of our scene are partially occluded by objects and receiving a reduced portion of the full power of the light. This solution reintroduces noise because we need many samples to gain a better approximation of the lights being hit, however we can drastically reduce it this time by using importance sampling. Rather than casting rays randomly using Monte Carlo style sampling, we can choose a direction that we know will hit the surface of a light if there are no occluders. We implemented an importance sampling algorithm for sphere lights which randomly samples the hemisphere of the light around the direction from the light to the surface point.

Figure 4.3 shows a comparison of the scene rendered using one versus two random samplers per light. The scene using two samples appears slightly brighter because on average more rays are finding the light source. However, once our post processing stack was implemented we found that our results converged best when using a single sample because we could factor

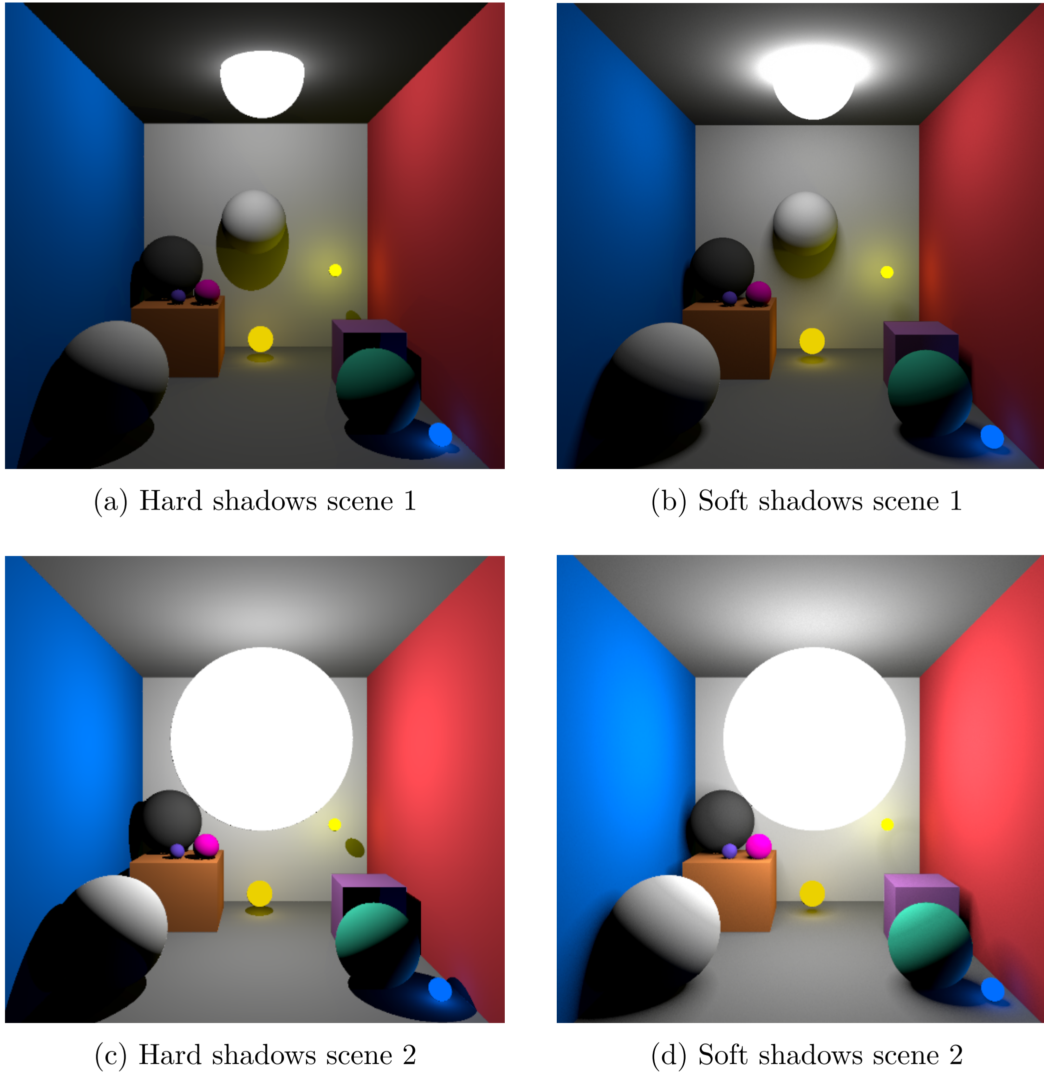


(a) One sample per light (b) 2 samples per light

Figure 4.3: Comparison of scene rendered using a single pseudo random sample per light versus two.

in additional samples over time rather than trying to do too much work in a single frame. Despite the noise present, we can see that the form of the shadows appears much more realistic because of the presence of the penumbra.

Figure 4.4 shows a comparison of a fully converged render using area lights versus raster lights in two scenes. The soft shadows are crucial for the quality of our output, especially the closer or larger the light source is. When a light source is far away or is smaller, the difference between using point or area lights diminishes because small or far lights act more similar to points. A future section in this chapter, as well as Chapter 5, will discuss how we could take this lighting model further to make the scene even more believable, but the discussion depends on the post processing stack so it will be presented later.



(a) Hard shadows scene 1 (b) Soft shadows scene 1

(c) Hard shadows scene 2 (d) Soft shadows scene 2

Figure 4.4: A comparison of hard and soft shadows in two scenes.

Reflections

Simulating reflections is difficult because we cannot be sure that the area seen in the reflection is also seen by the camera. In rasterization this is a problem because objects outside of the camera frustum cannot be properly projected and are typically culled even before that would be an option. However, since we are ray tracing, we can send rays to query the scene outside of the camera's frustum and no projections are needed. In fact, we have an opportunity to

save computations here if we break the rendering into two passes and attempt to query any results which have already been computed and cached. Ray traced reflections are expensive but physically accurate and robust against extreme camera angles and positions which are enough to break down screen space approximations.

A reflection occurs on a specular surface and we can simulate this by bouncing the query ray off the surface using the law of reflection and querying the new hit for material properties. Our processes is recursive up to a maximum reflection depth, at which point we terminate the loop and consider the albedo and lighting of the last hit surface. With each specular bounce, we keep track of a running specularity attenuation so that when we render the ultimate surface on the reflective surface, we know how much to blend in the new data. An objects material properties determine how well it reflects light based on a specularity coefficient.

The path tracer considers both an object's roughness and specularity, where the specularity gives the probability that a ray bounces specularly off a surface while the roughness determines a blending factor between a random diffuse bounce direction and a specular direction for rays which are not specularly bounces. Both of these properties are needed to simulate a range of believable materials but we drop the roughness term in our approximation to remove noise. We lose the ability to simulate mostly-diffuse reflective surfaces but we gain noise free glossy and pure reflections.

