Tracing rays: the past, present and future of ray tracing performance

JCW Kroeze

A mini-dissertation submitted in partial fulfillment of the MSc-degree in the School of Information Technology at the Vaal Triangle Campus of the North-West University

Study leader: **Prof DB Jordaan**

VANDERBIJLPARK

November 2010

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The metrics used to compare the performance of various ray tracers in the literature are flawed because they are non-standard and depend on the hardware configuration of the specific system used to gather data. A different way of measuring the relative performance of ray tracing algorithms is proposed and tested across several hardware platforms using correlation co-efficients.

OPSOMMING

Die maatstawwe in gebruik deur die “ray tracing” navorsings gemeenskap is gebrekkig. Daar is geen standaard maatstaf nie, en dié in gebruik is afhanklik van die hardeware in gebruik deur die toets-stelsel. ‘n Nuwe metode word voorgestel wat onafhanklik van hardeware spoed is. Hierdie metode word getoets deur statistiese metodes op verskeie rekenaars.

CHAPTER 1: INTRODUCTION

1.1 Background

Ray tracing is a global illumination algorithm[1] that renders scenes by modelling the behaviour of light based on the study of classical ray optics (Whitted, 1980). Consequently, it creates highly realistic images at the cost of great computational complexity (Georgiev & Slusallek, 2008; Wald *et al.,* 2007; Whitted, 1980).

The ray tracing algorithm was first implemented by Turner Whitted in 1980 (Whitted, 1980). In the original paper, Whitted states that the algorithm is unfortunately very slow but expresses a hope that it can be sped up (Whitted, 1980).

Since 1980, a lot of research has been conducted in order to find a way to speed up the naïve ray tracing algorithm. This trend continues even today as researchers seek to design faster and faster ray tracers. Recent attempts include Bikker's (2007) description of a real-time game engine based on ray tracing, the SIMD optimizations described by Overbeck *et al*. (2008) and the hybrid renderer described by Christensen *et al.* (2006).

There is no literature, however, that focuses on measuring the performance of various optimization techniques. Researchers usually state the performance improvements they achieved over more naïve algorithms in terms of how much faster the algorithm is. However, this can lead to confusion. Because authors often use different hardware configurations and render different scenes some of these figures are not comparable. Oftentimes, authors do not include information on the memory usage of their algorithms. Also, papers often focus on the implementation of

1 The term “global illumination algorithm” refers here to that set of algorithms that take inter-object interactions into account when rendering a scene. Therefore a global illumination algorithm will accurately render effects such as the reflections of objects in other objects.

certain techniques and not on their performance. Often the manner in which the data was gathered is treated only in a cursory fashion. More in-depth, rigourous and extensive figures will indicate which areas of a ray tracer should be improved in projects that are behind schedule and may indicate where more research should be done.

It is this gap in the ray tracing literature that the author wishes to address with his research.

1.2 Problem Statement

There is a lack of rigourous, complete and comparable experiments in the literature that compare existing ray tracing optimization techniques and therefore a lack of reliable figures on their relative performance.

1.3 Main Research Question

How can various improvements to the naïve ray tracing algorithm be compared in a fair manner?

1.3.1 Secondary Research Questions

1. Which variables can affect the performance of a ray tracing engine?
2. Which variables or circumstances make the greatest difference to a ray tracing engine's performance?
3. How can most of these variables be eliminated to provide reliable performance figures?
4. Is it possible to rigorously compare ray tracing engines with no statistical or environmental bias?

1.4 Hypothesis

The figures currently available for the performance difference between various ray tracing optimizations are incomplete, inaccurate, or suffer from statistical or environmental bias and do not indicate the true relative performance of different ray tracing algorithms.

1.5 Method of Investigation

1.5.1 Literature Review

During the course of the proposed study available literature on ray tracing techniques and algorithms will be studied in order to gain an overview of the field and to see how problematic the data may be. Evidence from this review will be used to support the case for a new technique of comparison in the ray tracing research community. The literature review will also serve as a guide during implementation of the empirical component – a thorough survey will indicate what is common or typical in ray tracing research.

1.5.2 Case Study

The proposed study will consist mostly of an empirical experiment. This experiment will start with creating an adaptable core ray tracing engine to serve as the basis of the rest of experiment. This ray tracer will then be used to investigate a fair way of comparing competing ray tracing engines.

1.6 Contribution to The Field of IT

This study will contribute to the field of IT by providing a reliable way of comparing the performance of ray tracing optimizations. This will indicate where research into the problem of ray tracing performance will be most productive. The study will also contribute to an understanding of experimental rigour in IT, by exploring the degree of rigour available in the field and its possible advantages.

1.7 Layout of The Study

Chapter 2 will discuss the literature reviewed for this study. It will focus on the evolution of the ray tracing algorithm, recent developments and how studies gather and compare data.

Chapter 3 will describe the data gathering method used for this study. It will discuss the way the experiments were set up and what mathematical methods were used to process the raw data.

Chapter 4 will describe the data analysis method used for the study. It will briefly discuss the raw data gathered and identify any apparent patterns. Mathematical analyses of the data will be carried out during this chapter.

Chapter 5 will draw a conclusion from the data analysis and will recommend future research opportunities, identify any limitations to the study and provide any final information.

CHAPTER 2: LITERATURE REVIEW

2.1 Introduction

The ray tracing algorithm was invented in 1980 by Turner Whitted (Whitted, 1980) as an improved shading model for computer generated graphics. It was largely based on previous work done by Arthur Appel, who described a method to render machine solids by casting rays of light into a scene (Appel, 1968). This idea is the inspiration behind ray tracing – which Whitted eventually extended by adding reflection, refraction and other refinements to create a compelling alternative to rasterization that stubbornly refuses to be quick enough.

At the time, ray tracing produced more realistic images than other techniques could, because it used information about an object's setting while rendering it (Whitted, 1980). Such rendering techniques are called “global illumination algorithms”, because they make use of global data. Less realistic techniques available in 1980 – and the algorithm used by graphics cards today – use only aggregate data local to the object being rendered (Whitted, 1980). These techniques are called “local illumination algorithms”.

This realism came at a price, however, since the naïve ray tracing algorithm described by Whitted (1980) was very slow. The trade-off is simple: the images produced by a ray tracer are more realistic because it uses more information, but they are also slow to compute since this extra information must be processed.

In his original paper, Whitted expressed a hope that future versions of the algorithm will be faster and suggested that using picture coherence will be the way to gain performance benefits (Whitted, 1980). Improving his algorithm has occupied the computer graphics community for decades, and has recently begun to yield exciting results.

In the coming sections of this literature review the algorithms, data structures and techniques that have provided these results will be discussed together with the historical advances that made them possible.

This chapter draws heavily on two papers read at the IBIMA 15 conference in partial fulfilment of degree requirements (Kroeze *et al*., 2010a; Kroeze *et al*., 2010b).

2.2 The Naïve Ray Tracing Algorithm

In order to understand the advances that have been made in the field of ray tracing, it is important to understand its history. The various improvements to the algorithm will be meaningless without a discussion of its basic form.

Simply put, Whitted's original algorithm gathers the global illumination information needed to accurately shade a pixel by creating a tree of rays and then traversing this tree (Whitted, 1980). During the traversal, the illumination information at a specific point is transported to any points that are affected by reflected or transmitted light from the original point (Whitted, 1980).

Whitted (1980) proceeds to derive a system of equations which will allow this tree to be constructed. By projecting a non-normalized ray vector from the viewing direction onto the surface normal and using the fact that the angle of incidence equals the angle of reflection Whitted (1980) derived the following equations to describe the direction of reflection:

- $\vec{V'} = \frac{\vec{V}}{\left|\vec{V} \cdot \vec{N}\right|}$ where $\vec{V}$ is the viewing vector and $\vec{N}$ is the surface normal,
- $\vec{R} = \vec{V'} + 2\vec{N}$ where $\vec{R}$ is the direction of reflection.

These equations are derived more formally in Appendix A.

It is initially unclear why Whitted (1980) thought that $\vec{V} \cdot \vec{N} < 0$. Now, $\vec{V} \cdot \vec{N} = \left\|\vec{V}\right\|\left\|\vec{N}\right\|\cos\theta$ where θ is the angle between $\vec{V}$ and $\vec{N}$. But, $\left\|\vec{V}\right\| > 0, \left\|\vec{N}\right\| > 0$ since these are the lengths of the respective vectors and therefore > 0. So the sign of the equation, which is the portion under scrutiny, is completely determined by the sign of $\cos\theta$. Therefore: $\cos\theta \leq 0 \forall \theta \in \Re, \theta \leq \frac{-\pi}{2} \vee \theta \geq \frac{\pi}{2}$.

As mentioned earlier, the surface normal vector ($\vec{N}$) must be altered to point in the direction of the viewing vector, otherwise the surface will appear to point away from the viewer when viewed from one direction. Since surfaces actually have two faces, this is undesirable. This results in a situation as shown in Figure 1.

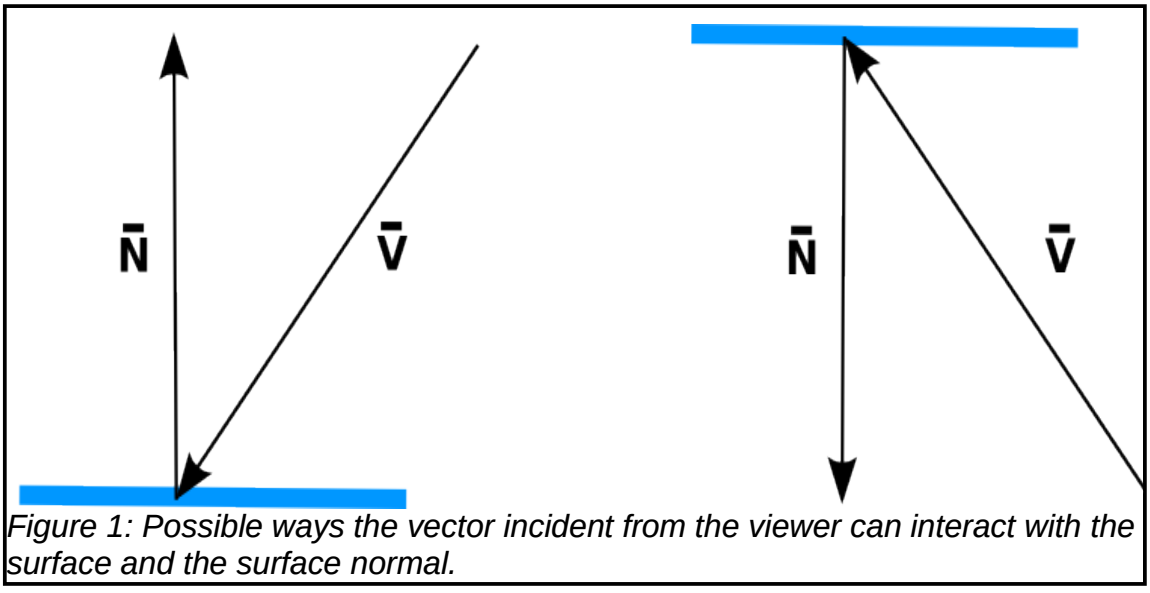

Figure 1: Possible ways the vector incident from the viewer can interact with the surface and the surface normal.

When we re-arrange the vectors to visualize the correct angle between them in terms of vector mathematics (namely θ), we arrive at a situation such as the one in Figure 2:

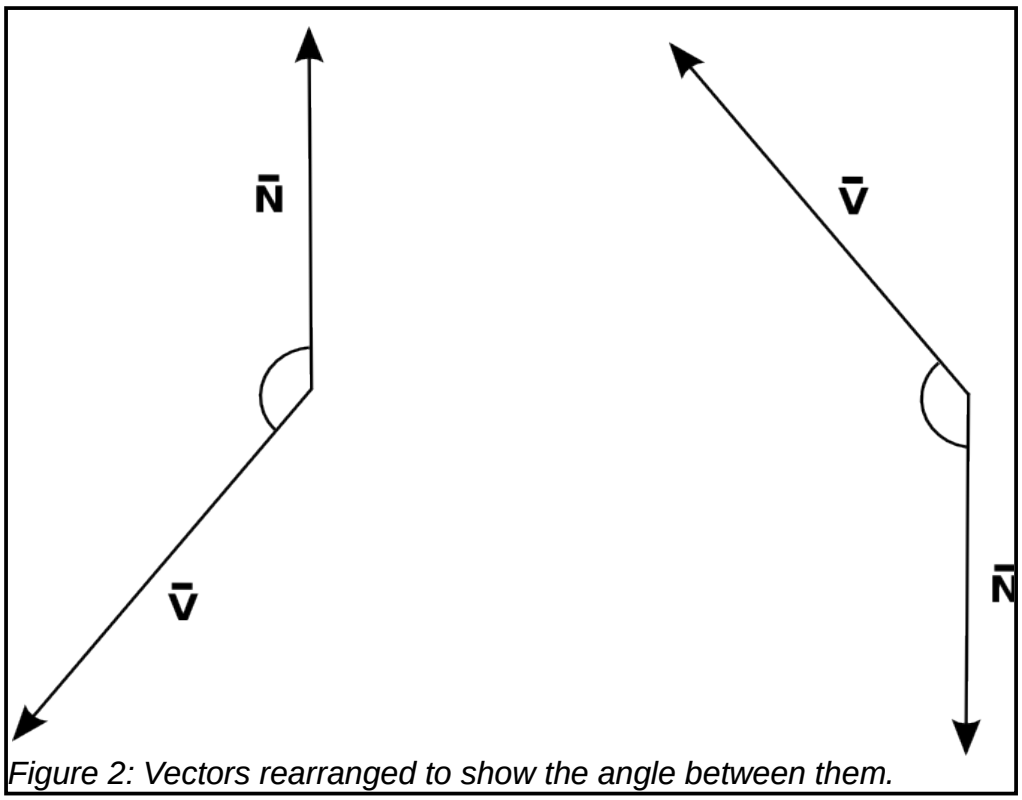

Figure 2: Vectors rearranged to show the angle between them.

Therefore, the direction of $\vec{V}$ causes the angle with the manipulated surface normal to necessarily be greater than $\frac{\pi}{2}$ in magnitude which means that $\vec{V} \cdot \vec{N}$ will always be negative.

The discussion so far has centred on the derivation of the reflection vector. It now focuses on the refraction vector.

