Sorted Shading for Uni-Directional Pathtracing

Joshua Bainbridge

Masters of Science,
Computer Animation and Visual Effects

Bournemouth University

August, 2015

Contents

Table of contents i
List of figures iii
Abstract iv
Acknowledgements v

1 Introduction **1**

2 Related Work **3**

3 Theory **6**
3.1 Radiometry 6
3.2 The Rendering Equation 7
3.3 Monte Carlo Integration 10
3.4 Multiple Importance Sampling 11
3.5 Russian Roulette Termination 12

4 Algorithms **13**
4.1 Uni-directional Pathtracing 13
4.2 Sampling Strategies 14
4.2.1 Next Event Estimation 14
4.2.2 Combining Strategies 14
4.3 Sorted and Deferred Shading 15

5 Implementation **18**
5.1 Libraries 19
5.2 Requirements 20
5.3 Design 21
5.4 Details 23
5.4.1 Pathtracer Class 23

5.4.2 Parallel Processes 25
5.4.3 Modular Interface 27
5.4.4 Scene Description 30
5.5 Example Program 31

6 Results **32**
6.1 Variance Reduction 33
6.2 Coherent Rendering 33

7 Conclusion **37**
7.1 Further Development 37

References **38**

A Detailed Diagrams **41**

B Example Scene File **43**

List of Figures

3.1 Definition of radiance (Dutre *et al.*, 2006) 7
3.2 Different importance functions (Dutre *et al.*, 2006) 10

4.1 Pathtracing with next event estimation (Dutre *et al.*, 2006) . . 15

5.1 Simplified diagram of the main system 22
5.2 Overview of the Pathtracer class 23
5.3 Simplified diagram of the module interfaces 27

6.1 Physically correct image created using the implementation . . 32
6.2 Equally weighted integrals 33
6.3 Multiple importance sampling 34
6.4 Performance test scene 34
6.5 Batch size against run time with a depth of two 36
6.6 Batch size against run time with varying depth 36

A.1 Detailed diagram of the main system 41
A.2 Detailed diagram of the modules 42

Abstract

Physically based models for light transport using ray tracing have become standard practice in industry as a result of changing demands and computational power. This approach simplifies pipelines and naturally solves many issues found with rasterized methods at the expense of computation. There has been much research and development into reducing such cost although there are other factors that contribute to latency, such as memory access and cache coherence of large production data sets. With ever increasing scene complexity, out of core access to geometry and texture data become an issue. This thesis investigates a current solution to this problem as described by Eisenacher *et al.* (2013) in combination with other well established optimisations for production rendering. This solution involves sorting rays and hit points into batches that are streamed from disk into main memory, allowing for optimal ray traversal and perfectly coherent shading. As a result we negate the need for such methods as irradiance caching that compromise artistic intent and limit scalability.

Keywords: light transport, rendering, memory coherency

Acknowledgements

This thesis would not have been possible without the help and guidance of Jon Macey, Mathieu Sanchez and Ian Stephenson. I would also like to thank Georg Finch for his encouragement over the last couple of years as well as friends and family for their continued support.

Chapter 1

Introduction

In recent years there has been a renaissance in research for efficient light transport methods in computer graphics. This has resulted in unbiased ray tracing for image synthesis and global illumination becoming popular within film production and the creative industries. Reasons for this are found in the physically based nature of these approaches, producing more plausible results and allowing for greater complexity. They also replace previously biased methods and as a result they simplify pipelines and improve scalability at the cost of computation. However the needs of production rendering and that of predictive rendering are very different with much research being concerned with the later. Developing more consistent estimators in relation to wave and quantum optics don't necessarily benefit production, as it is not concerned with scientific analysis but rather visual fidelity. Production needs and as a result research are usually characterised by four distinct areas: sampling strategies, efficient ray traversal, scalability and memory performance. With advancement in silicon processors and their scalability as well as the ever increasing demand on geometric and shading complexity, memory performance becomes an issue. This is potentially compounded with incoherent and out-of-core access commonly found with ray traced methods.

This thesis will focus on the development of a rendering architecture based upon that described in Eisenacher *et al.* (2013) which mitigates the issues surrounding memory performance when processing large data sets. The implementation involves a two stage sorting procedure of large and out-of-core ray batches to obtain coherency in both ray traversal and surface shading. An optimised ray tracing kernel takes advantage of the coherence

to efficiently traverse scene geometry while sequential and vectorized texture lookup improves cache coherency when shading. This is then implemented around a uni-directional pathtracer with modern sampling optimisations and a highly parallel, concurrent and re-entrant design. Current standards in colour and image management are also taken into account during development of a command line tool for testing.

An overview of the development of ray traced solutions towards image synthesis is covered in chapter two while the relevant theory behind physically based rendering is explained in chapter three. In chapter four we develop upon the theory discussed in chapter three and describe an algorithmic solution to the lighting integral as well the architecture that is later implemented. Then for chapter five there will be a detailed explanation of the implementation itself and it's design methodology. Finally in chapter six we will conclude with suggestions for further development and optimisation.

Chapter 2

Related Work

Ray tracing for computer graphics was originally suggested by Whitted (1980) and has become a very popular area of research. This is partly due to the simplicity and elegance of its design as well as the visual fidelity it can reproduce. After this initial paper, it was developed upon by Cook *et al.* (1984) with distributed ray tracing that used stochastic sampling to produce a more accurate lighting model. This allowed for such physical phenomenon as depth of field, motion blur and glossy surfaces but also introduced variance. As a result it meant that multiple samples needed to be taken to resolve an image, increasing the computational cost.

Later Kajiya (1986) proposed the rendering equation that allowed for a more physically based representation of energy redistribution and transport in non-participating media. The same paper also introduced an algorithmic solution to this integral equation called uni-directional pathtracing that solved the problem with a Monte Carlo based numerical integration. This extended the concept of a ray into a path that was randomly chosen and accumulated radiance throughout a scene. Pathtracing can produce very plausible results in comparison to previous ray tracing and rasterized approaches but was too computationally expensive to be practical in a production environment.

Sampling optimisations were then introduced in the seminal paper by Veach (1998) that took advantage of both classical importance sampling as well as methods for combining multiple sampling strategies called multiple importance sampling.

Other biased approaches are also suggested such as irradiance caching which was originally proposed by Ward *et al.* (1988) and methods based on nearest neighbour estimation called photon mapping, introduced by Jensen (1996). These amortised the rendering equation and will cache forms of light transport to save on computation. When used with traditional ray tracing they are referred to as multipass methods and require pre-render passes and a specific shader design. This can improve performance considerably but also complicates production pipelines and compromises artistic intent.

Veach also defined the path integral formulation of the rendering equation to combine both multiple importance sampling with a bi-directional path tracing algorithm. This was then further developed with Markov Chain Monte Carlo sampling strategies that was originally noted in Kajiya (1986). More recently there has been work done on combining bi-directional pathtracing and photon mapping in Georgiev *et al.* (2012), known as vertex connection and merging.

An overview of these different solutions to the rendering equation can be found in both Dutre *et al.* (2006) and Pharr and Humphreys (2010). Many of these methods significantly improve sampling performance for scenes with complex light transport. However for scenes that don't exhibit such complexity, it incurs considerable computational overhead in comparison to standard uni-directional pathtracing. Another issue is that these advanced sampling strategies and algorithms only solve the sampling problem. They do not address ray traversal, memory performance or scalability.

There has been work on utilising single instruction multiple data to improve ray traversal. This research has a direct impact on production rendering as it does not address corner case scenarios but rather the overall performance of the system. Ray packets were introduced with Wald *et al.* (2001) which aligned rays into vector lanes for instruction level parallelism on modern processors. This meant that multiple rays could be tested against a single node within an acceleration structure simultaneously, resulting in faster traversal. This however relies on ray coherence in both location and direction which is unlikely due to the random nature of pathtracing algorithms. Without coherence the effectiveness of this optimisation decreases and sometimes negatively affects performance as vectorized instructions can have potential overheads.

An alternative to this is described in Wald *et al.* (2008) where the nodes of a bounding volume hierarchy are vectorized allowing for better performance with both coherent and incoherent rays. Results have shown however that this approach does not perform as well as using coherent ray packets.

A highly optimised framework for ray traversal called Embree is described in Wald *et al.* (2014) which implements both ray packets and vectorized bounding volume hierarchies while automatically optimising for the processing architecture. This however is still dependent upon ray coherence for optimal results.