To calculate the final color of a pixel which depends on reflections, we first bounce a reflection ray off the surface to get the position of the reflected hit. With this, we can attempt a projection onto the virtual image plane and if it is within its domain we can check the cache to see if we have already calculated lighting data for that point. We need to compare the distance to the reflected point from the camera with the depth of the cached hit point at the related pixel in the geometry buffer to ensure that the point is visible and not occluded

by other geometry, but if it passes, then we are done and simply return the cached data. If we are not able to find a color in the cache then we must calculate the lighting data using the techniques presented previously.

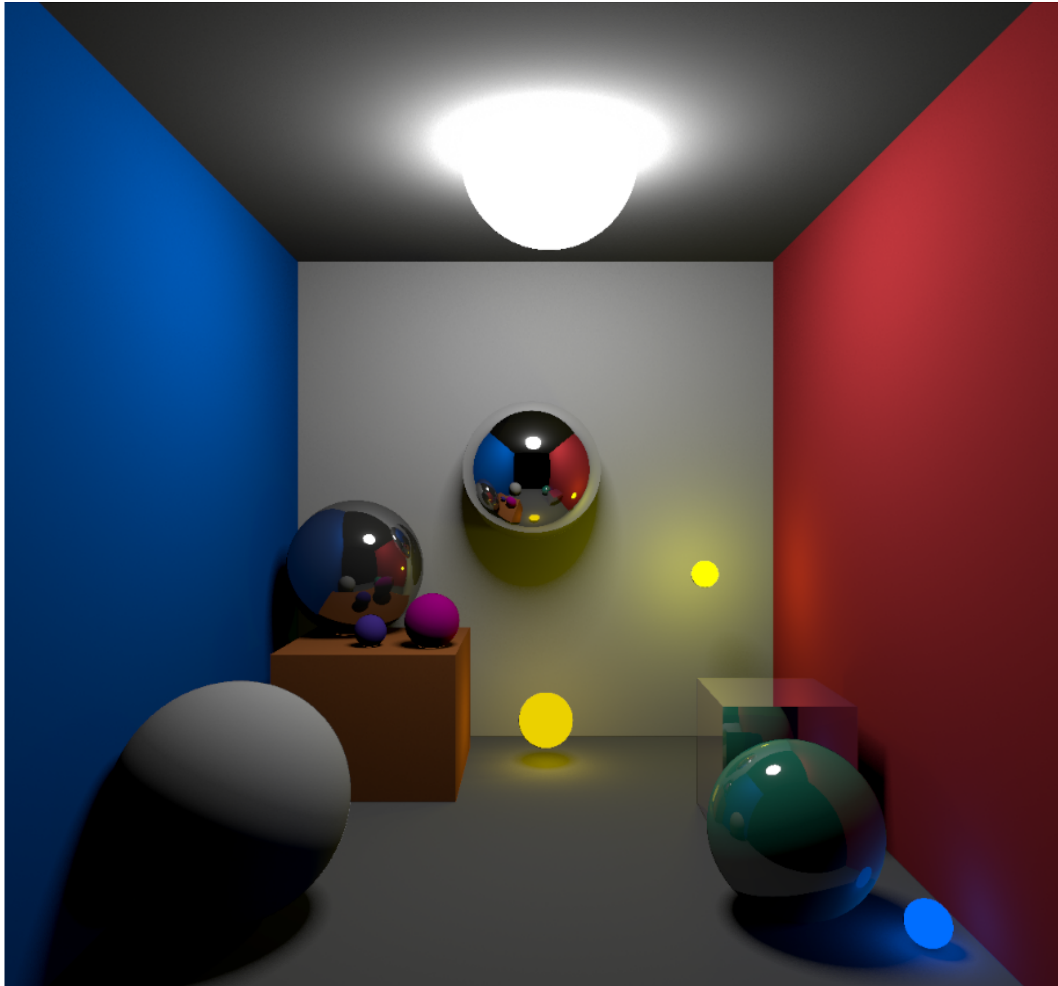


Figure 4.5: Scene rendered with reflective surfaces.

Figure 4.5 shows the scene rendered with reflections. Figure 4.6 shows a closeup portion of a render not using the post processing stack to demonstrate both the recursive reflections and how our choice to drop the roughness term removes all noise from pure reflective surfaces. Glossy reflective surfaces can still be noisy because

We have several parameters which can affect the fidelity of the simulation related to reflec-

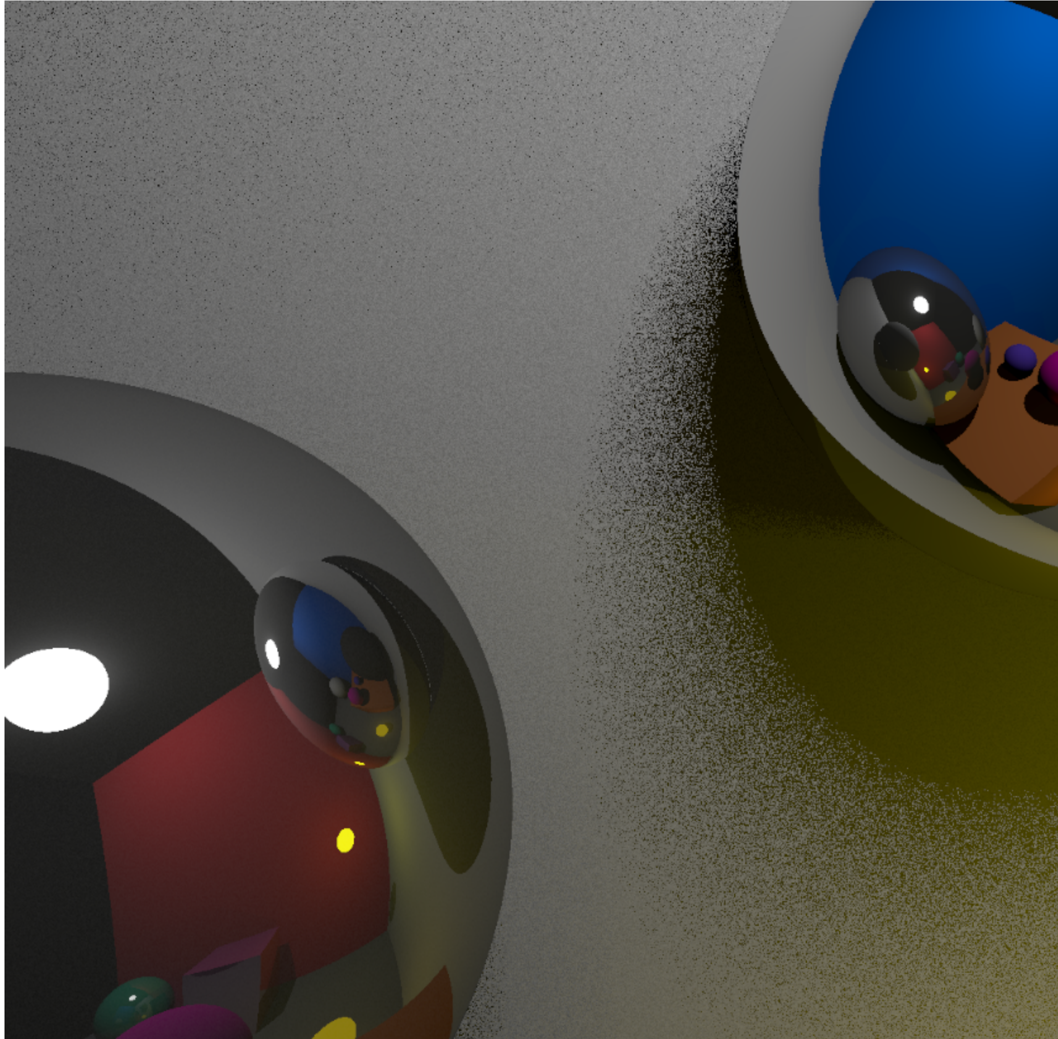


Figure 4.6: Recursive reflections with a depth of 2. Note that when either of the spheres renders itself reflected on its surface, it now appears as a diffuse surface. Also note that the presence of noise in the scene does not affect the purely reflective surfaces.