Using Snell's law, trigonometry and vector mathematics, Whitted (1980) derives the following set of equations:

- $k_n = \frac{\eta_2}{\eta_1}$ where k_n is the index of refraction, n_1 is the index of refraction of the incident substance – S_i in the diagram (refer to Appendix A) – and n_2 is the index of refraction of the refractive substance – S_r in the diagram (refer to Appendix A),
- $k_f = \pm\left[k_n^2 \left\|\vec{V}'\right\|^2 - \left\|\vec{V}' + \vec{N}\right\|^2\right]^{-1/2}$ and finally,

- $\therefore \vec{P} = k_f\left(\vec{V'}+\vec{N}\right)-\vec{N}$ where $\vec{P}$ is the direction of refraction.

The derivation of all these equations is given more rigorously in Appendix A.

While Whitted's derivations are only stated and not fully justified (Whitted, 1980), they can nonetheless be proven correct quite easily.

2.3 Vector Calculation Improvements

While Whitted's derivation of the reflected and refracted vectors for ray tracing is accurate, it may not be optimal. Heckbert (1989) states that Whitted's derivation results in two square root operations, eight divisions, 17 multiplications and 15 additions every time both reflected and refracted vectors are calculated.

This is one of the simplest optimizations to the naïve ray tracing algorithm. Heckbert (1989) derives a separate set of equations to calculate the reflected and refracted rays that uses only one square root operation, one division, 13 multiplications and 8 additions, which should be faster than Whitted's equations.

Alternatively, Heckbert (1989) proposes a different set of equations that reduces the amount of multiplications to eight, but increases the amount of divisions to four. Depending on the hardware of a particular system, either of these methods may be faster (Heckbert, 1989).

These derivations suffer from the same problem as Whitted's derivations, in that they are not formally derived where Heckbert states them (Heckbert, 1989). Heckbert (1989) provides a comprehensive overview of the derivation, but not a complete and rigorous write-up. Such a complete derivation may be found in Appendix B.

While Heckbert (1989) analyses these equations on a purely theoretical basis, he does not confirm his analysis with empirical observation. While measuring the

amount of operations in a calculation will certainly give a good idea about the relative speed of a certain algorithm, there are many factors that can influence its performance beyond raw calculation count. Examples of such factors are CPU pipeline effects, caching effects and branch prediction accuracy. These are highly complex influences and the only way to determine their effects on each set of equations is to test them side-by-side.

2.4 Performance Techniques

Even in 1980, Whitted could see that the way to improve the performance of his ray tracing algorithm would be to exploit picture coherence (Whitted, 1980). This is the phenomenon where very little data changes between locations that are close together in an image. Any way to exploit this coherence would lead to calculations that can be shared between neighbouring locations or cache optimizations that can yield great performance benefits.

The way to do this in ray tracing turns out to be a method that exploits the SIMD processor technology. SIMD stands for Single Instruction, Multiple Data. Overbeck *et al*. (2008) discuss different algorithms that use SIMD to exploit image coherence and yield performance benefits for the various ray tracing algorithms they researched.

The CPU is very important for the performance of the ray tracing algorithm due to the high number of calculations that each ray intersection generates. Therefore, it is vitally important to optimize most of the CPU instructions produced while coding a ray tracer. This is why the SIMD optimization approach works so well. Unfortunately, it relies mostly on hand optimization and assembly code programming. Therefore, the ability of modern C++ compilers to create highly optimized code is not utilized.

The approach taken by Georgiev and Slusallek (2008) addresses this problem. They construct a ray tracer that is highly adaptable by using C++ template programming (Georgieve & Slusallek, 2008). They justify this decision by noting that some

applications trade performance for adaptability by producing overhead associated with the traditional ways of making programs more flexible – these being approaches such as APIs, or polymorphism (Georgieve & Slusallek, 2008).

Because template programming in C++ produces intermediate code, the compiler has an opportunity to optimize even at the connecting points between interchangeable components – something that cannot be done in an object oriented environment.

One can critise this approach by noting that it increases compile time drastically and that the flexibility in the system can only be configured by someone who has access to the source code and who is able to recompile the program.

Of course, this sort of system is ideal for testing various algorithms and other interchangeable parts, as it does not incur any run-time overhead.

2.5 Acceleration Data Structures

The use of data structures aimed at accelerating ray tracing algorithms has been a focus of the research area over the last decades. This method usually subdivides the three dimensional space rendered by a ray tracer. An algorithm then determines if the current ray might intersect any object contained in each space. If the ray misses a particular subspace then the objects in that subspace can be disregarded. Since ray tracers typically spend most of their time intersecting rays against objects (Whitted, 1980; Fussel & Subramanian, 1988), this optimization proves very effective.

There are two major techniques used by acceleration data structures found in the ray tracing literature. These are space partitioning (including *k*-d, oct- and BSP trees) and bounding volumes (Subramanian & Fussel, 1990a). While the former seeks to divide space into a series of partitions the latter seeks to encase objects in a series

of hierachically arranged bounding volumes that become progressively smaller to more tightly bound an object as a traversal of the data structure moves down the hierarchy (Subramanian & Fussel, 1990a). Either approach leads to fewer intersection tests (Subramanian & Fussel, 1990a:2). The latter technique, in particular, can greatly reduce the number of rays that are considered, but never intersect any object in the scene (Subramanian & Fussel, 1990a).

This section of the literature review will describe each of the data structures that have been created to accelerate the ray tracing algorithm. It will discuss the basic formulation of each structure and the algorithms that construct and query it. The state-of-the-art regarding a particular structure will also be reviewed.

2.5.1 K-D Trees

One well known acceleration data structure is the *k*-d tree. A *k*-d tree subdivides a space according to any number of dimensions (Fussel & Subramanian, 1988) – hence the name: it is a *k*-dimensional tree. The *k*-d tree was introduced by Bentley (1975) and was originally intended for generic uses that involve multiple dimensions, such as records in a file that may have an arbitrary amount of attributes (Bentley, 1975). Bentley's work was later adapted for use in ray tracing by Fussel & Subramanian (1988).

A *k*-d tree achieves its multidimensionality by cycling through the dimensions it is tasked with indexing – each level of the tree is assigned a "discriminator" which indicates a dimension (Bentley, 1975). The discriminator for the root node is 0 (Bentley, 1975). If the amount of dimensions for the tree is k and the current level is l then its discriminator (d) is defined by $d = l \bmod k$.

The *k*-d tree is binary – each internal node of the tree has only two branches (Bentley, 1975). One branch (typically called the left branch) contains all nodes where dimension d is less than that of their parent, the converse holds for the other, "right-hand" branch (Bentley, 1975).

As mentioned previously, Fussel and Subramanian (1988) were the first to realize the use of the *k*-d tree in the area of ray tracing. By following their method the *k*-d tree can be used for a ray tracing application by considering the dimensions indexed by the tree to be the X, Y, Z dimensions of a three dimensional Cartesian space. Each internal node will then divide its children into those in front, behind, below, above, to the right or to the left of some infinite plane that is orthogonal to one of the axes (Fussel & Subramanian, 1988:5-6). This will result in a binary division of the 3D space at each step – one subspace on either side of the plane.

The method described by Fussel & Subramanian (1988) differs somewhat from the original *k*-d tree defined by Bentley (1975). Instead of cycling through the available dimensions using a discriminator as in a normal *k*-d tree, the discriminator is chosen at each level based on the dimension which is able to provide the most even division of objects (Fussel & Subramanian, 1988:3). Furthermore, Fussel & Subramanian's algorithm fills each internal node with a separating plane (Fussel & Subramanian, 1988), where internal nodes in the original *k*-d tree specification are objects in the data set – i.e. in the original *k*-d tree objects *are* “splitting planes” (Bentley, 1975). Lastly, their method does not use any of the algorithms used for traversal described by Bentley (1975; Fussel & Subramanian, 1988), this is probably because Bentley's efforts where focused on queries for sets of data, whereas in ray tracing only the nearest intersection is required. This leads to a much simpler traversal method which also allows for early termination (Fussel & Subramanian, 1988).

It is doubtful whether the term *k*-d tree is even applicable for the structures commonly used in ray tracing, since they bear so little resemblance to the structure proposed by Bentley (1975). These structures should perhaps have been named “binary spatial subdivision trees” or something similar. This is a moot point however, as the use of the term has become entrenched in the ray tracing literature.

The algorithm for determining the plane used to split a space in two is a simple binary search for each of the three dimensions (Fussel & Subramanian, 1988). This

means that the algorithm is at least $O(n^3)$ in terms of time complexity. It is likely that this characteristic is to blame for the long construction times observed by Fussel & Subramanian (1988) during their experiments.

When traversing the *k*-d tree, Fussel & Subramanian's method considers only the ray segment that is within the bounds of the scene (Fussel & Subramanian, 1988). The traversal begins at the root node of the *k*-d tree and proceeds recursively until a leaf node is located that is intersected by the current ray (Fussel & Subramanian, 1988). If the current ray segment is located entirely above, below, in front of, behind, to the left or to the right of the current node, only the relevant subtree will need to be examined (Fussel & Subramanian, 1988). This eliminates an entire subtree of objects. If the ray crosses the plane splitting the scene at the given node, both subtrees will be examined, starting with the subtree closest to the eye (Fussel & Subramanian, 1988). This is because only the first intersection needs to be found. If an intersection is found in the closer of the two subspaces, the algorithm can skip examining the second (Fussel & Subramanian, 1988).

Such an approach reduces the amount of ray-object intersections that must be done and this is the only reason for its increased performance (Fussel & Subramanian, 1988). However, the technique described by Fussel & Subramanian (1988) has a severe weakness. The algorithm under-performs in situations where most of the rays do not hit any object in the scene. (Fussel & Subramanian, 1988). Their choice of fitness function (termed "*fom*" in the paper) can also cause significant problems because it chooses subdivisions based on equality – whether the plane in question divides the available objects evenly. Since it is desirable to even out the size of the subdivided spaces and not the amount of objects they contain it could lead to odd subdivisions (Fussel & Subramanian, 1988). An example is a scene where a large object should be left on one side of a subdivision. Fussel & Subramanian's algorithm (1988) will instead add this large object to a collection of smaller objects – decreasing the performance of the ray tracer. Curiously, Fussel & Subramanian (1988) provide no figures supporting their statement of this flaw in their algorithm (Fussel & Subramanian, 1988).

Furthermore, Fussel & Subramanian (1988) merely provide the time taken by their algorithm and state that it is faster or slower than some of their contemporaries' algorithms (Fussel & Subramanian:1988). While the times can be compared with the algorithms they cite by looking at the results in their respective papers, Fussel & Subramanian (1988) use an adjusted metric since they had no access to the processor they were planning to use. They also do not state the performance achieved over a naïve implementation (Fussel & Subramanian: 1988). These deficiencies can be corrected by implementing their algorithm and testing it against several others on the same hardware and over a number of sample runs. Fussel & Subramanian (1988) also do not provide details on memory or disk usage, which could impact our understanding of their algorithm's efficiency. Detailed execution profiles are also not provided (Fussel & Subramanian, 1988).

Even though this research was conducted in 1988 (roughly 22 years ago) it is still relevant today. For example, a state of the art report on ray tracing algorithms published in 2007 discusses the tradeoffs associated with kd-trees as opposed to other acceleration data structures (Wald *et al*., 2007).

That said, it would be meaningless to conduct research on algorithms that are 22 years old if there have been no significant advances in their performance since then. However, *k*-d trees are certainly an active research topic in the ray tracing community, making Fussel & Subramanian's papers invaluable references. One example of the continuing research on *k*-d trees is the fact that the problems mentioned earlier with Fussel & Subramanian's fitness function have since been addressed with the introduction of the Surface Area Heuristic (SAH).

This heuristic assumes that the probability for a ray to hit an object in the scene is proportional to the surface area of that object (Wald *et al.*, 2007). An algorithm using the heuristic maintains a *k*-d tree that contains the objects in the scene. Each leaf node of the tree contains a certain amount of objects that are contained within its bounding volume, just as before. Using the SAH, we can now calculate the expected

computational cost of splitting a node into two, versus keeping it intact. Which of these two costs is lower will determine the action taken for that node (Wald *et al.*, 2007). This is a much more accurate heuristic than the one proposed by Fussel & Subramanian (1988), but it is also much more computationally complex.

Of course, the papers mentioned thus far are merely the foundational papers describing naïve *k*-d tree algorithms. Recent advances in *k*-d tree technology have done much to improve their performance and the performance of the ray tracing engines that they support. For example, Subramanian & Fussel (1990b) discuss alterations to their original algorithm for use in sparse data sets. They add a bounding box to each of the internal nodes of the *k*-d tree in order to quickly cull a large number of rays (Subramanian & Fussel, 1990b). This increases performance in cases where a large number of rays are missing all the objects in the scene, which was one of the problems with their earlier algorithm.

Subsequent experiments supported this addition to the original *k*-d tree algorithm – the addition of bounding boxes was found to bring the performance of the *k*-d tree on sparse data sets in line with the performance of BSP trees on the same data sets (Subramanian & Fussel, 1990a). BSP trees continued to outperform *k*-d trees on these sorts of data sets since they handle void spaces better, but the performance gap is made very small if the *k*-d tree uses bounding boxes (Subramanian & Fussel, 1990a).

2.5.2 Bounding Volume Hierarchies

Bounding Volume Hierarchies (BVHs) are another type of acceleration data structure available for increasing the performance of ray tracers. While *k*-d trees are seemingly preferred by the ray tracing community (Wald *et al.*, 2007), it has been stated that BVHs offer similar results to *k*-d trees (Wald *et al.*, 2007). In addition, they seem to be better suited to ray tracing dynamic scenes where objects are moving in a three dimensional space (Wald *et al.*, 2007).

2.5.3 B-KD Trees

As discussed above, kd-trees and BVHs have their own unique advantages. These are not mutually exclusive however. Woop, Marmitt and Slusallek (2006) combine the strengths of both approaches to describe the “Bounded K-D Tree”, or B-KD Tree.

A B-KD Tree is similar to a normal *k*-d tree, but instead of containing just one plane splitting a 3D space into two for each internal node, a B-KD tree uses two planes to bound a subspace (Woop *et al.*, 2006). Therefore, a B-KD tree splits the available 3D space into two disjoint subspaces at each internal node of the tree.

On the one hand, a B-KD tree retains the approach of considering only a single dimension at each internal node from the *k*-d tree (Woop *et al.*, 2006). This simplifies traversal of the tree as opposed to BHVs (Woop *et al.*, 2006). On the other hand, a B-KD tree uses a hierarchical bounding approach, which means that sets of objects that are moving together can be updated together, conferring advantages for dynamic scenes (Woop *et al.*, 2006).