The issue of memory coherence when ray tracing was addressed in Hanrahan (1986) where breadth first traversal was used to allow for geometric cache coherency. Memory limitations meant that this was done for every individual scanline which gave significant increase in performance but was not optimal. Ray queueing at grid boundaries was introduced by Pharr *et al.* (1997) which focused on efficient caching of out-of-core geometry. This was also a breadth first approach and as a result has the same issues with memory limitations when sorting rays. They mitigate this problem by taking advantage of computational decomposition and storing path information such as probability and importance with the original samples. This however means that other common methods used within global illumination algorithms such as adaptive light sampling can not be implemented.

This research into memory coherence has shown significant improvement in performance when considering ray traversal of scene geometry. However there has been little work on explicitly improving shading and texture coherence apart from that found naturally as a result of sorting for traversal. This has been the primary focus of Eisenacher *et al.* (2013) in which they present a solution to coherence for both ray traversal and surface shading. They use a breadth first approach and stream compressed ray batches to disk to handle memory limitations. Each batch will pass through a two stage sorting procedure to find coherence in both ray traversal and hit point shading. This has shown to substantially improve performance when handling data sets with considerably large and layered textures during the shading process.

Chapter 3

Theory

Physically based rendering involves accurately modelling the transport of electromagnetic radiation through space and its interaction with objects and volumes. The energy is measured across a surface to resolve an accurate two dimensional representation of a three dimensional scene. This scene is usually defined as surfaces that reflect, transmit and radiate energy using parametric data as well as statistical models to describe the surfaces on a micro level.

3.1 Radiometry

Here we will define light as the visual range within the electromagnetic spectrum and use the geometric model as described in Dutre *et al.* (2006). Light can be measured according to several radiometric quantities. The most basic of these is flux which is commonly written as Φ and is represented in watts. Irradiance and radiosity are the incident and exitant quantities with respect to unit area and are both expressed as watts/m^2. Radiance is the most applicable measurement in light transport and is expressed as flux per unit projected area per unit solid angle.

The reason for radiance being commonly used when discussing light transport is that it is invariant along straight lines when not taking into account participating media. If the incident and exitant positions of radiance are reversed or the distance changed, the amount of energy measured is constant.

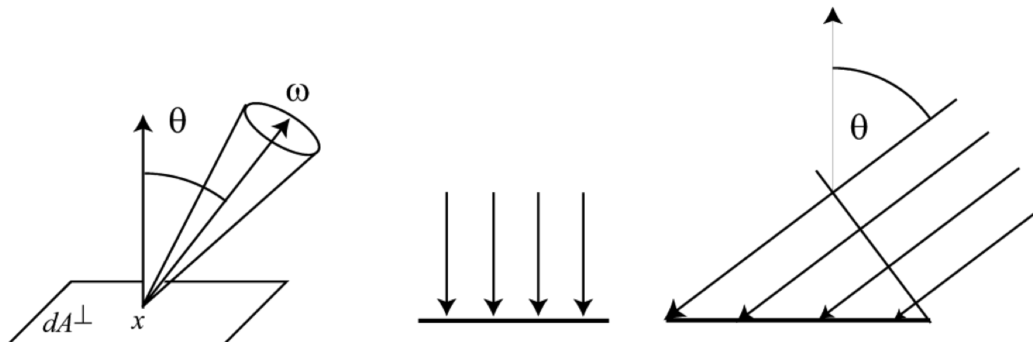


Figure 3.1: Definition of radiance (Dutre *et al.*, 2006)

When considering radiance, the projected area as seen in figure 3.1 is equivalent to the cosine of the surface normal and the incident direction. This is due to the projected area distributing the energy across a large surface and as a result reducing irradiance which is measured per unit area.

3.2 The Rendering Equation

A mathematical formulation for the distribution of light in a graphical context was first introduced by Kajiya (1986) and named the rendering equation. This describes the radiance at a position on a surface as the sum of emitted and reflected energy in a particular direction. We also make the assumption that light travels instantaneously, is not spectrally dependent and travels through a vacuum without interacting with participating media. The rendering equation can be written as

$$L(x \to \Theta) = L_e(x \to \Theta) + \int_{\Omega_x} L(x \leftarrow \Psi) f_r(x, \Psi \to \Theta) cos(N_x, \Psi) d\omega_\Psi \quad (3.1)$$

where $L(x \to \Theta)$ is the total radiance in a direction and equals the summation of both the emitted and reflected energy. Emitted radiance is written as $L_e(x \to \Theta)$ while reflected radiance is an integration of three parts across the unit hemisphere above the surface. The first part being written as $L(x \leftarrow \Psi)$ is the incident radiance from all directions. This is then scaled by $f_r(x, \Psi \to \Theta)$ which represents the bidirectional reflectance distribution function of that position. The final part is the cosine term which converts the surface irradiance into radiance, accounting for the projected area.

As there is an unknown quantity of radiance on both sides, it is known as a Fredholm equation of the second type, making it a recursive integral. This becomes an issue for computing an unbiased estimate, but this will be addressed later in section 3.5 of this chapter.

A bidirectional reflectance distribution function (BRDF) describes how a surface reflects the incident light and must have at least two properties as stated in Pharr and Humphreys (2010) to be considered physically correct. The first of these is called Helmholtz reciprocity which states that if both the incident and exitant directions are interchanged then the amount of light reflected is the same. The second property is the law of energy conservation which means that the total amount of energy reflected in all directions must be equal to or less than the total amount of energy received from all incident directions.

Another equation known as the response function measures the propagation of importance instead of radiance in the opposite direction. This can allow for a more intuitive description of the problem as computation that has no effect on the image plane is never taken into consideration. It is written as

$$W(x \rightarrow \Theta) = W_e(x \rightarrow \Theta) + \int_{\Omega_x} W(x \leftarrow \Psi) f_r(x, \Psi \leftarrow \Theta) cos(N_x, \Psi) d\omega_\Psi \tag{3.2}$$

where the bidirectional reflectance distribution function is reversed and all radiance terms are replaced with their equivalent variant for importance.

In Dutre *et al.* (2006) other formulations of this equation are described that can become useful when efficiently solving the problem. The original rendering equation takes the integral of radiance across the hemisphere but it can also be described as the integration of all visible surfaces at that position. This is called the surface area formulation and is written as

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + \int_A L(y \rightarrow \overrightarrow{yx}) f_r(x, \Psi \leftrightarrow \Theta) V(x, y) G(x, y) dA_y$$
$$G(x, y) = \frac{cos(N_x, \Psi) cos(N_y, -\Psi)}{r_{xy}^2} \tag{3.3}$$

where $V(x,y)$ is the visibility term between both positions and $G(x,y)$ is the geometric term that defines the probability of radiosity being received from that surface area.

The surface area formulation allows for a decomposition of the rendering equation as shown in Dutre *et al.* (2006) into separate integrals for both direct and indirect contribution. This can be written as

$$\begin{aligned} L(x \rightarrow \Theta) &= L_e(x \rightarrow \Theta) + L_r(x \rightarrow \Theta) \\ L_r(x \rightarrow \Theta) &= \int_{\Omega_x} L(y \leftarrow \Psi) f_r(x, \Psi \rightarrow \Theta) cos(N_x, \Psi) d\omega_\Psi \\ L_r(x \rightarrow \Theta) &= L_{direct} + L_{indirect} \end{aligned} \tag{3.4}$$

where direct and indirect integrate according to both the surface area and hemispherical formulation respectively as

$$\begin{aligned} L_{direct} &= \int_A L_e(y \rightarrow \overrightarrow{yx}) f_r(x, \overrightarrow{xy} \rightarrow \Theta) V(x,y) G(x,y) dA_y \\ L_{indirect} &= \int_{\Omega_x} L_i(x \leftarrow \Psi) f_r(x, \Psi \rightarrow \Theta) cos(N_x, \Psi) d\omega_\Psi \\ L_i(x \leftarrow \Psi) &= L_r(r(x, \Psi) \rightarrow -\Psi) \end{aligned} \tag{3.5}$$

and $r(x,\Psi)$ is a position on the hemisphere. This means that separate sampling strategies can be used for each integral and correctly weighted according to some sort of heuristic.

The path integral formulation was introduced by Veach (1998) and is another useful representation of the original rendering equation. It describes the throughput of light through all possible paths of any length within global path space. This is better suited than the original recursive equation for discussion of more complex algorithms such as bi-directional pathtracing. It can be written in the form of

$$\Phi(S) = \int_{\Omega^*} f(\bar{x}) d\mu(\bar{x}) \tag{3.6}$$

where Ω^* denotes the path space and $\bar{x}$ is a path of any length with $f(\bar{x})$ being the throughput of radiance along that path. This throughput takes

into account all weighting along the path as well as the light radiance and camera importance at both ends.