tions. First, as mentioned above, we must choose a reflection depth where a higher number increases the number of recursive reflections we can render at the cost of performance. We can also choose to use raster style lights in reflected surfaces rather than area lights. This optimization is good in the general case because reflections are frequently small in screen space, but in an application where large reflective surfaces are frequently visible, it might be better to calculate the more physically accurate area lights in reflections. We can also

choose the number of light samples to include in reflection calculations independently of the main lighting parameter.

4.2.2 Post Processing

There are two main components in the post processing our solution performs. The first is temporal accumulation. An effective way to simulate gathering a greater number of samples is to keep the results from previous frames. As long as nothing has moved between the previous frame and the current one, we can blend the frames together to allow past samples to contribute to the render. This is exactly how the path tracer works and if we continuously do this, its not just the past frame contributing, but all past frames. Since we are working in dynamic scenes however, we need to invalidate the blending if anything, including the camera, moves. Because of this, temporal accumulation is only a viable solution if its effects are very fast. The less noisy the input image is, the faster temporal accumulation will appear to work because less samples are needed to finish the convergence. This may seem cyclical, but really it indicates that temporal accumulation should be added to polish the image, not to fully denoise it.

To carry the brunt of the denoising work, we implemented a custom solution. Given a target frame rate (normally 60 frames per second in commercial video games) we can determine if we have time left over at the end of the frame. If we do, we can cast additional rays to attempt to clean up the noisiest portions of the render. To do this we need two things; one, a way to generate a mask to tell where the noisiest areas are, and two, a way to estimate how much work we have time to do.

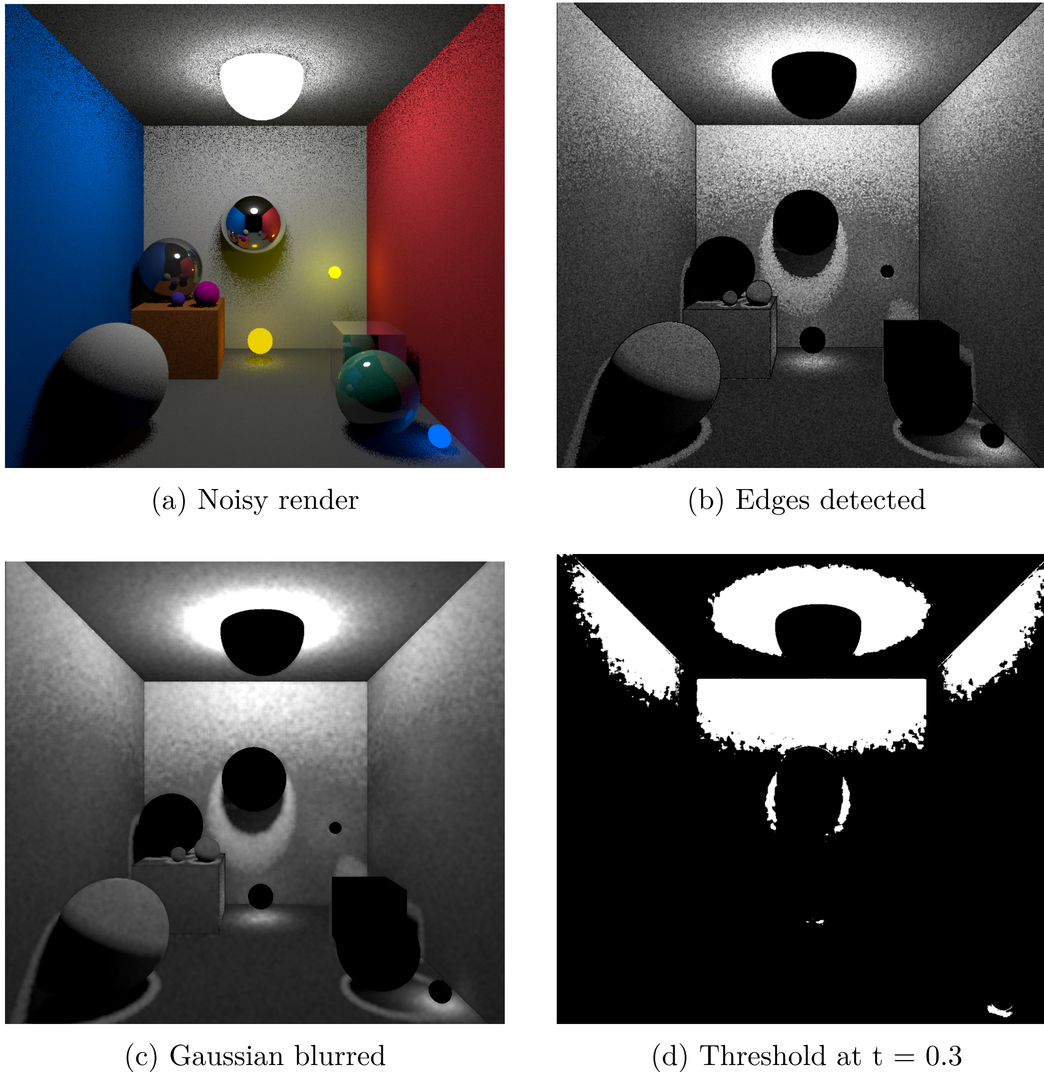
Noise Mask Generation

To generate a noise mask, our key insight was to use what is typically a problem in image processing. When performing edge detection on an image using a Sobel filter, high frequency noise is typically a problem and can lead to false positives. The solution normally involves convolving the image with a gaussian kernel which acts as a low pass filter before performing edge detection. However, in our case we actually want to detect noise so doing exactly the opposite proved very effective from our experiments.

We first run an augmented Sobel edge detection kernel on the noisy image to pick up the areas with the highest rates of change [27]. Running an Sobel filter like this would also pick up on actual edges in the scene which we do not want because ultimately it will lead to wasting computations by sending additional clusters of rays at spots which do not require them. To filter these out as best as possible, we filter the edge detection kernel by surface normal alignment and whether the surface is a light or not. If the normal vectors of two surfaces are not sufficiently aligned, we predict that this is an actual edge rather than a noise spike and discard it. We always discard pixels which are light sources because the illumination also would artificially trigger edges to be detected.