This is accomplished with a simple bottom-up update of the bounding planes encountered at each internal node and does not change the tree's shape (Woop *et al.*, 2006). While this is sufficient for situations where the hierarchical structure of the scene geometry changes little during the course of an animation, it is likely to break down in situations where objects move less coherently (Woop *et al.*, 2006). An example would be a scene where an object detaches from its parent and attaches to some other object.

The B-KD tree also provides mesh duplication through transformation nodes, an approach that might save some memory (Woop *et al.*, 2006). Finally, B-KD trees add ordering to the children of internal nodes (Woop *et al.*, 2006). The internal nodes are placed in the same order as they would be struck by a ray travelling along a certain axis in either the positive or negative direction (Woop *et al.*, 2006). If an example ray

is travelling in the opposite direction to this order, only the traversal order needs to be changed.

Woop *et al.* (2006) discusses only a hardware implementation of the B-KD tree. Indeed, the structure was invented as an efficient and easy to implement hardware solution for their ray tracing hardware prototype (Woop *et al.*, 2006). It would be instructive to test their data structure against other acceleration structures in use by implementing it in software in order to measure how much performance accrues to it by virtue of the data structure itself, and how much is due to its hardware implementation.

Furthermore, Woop *et al*. (2006) only provide data for the relative time various parts of the algorithm took during their sample runs together with overall frame rates. They scale many of their results to a 66Mhz clock speed to be comparable to their hardware implementation (Woop *et al.*, 2006). Comparing the algorithm to others on the same hardware in the same conditions would provide a clearer understanding of its relative merit. It is unclear from their paper whether any scaling was done for their PC implementation. Regardless, there are too many variables being changed for their data to be meaningful for the purposes of comparison – the data points are gathered from three different algorithms, using different data structures, on different hardware configurations (Woop *et al.*, 2006). For the purpose of determining the performance of the B-KD data structure alone it would be more revealing to change a single variable at a time. Their comparison of these different ray tracers is also lacking. Woop *et al.* (2006) provides only a single table comparing the ray tracers in three scenes. It is unlikely that these three scenes will be representative enough, or that they will exercise edge cases properly. Like most ray tracing research papers they provide no statistics relating to memory use (Woop *et al.*, 2006). There is also very little about the experimental set up in their paper (Woop *et al.*, 2006).

Interestingly, it seems as if B-KD trees have largely been left by the wayside of ray tracing research. They are mentioned in the literature, but it seems as if there has been very little work done since their introduction by Woop *et al.* (2006). Detailed

analysis of the algorithm's performance under various conditions may motivate more researchers to spend time developing an understanding of it, should it prove to be useful for more than prototype hardware development.

2.5.4 BSP Trees

The binary space partition (BSP) tree subdivides space recursively and also makes use of a tree data structure (Subramanian & Fussel, 1991) like a *k*-d tree. The difference between the two data structures is in the amount of splitting planes. Whereas the *k*-d tree uses only one splitting plane, the BSP tree uses three to subdivide a subspace into four equal subspaces (Subramanian & Fussel, 1991). Note that a *k*-d tree's subdivision does not necessarily result in equal sized voxels. An advantage of using the BSP tree is that its construction algorithm ignores subspaces that are empty – this can lead to great performance gains. Normally, it is quite difficult to determine the level of subdivision that will be optimal for a BSP tree (Subramanian & Fussel, 1991), but a heuristic can be used to automatically decide this with good results (Subramanian & Fussel, 1991). The BSP tree suffers from a distinct disadvantage in scenes that are less regular as many objects may end up in a small collection of nodes in which case there is little gain in the subdivision.

2.5.5 Octree

The octree is very similar to the BSP tree discussed above; except that it partitions space thrice per level and uses a hash table structure instead of a tree (Subramanian & Fussel, 1991). Since they are so similar, it is expected that the octree and the BSP tree share most advantages and disadvantages. Of course, the only way to be sure of this ascertion is to rigourously test the two data structures.

2.5.6 Regular grids

Regular grids are one of the simplest acceleration data structures. They divide a three dimensional space into a cube of voxels that contain references to the items that are at least partially contained within them (Subramanian & Fussel, 1991). The traversal method for this structure is a modification of Bressenham's algorithm

(Subramanian & Fussel, 1991). This approach is easy to implement, but underperforms on scenes where objects are not uniformly distributed. It is difficult to determine the resolution of the grid (the amount of voxels that should be generated) *a priori*, but a useful heuristic was introduced by Subramanian & Fussel (1991) that helps to choose this value.

2.6 Heuristics for Acceleration Data Structures

The data structures in the previous section are designed to increase the performance of a given ray tracer by making the cost of intersection testing easier (Subramanian & Fussel, 1991). The reason for this focus is that this cost is the dominating factor in ray tracing performance (Subramanian & Fussel, 1991). Essentially, the extra performance is gained by caching data, or creating extra data for the scene in some other way. Eventually though, the cost of constructing, maintaining and traversing the data structure must become greater than the amount of time saved by its use.

It is possible to derive a formula that will indicate when the cost of further complicating a data structure is greater than the expected performance increase (Subramanian & Fussel, 1991). This formula is the sum of the cost of finding the first intersection for a ray and the cost of traversing the data structure (Subramanian & Fussel, 1991). By minimizing this function, a ray tracing algorithm that uses such a data structure can be sure that it is gaining performance instead of losing it.

Because the function needs to be quick to calculate (otherwise it will impact the performance of the algorithm) and has to use *a priori* data (all data is only available after a complete ray trace – which is what the function is trying to speed up), it is rather inaccurate (Subramanian & Fussel, 1991). Luckily this is not too much of a problem because the formula as derived by Subramanian & Fussel (1991) closely matches the overall *profile* of the algorithms studied (Subramanian & Fussel, 1991). That is, the local minima in the formula derived by Subramanian & Fussel (1991) more or less match the local minima that were experimentally determined

(Subramanian & Fussel, 1991). This is probably the best that can be expected given such limited data. However, an adaptive approach that gathers data as the rays are being processed might fare better. Note that Subramanian & Fussel (1991) do not supply an approach for using their formula to optimize an algorithm; they merely test its predictive power (Subramanian & Fussel, 1991). A heuristic based on their formula will be valuable indeed. Subramanian & Fussel (1991) also do not provide performance comparisons with previous versions of the algorithms they test, nor do they provide relative speeds, memory usage figures or profile data. These figures would be interesting to study.

The original ray tracing *k*-d tree algorithm proposed by Fussel & Subramanian (1988) has a built in heuristic to stop subdivision (Fussel & Subramanian, 1988). It was found to be just as efficient as the formula discussed above (Subramanian & Fussel, 1991), validating this aspect of their *k*-d tree algorithm experimentally.

2.6.1 General Discussion of Acceleration Data Structures

This section has discussed a number of acceleration data structures that are in use, or were used for the acceleration of ray tracing. The *k*-d tree, BSP tree, octree and others have been described and dissected.

From some of the available literature, however, it seems as if the data structures in use are not quite so clear cut. It is possible to use a *k*-d traversal with a BSP tree and an octree, and it is possible to use the surface area heuristic with a *k*-d tree (Subramanian & Fussel, 1990a). Indeed it seems as if a *k*-d type tree, using bounding boxes and the surface area heuristic had emerged as the best known data structure for ray tracing by the end of 1990 (Subramanian & Fussel, 1990a).

However, it would be a mistake to declare the *k*-d tree the victor here, since it rather appears that a mixture of techniques had triumphed over more naïve approaches. For example, while researchers believe that the *k*-d tree is the best option for general

use, the grid structure is better suited to scenes that are more uniform (Foley & Sugerman, 2005).

2.7 GPU Techniques

Recently there has been a lot of interest in the execution of ray tracing algorithms on the graphics processing unit (GPU) present on current graphics cards (Horn *et al.*, 2007). Since these cards are usually built to parallel process huge volumes of data at interactive speeds, they may prove to be a good platform for the ray tracing algorithm, which is inherently very parallel.

Another attractive aspect of GPU based ray tracing is the fact that GPUs are very good at generating rasterized images of three dimensional scenes very quickly. This ability can be used to quickly determine the first hit location for a large collection of rays (Horn *et al*., 2007; Purcell *et al.*, 2002). Since this is a major part of the work done by a ray tracer, it should speed up computation significantly. The ray tracing algorithm can then be used for the parts it excels at: perfect specular reflection, refraction, shadows, caustics and the like. In addition, the graphics card can be used to perform basic and advanced shading operations (Horn *et al*., 2007), since all ray tracers require some form of shading this capability makes graphics cards attractive platforms for ray tracing.

There is also evidence that GPUs are faster than central processing units (CPUs) for at least some tasks (Buck *et al.*, 2004). There is also the fact that graphics cards have advanced faster than CPUs in the past, since they can always incorporate more pipelines, while it is harder to add more transistors to a CPU (Purcell *et al.*, 2002). It is unclear that this argument still holds in the current day however, since the rise of multi-core CPUs has brought some measure of scalability to the CPU.

2.8 Early Predictions

Prior to the emergence of viable GPU architectures, simulation of the GPU architectures that would emerge in the future managed to predict many of their performance aspects. It was predicted that a GPU that was capable of branching would be faster than a GPU without it (Purcell *et al*., 2002). According to Purcell *et al*. (2002) this would be due in part to extra work and to the coherence that is lost when not using the looping algorithm that branching allows (Purcell *et al*. 2002), whereas Foley and Sugerman (2005) put the inefficiencies in a non-branching architecture down to the data that must be recirculated for every ray. This was later confirmed and the performance gains from branching were estimated at a 25 times speed increase (Horn *et al*., 2007).

It was also predicted that secondary and shadow rays would be less cache friendly than the primary rays that spawned them (Purcell *et al*. 2002) as was later confirmed (Horn *et al*., 2007). Naturally this is a problem on the CPU as well, but because GPUs are so parallel and based on the very idea of coherence, it is a bigger problem on the GPU (Horn *et al.*, 2007; Carr *et al*., 2002).

Acceleration data structures were first implemented on a simulated GPU architecture by Purcell *et al.* (2002; Horn *et al.*, 2005). Their simulation was also the first GPU algorithm to make use of a uniform grid. They note that this structure performs poorly on some scenes (Purcell *et al*. 2002). Interestingly, Purcell *et al*. (2002) proposed the use of the rasterizer on the graphics card to traverse a uniform grid acceleration structure. To the best of the authors' knowledge, this approach has not been implemented.

2.9 The Stream Model

The previous section has discussed some of the advantages that might be realised with the use of a GPU ray tracer. While these advantages are attractive in theory, extracting them in practise has proven to be more difficult.

In part, this difficulty is due to the fact that graphics cards express their programmable units in terms of graphics concepts such as textures and shaders. This is not ideal for the design and implementation of a ray tracer, since these concepts do not map well to ray tracing. It makes more sense to view a GPU as a streaming processor in which data is modelled as streams with specific dimensions that flow through a sequence of kernels (Purcell *et al*., 2002). Each kernel then performs operations on its input stream and produces an output steam that serves as input to the next kernel (Buck *et al*., 2004).

The stream model has several advantages: it encourages independent execution which increases parallelism, it forces kernels to do many calculations versus memory bandwidth utilized and it hides memory latency with the use of pre-fetching (Purcell *et al*., 2002).

In order to capture these advantages and ease the implementation of general algorithms on the GPU a programming environment such as Brook is important. Brook allows programmers to express their algorithms in terms of the streaming model (Buck *et al.,* 2004) and was implemented on the GPU and tested with a ray tracing algorithm as early as 2004 (Buck *et al*., 2004). Brook would prove to be influential in the early research on GPU ray tracing, as it was used by both Foley and Sugerman (2005) and Horn *et al*. (2007) for their implementations.

Brook has not seen widespread use in the most recent papers, this is likely due to the increasing ease of programming that recent GPUs offer.

2.10 Initial Hardware Implementations

To the extent of the authors' knowledge, the first use of graphics card hardware in ray tracing was the use of the cards' rasterization capabilities to speed up the calculation of eye rays' first hit with scene geometry (Carr *et al.*, 2002). This was the only part of the ray tracing process accelerated by the graphics card in that specific approach (Purcell *et al.*, 2002). This approach has the advantage that the CPU can

be used for the tasks it is best suited for: complex algorithms and data structures, while the GPU can be used for the parallel and repetitive tasks for which it was intended (Carr *et al*. 2002). Carr *et al*. (2002) achieved good results with this approach, but their ray tracer's performance was limited by the slow transfer rates between video card and CPU that was the case at the time. Given the recent advances in the technology bridging GPUs and CPUs in the PCI express specification, this approach could be revisited.

The first GPU ray tracing algorithm to make use of the *k*-d tree was described by Foley and Sugerman (2005; Horn *et al*., 2005). Due to memory limitations imposed by the GPU hardware the generic *k*-d tree algorithms had to be adapted to run without a stack (Foley & Sugerman, 2005). Typically, an optimized *k*-d tree will process the child of a node nearest to a ray first and place the further child on a stack (Horn *et al*., 2007). These stack operations can be eliminated by keeping track of the start and end points of a specific ray, and updating the start point to equal the start of the next child's extents when the algorithm finishes with a leaf node (Foley & Sugerman, 2005). When the algorithm then reaches a leaf node with no intersections, it can simply restart from the root and quickly find the node it should search next – this technique is called *kd-restart* (Foley & Sugerman, 2005). By further manipulating these start and end points, the algorithm can determine the parent of the next node to be searched, eliminating a couple of traversal steps (Foley & Sugerman, 2005) – this optimization is termed *kd-backtrack*. There is one major problem with *kd-backtrack* however, as this strategy requires 256 extra bits of storage (Foley & Sugerman, 2005). This cost would prove too large for Horn *et al.* (2007), who were worried about the effects it would have on packetization and bandwidth. All in all, the loss of a stack only increased the cost of *k*-d tree traversal by a linear factor (Foley & Sugerman, 2005).

A year later, Carr *et al*. (2006) developed a method based on the idea of storing an acceleration structure in a MIP map texture as a geometry image. Their method was able to ray trace dynamic scenes and was competitive with other techniques at the time (Carr *et al.*, 2006). Unfortunately, they could only ray trace scenes containing a

single mesh with no sharp edges (Carr *et al.*, 2006, Popov *et al*., 2007). This is probably why their method has fallen by the wayside, despite having competitive performance characteristics for the techniques of the time. It is also likely that the community's familiarity with *k*-d trees pushed research in that direction, rather than into novel approaches.