3.3 Monte Carlo Integration

Solving the rendering equation in all but the most trivial of scenarios involves numerical integration and is done using a Monte Carlo estimator. This can be written as

$$\langle I \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i, y_i)}{p(x_i, y_i)} \tag{3.7}$$

for a two dimensional integration where $f(x_i, y_i)$ is the function and $p(x_i, y_i)$ is the probability density (PDF) of the sample taken. The advantage of Monte Carlo methods for numerical integration is that it makes expanding into higher dimensional space relatively simple and computational linear.

An optimisation noted by Kajiya (1986) is that importance sampling can be used to substantially reduce variance within the estimator. This involves sampling according to a probability density function that is equal or similar to the function being evaluated. The value is then divided as seen in equation 3.7 by the probability, resulting in an unbiased estimate. If however the probability density function from which a sample is created does not resemble the function being evaluated, then the variance will increase.

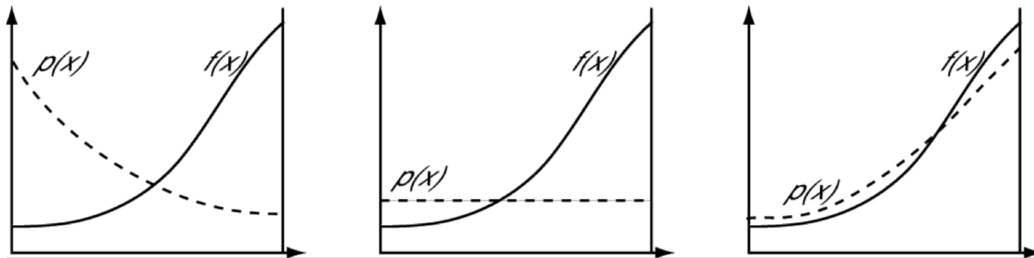


Figure 3.2: Different importance functions (Dutre *et al.*, 2006)

Sampling according to the function itself can be very difficult or impossible, so usually a similar function is used that has a known probability density.

This function is called an importance function and is illustrated in figure 3.2 with a worst to best case scenario from left to right respectively. The function is then converted into a cumulative distribution function (CDF) and finally inverted to allow for uniform variables to be mapped onto the correct distribution.

3.4 Multiple Importance Sampling

Multiple sampling strategies can be used to evaluate the same rendering equation as is seen with equation 3.1 and equation 3.3. These different strategies can both be importance sampled but only one might be effective. Multiple importance sampling is a method introduced by Veach (1998) as a means to sample multiple strategies and weight them according to their respective probability. When using multiple importance sampling, the estimator is written as

$$\langle I \rangle_{MIS} = \sum_{i=1}^{m} \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(X_{i,j}) \frac{f(X_{i,j})}{p_i(X_{i,j})} \tag{3.8}$$

where we have m different sampling strategies. The $w_i(x)$ function is a weighting term called the balance heuristic and is described as

$$w_i(x) = \frac{n_i p_i(x)}{\sum_{k=1}^{m} n_k p_k(x)} \tag{3.9}$$

where the denominator takes into account all other sampling strategies. Although the balance heuristic is proven by Veach (1998) to be near optimal in most scenarios, another combination strategy named the power heuristic can sometimes be more effective. This is best used when noise is evident from low-variance problems as it will favour sampling strategies that match the integral very well. The power heuristic is written as

$$w_i(x) = \frac{[n_i p_i(x)]^{\beta}}{\sum_{k=1}^{m} [n_k p_k(x)]^{\beta}} \tag{3.10}$$

where the probability of the sampling strategy is raised to the power of β which is divided by all strategies that are also raised to the same power. Veach recommends $\beta = 2$, as this performs well in most scenarios that the balance heuristic does not.

3.5 Russian Roulette Termination

As the rendering equation is a recursive integral it is clear that a fixed stopping procedure will introduce bias into the estimate. This can be solved by using a technique called russian roulette termination. Using russian roulette also has the benefit of terminating paths that have a low probability of giving a significant contribution but will also increase variance. This technique involves picking an absorption probability and a uniform random variable between zero and one. If the random variable is below the absorption probability then the evaluation is terminated, contributing nothing to the estimate. Compensating for this loss of energy is done by scaling the contribution when the random variable is above the absorption probability by the multiplicative inverse of the absorption probability itself. The estimator can be written as

$$\langle I \rangle_{RR} = \begin{cases} \frac{1}{P} f(\frac{x}{P}) & \text{if } x \leq P \\ 0 & \text{if } x > P \end{cases} \tag{3.11}$$

where P is equal to one minus the absorption probability. The absorption probability can be any value but in Dutre *et al.* (2006) it is noted that using the hemispherical reflectance of the surface will produce better results than a constant variable.

Another improvement noted by Veach (1998) is defining the absorption probability as $\min(1, t_i/\delta)$ where t_i is the original probability and δ is a threshold that can be used to control variance. This prevents russian roulette from increasing variance until the original probability is below the threshold.

Chapter 4

Algorithms

In the previous chapter the integral equation to light transport was described as well as various improvements that all maintain an unbiased estimate when numerically integrating. Efficiently computing this integral can be very expensive so consideration of memory performance and sample contribution must be taken into account when considering an algorithmic solution.

4.1 Uni-directional Pathtracing

When Kajiya originally published his paper (Kajiya, 1986) he also proposed an algorithm to solve the rendering equation. This is called uni-directional pathtracing and uses a Monte Carlo estimator to solve the recursive integral.

This took work done previously in radiosity approximation and adapted it to ray tracing so that only visible points from the camera need computation. The concept of a ray was extended into a path that represented a set of sequential rays. This path would be constructed with each intersection creating the next ray within the set according to some importance function. This importance sampling technique is only effective if the importance function resembled the BRDF, increasing variance if not. Each segment is an estimate of the reflective integral while any radiance found being emitted from the surface contributes to the estimator while taking into account the path's weight and probability. Evaluating the path as a whole means finding the product of its initial state and each transition until the contributing state is reached.

This becomes computationally very expensive as each segment or ray requires scene traversal without any guarantee that it or any resulting ray will contribute to the final estimate. Multiple paths are then averaged using a Monte Carlo estimator to produce the final pixel value.

4.2 Sampling Strategies

Much variance in the original pathtracing algorithm is due to only sampling with an importance function based upon the BRDF of the surface. Another approach is to sample direct and indirect lighting as described in equation 3.5 and combine the two results.

4.2.1 Next Event Estimation

Sampling the both integrals separately is called next event estimation and involves creating a deterministic ray for each intersection towards a stochastic sample upon the light's surface to determine its visibility. Then evaluating the light's radiosity as well as its probability in respect to the projected solid angle. These satisfy both the geometric and visibility terms when integrating across all visible surfaces. The BRDF is then used to continue the path. This can be seen illustrated in figure 4.1 with each intersection tracing deterministic rays towards the light. Both of the sampling strategies are then combined to produce an unbiased result.

This approach however can also increase variance if the BRDF has a specular surface distribution and the projected solid angle of the light area onto the hemisphere has an even distribution. In this scenario the BRDF sampling strategy better matches the direct lighting integral than sampling according to the area formulation. This can be improved by allowing both the light and BRDF sampling strategies to contribute to both the direct and indirect integrals and weighting their contribution.

4.2.2 Combining Strategies

This weighting can be determined by using multiple importance sampling as described in equation 3.8 with either the balance or power heuristic. This will

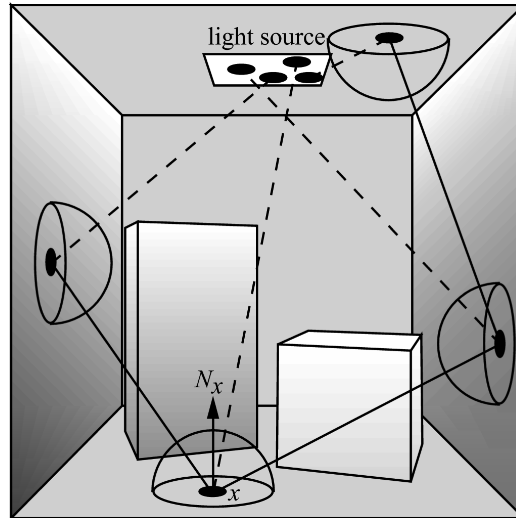


Figure 4.1: Pathtracing with next event estimation (Dutre *et al.*, 2006)

result in a weighting that takes advantage of both strategies and maintains an unbiased estimate. When a highly specular surface is being shaded with area lights that project evenly across the hemisphere, the heuristic will give more weight to contributions from the BRDF sampling strategy. When the surface has a very diffuse distribution and the area lights projected area across the hemisphere is small, then the heuristic will weight more towards the contribution from the light sampling strategy.