After initial edge detection, we then blur and threshold the image. We use a separable gaussian blur to patch small gaps in the mask created by the nature of the noise it it detecting. We then perform a threshold against a constant to discard portions of the mask which we wish to cull. This threshold constant gives us an excellent control handle which can react by raising or lowering based on how much work we predict we can do at the end of a frame.

Figure 4.7 shows the full pipeline that takes in a noisy image and creates a mask tailored to the threshold constant. Comparing the output mask and the input image, we can see

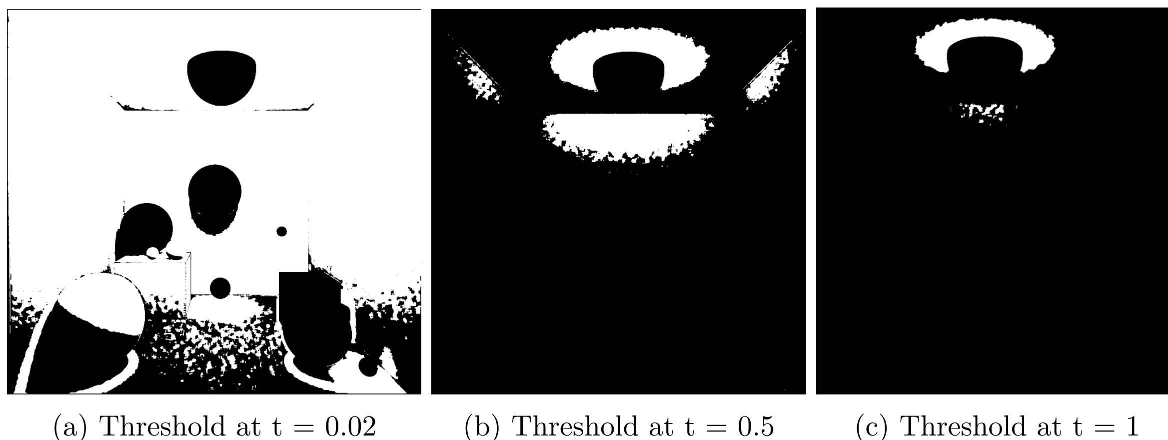


(a) Noisy render (b) Edges detected

(c) Gaussian blurred (d) Threshold at t = 0.3

Figure 4.7: Full noise mask creation pipeline.

that the mask generated matches the noisiest portions of the image very well. The biggest problem with this technique is that the edges of the noise mask itself are noisy, however this is exactly the type of work that the temporal accumulator can handle well. Figure 4.8 shows the mask output at various thresholds. The threshold does not seem to apply a linear mapping with respect to how much of the mask is culled, but this did not seem to affect the output.



(a) Threshold at t = 0.02 (b) Threshold at t = 0.5 (c) Threshold at t = 1

Figure 4.8: Various threshold values.

Work Estimation

With the noise mask created, we need a way to estimate how much work we can do without dropping below our target frame rate. We implemented this control loop using a PID controller and the current frame rate as the feedback variable. The PID controller updates the mask threshold dynamically to grow or shrink the area that is receiving additional rays in order to accelerate the number of samples that the noisiest areas get. When computing additional samples, a post processing pass will run anywhere the mask has a true value. When such a location is detected, that pixel will send eight additional samples to attempt to converge the irradiance estimation on that surface. The selection of eight samples is balanced by the tuned PID constants which together estimate how much work we can do at the end of a given frame.

Since the current frame rate is the feedback variable, our system automatically throttles based on the current performance. If the GPU were handling sporadic asynchronous tasks our system would respond by raising the mask threshold. This also means that with a more complex scene, worse hardware, additional GPU load, low power device, or any other scenario which would cause lower frame rates, our renderer will appear slightly more noisy

to maintain a real time frame rate.

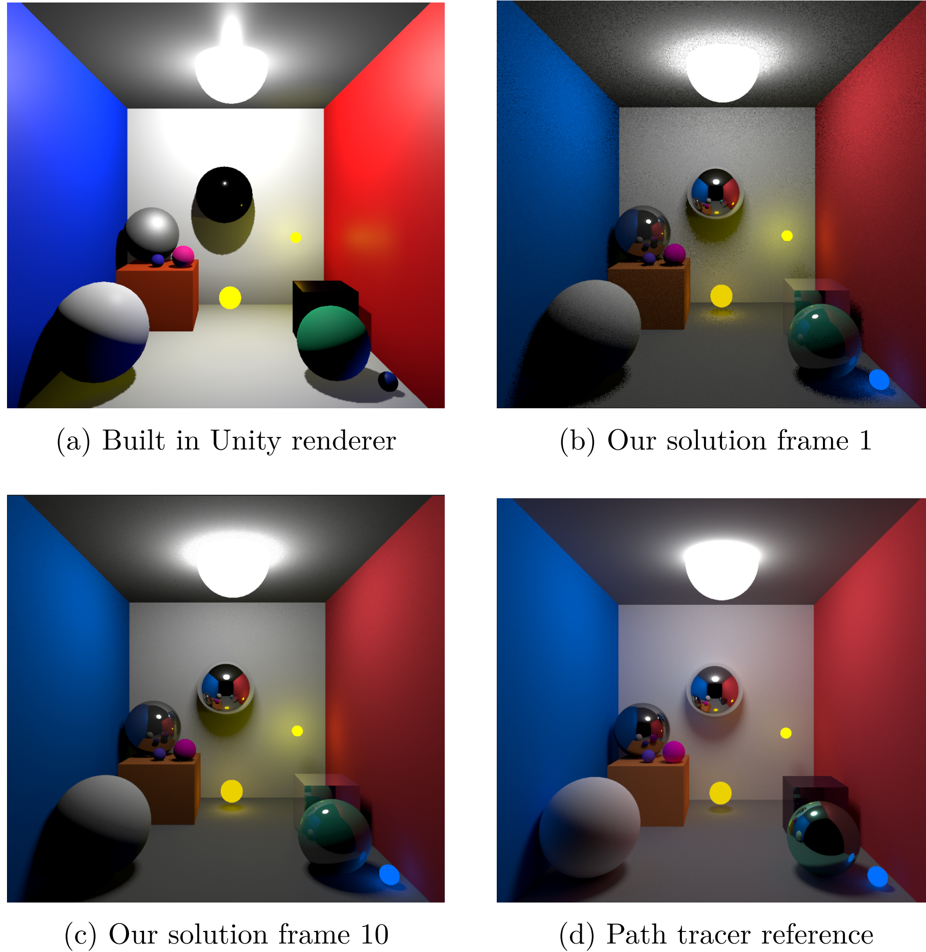
4.3 Results

To present the results, we will show the speed up over the reference model and compare the outputs. We will show that with the most accurate settings selected, the output is close to that of the reference model and appears to have a higher visual fidelity than the built in Unity renderer. In addition, we will present the limits of this model and also show the impact of the post processing on frame rate and visual fidelity.