Around the same time Huang *et al*. (2006) developed the traversal field method. This method constructs a series of ray relays at the faces of the bounding boxes that enclose objects (Huang *et al*., 2006). These relays then sample all the possible incoming directions of rays and associate them with the triangles they would intersect (Huang *et al*., 2006). While their method had a good performance profile when measured against the efforts of Carr *et al*. (2006), it required user intervention (Huang *et al*., 2006) and was subject to aliasing effects caused by the sampling nature of the algorithm (Huang *et al*., 2006). The algorithm also had difficulty dealing with convex objects (Huang *et al*., 2006) and experienced severe performance and memory footprint penalties when the amount of triangles in a scene reached 2^{16} (Huang *et al*., 2006). These difficulties are likely the reason that researchers didn't explore this algorithm further.

The performance figures comparing GPU raytracing to CPU raytracing were disappointing at this point in history. Foley and Sugerman (2005) report that their implementation is an order of magnitude slower than a CPU implementation. This large discrepancy was reportedly due to data recirculation (Foley & Sugerman, 2005) – a problem that was later solved by the use of the new looping features on more modern cards (Horn *et al.*, 2007). Zhou *et al.* (2008) summarily states that the algorithms described in Carr *et al*. (2002), Carr *et al*. (2006) Purcell *et al.* (2002), and Foley and Sugerman (2005) are slower than heavily optimized CPU ray tracers. However, Buck *et al*. (2004) claim significant improvement over a fast CPU implementation on graphics cards with lots of memory bandwidth, but their figures compare ray-triangle intersection per second, rather than the more common and appropriate frames per second. It is uncertain whether their algorithm outperformed

the CPU algorithm in terms of animation speed as their focus wasn't on ray tracing, *per se*.

2.11 Advanced Implementations

The case for GPU ray tracing became much stronger in 2007 with the introduction of at least three algorithms that outperformed CPU ray tracers – Horn *et al*. (2007), Chen and Liu (2007) and Popov *et al*. (2007). Horn *et al*. (2007) achieved nearly double the performance for a single Opteron 2.4 GHz CPU, which is encouraging. Unfortunately there are no figures comparing the performance of their algorithm to recent CPUs.

This algorithm consists mainly of refinements to the approach suggested by Foley and Sugerman (2005). These refinements are called *push-down* and *short-stack* (Horn *et al*., (2007). The focus of these algorithms is to exploit the additional functionality that had been introduced into the programmable units on the graphics cards from 2005 till 2007 – e.g. looping and branching (Horn *et al*., 2007). The *short-stack* optimization provided the majority of the performance improvement – reducing the count of visited nodes by 48 – 52% over the *k*-d tree with *push-down*, which had already reduced counts by 3 – 22% (Horn *et al*., 2007).

These optimizations together with improvements in the hardware's computational power resulted in more than a 25 times performance increase over the work done by Foley and Sugerman (Horn *et al.*, 2007). Most of this performance improvement is due to the introduction of looping into the algorithm (this was previously impossible due to limitations present in the platform), which eliminated the data recirculation problems encountered by Foley and Sugerman (Horn *et al*. 2007).

That said, the hardware still proved to be problematic. The graphics card that was used by Horn *et al*. provided four wide SIMD instructions, but only two scalar operations could be performed at once (Horn *et al*., 2007), which slowed down the algorithm when compared with processors that are fully four wide.

The figures for the packetization introduced by Horn *et al*. (2007) are less rosy. While there is no real penalty or improvement when using ray packets that bounce only once on the GPU, packetization becomes more problematic when more bounces are added (Horn *et al.*, 2007). This is thought to be due to incoherent branching, which is a major problem on the GPU architecture due to its nature (Horn *et al*., 2007). Because of this problem and the limited register memory that is available on current graphics cards, the use of large ray packets is unfortunately unlikely (Horn *et al*., 2007). A modification to the *k*-d tree that results in larger leaves might alleviate this problem in the future (Horn *et al*., 2007).

Chen and Liu (2007) report that they were able to get a 62% - 157% performance boost over a pure CPU solution from just using the graphics hardware to speed up the first hit calculation, even when taking into account the overhead of transferring data between the graphics card and CPU.

At the same time Popov *et al*. (2007) developed an extension to *k*-d trees that significantly reduces the amount of work that is done while traversing the tree. In their algorithm, the *k*-d tree maintains “ropes” at its leaf nodes (Popov *et al*., 2007). These ropes link a leaf node's bounding box faces to the node that is on the other side of that face (Popov *et al*., 2007). This has a number of advantages: first, the resulting algorithm does not require a stack, which saves on memory bandwidth and second, it can reduce “down”-traversals by 5/6 over the method described by Foley and Sugerman (2005).

Popov *et al*. (2007) state that their GPU implementation of this algorithm outperforms the CPU implementation. Their figures also indicate that their algorithm beats the performance attained by the OpenRT system that is designed for CPUs (Popov *et al*., 2007). This is certainly encouraging, but a comparison with other heavily optimized ray tracers available at the time would have been welcome.

Curiously, the method described by Popov *et al*. (2007) doesn't seem to have penetrated the ray tracing research community, as their research is not incorporated into any later papers to the authors' knowledge. It seems like a very effective scheme, however, and more investigation should be done.

The difference in hardware and the algorithms used between the different papers in the literature muddy the waters significantly. There is a need for a standardized platform to compare different approaches on the same hardware.

2.12 Current State of The Art

Previous techniques did not fully exploit the highly parallel nature of modern GPUs. Zhou *et al.* (2008) describe a real-time *k*-d tree construction algorithm that is tailored to this type of architecture. The algorithm builds the tree in breadth-first order, instead of depth-first (Zhou *et al*., 2008). This leads to a large number of threads being spawned, taking advantage of the GPU's high parallelism (Zhou *et al*., 2008). In addition, the algorithm iterates over primitives for the top levels of the trees, making sure that the GPU is fully utilized for the complete run of the algorithm (Zhou *et al*., 2008:126:1).

These techniques are enough to bring their ray tracer up to speed with CPU techniques, as their results trump those of two recently published CPU-based results (Zhou *et al*., 2008).

The performance benefit for GPU over CPU ray tracers seems to be anything but clear cut. Even this algorithm (which is one of the fastest at the moment) is inferior to a CPU algorithm running on eight cores (Zhou *et al*., 2008) for at least one scene.

Using a simulator that makes very favourable assumptions about the memory bandwidth available on modern GPUs, it is possible to determine that current techniques are limited by the work distribution mechanism on modern graphics cards

(Aila & Laine, 2009), rather than the memory bandwidth available on these cards as is commonly thought.

Aila and Laine (2009) argue that the work distribution problem is caused by the fact that each ray is usually assigned as a packet of work to each of the pipelines on a GPU. However, GPUs execute the same instruction on each pipeline at the same time (SIMD). If one ray takes significantly longer to compute than another, then most of the pipelines will remain idle (Aila & Laine, 2009).

It is therefore possible that Zhou *et al*.'s algorithm is only utilizing a fraction of the graphics card's power. If this is the case, then GPU ray tracing performance could far exceed the performance of CPU algorithms in the near future. More research should be done to implement Zhou *et al*.'s algorithm using the work distribution method described by Aila and Laine (2009).

It is entirely possible, however, that Aila and Laine's findings (2009) are not applicable to the algorithm introduced by Zhou *et al*. (2008). Aila and Laine's technique described above makes many assumptions and therefore can only provide approximate data (Aila & Laine, 2009). Since the memory architecture of the simulator used by Aila and Laine (2009) is so optimistic, there is room for error in their conclusions.

That said, the results of the optimizations suggested by Aila and Laine (2009) are compelling. The situation described above can easily be solved by using persistent threads and utilizing speculative traversal (Aila & Laine, 2009). These improvements bring the performance of GPU ray tracers to within 10% of the estimated upper bound on performance as determined by Aila and Laine (2009).

Ironically, these modifications allow the GPU algorithms to reach an efficiency level where memory bandwidth may indeed become a problem (Aila & Laine, 2009).

Future advances in GPU memory bandwidth will therefore be very beneficial to ray tracing.

Kalojanov and Slusallek (2009) also developed a highly parallel construction algorithm, but for uniform grids. They reduce the problem of constructing a grid to a sorting problem, that is easily solved by an implementation of the radix sort algorithm present in the SDK they were using (Kalojanov & Slusallek, 2009). They store their acceleration structure in texture memory on the graphics card in order to make use of the speedy texture cache (Kalojanov & Slusallek, 2009). While their construction algorithm is very quick, the results from the ray tracer is not encouraging. Kalojanov and Slusallek (2009) state that their results are inferior to those already seen on the CPU. However, their ray tracer was not as sophisticated and optimized as the ones they were comparing against. Their true contribution is the fast construction algorithm, which looks very promising. Kalojanov and Slusallek's approach may be useful for dynamic scenes were the acceleration structure must be rebuilt quickly – as their approach can completely hide the computation done to upload new geometry to the GPU (Kalojanov & Slusallek, 2009). However, the memory problems they encountered (Kalojanov & Slusallek, 2009), together with the slow ray tracing speed of their approach will likely mean that their approach will not be used for complex scenes.

Most of these approaches have looked at ways to improve the amount of rays that can be traced per second. However, there are other factors impacting the performance of a GPU ray tracer that may become stumbling blocks in the future. Further improvements to the GPU ray tracing algorithm may include strategies for speeding up the rasterization step, early termination for shadow rays and using the GPU's advanced shading capabilities (Horn *et al.*, 2007). Research into these ideas may yield surprising gains.

2.13 Summary of Data Gathering Methods

The preceding sections have looked at the development of ray tracing from the perspective of various improvements and inventions. This section will take a high-level view to illustrate the flaws inherent in the current research paradigm. It will first focus on the GPU ray tracing research community and will then focus on the more general community.

	Carr *et al*. (2002)	**Purcell *et al*. (2002)**	**Buck *et al*. (2004)**	**Foley & Sugerman (2005)**	**Carr *et al*. (2006)**	**Huang *et al*. (2006)**
Data Structure	Octree & 5-D ray tree.	Uniform grid.	Uniform grid.[2]	*K*-D tree.	Bounding volume hierarchy.	Traversal field.
Focus of Research	Performing ray-triangle intersection on the GPU.	GPU simulation.	Measuring the performance of the Brook programming environment.	Application of the k-d tree acceleration structure to GPU ray tracing.	Storage of acceleration structure in texture memory.	Development of the traversal field structure and ray relays.
Interactive Rendering Speeds Achieved	No.	No.	No.	No.	No.	No.
Approximate FPS	N/A.[3]	N/A.[4]	N/A[5]	~1[6]	N/A.[7]	2 – 10.[8]

Table 1: A comparison of some GPU ray tracing papers.

Table 1 and table 2 summarize the approaches used by each of the papers discussed earlier. Almost every study introduces its own take on performance

2 Buck *et al*. (2004) state that they based their ray tracer on Purcell *et al*.'s work, therefore it is assumed that they used the same acceleration structure.

3 FPS is not stated, but the ray tracer achieved speeds of 100 000 – 200 000 rays per second which far exceeded the CPU ray tracers available at the time.

4 The research does not include any timing information.

5 The research contains no timing information, but states that between 45 and 186 ray-triangle intersections were performed per second (Buck *et al*., 2004).

6 There is no data about FPS in the research *per se*, but the ray tracer described achieved rendering speeds of ~950 ms on the most complex scene rendered.

7 The research does not include any data on frames-per-second achieved, but states that an image was rendered at 1272 x 815 in approximately half a minute.

8 While the research does not include any data on FPS, it states that the ray tracer involved could compute an image in ~100 – 450 ms for one of the scenes. However, this data is only for eye rays which makes it an ineffective measure.

enhancement, ignoring many of the advances, observations and improvements that were made previously – promising results from a previous study are rarely developed further.

	Horn *et al.* (2007)	**Chen & Liu (2007)**	**Popov *et al*. (2007)**	**Zhou *et al.* (2008)**	**Aila & Lane (2009)**	**Kalojanov & Slusallek (2009)**
Data Structure	*K*-D tree.	Bounding volume hierarchy.	*K*-D tree with "ropes".	*K*-D tree.	BVH.	Uniform grid.
Focus of Research	Application of Foley and Sugerman's work (2005) to a branching GPU architecture.	Use of the hardware Z-buffer algorithm to speed up first hit calculations.	Development and performance analysis of the improved K-D tree structure.	*K*-D tree construction improve-ments.	Work distri-bution improve-ments.	Fast construction of uniform grid.
Interactive Rendering Speeds Achieved	Yes.	Yes.	Yes.	Yes.	Yes.	Yes.
Approx. FPS	N/A.[9]	~10 depending on scene[10]	4.0 – 12.7[11]	4.8 – 32.0[12]	N/A[13]	3.5 – 7.7[14]

Table 2: A comparison of some more recent GPU ray tracing papers.

This is not the only problem, however. There is also a great deal of variation in the experimental methods used by each paper. No agreement has been reached in the GPU ray tracing community regarding an acceptable standard performance metric or a set of representative and common testing scenes. This will be illustrated by tables 3 and 4.

9 The research claims interactive rendering rates and a sustained rate of 15 million rays per second, but makes no mention of any timing information (Horn *et al.,* 2007).

10 There's no timing information in the research, but it does briefly state a computation time of 115ms on the Stanford bunny scene (Chen & Liu, 2007).

11 These figures are for the ray tracer running on four different scenes with secondary rays and packet tracing (Popov *et al*., 2007).

12 Four dynamic scenes at 1024 x 1024 resolution (Zhou *et al.*, 2008).

13 The research reported 20-40 million rays per second presumably with secondary rays (Aila & Lane, 2009).

14 This measurement is only for the generation of eye rays (Kalojanov & Slusallek, 2009).

	Carr *et al*. (2002)	**Purcell *et al*. (2002)**	**Buck *et al*. (2004)**	**Foley & Sugerman (2005)**	**Carr *et al*. (2006)**	**Huang *et al*. (2006)**
Performance Metric	Rays / second.	SIMD efficiency, traversal steps and intersections.	Ray / triangle intersections per second.	Elapsed milli-seconds and various traversal counts.	Elapsed milli-seconds.	Rays / seconds and inter-sections / ray.
	Horn et al. (2007)	**Chen & Liu (2007)**	**Popov et al. (2007)**	**Zhou et al. (2008)**	**Aila & Lane (2009)**	**Kalojanov & Slusallek (2009)**
Performance Metric	Frames per second and millions of rays / second.	Elapsed seconds and percentage speed-up.	K-d tree statistics, traversal steps and frames per second.	Elapsed seconds and frames per second and speed-up factor.	SIMD efficiency, millions of rays / second and percentage of simulated performance.	Frames per second and milliseconds.