4.3 Sorted and Deferred Shading

Uni-directional pathtracing with next event estimation and multiple importance sampling is a robust solution to the sampling problem as well as being highly scalable. Latency that results from inefficient sampling has been recognised and substantially reduced but with larger data sets, other forms of latency have become very apparent. Production scenes are constantly becoming more complex and the incoherent nature of pathtracing and the resulting memory access have become a real problem when traversing and shading.

In the paper (Eisenacher *et al.*, 2013) an architecture for coherent uni-directional pathtracing is described to mitigate this problem with memory coherency. Traditionally a depth first approach is used when pathtracing where each path is constructed, evaluated and then discarded. This prevents any form of sorting for coherency between rays when considering traversal or shading, resulting in very incoherent memory access. What Eisenacher *et al.* (2013) proposes is a breadth first approach where rays from multiple paths are processed according to a two stage sorting procedure. Allowing enough rays to fit within memory requires them to be stored in directional batches that are streamed to and from disk. During the shading procedure each ray can create a subsequent ray that is feed back into another batch representing the last segment of a path. These two stages are referred to as ray sorting and deferred shading respectively.

The ray sorting stage first takes rays from either the camera or a surface being shaded and compresses them for streaming. This is very important as the primary latency cost for coherency is not sorting but rather writing and reading batches to disk. Rays are compressed and stored in local thread buffers before being added to one of six cardinal direction bins. Every bin will hold a single batch of rays in a memory mapped file with an incremented atomic pointer to allow for lock free write access. When a bin's capacity is reached it's scope is locked while the file is closed and its address gets added to a batch queue. When this is done, a new file is created and the scope is unlocked allowing for compressed rays to be written.

Before traversal an address is taken from the batch queue, rays are then loaded and decompressed into main memory. Concurrent loading and processing of ray batches are used to minimise thread downtime. Batches are first sorted using a recursive median partition according to position until a subset of no more than 4096 rays are found. Then they are further sorted according to direction until groups of 64 coherent rays are produced. This then allows for a memory coherent traversal of an acceleration structure using vectorized packets or bounding volume hierarchies. Taking advantage of the ray coherency can substantially reduce traversal times.

When traversal of the batch is complete and the intersection hit points are found, then the differed shading stage begins. Each intersection from the batch will have some differential information that is used for sorting.

Using this information, all intersections are sorted according object and then primitive identification.

This results in faces that are intersected being only accessed once during a single batch. When shading the surface at each intersection point, per-face textures are loaded with perfect coherence and whole textures are loaded with minimal overhead in regard to memory performance. Texture lookup can also take advantage of vectorization to maximise performance. Finally new rays that are created to continue the path, are compressed and added to the threads local buffer which is later passed to the directional bins.

Chapter 5

Implementation

This thesis is concerned with the development of a uni-directional pathtracer based upon the architecture described in Eisenacher *et al.* (2013). The language chosen for development is C++03 and other third party libraries are used for added functionality.

The intended outcome is to produce a robust architecture as a programming library that can handle large data sets while still maintaining efficiency and being extendible. The requirements and standards of production environments have been taken into account within the design and when adding dependencies. The scope does not cover the implementation of a graphical tool for end users nor the implantation of advanced modules or shaders. However the design accommodates for the extension of the pathtracer using a modular interface of which basic examples have been implemented throughout.

Despite previous work done on similar projects, almost none of the code written for such projects has been used here. The only notable exception to this is the ThinLensCamera class which has been extensively modified as well as the Common utility header that has also been extended. There is a generic Framebuffer class that is used in the example program to visualize the output. This is also a development upon something very similar originally developed for a previous project.

5.1 Libraries

Dependencies were chosen to conform to industry standards as well as reduce the amount of development for things that the project was not directly investigating. The most notable of these is a hit testing library named Embree that is maintained by Intel for CPU architecture. Embree is a highly optimised system for constructing acceleration structures and traversing rays to find closest intersections. This performance however is dependent upon ray coherence and as result naturally benefits from any improvements in this area. The project requires that the RTCORE_ENABLE_RAY_MASK option be enabled when compiling Embree as it takes advantage of the libraries masking functionality to avoid float point precision errors. Required dependencies are as follows:

- Boost 1.55
- TBB 4.2
- Embree 2.6.1
- OpenEXR 2.2
- OpenColorIO 1.0.9
- OpenImageIO 1.5.13
- Yaml-Cpp 0.5.1
- Eigen 3.2.4
- Python 2.7
- GLFW 3.0
- GLEW 1.11

All libraries that can be found within the VFX Reference Platform conform to the versions stated in the CY2015 standard. When building the project, the CMake system with a version of at least 2.8 is recommended and all non-standard modules are included with the source code. Another third party library named tinyobjloader is also included with the source for loading geometry data.

5.2 Requirements

Development goals have been oriented around production needs rather than implementing research for corner case scenarios. These needs in a broad sense can be categorised into three distinct areas. These being extendibility, iteration and system performance as described by Křivánek *et al.* (2013). Performance can be further decomposed into sampling strategies, efficient ray traversal, scalability and memory performance. Colour pipelines are another issue that should be addressed when working with and creating images.

When considering the complexity of industry pipelines it is important to allow for the extension of a system through generic interfaces. This is accomplished here with the use of a modular design inspired by the Physically Based Rendering Toolkit (Pharr and Humphreys, 2010). There are a total of seven different module types that allow the extension of lights, cameras, filters, objects, shaders, samplers and textures.

Iteration is also important to allow for fast creative feedback of artistic changes. This has been accommodated for using a two stage sampling routine, enabling a user to trade in overall efficiency for iteration speed.

With the last requirement being performance, all sampling techniques from the previous chapter have been used to allow each individual sample to have a greater contribution with minimal overhead. The system is highly parallel and concurrent while implementing the design described in Eisenacher *et al.* (2013) which has the added benefit of allowing for faster traversal when using the Embree library.

The colour pipeline assumes that all calculations are performed in float point linear space. This means that all textures are converted to linear space and mipmapped using the maketx command line program before rendering. The maketx program is distributed with OpenImageIO which needs to be compiled against OpenColorIO. The implementation will then either display the image using sRGB colour space or save to file in linear space after rendering is complete.

5.3 Design

When designing the implementation a balance between two paradigms were taken into account. The first being object orientation which encourages encapsulation and extendibility through polymorphism. This allows for abstraction at a higher level of the system without affecting performance critical processes. However the object oriented paradigm amortises data and as a result degrades performance with cache misses while also preventing instruction level parallelism. This has meant that a data oriented paradigm has been used on more performance critical areas of the project.

Data oriented programming structures computation around data with access being local in time and address space. This means representing data in continuous arrays that are processed using iteration, minimising the chance of a cache miss and reducing instruction calls to polymorphic classes. Another benefit to data orientation is that it makes instruction level parallelism through vectorization more applicable. Generally throughout the project there has been an effort to maintain locality in both access and location of information. Functions are designed to take arrays of continuous data over being called iteratively and being passed individual elements when there is a chance of multiple elements needing to be processed.

Scalability has been considered from the beginning as thread synchronisation can have a considerable impact on performance. Workloads for a parallel task are also rarely well balanced so the project uses a task based system that amortize these workloads and allows for better thread utilisation. The Threaded Building Blocks (TBB) library was chosen to handle tasks which also provided thread safe containers for data synchronisation. When local thread storage is required, TBB also provides templated classes to access local thread data, avoiding contention entirely. This is used when accessing random number generators or texture caching systems. Information that represents the scene description is considered immutable during processing to prevent data races.

Concurrency is a broader term than parallelism which is the execution of similar tasks simultaneously. The difference is that a concurrent system refers to one where tasks can work independently from each other, with the possibility of working simultaneously. These tasks however are not necessarily similar

or improving performance but are a way in which a process is performed. An example of where this concept is implemented can be given in how batches are loaded and processed. When processing a single batch many tasks are used in parallel while another task that is potentially working simultaneously, loads the next batch to be processed. This results in improved performance as it prevents threads from hanging while waiting for the operating system to read batches from disk.