4.3.1 Comparison to Reference

Figures 4.9, 4.10, and 4.11 show comparisons of the Unity built in renderer, our solution on frame 1 and frame 10, and our path tracer reference. Our solution provides a good approximation of the reference solution while converging about 3000x faster based on the time difference between most visible noise being suppressed. The biggest differences between the approximation and the reference model are due to the absence of a global illumination system. This would provide color bleeding and softer, scattered lighting which would add to the believability of the renders. Our light attenuation function, while fast, is also not a perfect approximation of light and noticeable differences show up where the lights hit

However, as seen from the test environments shown in the figures above, our renders appear to have a higher fidelity than rasterization is able to provide. This is due to the absence of real reflections in the raster engine and also due to the method of sampling lighting that we are using. Our model balances visual fidelity and speed by supplementing realistic lighting simulations with optimizations and approximations.



(a) Built in Unity renderer (b) Our solution frame 1

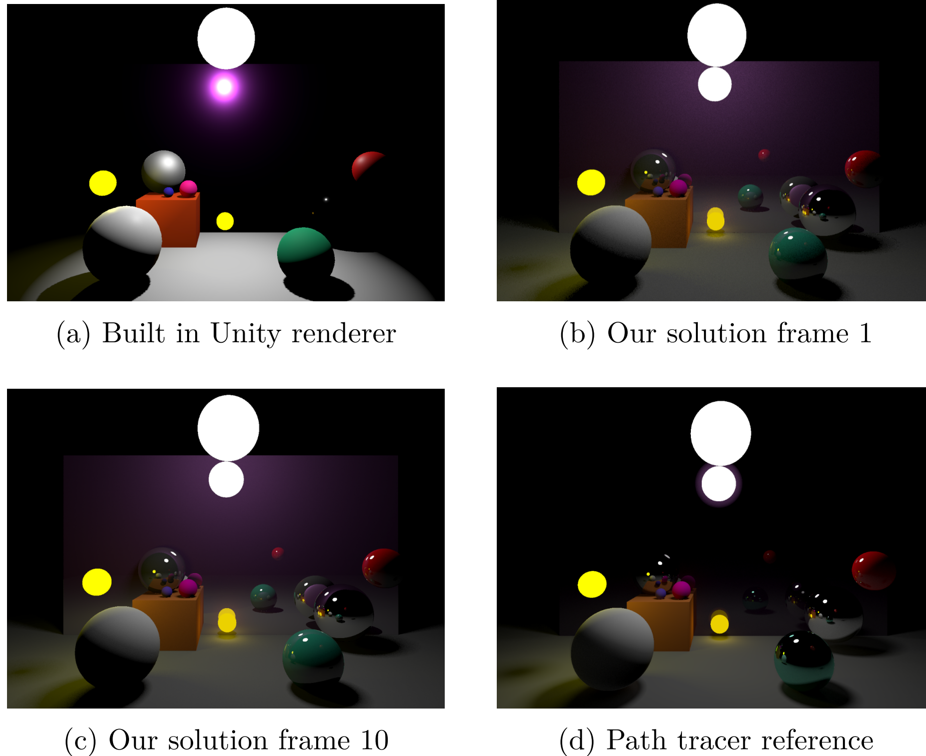
(c) Our solution frame 10 (d) Path tracer reference

Figure 4.9: Reference environment 1.

Figure 4.12 shows the differences between the first frames of the path tracer and our hybrid approach. Our model is able to converge far quicker than the reference model due to the tradeoffs in accuracy that we made. Not only does it converge in far fewer frames, it also renders those frames faster.

Table 4.1: Average frame rates in each reference environment with the path tracer simulating 2 samples per pixel with 4 bounces each.

	Environment 1	**Environment 2**	**Environment 3**
Hybrid	160 FPS	263 FPS	168 FPS
Path Tracer	135 FPS	170 FPS	46 FPS



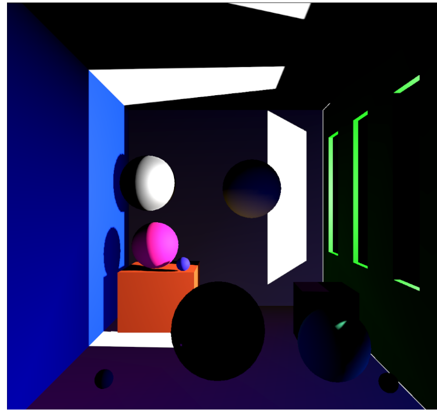
(a) Built in Unity renderer (b) Our solution frame 1

(c) Our solution frame 10 (d) Path tracer reference

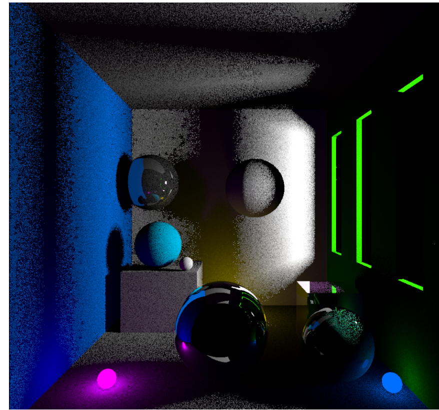
Figure 4.10: Reference environment 2.

Table 4.1 shows the average frame rates for each of the shown reference environments. Not only do we see that our model out performs the reference in terms of speed, we can also see that in reference environment 3, which has more triangles than the others, our model is far more resistant to jumps in geometry than the path tracer. This is because, as previously mentioned, the intersection calculations are the most expensive part of a ray tracing engine and our approximation casts far fewer rays than the path tracer because of the importance sampling we use.

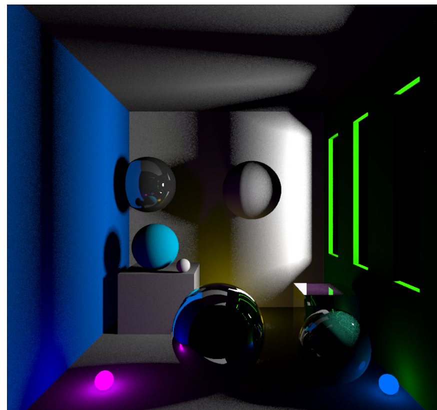
However, the computational costs are not the only measure. We actually see that despite our model being more resilient in environment 3, our results appear visually further from the reference than in the others. Some environmental situations are difficult for our hybrid approach to handle because of our approximations. Environment 3 is lit primarily by indirect