Table 3: A comparison of the performance metrics used by some GPU ray tracing papers.

Rays per second, elapsed time and frames per second are used as metrics several times, but there is still very little unification between papers. This means that it is very difficult to compare the performance of one ray tracer to another.

It is also unclear which of these measurements is the best, and if any of them are suited to the comparison of experimental results. There is a need for research to be conducted to investigate which of these measurements describes the performance of a ray tracer in the most precise manner. Such a metric will have to eliminate as many variables as possible.

	Carr *et al*. (2002)	**Purcell *et al*. (2002)**	**Buck *et al*. (2004)**	**Foley & Sugerman (2005)**	**Carr *et al*. (2006)**	**Huang *et al*. (2006)**
Scenes	"Teapot room", "office" and "soda hall".	"Soda hall", "forest" and "bunny".	"Glassner"[15]	"Robots", "kitchen", "Cornell box" and "Stanford bunny".	"Stanford bunny" and "Mult."	"Desk", "cube", "teapot", "bear", "venus", "simplified bunny", "approxi-mate bunny", "teapot house" and "bunny couple".
	Horn et al. (2007)	**Chen & Liu (2007)**	**Popov et al. (2007)**	**Zhou et al. (2008)**	**Aila & Lane (2009)**	**Kalojanov & Slusallek (2009)**
Scenes	"Cornell box", "kitchen", "robots" and "conference".	"Bunny", "dragon" and "easter".	"Shirley6", "bunny", "forest" and "conference".	"Toys", "museum", "robots", "kitchen", "fairy forest" and "dragon".	"Conference", "fairy" and "Sibenik".	"Thai statue", "soda hall", "conference", "dragon", "fairy forest", "sponza", "ruins".

Table 4: A comparison of the scenes used by some GPU ray tracing papers.

Like the performance metrics, there is a wide variety of scenes in use by the ray tracing community. While several scenes are used repeatedly, there is still too little correlation to make comparisons easily.

If we are to obtain meaningful experimental results that are comparable, then all variables must be controlled for. Certainly the use of certain scenes is one such variable. Haines (1987) and Lext, Assarsson and Möller (2001) have made some progress towards this ideal, but their scenes are seldom used: as shown by table 4 – only "kitchen" and "robots" from Lext *et al*.'s library is used 3 times.

The viewpoint from which a scene is rendered is also important. Most of the papers surveyed did not specify this viewpoint, even though it is an important variable. More care should be taken in the future with regards to stating this.

15 It is unclear whether any other scenes were used. However, this scene name is mentioned on page 780 and the performance graphs suggest that only one scene was used.

	Carr *et al*. (2002)	**Purcell *et al*. (2002)**	**Buck *et al*. (2004)**	**Foley & Sugerman (2005)**	**Carr *et al*. (2006)**	**Huang *et al*. (2006)**
GPU	Radeon 8500 / GeForce 3 / GeForce 4 Ti4600	Not stated.	Radeon X800 XT Platinum / GeForce 6800 (Pre-release)	256 MB ATI X800 XT PE	GeForce 7800 GTX (430 MHz clock, 1.2GHz memory clock)	256 MB NVIDIA 6800GT
CPU	Not stated.	Not stated.	3 GHz Pentium 4 (875P Chipset)	Not stated.	2.2 GHz Athlon 3500+	2 x 3.2 GHz Pentium 4
Memory	Not stated.	Not stated.	Not stated.	Not stated.	Not stated.	2 GB
	Horn et al. (2007)	**Chen & Liu (2007)**	**Popov et al. (2007)**	**Zhou et al. (2008)**	**Aila & Lane (2009)**	**Kalojanov & Slusallek (2009)**
GPU	512 MB Radeon X1900 XTX (650 MHz clock & 750 MHz memory clock)	Radeon X300SE	GeForce 8800 GTX	768 MB GeForce 8800 ULTRA	GeForce 285 GTX	1 GB GeForce 280 GTX
CPU	2 x 2.4 GHz Core2 Duo	1.8 GHz Athlon64 3000+	2.6 GHz Opteron	3.7 GHz Xeon	Not stated.	4 x 2.66 GHz Core2 Quad
Memory	Not stated.	Not stated.	Not stated.	Not stated.	Not stated.	Not stated

Table 5: Hardware used by different GPU ray tracing papers.

Table 5 illustrates the wide variety of hardware used to test the performance of the various algorithms discussed in the papers above. The great difference between the performance of the various components identified obscures the differences between the performance of the algorithms discussed.

2.14 Other Ray Tracing Research

The previous section discussed papers published on the topic of GPU ray tracing and highlighted some of the flaws in the current research paradigm. While these results are interesting, they are only applicable to one subfield of the ray tracing research community. In the interest of fairness, this section will sample six papers from various other fields in the ray tracing research. This will help to determine whether the same problems are present in the wider context of ray tracing.

	Georgiev & Slusallek (2008)	**Overbeck *et al.* (2008)**	**Wald & Slusallek (2001)**
Performance Metric	Frames per second.	Frames per second & performance benefit.	Frames per second & microseconds per primary ray.

Scenes	“Sponza”, “conference” and “soda hall”.	“ERW6”, “toasters”, “fairy” and “rings”.	“MGF office”, “MGF conference”, “MGF theater”, “Library”, “Soda Floot 5” and “Soda Hall”.
GPU	Not stated.	Not stated.	NVIDIA GeForge II GTS
CPU	2 x 2.6 GHz Core2 Duo[16]	8 x 2.0 GHz Intel Xeon	800 MHz Pentium III[17]
Memory	Not stated.	Not stated.	256 MB
	Wald, Mark, Günther, Boulos, Ize, Hunt, Parker & Shirley (2007)	**Havran, Herzog & Seidel (2006)**	**Wächter & Keller (2006)**
Performance Metric	Frames per second.	Time to construct the data structure and render the scene along with several other metrics.	Frames per second and milliseconds required to render.
Scenes	“ERW6”, “conference”, “soda hall”, “toys”, “runner” and “fairy”.	“Conference”, “bunny”, “armadillo”, “dragon”, “buddha”, “blade”, “robots”, “museum” and “kitchen”.	“Shirley Scene 6”, “Dragon”, “Buddha”, “Kitchen”, “Conference”, “Bunny”, “Car 1” and “Blender Suzanne”, BART scenes and “UTAH Fairy Forest”.
GPU	Not stated.	Not stated.	Not Stated.
CPU	2.6 GHz Opteron	2 x 3800+ AMD Athlon 64[18]	2.8 GHz Pentium 4HT / 2.6 GHz Opteron[19]
Memory	Not stated.	Not stated.	Not stated.

Table 6: Performance metrics, scenes and hardware used in six ray tracing papers.

Note that the information in table 6 is for the sections of the papers discussing total performance measurements only. Other sections dealing with the performance of specific elements and the like have been ignored.

The papers covered in table 6 are much more homogenous with regards to the performance metrics used. Only one paper (Havran *et al*., 2006) uses a different metric than frames per second. Since these papers are slightly more recent than the GPU papers discussed earlier, the dominance of a single metric is an encouraging sign that the research community is settling on a common performance metric. This will greatly simplify the comparison of results between papers.

16 One of the algorithms was tested on a different hardware platform, although the authors of the paper do not provide specifics (Georgiev & Slusallek, 2008:121).

17 The researchers also used the SGI Onyx-3 and the SGI Octance graphics supercomputers for rasterization performance figures (Wald & Slusallek, 2001).

18 While the hardware supported symmetric multiprocessing the authors chose not to take advantage of it (Havran *et al.*, 2006).

19 The research contains reference to previous work done on the latter processor (Wächter & Keller, 2006).

The hardware used is also more homogenous – all of the CPUs (save one) are in the 2.0 GHz to 3.0 GHz range. Unfortunately, the amount of cores in use ranges from one to eight. The single paper from 2001 that uses a slow single-core 800 MHz processor is a reminder that performance metrics should be hardware independent in order to remain relevant into the future and allow researchers with access to poor hardware to fairly represent their findings.

Since the ray tracing algorithm is inherently very parrallel, the amount of cores available is a very important variable. Since the research in question focuses on finding better ray tracing algorithms and not better ray tracing CPUs, it would be ideal to eliminate this variable from the experiments.

It is also troublesome that the GPU used and the memory installed on the testing platforms is not mentioned at all. Every component in a computer has some effect on its eventual performance. The descriptions of the CPU alone that is often encountered in the literature may be too simple to give an accurate idea as to the performance of the testing platform. Ideally, a performance metric would ignore these complicated issues entirely, making life easier for the researchers involved while also providing a clearer idea about the performance of a given algorithm.

These goals are hard to achieve, since the efficiency of a given computer is a very complex thing to measure or standardize. It would also be impractical to require researchers to use a single, pre-determined hardware platform for their measurements. As such a platform would age, its components would eventually be hard to come by and many researchers would balk at the extra effort it would entail.

Another solution to the problem is needed, one which any researcher can easily implement and which would give consistent and clear results. The rest of this report discusses a possible solution that would meet these criteria.

2.15 Conclusion

This chapter has presented the development of ray tracing algorithms from their inception to the current state of the art. It has highlighted several problems in the literature that pertain to the way that results are presented.

Much progress has been made with regards to the performance of ray tracers – both on the CPU and on the GPU – but there is considerable confusion in the existing literature. The experimental set up for most of the papers that have been reviewed here is ad-hoc. Two different algorithms are sometimes compared by their performance on completely different hardware platforms.

Advancing hardware is also an issue. It is difficult to compare experiments in this field because the algorithms are so heavily dependent on the newest technology.

All of these issues make comparisons between algorithms very difficult – even for papers that were published only in the last three years.

If the ray tracing community is to learn which algorithms are effective, then it must find a way to compare the results from different studies in a fair way. Currently, the literature lacks a methodology that is capable of achieving this.

Such a methodology will need to find performance metrics that are independent from the underlying hardware and the properties of specific scenes. It will need to be widely acceptable and must be easy to use so that it will be used consistently.

It is the author's belief that research into such a methodology will benefit the ray tracing community and may lead to great advances in the field.

CHAPTER 3: DATA GATHERING METHOD

3.1 Introduction

This chapter will discuss the data gathering method the study used. The study's secondary research questions centred on the applicability and use of rigorous, scientific experiments to Computer Science. Therefore, this chapter specifies the procedure followed in great detail. The experiments conducted to gather data for the study are also set up with great care to any external variables that might affect their outcome. Any remaining variables will be stated.

One secondary research question of the study inquired as to the variables present when conducting the types of experiments presented here and how to control for them. As such, the variables encountered when setting up experiments and the methods used to eliminate them are listed in this chapter.

Some discussion will also be provided regarding which variables affect a ray tracer's performance the most and why they are important.

Naturally, the hardware testing platform for the study will also be described in great depth.

Finally, the exact approach to data gathering in this study will be detailed.

This chapter draws heavily on a paper read at the IBIMA 15 conference in partial fulfilment of degree requirements (Kroeze *et al*., 2010a).

3.1 Variables

Computers are complicated machines. Details of architecture and implementation can affect the relative performance of various techniques and approaches. Modern processors have very advanced pipelines, caches and prediction logic – these technologies often play a big role in applications (like ray tracing) where peak performance is paramount. Last, but not least, the effects the compiler's optimizer can have on the performance of code can also cause surprising results.

In order to provide reliable results, a scientific study must carefully eliminate any and all of these variables from each run of an experiment. This ensures that the study obtains comparable information and that algorithms are not penalized or advantaged by some external factor.

This section will detail the variables encountered during the experimentation phase of this study and discuss how they were eliminated.

3.1.1 Compiler Optimization And Debugging Symbols

The optimizations that some C++ compilers can apply to source code during the translation to machine code can sometimes cause the performance profile of a program to change radically. Likewise, the addition of debugging symbols can add significant overhead, or cause unexpected bugs. These occurrences are most often observed in multi-threaded applications. These types of programs are very sensitive to timing, so a small alteration to a program's machine code can cause previously hidden race conditions to appear.

During the development of this study's testing platform, this effect was observed. When the code was switched to a threaded form running in parallel on four CPUs the debug version of the program actually ran slower, whereas the optimized version obtained a 2x speed increase.

The difference in the performance profiles for these two versions and the different ways they are compiled are shown in Figures 4 & 5.