Allowing for progressive rendering meant that combining samples to represent the colour of a pixel is done in two separate stages. The first stage involves creating multiple samples on a sub-pixel level and then filtering them to produce a result. This result is then combined with previous results and averaged for every iteration over the image. Larger sample counts and lower iterations will allow for better coherency between the rays as well as more advanced sampling and filtering routines. Lower sample counts and more iterations allow for a faster progressive feedback while losing some performance. This design also allows for the system to be re-entrant as the processing can be stopped between iterations and started again at a later time.

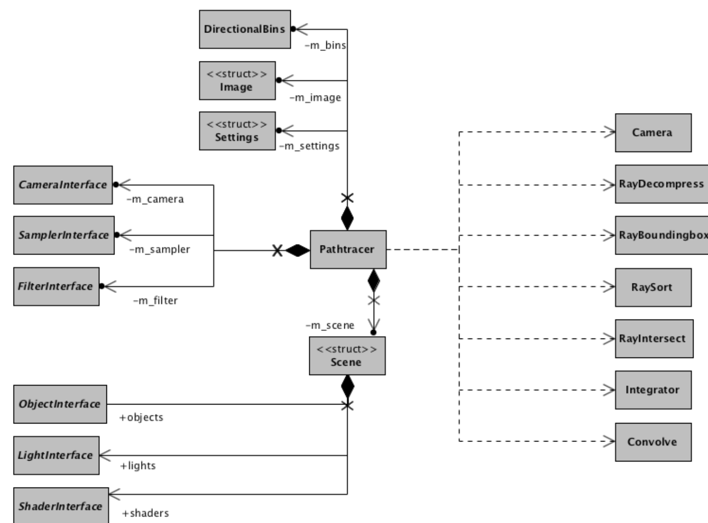


Figure 5.1: Simplified diagram of the main system

5.4 Details

Development of the implementation was structured around the Pathtracer class which is also used as the main interface when integrating the library into other software. Everything is written in accordance to the NCCA coding standard while also using namespaces to prevent pollution of global context. Shared headers, utilities and linear algebra types are all found in the Common header file which is included throughout the code base.

5.4.1 Pathtracer Class

The Pathtracer class is the only one needed to be included in external applications to render an image. Construction involves passing a path to a scene description file as a std::string. The scene is then passed using the Yaml-Cpp library with each constructible object handling its own construction. Required objects also have defaults that are used when that particular object is not found within the scene file. The internal image can be accessed and reset using the image() and clear() functions respectively. Terminating the pathtracers execution can be done using terminate() while querying its current state is done using active().

```
Pathtracer
-m_bins : scoped_ptr<DirectionalBins>
-m_settings : scoped_ptr<Settings>
-m_image : scoped_ptr<Image>
-m_scene : scoped_ptr<Scene>
-m_camera : scoped_ptr<CameraInterface>
-m_filter : scoped_ptr<FilterInterface>
-m_sampler : scoped_ptr<SamplerInterface>
-m_thread_texture_system : LocalTextureSystem
-m_thread_random_generator : LocalRandomGenerator
-m_batch_queue : concurrent_queue<BatchItem>
-m_terminate : atomic<bool>
+Pathtracer(_filename : string &)
+~Pathtracer()
+image(_pixels : float **, _with : int *, _height : int *) : void
+clear() : void
+active() : bool
+terminate() : void
+process() : int
-construct(_filename : string &) : void
-cameraSampling() : void
-batchLoading(batch_info : BatchItem *) : bool
-fileLoading(batch_info : BatchItem &, batch_compressed : RayCompressed *) : void
-rayDecompressing(batch_info : BatchItem &, batch_compressed : RayCompressed *, batch_uncompressed : RayUncompressed *) : void
-raySorting(batch_info : BatchItem &, batch_uncompressed : RayUncompressed *) : void
-sceneTraversal(batch_info : BatchItem &, batch_uncompressed : RayUncompressed *) : void
-hitPointSorting(batch_info : BatchItem &, batch_uncompressed : RayUncompressed *) : void
-surfaceShading(batch_info : BatchItem &, batch_uncompressed : RayUncompressed *) : void
-imageConvolution() : void
```

Figure 5.2: Overview of the Pathtracer class

Creating an image is done using process() which will render the scene and store the result internally. This method will first create the primary rays

from the camera, writing them as batches to disk and storing the reference as a BatchItem in a concurrent queue. When the queue is finished being populated, batches are taken from the queue and processed in a while loop. While batches are being processed the next batch is being read in using a Boost thread in a concurrent manner. When the queue is found to be empty, the DirectionalBins object is flushed to populate the queue with another BatchItem. DirectionalBins will decide upon which internal bin to flush to the queue based upon the bins size. Deciding upon the correct bin can have a substantial impact upon performance but taking the largest has proven to give best results.

Within the while loop the pathtracer will decompress the batch and call four methods that are described in Eisenacher *et al.* (2013). These methods are raySorting(), sceneTraversal(), hitPointSorting() and surfaceShading() with each being amortised into tasks and processed in parallel. The first method takes the decompressed ray batch and sorts the results with a recursive median partition. Scene traversal uses the Embree library to populate the decompressed ray batch with intersection data. The third method uses a parallel_sort algorithm provided from the TBB library to sort rays according to both the geometry and then the primitive identification numbers found during traversal. Surface shading then uses a custom Range type that allows TBB to divide the batch into tasks according to intersected objects and then integrates the light distribution at each intersection point.

As each batch is taken from the queue and processed, new batches are created from the surfaceShading() method and added to the queue. When all batches have been processed and each path has been terminated the final image is then filtered. This is done using the imageConvolution() method which uses a filter to convolve the samples and stores the resulting image internally.

Compressed rays are streamed to and from disk while being managed by the DirectionalBins class. This class handles the synchronisation of adding ray buffers to the six cardinal direction bins as well as writing these bins to disk when their capacity is used. Flushing the largest bin to disk when new batches are required is also handled internally within the class.

5.4.2 Parallel Processes

This is a broad outline of the Pathtracer classes process() method although it can be described in more detail when looking at the individual classes that are integrated into TBB to allow each part of the method to be parallelised.

The Camera class is used with a parallel_for loop and a blocked range to divide the image pixels in tasks that are processed in parallel. Then for all samples within a pixel, the sampler interface creates sample positions. These samples are then used by the camera interface to generate compressed rays which are stored within the tasks buffer. When the task has finished creating all rays for that block it gives the buffer to the DirectionalBins object to stream the data to disk.

When ray batches are loaded they need to be decompressed before being processed. This is done using the RayDecompress class and a parallel_for loop with a standard range. The range is divided automatically using a auto_partitioner which attempts to reduce splitting while also allowing for task stealing between threads to balance workload.

Before ray batches can be sorted for traversal their bounding box needs to be computed. This is accomplished using the RayBoundingbox class and a parallel_reduce algorithm with a standard range. This will divide the batch into tasks where each sub-range computes a bounding box according the the ray positions. When the task is has completed, the bounding box is combined with that of other tasks in a hierarchical structure. This process allows for best use or parallel computation.

Sorting the ray batches for traversal is done using the RaySort class which takes the bounding box created by the RayBoundingbox class and divides it using a median partition along the longest edge. This class acts as a recursively parallel functor with the use of the parallel_invoke algorithm which allows for parallel branching. Every invocation of the functor sorts the sub-range according to ray position with a parallel_sort algorithm. Task scheduling between the parallel_invoke and parallel_sort tasks is then handled with TBB for optimal results. Rays are sorted according to position until the sub-range falls below 4096, as sorting is then done according to ray direction until coherent groups of no more than 64 rays are found.

Scene traversal is then done using the RayIntersect class that uses a parallel_for algorithm and a standard range. The Embree library then takes the sorted and uncompressed ray batch and creates intersection data. The configuration of Embree has been set to allow for optimal traversal with coherent rays to take advantage of the sorting procedure.

Uni-directional pathtracing is performed within the Integrator class during surface shading. This class uses the parallel_for algorithm and a blocked range with a partitioner template called RangeGeom. The RangeGeom template will split the computational workload according to some comparable value of the data being processed. The result will be tasks working on data with a continuous value although ranges that are too large will still be divided to distribute workload. This requires that the data be sorted according to this value before being partitioned. Sorting of the intersected ray batches is done using the parallel_sort algorithm.