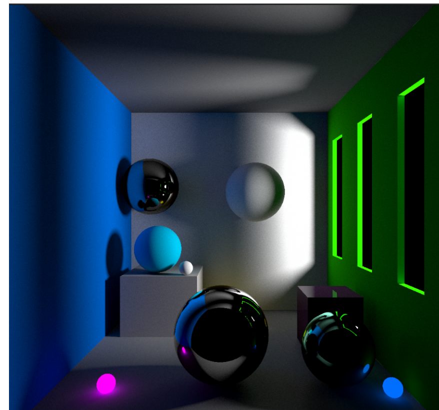
(a) Built in Unity renderer



(b) Our solution frame 1



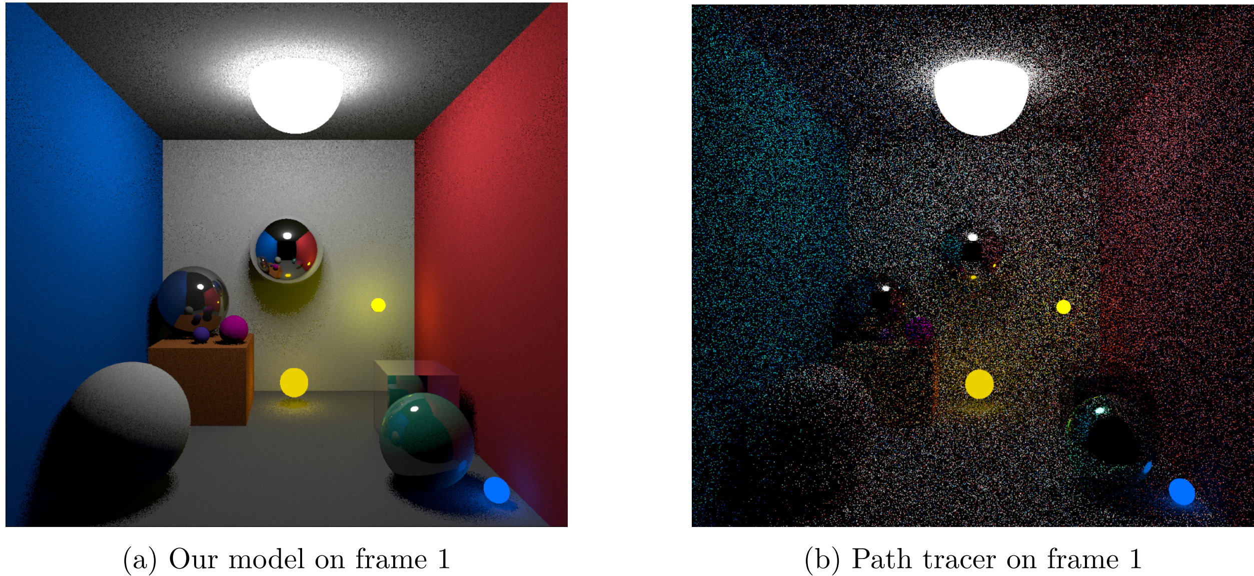
(c) Our solution frame 10



(d) Path tracer reference

Figure 4.11: Reference environment 3 (lit by offscreen light on the right).

lighting bouncing after coming through the windows. Since our model does not simulate global illumination, this is one of the worst cases for it. Another bad case is one with many light sources. While the path tracer performance does not depend on the number of light sources at all, since we are importance sampling each light, our algorithm is $O(N)$ where N is the number of lights. When tripling the number of lights in environment 3, performance drops to an average of 70 FPS.



(a) Our model on frame 1 (b) Path tracer on frame 1

Figure 4.12: Comparison of rendering a single frame from our model versus the reference model.

4.3.2 Limits

A bounding volume hierarchy would be necessary to raise the upper polygon limit of our model to a realistic value given today's standards. A scene will typically have hundreds of thousands of triangles with some well managed scenes achieving well into the millions by utilizing additional culling techniques like level of detail meshes, frustum culling and more. While our hybrid approach is not able to achieve this level of geometry, if we implemented some acceleration structure like a bounding volume hierarchy, it is likely that we would be able to achieve similar goals. Chapter 5 will discuss this in more detail, while here we will present the results of a geometry stress test.

We tested the upper limit of our solution by adding triangles to the scene until the average frame rate dropped below 60 FPS. Figure 4.13 shows the scene with all the triangles we managed to render before this happened. This frame contained about 21 thousand triangles and was rendering at an average frame rate of 66 FPS. Our path tracer rendered exactly the

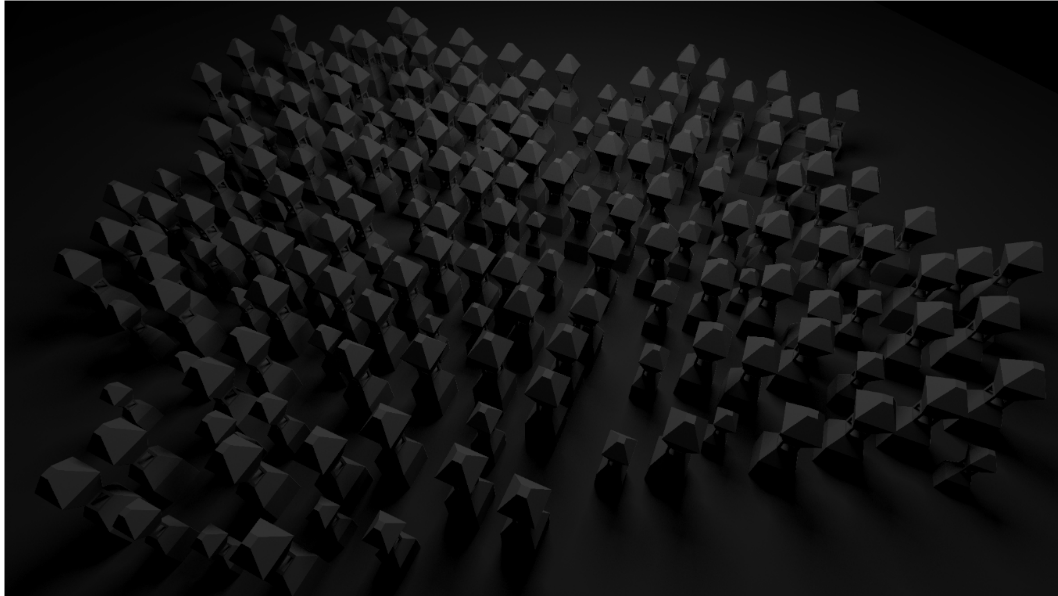
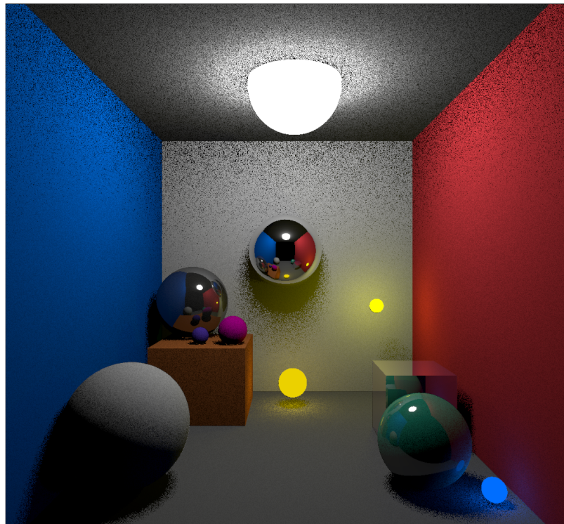


Figure 4.13: Geometry stress test with just under 21 thousand triangles.

same scene from the same position at about 4 FPS. We also added additional lights to the scene since our hybrid solution's performance depends directly on that and saw that with two lights in this scene, performance dropped to 53 FPS, and with 4 lights it dropped further to 37 FPS. On the other hand, our path tracer is independent of the number of light sources and remained at an average of 4 FPS.