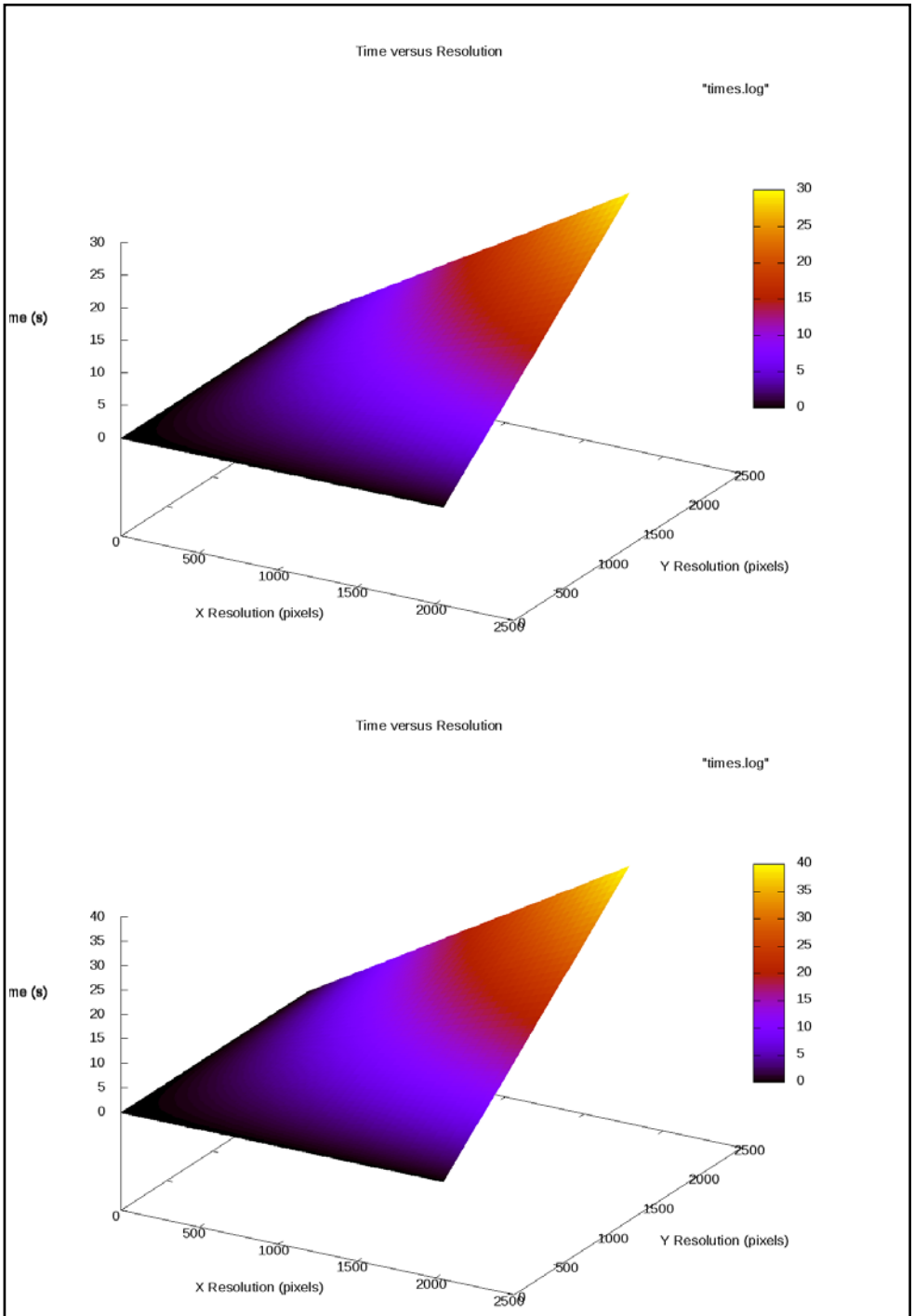

Figure 3: (a) Performance profile for version 1.1.2 (top) and (b) version 1.2 (bottom). These graphs are taken from the debug version.

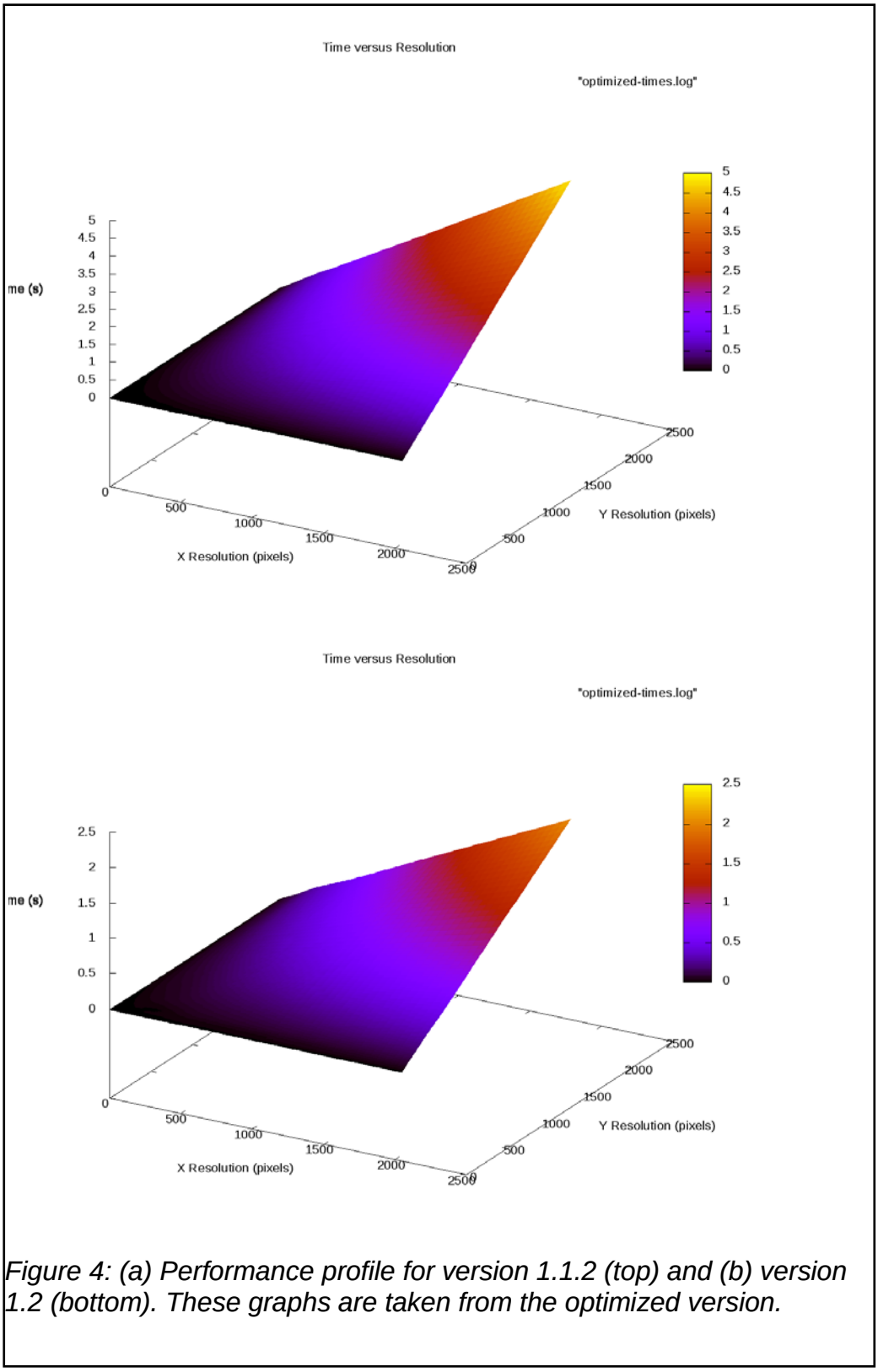

Figure 4: (a) Performance profile for version 1.1.2 (top) and (b) version 1.2 (bottom). These graphs are taken from the optimized version.

Note the steeper slope for the debug version that includes threading. This likely indicates additional overhead being present in the program. It is possible that the

debugging symbols affect the performance of threaded applications to a greater degree than others.

Also note the great effect the debug symbols have not only on the absolute running time of the application, but also on the steepness of the performance curves. This illustrates the effect that these debug symbols can have on the measuring of a program's performance.

The effect is so great that their use can lead scientists astray when doing research. Looking at figure 3, one would conclude that the version without threads is actually running faster. However, the optimized version is the one that really matters, since any type of production application using the results of research will be compiled in this way.

Given the distorting effect that is possible with these debug symbols / optimization passes, this study will use the timing results from runs with no debug symbols or optimization passes. Since this type of compilation will change the source code the least, it is felt that it will lead to the least amount of distortion when discussing the performance of specific techniques or algorithms. In this way, the study can eliminate another variable from the experimental stage. Of course, the different compilation options that other authors have used should be taken into account when comparing the results of this study with the results of others. In that case, it may be illuminating to compare the results of combinations of different compilation options used to the results in question. Unfortunately, the compilation options used during a study are seldom stated, making these comparisons quite difficult.

Unfortunately, this approach will lead to a loss of the profiling information that is supplied by these debug flags.

3.1.2 Object Orientation

The use of the object oriented approach has been controversial in terms of performance. It certainly adds a measure of overhead to any program, but the extent

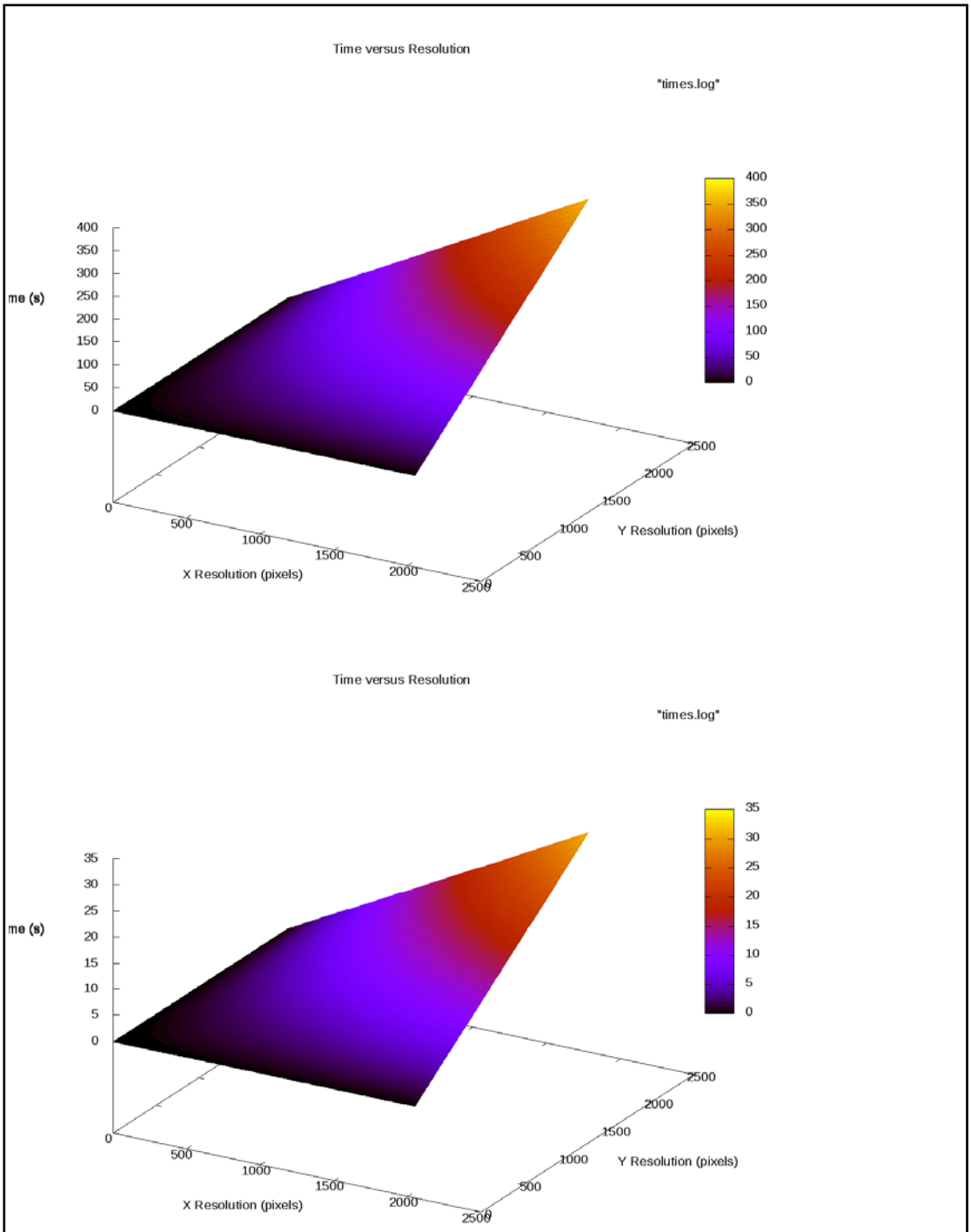

Figure 5: (a) Performance profile for version 1.0 (DEBUG). Slow Vector class (top). (b) Performance profile for version 1.1 (DEBUG). Faster SimpleVector class (bottom).

of the impact is disputed.

While creating the testing platform for this study, it was discovered that a highly generic and complicated class was responsible for a sizeable performance penalty. The class Vector3 depended on a generic Matrix class. Most of the Vector3 methods were simple calls to underlying Matrix methods. The Matrix class also dynamically allocated all its data on the heap. Unfortunately, all this lead to an extreme amount of overhead.

Since a ray tracer has no need for matrix operations and uses only a couple of simple vector operations it was decided to create a stripped down version that would have the minimal amount of overhead.

The new SimpleVector class would store all its data on the stack. Since vectors are created one-by-one and quickly destroyed by a ray tracer this would not cause a stack overflow. SimpleVector also exposed its private data directly to avoid the overhead that is associated with getters and setters. Since this class is used in the highly critical inner loop of the ray tracer, its performance is paramount. Therefore, this violation of encapsulation was felt to be justified.

Results from this process are shown in Figure 5.

Since object oriented techniques can add this level of overhead to a program, their use will be restricted for the testing platform. While most classes will closely adhere to object oriented design principles, the ones used in the innermost loop of the program will have a design that errs on the side of performance.

It is possible to over-optimize here, however. Subsequent experimentation with removing getters and setters from some classes yielded only marginal benefit. Figure 1(a) shows the performance profile of the program after this process was

completed, and a Colour class introduced. The latter addition added some three seconds to the execution time (this addition was the difference between version 1.1 and version 1.1.1). The simplification of the other classes yielded about a four second saving (this was the difference between version 1.1.1 and version 1.1.2).

The performance benefits from these wide scale simplifications are questionable and can be avoided in the future, but the changes were kept for the simplifying effect they had on the source code.

3.1.3 Standardized Testing Frameworks

Several sets of standard testing scenes for ray tracers have been suggested in the literature. Since the testing to be performed for this study focuses more on hardware differences than oddities in the ray tracers themselves, a reasonably taxing and representative array of tests will be suffificient.

The testing framework must be able to load each of these scenes, so it will need a file handler for each of the file types used in the literature.

All the scenes to be used will be collected into a “scene battery”. Each ray tracer will then render each scene in the battery and the time it took to render the scene will be recorded.

3.1.4 Resolution

Since a ray tracer must trace at least one ray per pixel in the target image, any ray tracing algorithm must be at least linear in the amount of pixels. Because of this, resolution obviously affects the absolute running time of a ray tracer. It is also quite likely that an increase in image resolution will have a disproportionate effect on the performance of some ray tracers because they may be using super-sampling to anti-alias an image, or they may iterate over more objects checking for intersections because they have a more innefficient data structure. It is also possible that this

effect is different on various hardware platforms – due to the difference in raw speed, or because some operations execute faster on some CPUs than on others.

However, since the benchmarking tool used in this study is a ray tracer itself, all these factors should already be taken into account. For this reason, and because the naïve benchmark ray tracer executes so slowly, a relatively small resolution of 512x512 was arbitrarily picked for the experiments executed in this study.