The Integrator class first loads in or creates thread local storage. Then it finds the object intersected which is guaranteed to be the same for all intersections within this task due to the RangeGeom partitioner. Then it proceeds through five parts before completing, taking advantage of the shared intersection information between the rays. First if an object was not hit then the task completes, terminating all rays. Then if the object hit was a light, then the radiance transmitted through the path represented by each ray is weighted and balanced with multiple importance sampling. The resulting value is then added to the sample and all paths are terminated, completing the task.

When the procedure has passed both first and second parts, it is known that an intersection with an object has occurred. The third part is cloning the surface shader and calculating the colour coefficients of all intersections with perfect coherence. These values are then stored internally within the shader and used in the last two parts of the procedure. Then we integrate across the combined surface area of all lights with an even pick probability. The chosen light for each ray intersection is sampled and tested for visibility using a ray traced approach. Geometry masks are used so that visibility testing does not have to consider float point precision errors that result in self intersections. Radiance from the light is then weighted according to the surface shader

and the throughput of the represented path. Finally the result is balanced for multiple importance sampling and added to the sample. The last part continues the random walk of the path by sampling the surface shader and adding compressed rays to the tasks buffer. Before the task completes, the buffer is given to the DirectionaBins class which stores the new rays internally.

When the batch queue is completely empty and all samples have been evaluated, the Convolve class is used to produce the final colour. This class uses the parallel_for algorithm and a blocked range in the same manner as the Camera class. It then passes the blocked range to the filter interface which stores the resulting colour information within the Pathtracer's image structure.

5.4.3 Modular Interface

There are seven modular components to the system and basic examples of each module type have been developed. These modules allow the system to be extendible but for performance purposes take ranged data where appropriate. This adds flexibility and also means that other compute devices can be used to accelerate modules internally if required.

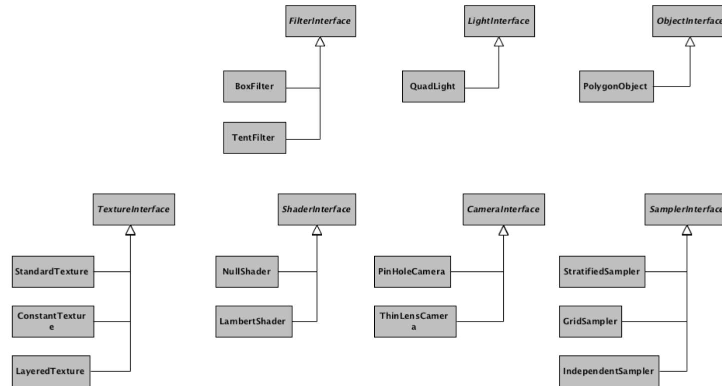


Figure 5.3: Simplified diagram of the module interfaces

The interface for a light module requires that two methods are implemented for multiple importance sampling. When an intersection requires radiance from a light, the illuminate() method is used and passed the in-

tersection position. This method then returns the direction to a sample on the lights surface area, the distance to that sample, radiance leaving the sample and finally the probability density in respect to the hemisphere. The second method is called radiance() and is called when a light is intersected. It takes the direction of the intersecting ray and returns the light radiance, the probability in respect to surface area and the cosine angle between the reversed ray direction with the light normal. The QuadLight class is an implementation of the light interface and represents a planar light which illuminates in single direction.

Primary rays are generated using a camera module that inherits from the camera interface. This interface requires a single method to be implemented called sample() which takes a range of sample positions and produces compressed rays. There are two example implementations of this module type being the PinHoleCamera and ThinLensCamera classes. The PinHoleCamera class creates a perspective project upon the image plane through a single nodal point. This class has a variable focal length that is measured in millimetres and assumes a film plane size of 35mm horizontal. When depth of field is required, the ThinLensCamera can be used which extends the PinHoleCamera class with variables for focal depth and aperture size.

When convolving the image a filter module is passed samples and produces colour values for each pixel. The filter interface requires a convolve() method to be implemented which takes the samples that are to be filtered as well as an image to store the results. Along with this, the images width, height, sample count as well as the blocked range to compute must also be passed. There are two example implementations that are the BoxFilter and TentFilter classes. The BoxFilter class will evenly average all samples within the pixel while the TentFilter class should produce better result and considers samples in neighbouring pixels.

Geometric objects are represented with object modules and must inherit from the object interface. This interface requires that texture() and shader() methods are implemented which retrieve the texture coordinate and shader index respectively. The texture() method requires a primitive identifier and barycentric coordinate to find the corresponding texture coordinate for an intersection. The PolygonObject class is an implementation of the object interface and represents triangulated meshes built from vertices, texture coor-

dinates and face indices.

Shading of surfaces is described through shader modules that inherit from the shader interface. The shader interface requires that five methods be implemented. These are clone(), initialize(), continuation(), evaluate() and sample() with the last two being used for multiple importance sampling. The clone() and initialize() methods are used to duplicate the shader and lookup texture values for each range of hit points being processed within the Integrator class. This is so that thread contention is avoided and to allow textures to be accessed in perfect coherence with vectorised instructions. The initialize() method takes the amount of texture coordinates to be evaluated, the texture coordinates themselves and a texture cache. The continuation() method allows the shader to give an approximation of the surfaces reflectance. This is used when evaluating for russian roulette termination within the Integrator class.

When lights have been deterministically sampled, the radiance distribution on the surface is found by calling the evaluate() method. This takes the colour index of the intersection created during the initialize() method, input light direction, output light direction and surface normal. This returns a weighting distribution, the cosine angle of the sample direction with the surface normal and finally the probability density of the sample direction in respect to the hemisphere. The sample() method is used to create a direction according to some probability that resembles the surface distribution. This takes the colour index of the intersection created during the initialize() method, output light direction and surface normal. Unlike the evaluate() method it does not take an input light direction as this is what it returns. The method will also return a surface weighting distribution, the cosine angle of the sample direction with the surface normal and finally the probability density of the sample direction in respect to the hemisphere. The weighting distribution returned by both evaluate() and sample() must conform to the requirements of a BRDF in their physical correctness so that bias is not introduced.

There are two shader implementations being the NullShader and LambertShader classes. The NullShader class does not contribute any radiance to a scene and is used as an index when mapping objects to lights. Perfectly diffuse surfaces are created using the LambertShader class which gives a perfectly lambertian distribution. When sampling this shader it will give a

direction and a probability density proportionate to the cosine distribution around the surface normal as described in Dutre *et al.* (2006).

Sampling the image is done using a sample module that inherits from the sampling interface. This only requires a method called sample() be implemented that takes the base sample count and will return sample positions across the pixel area. There are three implantations of this module being the IndependentSampler, GridSampler and StratifiedSampler classes. When stochastic sampling is required with each sample having no relation to other samples, the IndependentSampler class should be used. Both the GridSampler and StratifiedSampler classes will structure samples evenly across the area of a pixel with the later stratifying their positions.

Textures are handled with texture modules that inherit from the texture interface. They have the same requirement as shader modules for implementing both the clone() and initialise() methods. They also require a colour() method that is passed a single colour index and returns the texture value created during initialise(). There are three texture modules implemented being the ConstantTexture, StandardTexture and LayeredTexture classes. When dealing with constant colour, the ConstantTexture class is used which does not store any values during initialisation and returns just a single colour member. The StandardTexture class is used when dealing with variant texture values from a file. When looking up textures from file, the OpenImageIO library is used to handle caching and access. As textures are accessed with perfect coherency, vectorised instructions are used by OpenImageIO to increase performance.

5.4.4 Scene Description

Scene construction is accomplished with a description file that is passed to the Pathtracer class on construction. This file description format was designed to be clear and easy to read while also being reasonably flexible. Paths within the file to external object and texture files are relatively referenced. It was decided that the language for this description and for possible re-serialisation would be YAML due the clarity of the language and its actively supported libraries. This can be seen in an example file included within Appendix B.

5.5 Example Program

A command line program was developed as an example of integrating the library into an external program and was also used for testing purposes. This program takes a relative path to a scene description file and by default will display a progressively rendered image using a framebuffer. The resulting image can also be saved to disk using the '–output' argument followed by a file name. Help information can be displayed with the '–help' argument.

The Framebuffer class is compiled and loaded as a separate library and was originally developed for a previous project but was designed to be generic. When the framebuffer is initialised it is passed a pointer to the Pathtracer class and returns a pointer to the context. Using the context pointer, the default key callbacks are then overridden to allow users to terminate the pathtracer's execution. This allows the Pathtracer class to close correctly and clean-up any batches left on disk. The pathtracer's process() method is then called in a loop on a separate thread while events are polled on the main thread. Colour is displayed using an sRGB gamut when using the Framebuffer but is left as linear when a file is saved.