An edge case that presents a significant challenge is when the full virtual image plane is filed with a single detailed mesh. When this happens, every pixel must test every triangle in that mesh. We tested and saw that the frame rate dropped to an average of 45 FPS from 66 FPS when the camera was positioned so that a single mesh from Figure 4.13 filled the entire screen. Each of the meshes depicted in that scene contain 104 triangles and while this is not high resolution by modern standards, without an acceleration structure it demands a huge amount of work from our ray tracer.

4.3.3 Post Processing


(a) Without post processing. Average FPS of 395

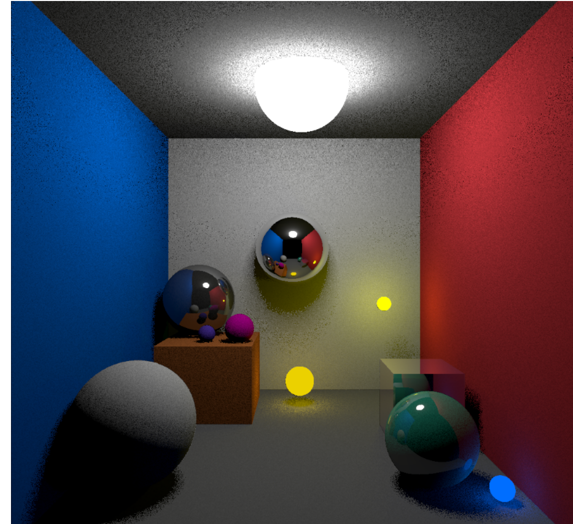

(b) With post processing. Average FPS of 170

Figure 4.14: Demonstration of the effect of the post processor.

The post processing impacts the frame rate heavily but this is purposeful. Recalling the way it works, it estimates the amount of additional work in can do in a frame without dipping below a target frame rate. Figure 4.14 shows the output with and without the post processing. The frame rate drops from an average of 395 FPS without it to an average of 170 with it due to the additional rays being sent in the noisiest areas around the light source. However, this is because it's launching an additional eight samples per pixel in the mask, so despite the frame rate drop it will converge much faster than without it. Specifically, it takes about 150ms across 50 frames for an image to converge without post processing, but only 59ms across 10 frames with it. Figure 4.14 supports these results as we can see that the image with the post processor appears far less noisy after a single frame than the image without it. One down side to this post processor is that the PID control constants need to be tuned on a per-machine basis. This is not as intuitive as the other settings available with our solution and is the biggest downside to using PID.

4.4 Insights

This solution, while substituting approximations in many places, meets our goal of producing images with higher visual quality than rasterization can do while still running at real time rates without hardware accelerators. However, there are a few things which could be improved to increase the quality of this simulation. First, one of the biggest sources of noise currently occurs when lights intersect the geometry of other objects in a scene. This is because we naively sample a random point on the surface of the hemisphere around the direction from the light's center to the point being lit. If instead we were able to detect this and only sample portions of the light which are possible to hit, we could reduce the noise by a lot. In addition to this, we could potentially completely eliminate noise by developing some heuristics to determine the portion of a light visible from a certain point and use that to create the penumbra regions rather than random samples. This could look like borrowing more ideas from rasterization such as using the stencil buffer to project light sources against everything else in the scene.

Another idea that could dramatically reduce the noise would be to use screen space caches instead of temporal accumulation. This idea is similar to the work presented by [30] and could allow us to keep cached data across frames despite motion. The key idea is to project the screen space cache from the current frame into the last one given the camera transformation deltas. This would allow us to save the samples accrued for scene points despite camera motion and potentially even for points which leave the view port before returning.

Chapter 5

Conclusion

With Chapter 4, we were able to meet our goal of producing images with a higher visual fidelity than what rasterization can produce while still achieving real time frame rates. Our solution is able to simulate lighting, shadows, and reflections, for diffuse materials with specular reflections. This could easily be extended to handle transparent materials or light shapes other than spheres as well. However, given more time, there are two natural continuations to the work done in this thesis, and those are constructing an acceleration structure and adding global illumination to our approach in Chapter 4.

5.1 Bounding Volume Hierarchy

While we did not have the time to implement a bounding volume hierarchy, these acceleration data structures are crucial to the deployment of any ray tracing engine. They raise the upper limit on the geometry count in a scene by orders of magnitude by drastically reducing the amount of ray-triangle intersection tests needing to be done. Ray-geometry tests are the most expensive part of a ray tracing algorithm, and of them ray-triangle intersection tests are typically the worst. A bounding volume hierarchy is similar to an octree in that it hierarchically breaks a tree like structure, such as a triangle mesh, into nested containers. We can then quickly query an object by traversing the bounding volumes using a much faster ray-box test before sampling the last few triangles which are contained by the leaves of the

bounding volume hierarchy. We therefore trade iterating and testing every triangle in the scene for an intersection per ray cast for querying a tree like data structure of faster box intersections.

[26] gives methods for forming dynamic bounding volume hierarchies which account for skinned meshes, those which deform over time. [3] also talks about scene level bounding volume hierarchies which consider objects in the scene rather than triangles in a mesh. Combining both of these ideas, we could drastically cut down on the number of rays being tested per frame. Our current method performs brute force ray casting by iterating over every triangle or sphere in the scene and testing against it regardless of if there is no change the ray could hit it. The one optimization we do currently make is testing against an axis aligned bounding box before sampling each triangle in a mesh.

In the previous chapter, we presented an edge case that occurred when a single high detail mesh filled the screen and was bad for performance. This is exactly the kind of case that a bounding volume hierarchy would handle well. Instead of testing every triangle in the mesh filling the camera's view, we would instead traverse the hierarchy before testing only the few triangles which were contained by the leaves of the hierarchy.

5.2 Adding Global Illumination to the Hybrid Approach

The absence of a global illumination system is responsible for the most apparent differences between our hybrid solution and our reference model. Without the ability to simulate light bouncing off of surfaces and indirectly illuminating others, we cannot achieve anything resembling a fully realistic image. Our goal in this work was to produce images with a higher fidelity than rasterization outputs; however, in the future, we could update our solution to render outputs which are closer to the realism displayed by the path tracer by incorporating

global illumination.