3.1.5 Image Quality

There exists some techniques for ray tracing that sacrifice image quality for increased performance. Since this complicates the experiments and adds yet another variable these techniques are considered beyond the scope of this study.

3.1.6 Caching Effects

Modern operating systems usually cache very aggressively. Due to the high cost of disk I/O, a program can sometimes run much faster the second time through, since it does not need to access its files from the disk itself due to them being cached in primary memory. This is a complex issue to solve from an experimental standpoint. Fortunately, the data files used by the programs are relatively small – they are only a couple of megabytes in size. The actual ray trace for the benchmark takes so long, that it dominates computation time. Since these twin factors are quite powerful, the question of caching effects has been ignored in this study. This is a possible area for future improvement of the results obtained herein.

3.1.7 Disk Access And IO

Since the study wishes to investigate the relative performance of algorithms, it should not take into account the time it takes to read in files and output any results to disk. This time can be highly variable depending on where the disk heads lie and the amount of disk contention. It is therefore better to eliminate this variable completely.

Output of results can be ignored, because it can be easily turned off in the program, once it is certain that the program actually *produces* the correct results. The image could still be computed, but not written to disk.

However, in the interest of scientific openness and so that peers can investigate the images produced by the ray tracers used for themselves, this refinement will not be used. Since the images being written to disk are very small (512x512 pixel PNG files), it is hoped that the asynchronous IO capabilities of the operating system will take care of this problem. This is a possible area to improve with future research.

Reading in of files cannot be avoided in the same way however, as the scenes used for testing will typically be stored in external files that must be input to the program in some way. This will remain a variable that cannot be eliminated at the moment, but the existence of the operating system's drive cache and the small size of the scene description files will hopefully counteract any effects.

3.1.8 Console IO

Console IO is at least an order of magnitude slower than the calculations used for ray tracing. This is due to the system and hardware calls involved in writing to a terminal. To prevent the program waiting for output to a console it will emit no messages. Only the test suite's driver application will emit records to the disk, and messages to the console *between* runs.

3.1.9 CPU Contention

To eliminate most of the chances of other processes using the CPU while the tests are in progress most of them will be closed. The ideal way of doing this is to use a Linux system booted into single user mode with all processes stopped that can be stopped. Ideally, only **init**, **tty** and **bash** should be running. Regardless, this step will at least minimize the amount of CPU contention that might affect the execution time of the programs.

3.1.10 Temperature

Modern CPUs often come with thermal throttling technology – when the CPU reaches a certain temperature, the motherboard will lower its clock speed in an attempt to save the CPU from overheating and failing.

Unfortunately, the author did not have access to testing platforms that guaranteed cool operating temperatures, so the possibility of thermal throttling affecting the results exists.

Monitoring applications could be installed to monitor and record the CPU's temperature during the tests so that the efficacy of this control can be assessed. However, this step will produce some CPU contention, which should be avoided.

However, since the tests run for so long, it is expected that any thermal throttling effects will be present in all the tests, and therefore a constant factor. Therefore, no special measures were taken to control for these effects.

Ideally, the tests conducted in this study should be repeated on platforms were the CPUs are guaranteed to remain cool enough that thermal throttling will not be utilized.

Ambient temperature should not present a problem, as it is the absolute temperature of the CPU itself that is important. As long as the latter remains more-or-less constant it should not interfere with the tests.

3.2 Hardware

The performance figures for this study will be meaningless without a detailed discussion of the hardware on which the study will execute its experiments.

Computers are very complex and even small alterations to their hardware configuration can have far reaching consequences.

Briefly, the test platforms for the study's experiments have been set up in this fashion:

	Straylight	Neolith (Laptop)	Monolith
CPU	4 x 2.6GHz Intel i5 750	2 x 1.6 GHz Intel Core2 Duo T5500	1 x 3.4 GHz Intel Pentium 4
CPU Cooler	Thermalright MUX-120	Stock	Stock
GPU	NVIDIA GT250	Intel Mobile GM965	ATI Radeon 9600
Memory	4 GB DDR3 1333 MHz	1 GB DDR2 667 MHz	1 GB
Hard Drive	1 TB Seagate SATA 3GB/s	120 GB Seagate Momentus SATA / 1. GB/s 5200 RPM	160 GB Seagate Barracuda Ultra ATA / 100MB 7200 RPM

Table 7: Hardware installed on the test platforms.

Of course, these figures are merely approximate. A standardized benchmark will be much more illuminating for comparison with other test set ups.

3.3 Test Scenes

There is very little published research regarding common test scenes for testing ray tracer performance. The earliest collection of test scenes seems to be the “standard procedural database” (SPD) created by Eric Haines (1987). This database contains: a three dimensional version of the Sierpinski Gasket that was used at the time by several researchers, a fractal mountain with three glass orbs used to test reflection and refration, a tree model, a set of ten-sided rings that should stress most ray tracers and a scene containing many gears to test how various algorithms handle concave polygons (Haines, 1987:4).

Haines's database is published in the neutral file format (NFF) that is human-readable and very easy to parse. Because of the ease of use of these scenes they will be used during the empirical part of this study.

When ray tracers with interactive frame rates became a reality, Lext *et al.* (2001) published a set of scenes to test ray tracers on animated scenes. This suite of scenes is called the “benchmark for animated ray tracing” (BART).

Lext *et al*. (2001) identify eight stresses that a ray tracer could encounter: hierarchical animation, more random animations, the “teapot in a stadium” problem, bad temporal coherence, high memory utilization, overlapping bounding volumes, changes in the concentration of objects in specific parts of a scene and a large number of lights.

BART is designed to present a number of stresses in each scene to a ray tracer (2001). Therefore BART is similar to a stress-test. It is aimed at finding weak spots in a ray tracing algorithm. One criticism that may be levelled at the suite is that because multiple stresses are present in each scene it may prove hard to isolate the real problem a ray tracer is encountering.

BART is represented in the “animated file format” (AFF) which is an extension of the “neutral file format” (NFF) created by Haines (Lext *et al*., 2001). However, since the addition of animation would complicate the simple ray tracer greatly, and also increase the time needed for experiments, this suite will not be used in the empirical section of this study.

While SPD and BART seem to be the only attempts at proposing a standard test suite, there are many other scenes that are used in the literature. The office, conference and theatre scenes from the MGF repository are popular (Wald & Slusallek, 2001; Georgiev & Slusallek, 2008). A model of the Soda Hall building is also often used (Wald & Slusallek, 2001; Georgiev & Slusallek, 2008). A scene of an atrium named “Sponza” is also used regularly (Georgiev & Slusallek, 2008). The “toasters” and “fairy” (Overbeck *et al*., 2008; Kalojanov & Slusallek, 2009) scenes are often used. There are many more scenes that are less commonly used by the ray tracing community. This proliferation of test scenes suggests that an attempt should

be made to create a general test suite. This will ensure that results are more easily comparable.

For the purposes of this study, however, only the SPD scenes will be used. They are considered to be representative enough. A survey of all models currently in use would be a daunting task indeed.

3.4 Measurements

In the ray tracing community it is quite common to measure the performance of a ray tracer by the amount of frames it can render per second. This figure is known as the “frames per second” (FPS). While this figure indicates the overall performance of a given ray tracer, it can be misleading to compare the FPS achieved by two different ray tracers. This is because those ray tracers may have executed on different hardware platforms. Even the smallest difference between two computers can have an impact on performance. Since empirical observations rely on eliminating as many variables as possible, this situation is unacceptable. There should be a way of measuring the performance advantages of one method over another without succumbing to this problem.

Lext *et al*. (2001) propose that researchers should use the deviation from the average frame time as a performance measurement together with a measure of the continuity of frame render times over frames. They state that this measurement is relatively independent of hardware configuration (Lext *et* al., 2001). Since the experiments conducted for this study focus on single-frame scenes, this technique is not applicable. However, it is interesting as it may eliminate many hardware variables.

Since approximate rendering methods are considered outside the scope of this report, measurements such as the peak signal noise ratio (PSNR) and mean square error (MSE) discussed in Lext *et al*. (2001) will not be utilized.

The measurements used for the experiments conducted as part of this study will be quite simple. A basic ray tracer will be used to render a scene from the battery, then a more advanced ray tracer will used to render the same scene. The ratio of the time taken by the latter over the former, expressed as a percentage is the measure of that ray tracer's efficiency. The same procedure will be conducted for a slightly optimized version of the basic ray tracer, as a control.

This experiment will be conducted on each of the hardware platforms detailed above. The series of ratios from each platform will then be related together with a correlation coefficient (r^2). If this coefficient is close to one then the ratios from one platform are closely related to those of another, and we can conclude that the technique described above eliminates most of the variability inherent in the difference between hardware platforms.

3.5 Program Development

Prior to gathering the data needed for this study, a program will be developed to assist in this regard. This program will be a multi-core un-optimized C++ ray tracer that can serve as a baseline implementation of a ray tracer.

The program will be multi-core to accommodate various ray tracers. There are many ray tracers in the literature that are capable of running multiple threads to speed up the calculation of an image. However, scientists may wish to test other ray tracers that are incapable of this. To this end, the amount of threads the program generates can be set through the use of a constant in a C++ header file.

Various improvements can then be tested against the times achieved by this baseline program. This will ensure that the results from the study are comparable. Since only the relative increase in speed will be taken into account when reporting results, hardware variations should not contaminate the data.

This study restricts itself to examining the relative performance of CPU ray tracers for the empirical tests, therefore a GPU version of the benchmark program was not created.

In honour of William Gibson's widely known book “Neuromancer” the program was named “Straylight”.

In order to have a control experiment, some optimizations were added to Straylight that can be switched on and off with a command-line parameter. This “optimized” version will then also be tasked with generating the scenes in the test battery. Since the two programs are so close, their running times should be highly correlated. This serves as a “sanity check” to make sure the procedures in the experiment aren't causing errors.

Finally, the real experiment will be conducted by running POVRay on the test battery. POVRay was chosen because it is widely used for image generation and it seems to use a very different approach to a naïve ray tracer. POVRay is also very fast (the scenes only take a couple of seconds to render) – this will push the resolution for the technique to its limits.

A shell script will be used to power the data gathering process, it is shown in Listing 1.

```
#!/bin/sh

echo Starting test...

PREV_TIME=$TIME
export TIME="%e"

WIDTH=512
HEIGHT=512

rm straylight_naive.log straylight.log pov.log
rm straylight*.png
rm mount.png balls.png tetra.png tree.png rings.png gears.png

echo "FILE\t\tTIME" >> straylight_naive.log
echo "----\t\t----\n" >> straylight_naive.log

echo "FILE\t\tTIME" >> straylight.log
echo "----\t\t----\n" >> straylight.log

echo "FILE\t\tTIME" >> pov.log
echo "----\t\t----\n" >> pov.log

SCENES="mount balls tetra tree rings gears"

for s in $SCENES
do
	echo "Rendering $s using Straylight in naÃ¯ve mode..."
	echo -n "$s\t\t" >> straylight_naive.log
	time -a -o straylight_naive.log ../../../straylight -n -w $WIDTH -h $HEIGHT -f $s.nff -o
straylight_naive_$s.png
	echo "Finished $s."
done

for s in $SCENES
do
	echo "Rendering $s using Straylight in AABB mode..."
	echo -n "$s\t\t" >> straylight.log
	time -a -o straylight.log ../../../straylight -w $WIDTH -h $HEIGHT -f $s.nff -o straylight_$s.png
	echo "Finished $s."
done

for s in $SCENES
do
	echo "Rendering $s using POVRay..."
	echo -n "$s\t\t" >> pov.log
	time -a -o pov.log povray -W$WIDTH -H$HEIGHT -UV Version=3.1 Display=False POVRay/$s.pov 2>
/dev/null
	echo "Finished $s."
done

export TIME=$PREV_TIME

exit 0
```

Listing 1: Data gathering procedure.

CHAPTER 4: RESULTS

4.1 Introduction

The previous chapter discussed the data gathering method employed in this study. This chapter will discuss the results of that process. First, the raw data will be represented. A discussion will follow of the various methods used to extract meaning from this data, along with the findings from this analysis. Some source code will be provided to exactly specify the procedures used. Graphs will be used to represent the relationship between data sets in a visual manner.

This chapter draws heavily on a paper read at the IBIMA 15 conference in partial fulfilment of degree requirements (Kroeze *et al*., 2010a).

4.1 Caveats

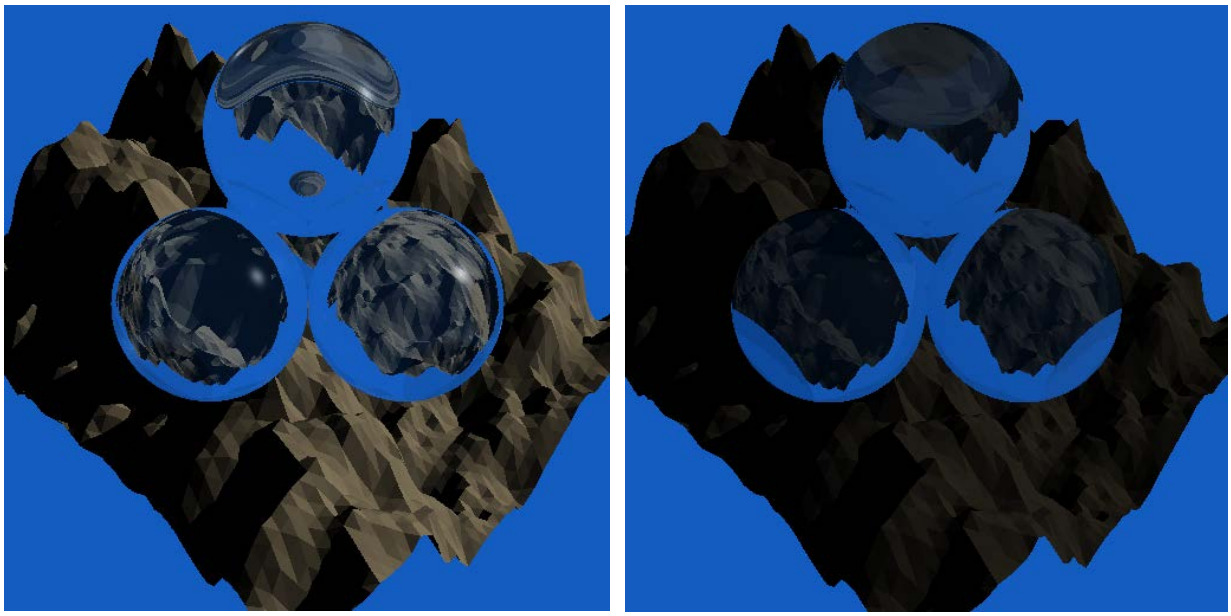

Figure 6: Discrepancies in Straylight render of the "mount" scene and POVRay render of the same scene.