When compiling the project it is advised to enable compiler optimisations as code has been written accordingly while some libraries used will also have a significant performance increase.

Chapter 6

Results

This chapter outlines the results found from using different sampling strategies as well as changing parameters to allow for memory coherence. All images are produced using the implementation described above and the command line program. Measurements taken at Cornell University gave a baseline to validate the physical correctness during development. The dragon model in the test renders was created using data from Stanford University's scanning repository. The images shown in figure 6.1 give an example of the working implementation.

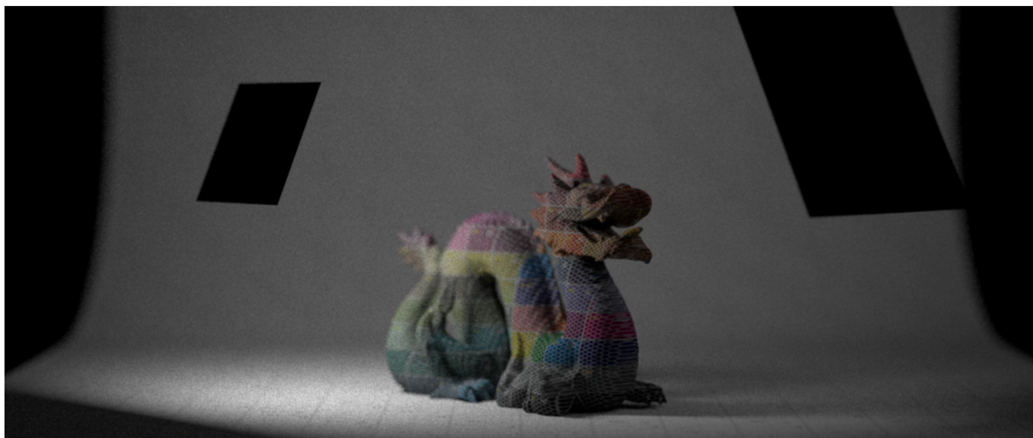
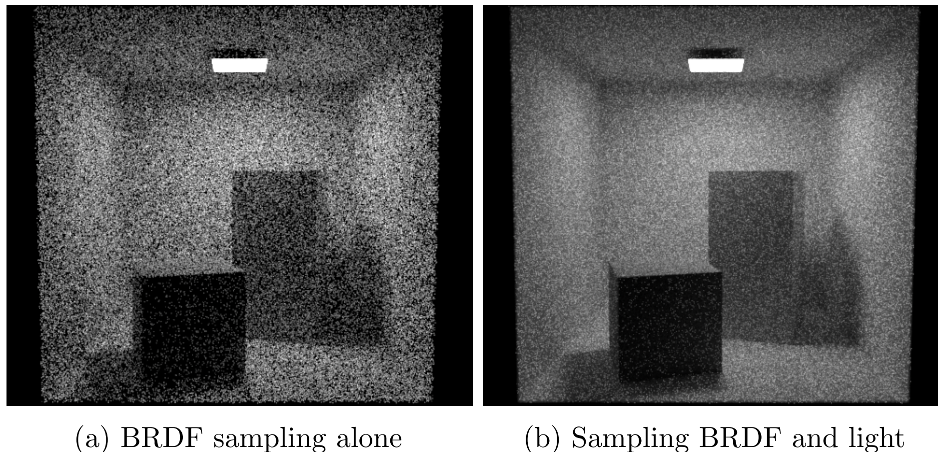


Figure 6.1: Physically correct image created using the implementation

6.1 Variance Reduction

Before optimising the system for coherency, the sampling strategies that allow for reasonable variance reduction needed to be implemented. Sampling the direct and indirect integrals separately and evenly weighting the results are shown to give significant variance reduction. This can be seen in figure 6.2 with the first image being a naive approach, sampling just using importance sampling of the surface. The second samples the light deterministically as well as the distribution on the surface and weights the results evenly.



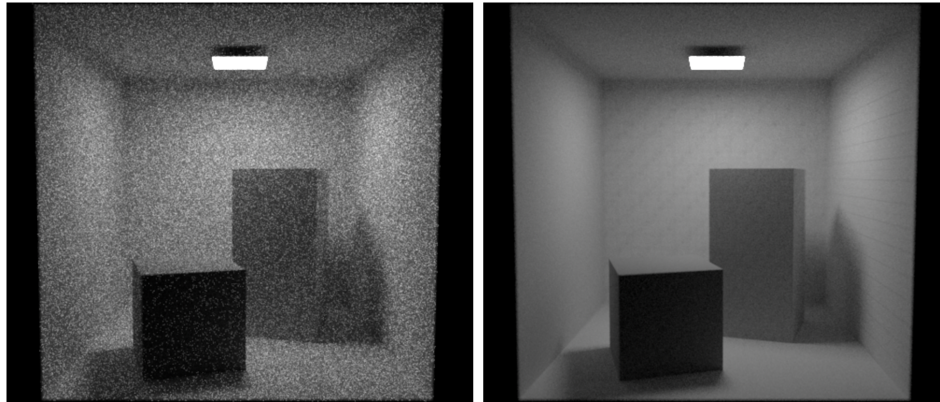
(a) BRDF sampling alone (b) Sampling BRDF and light

Figure 6.2: Equally weighted integrals

This is further improved with the use of multiple importance sampling. Weighting the contribution from both strategies according their respective probabilities can be seen to improve variance reduction further. This is shown in figure 6.3 with the first image being an equal distribution and the second using multiple importance sampling.

6.2 Coherent Rendering

The primary focus of this thesis was to see if sorting for coherence of a physically correct and unbiased pathtracing algorithm can increase performance. Testing this meant processing a scene with a considerable memory footprint.



(a) Equal sampling (b) Sampling with MIS

Figure 6.3: Multiple importance sampling

This was done procedurally with python by generating a data set containing 700MB of geometry data and 5.15GB of texture data. The scene contains 64 non-instanced models contained within a box, each with a unique shader. Every shader has four colour textures that are layered using three texture masks. A single light is used for illumination and the camera models a thin lens. This can be seen in figure 6.4 from a standard and zoomed view.

(a) Standard view (b) Zoomed view

Figure 6.4: Performance test scene

Tests were taken to evaluate performance, each with a varying batch size and ray depth. Batches are measured as an exponent of two and when set to smaller values will limit the coherency that can be found within each batch. However when using higher ray depths the coherency naturally is increased. The results are very encouraging and show that sorting for coherence can increase performance by an order of magnitude as seen in table 6.1.

Depth	Batch Size	Real	User	System
0	8	100.01	133.88	36.91
0	11	64.19	77.31	19.38
0	14	49.01	36.16	10.37
0	17	20.39	17.79	4.46
1	8	916.78	1968.96	505.95
1	11	211.46	538.35	138.72
1	14	46.73	122.50	33.08
1	17	26.82	67.16	18.07
2	8	1696.44	4401.65	1199.63
2	11	359.41	1021.20	275.96
2	14	70.06	203.51	55.36
2	17	42.84	119.60	33.20

Table 6.1: Performance test results

These results show the real, user and system time for each test. Physical time spent is represented by real time and is measured in seconds. This accounts for the parallelism on the four threads that were available on the test system. The user and system times represent CPU time within the process while outside of the kernel as well as time within the process and inside the kernel.

The performance seen in figure 6.5 shows that sorting rays for traversal and shading when accessing large data sets is substantially faster. Rendering with a smaller maximum depth as seen in figure 6.6 also gave a performance increase despite being more naturally coherent, even when indirect lighting is not evaluated.