While the experiment in Chapter 3 was a failure overall, it did provide a good approximation of color bleeding and therefore indirect illumination. With more research, it would be interesting to adapt that solution to augment our hybrid approach. While the irradiance volume as constructed in this paper would likely not work due to the desire for solutions which support fully dynamic scenes, the idea of caching irradiance for later use is still useful. Wright et al. [30] and [24] both provide interesting approaches which specifically target dynamic scenes as the caches are built and updated during runtime. Both of these works place the caches in screen space and on the surfaces of geometry respectively, but it would be interesting to consider keeping the idea of caching in a volume rather than on surfaces because it could potentially be more responsive to first shot lighting. Specifically, situations where there is no cache built up could benefit when an object appears in space and there has been no previous ability to cache based on its surfaces.

In this thesis we have presented our work in optimizing ray tracing to run in real time without hardware accelerators. We were able to provide a system which trades accuracy for speed while still producing a higher quality output than rasterization engines. We are able to simulate dynamic scenes with lighting, shadows and reflections at real time frame rates. Given additional time to incorporate an acceleration structure and further research to add a global illumination system, this solution could be considered a viable option for games and other real time applications that aim to approach realistic light transport by fast estimation.

Bibliography

[1] Monte carlo integration, 2020. URL https://www.cs.princeton.edu/courses/archive/fall20/cos302/notes/cos302_f20_lecture19_monte.pdf.

[2] Tavian Barnes. Tavianator.com, May 2011. URL https://tavianator.com/2011/ray_box.html.

[3] Erin Catto and Blizzard Entertainment. Dynamic bounding volume hierarchies, 2019.

[4] Manfred Ernst and Gunther Greiner. Multi bounding volume hierarchies. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 35–40. IEEE, 2008.

[5] Gene Greger, Peter Shirley, Philip M Hubbard, and Donald P Greenberg. The irradiance volume. *IEEE Computer Graphics and Applications*, 18(2):32–43, 1998.

[6] Warren Hunt and William R Mark. Ray-specialized acceleration structures for ray tracing. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 3–10. IEEE, 2008.

[7] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environments. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, page 133–142, New York, NY, USA, 1986. Association for Computing Machinery. ISBN 0897911962. doi: 10.1145/15922.15901. URL https://doi.org/10.1145/15922.15901.

[8] Henrik Wann Jensen and Niels Jørgen Christensen. Photon maps in bidirectional monte carlo ray tracing of complex objects. *Computers & Graphics*, 19(2):215–224, 1995.

[9] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, page 143–150, New

York, NY, USA, 1986. Association for Computing Machinery. ISBN 0897911962. doi: 10.1145/15922.15902. URL https://doi.org/10.1145/15922.15902.

[10] Linus Källberg and Thomas Larsson. Optimized phong and blinn-phong glossy highlights. *Journal of Computer Graphics Techniques Vol*, 3(3), 2014.

[11] Chun-meng Kang, Lu Wang, Yan-ning Xu, and Xiang-xu Meng. A survey of photon mapping state-of-the-art research and future challenges. *Frontiers of Information Technology & Electronic Engineering*, 17(3):185–199, 2016.

[12] Sanjeev J Koppal. Lambertian reflectance. In *Computer vision: a reference guide*, pages 729–731. Springer, 2021.

[13] David Kuri. Gpu ray tracing in unity – part 1, Dec 2023. URL https://www.gamedeveloper.com/programming/gpu-ray-tracing-in-unity-part-1.

[14] Sebastian Lague. Ray-tracing. URL https://github.com/SebLague/Ray-Tracing.

[15] Zheng Lyu, Krithin Kripakaran, Max Furth, Eric Tang, Brian Wandell, and Joyce Farrell. Validation of image systems simulation technology using a cornell box. *arXiv preprint arXiv:2105.04106*, 2021.

[16] Wojciech Matusik, Fredo Durand, and Jaakko Lehtinen. Global illumination and monte carlo. URL https://ocw.mit.edu/courses/6-837-computer-graphics-fall-2012/1b5985f78c68379e15543fe27d41b72c_MIT6_837F12_Lec18.pdf.

[17] Ment Reeze, Menno A. Veerman, and Chiel C. van Heerwaarden. Machine learning-based denoising of surface solar irradiance simulated with monte carlo ray tracing, 2025. URL https://arxiv.org/abs/2411.06574.

[18] Hiroyuki Sakai, Christian Freude, Thomas Auzinger, David Hahn, and Michael Wimmer. A statistical approach to monte carlo denoising. In *SIGGRAPH Asia 2024 Conference Papers*, pages 1–11, 2024.

[19] Peter Shirley. Ray tracing in one weekend. *Amazon Digital Services LLC*, 1(4), 2018.

[20] Peter Shirley and Kenneth Chiu. Notes on adaptive quadrature on the hemisphere. 1994.

[21] Peter Sturm. Pinhole camera model. In *Computer Vision: A Reference Guide*, pages 983–986. Springer, 2021.

[22] Peter Sturm and Srikumar Ramalingam. A generic concept for camera calibration. In *Computer Vision-ECCV 2004: 8th European Conference on Computer Vision, Prague, Czech Republic, May 11-14, 2004. Proceedings, Part II 8*, pages 1–13. Springer, 2004.

[23] Ping Tan. Phong reflectance model. In *Computer Vision: A Reference Guide*, pages 962–964. Springer, 2021.

[24] Wolfgang Tatzgern, Alexander Weinrauch, Pascal Stadlbauer, Joerg H Mueller, Martin Winter, and Markus Steinberger. Radiance caching with on-surface caches for real-time global illumination. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 7(3):1–17, 2024.

[25] Jonathan Thaler and TU Wien. Deferred rendering. *TU Wein: Vienna, Austria*, 2011.

[26] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics (TOG)*, 26(1): 6–es, 2007.

[27] Whittaker. URL https://pages.stat.wisc.edu/~mchung/teaching/MIA/reading/diffusion.gaussian.kernel.pdf.pdf.

[28] Turner Whitted. An improved illumination model for shaded display. In *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, page 14, 1979.

[29] Turner Whitted. An improved illumination model for shaded display. In *ACM Siggraph 2005 Courses*, pages 4–es. 2005.

[30] David Wright, Krzysztof Narkowicz, and Patrick Kelly. Lumen. URL https://advances.realtimerendering.com/s2022/SIGGRAPH2022-Advances-Lumen-Wright%20et%20al.pdf.

[31] Liew Wen Yen, Rajermani Thinakaran, and J Somasekar. Machine learning-based denoising techniques for monte carlo rendering: A literature review. *Machine Learning*, 16(2), 2025.

[32] Juyong Zhang, Bailin Deng, Zishun Liu, Giuseppe Patanè, Sofien Bouaziz, Kai Hormann, and Ligang Liu. Local barycentric coordinates. *ACM Trans. Graph.*, 33 (6), November 2014. ISSN 0730-0301. doi: 10.1145/2661229.2661255. URL https://doi.org/10.1145/2661229.2661255.

Appendices

Appendix A

Appendices I

A.1 A1

A.2 A2