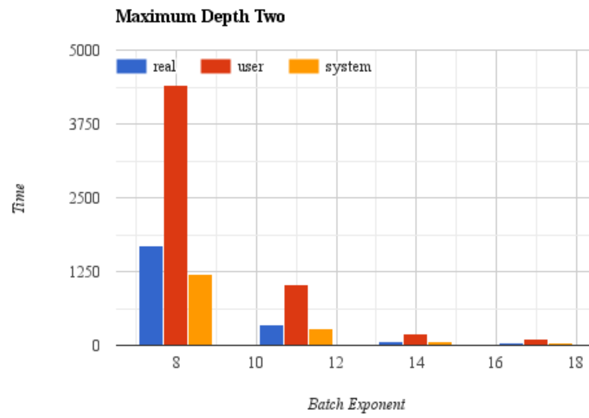
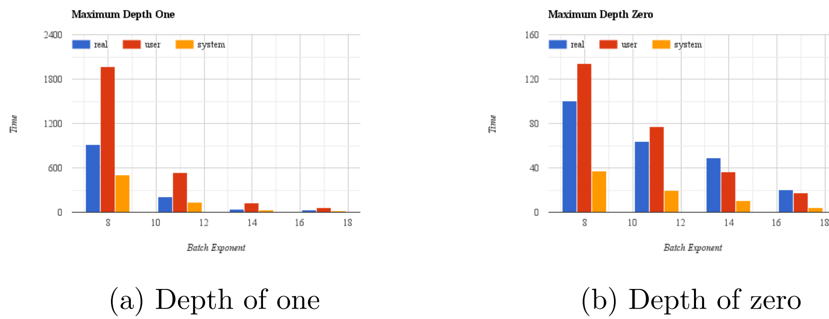


Figure 6.5: Batch size against run time with a depth of two



(a) Depth of one

(b) Depth of zero

Figure 6.6: Batch size against run time with varying depth

Chapter 7

Conclusion

Physically based rendering has become very popular within production but it is clear that rendering performance is not a solved problem and will continue being developed with ever increasing scene complexity. This project effectively showed that processing scene information without any regard to memory can have substantial performance costs. The implementation was completed successfully with respect to the original scope of the project while giving an opportunity to learn good development practice. It should be noted that there were some difficulties when creating the texture files, as a result of this they were not mipmapped. Testing and analysis could definitely be continued on higher performance hardware and with real world data. Further performance could be increased with the use of larger batches as the limited memory on the system used prevented sizes larger than that tested. Using solid state drives would have also benefited performance.

7.1 Further Development

There are multiple parts of this system that could be developed further to either give greater functionality or improve performance. More advanced implementations of the modules and particularly the development of advanced shaders would add much functionality to the system.

There is currently a known issue in that if a local thread buffer is set larger than the batch size, there will be a segmentation fault. This will happen when the sample count in relation to the bucket size is greater than the bin

exponent, although this could be handled with standard exception handling.

Geometry passing could be improved, replacing the tinyobjloader library with something more tailored. This would allow object passing to accept a greater range of formats as well as increasing performance with parallelism. Writing of batches to and from disk could be improved with a more advanced synchronisation between the threads. Another way to increase performance with regard to streaming batches would be to increase the compression of the rays themselves. Sorting intersections might be faster if a radix algorithm was integrated although this would use significantly more memory and the performance benefit would need to be tested. Packeting the rays before handing them over to the Embree library would allow for faster ray traversal as this would take better advantage of the rays coherency.

Finally, there has been much research done into the development of ray tracers in regard to graphic processing units. However this has not been adopted widely in production due to the limitations of massively parallel computation. Production rendering might seem well suited to this type of architecture when in fact there are issues that need to be addressed. Ray traversal is a massively parallel concept as long as all rays are traversing and intersecting homogeneousness data. This however is rarely the case with different types of geometry and shading models being used that cause divergence in the computation. Using the sorting procedure described by Eisenacher *et al.* (2013) and implemented here, a design could be developed to allow graphic processors to evaluate homogeneousness data.

Bibliography

Cook R. L., Porter T. and Carpenter L., 1984. Distributed ray tracing. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, New York, NY, USA. ACM, 137–145.

Dutre P., Bala K., Bekaert P. and Shirley P., 2006. *Advanced Global Illumination.* AK Peters Ltd.

Eisenacher C., Nichols G., Selle A. and Burley B., 2013. Sorted deferred shading for production path tracing. In *Proceedings of the Eurographics Symposium on Rendering*, EGSR '13, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association, 125–132.

Georgiev I., Křivánek J., Davidovič T. and Slusallek P., November 2012. Light transport simulation with vertex connection and merging. *ACM Trans. Graph.*, **31**(6), 192:1–192:10.

Hanrahan P., 1986. Using caching and breadth-first search to speed up ray-tracing. In *Proceedings of Graphics Interface and Vision Interface '86*, GI '86, Toronto, Ontario, Canada. Canadian Man-Computer Communications Socitey, 56–61.

Jensen H. W., 1996. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, London, UK, UK. Springer-Verlag, 21–30.

Kajiya J. T., 1986. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, New York, NY, USA. ACM, 143–150.

Křivánek J., Georgiev I., Kaplanyan A. S. and Cañada J., 2013. Recent advances in light transport simulation: Theory & practice. In *ACM SIGGRAPH 2013 Courses*, SIGGRAPH '13, New York, NY, USA. ACM, 4:1–4:5.

Pharr M. and Humphreys G., 2010. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition.

Pharr M., Kolb C., Gershbein R. and Hanrahan P., 1997. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co., 101–108.

Veach E. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford, CA, USA, 1998.

Wald I., Benthin C. and Boulos S., 2008. Getting rid of packets-efficient simd single-ray traversal using multi-branching bvhs. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE, 49–57.

Wald I., Slusallek P., Benthin C. and Wagner M., 2001. Interactive distributed ray tracing of highly complex models. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, London, UK, UK. Springer-Verlag, 277–288.

Wald I., Woop S., Benthin C., Johnson G. S. and Ernst M., July 2014. Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph.*, **33** (4), 143:1–143:8.

Ward G. J., Rubinstein F. M. and Clear R. D., 1988. A ray tracing solution for diffuse interreflection. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, New York, NY, USA. ACM, 85–92.

Whitted T., jun 1980. An improved illumination model for shaded display. *Commun. ACM*, **23**(6), 343–349.

Appendix A

Detailed Diagrams

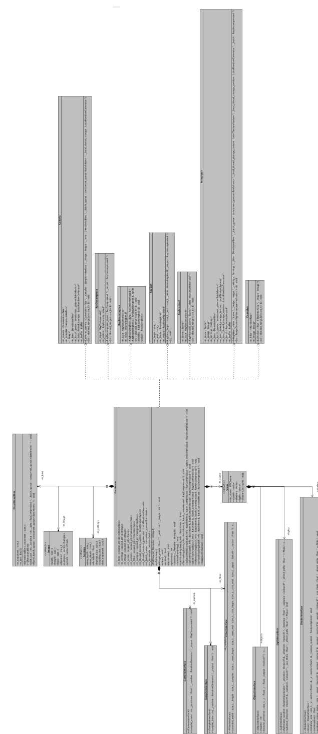


Figure A.1: Detailed diagram of the main system

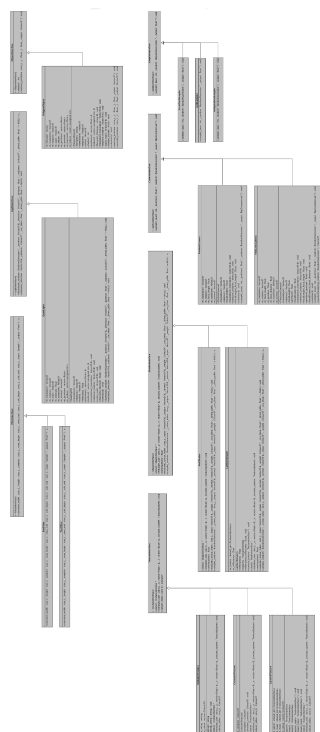


Figure A.2: Detailed diagram of the modules

Appendix B

Example Scene File

```
image:
  width: 1024
  height: 429
  sample base: 1

settings:
  min depth: 2
  max depth: 5
  threshold: 0.1
  bucket size: 32
  shading size: 4096
  bin exponent: 15

camera:
  type: PinHole
  translation: [ 0, 120, 1000 ]
  rotation: [ 0, 90, -3 ]
  focal length: 150

filter:
  type: Tent

sampler:
  type: Stratified
```

```
object:
  type: Polygon
  filename: objects/model.obj
  translation: [ 15, 0, 25 ]
  rotation: [ 0, 45, 0 ]
  scale: [ 10, 10, 10 ]
  shader: 0

shader:
  type: Lambert
  reflectance: 0.3
  texture:
    type: Layered
    upper:
      type: Standard
      string: textures/texture_one.tx
    lower:
      type: Standard
      string: textures/texture_two.tx
    mask:
      type: Constant
      colour: [0.5, 0.5, 0.5]

light:
  type: Quad
  translation: [ 100, 100, 35 ]
  rotation: [ 0, 135, -25 ]
  scale: [ 50, 50 ]
  intensity: 25

light:
  type: Quad
  translation: [ -120, 120, 50 ]
  rotation: [ 0, 45, -25 ]
  scale: [ 100, 100 ]
  intensity: 1
```