The difference in brightness should be ignored - this is caused by different default values for light source brightness in the two programs.

Before discussing the data obtained in the data gathering step, a couple of caveats are in order. As discussed in the previous chapter, some bugs were left in the program, since the author could not fix them in time.

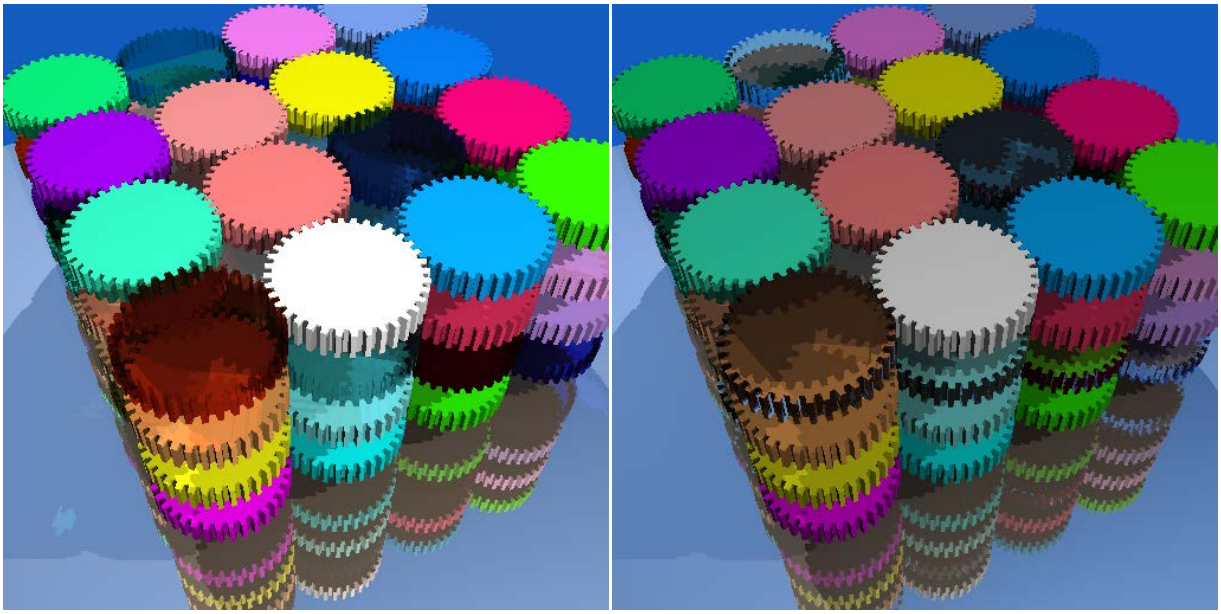

Figure 7: Discrepancies in Straylight render of the "gears" scene and POVRay render of the same scene.

The difference in brightness should be ignored - this is caused by different default values for light source brightness in the two programs.

These lead to small errors in the generated images and may affect the running time of individual scenes. It is doubtful, however, that these errors could cause much of a problem, since they affect a handful of pixels out of a big image only. In the interest of opennes, however, these discrepancies have been reported.

The author believes that these discrepancies is caused by the interaction of shadow rays and refractive (transparent) objects. Unfortunately, there was no time to rectify the problem.

The other scenes in the test battery render correctly.

4.2 Raw Data

Recall from the previous chapter that the benchmark program was executed for six different scenes on three different computers, along with two other programs – an optimized benchmark program which served as the control experiment, and POVRay that served as the actual experiment. The raw data from each of these runs is provided below.

Scene	Straylight Time	Optimized Time	POVRay Time
mount	4413.58	3276.52	1.55
balls	3558.23	3235.33	1.53
tetra	453.92	354.15	0.26
tree	4125.84	3280.51	0.84
rings	5325.36	4440.74	4.24
gears	7277.77	5376.92	11.91

Table 8: Execution times (in s) for each run on platform "Straylight".

Scene	Straylight Time	Optimized Time	POVRay Time
mount	11653.26	9114.84	2.93
balls	8536.61	8260.89	3.01
tetra	1132.5	917.28	0.48
tree	10018.26	9013.17	1.86
rings	13143.45	12018.99	8.36
gears	19666.15	14679.49	20.87

Table 9: Execution times (in s) for the "Neolith" platform.

Scene	Straylight Time	Optimized Time	POVRay Time
mount	9244.91	7772.33	3.07
balls	7114.89	6568.04	3.26
tetra	939.9	847.63	0.51
tree	8179.86	6678.17	1.91
rings	10093.67	9169.9	8.96
gears	16407.94	12860.03	21.52

Table 10: Execution times (in s) for platform "Monolith".

4.3 Data Analysis

The analysis performed on the data is relatively simple. For each scene in the test battery, the time it took to render using the optimized benchmark and the time it took

to render using POVRay is divided by the execution time for the benchmark itself. This ratio is multiplied by a hundred to give a percentage. This gives two columns of data for each test platform. Now, each of these columns should have a high correlation coefficient when compared with the corresponding column for a different test platform.

First, the ratios for the first step are provided:

Scene	Optimized Benchmark	POVRay
mount	90.93	0.042999
balls	78.02	0.057279
tetra	74.24	0.035119
tree	83.39	0.079619
rings	79.51	0.020359
gears	73.88	0.163649

Table 11: Percentages for "Straylight" platform.

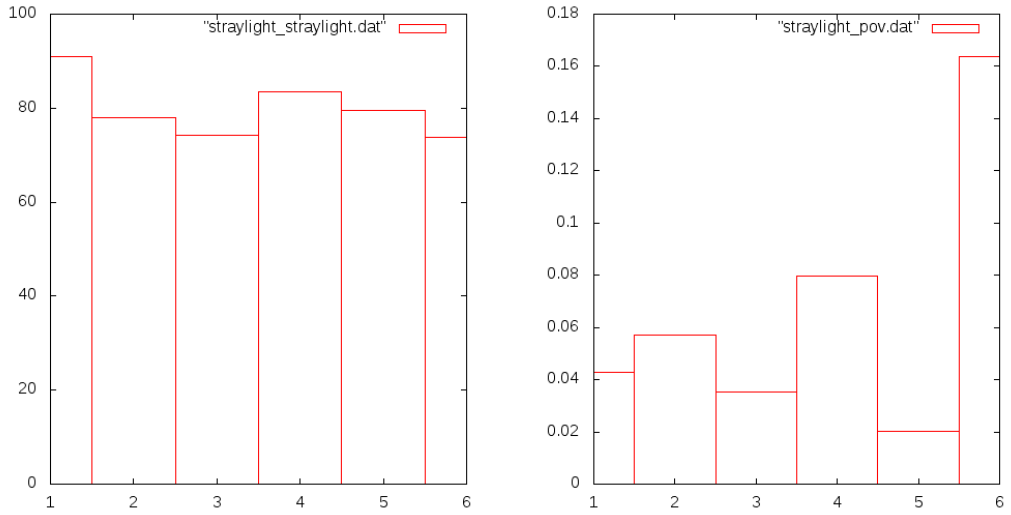

Figure 8: Percentage of benchmark time taken by (left) optimized benchmark and (right) POVRay on the "Straylight" platform.

Scene	Optimized Benchmark	POVRay
mount	96.770146	0.035260
balls	80.996026	0.042384
tetra	78.217083	0.025143
tree	91.444712	0.063606
rings	89.967419	0.018566
gears	74.643436	0.106121

Table 12: Percentages for "Neolith" platform.

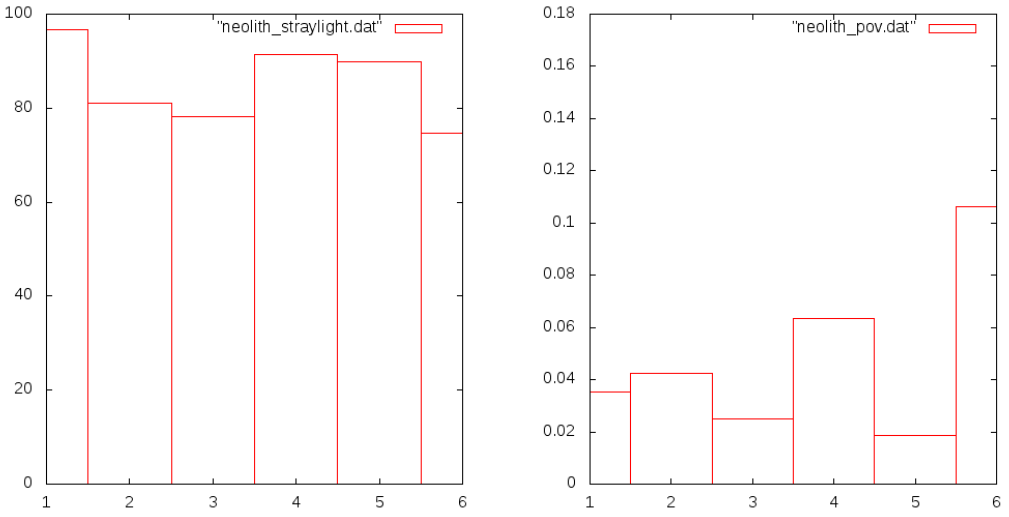

Figure 9: Percentage of benchmark time taken by (left) optimized benchmark and (right) POVRay on the “Neolith” platform.

Scene	Optimized Benchmark	POVRay
mount	92.314006	0.045819
balls	90.182998	0.054261
tetra	84.071451	0.033207
tree	90.848027	0.088769
rings	81.641617	0.023350
gears	78.376871	0.131156

Table 13: Percentages for "Monolith" platform.

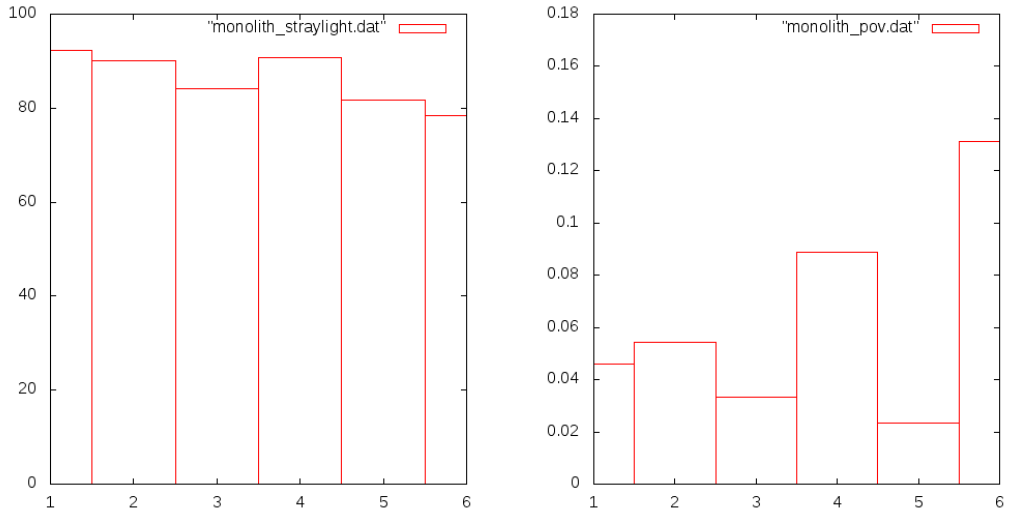

Figure 10: Percentage of benchmark time taken by (left) optimized benchmark and (right) POVRay on the “Monolith” platform.

Note that the graphs maintain the same basic shape across platforms. This is an encouraging sign, but is not statistically conclusive. In order to determine how similar the graphs are a tool such as the correlation coefficient (r^2) will be used. The previous processing step of dividing the running times and subsequently turning them into percentages serves to normalize the data, a necessary step for an r^2 analysis.

The correlation between the percentages for the “optimized benchmark” and “POVRay” between various platforms is the relevant figure. Choosing the “Straylight” platform as the baseline, the following correlation coefficients are found (rounded to the sixth decimal place):

	Neolith	**Monolith**
Optimized Benchmark	$r^2 = 0.869229$	$r^2 = 0.565845$
POVRay	$r^2 = 0.985712$	$r^2 = 0.958104$

Table 14: Correlation coefficients.

Over all, the correlation coefficients are relatively high, suggesting a strong relationship between the various percentage measurements on each of the platforms. The correlation coefficient for the “optimized benchmark” on the “Monolith” platform is rather low, however. Note that the second lowest correlation coefficient is also found for the “optimized benchmark” run on Neolith.

These runs took much longer to complete than the POVRay runs. Therefore, it is likely that errors in the process accumulate over time, causing the results for longer running tests to be skewed.

These errors may be caused by any of the variables that were not controlled for in the study. These variables are discussed in the previous chapter.

Luckily, the optimized benchmark test is not the sort of test that this technique will typically be used for in practice – the ray tracers that are commonly discussed in the

literature are typically much faster than this “optimized” benchmark. They are typically closer to the speed of POVRay, or even faster.

CHAPTER 5: CONCLUSIONS, LIMITATIONS AND FURTHER RESEARCH

5.1 Introduction

The previous chapter discussed the data analysis done during the study. This chapter will look at the conclusions that can be drawn from this analysis, highlight any limitations in the research and suggest avenues for future research.

5.2 Conclusion

As was discussed in the previous chapter, the technique described in this paper achieved a high correlation when measuring the performance of the same algorithm on different hardware platforms. While there were greater errors in the data set for the longer running tests, these are considered anamolous, since they do not represent reasonable run-times for real-world ray tracers.

Despite these minor problems, it should be pointed out that the correlation of raw speeds for two different algorithms on completely different hardware was not measured. It is likely that these figures would be extremely low and that the technique described in this study is a significant improvement.

As such, the author hopes that this technique will be improved and that it will be used in the future to make ray tracing performance measurements more meaningful.

For the reasons stated above, this paper comes to the conclusion that the use of a benchmarking program for comparing the relative performance of various ray tracing algorithms on diverse hardware is a feasible option. This approach may provide better results than comparing raw speeds that have not been adjusted for. In the latter case, confusion may arise due to the fact that the variability in hardware has not been accounted for.

This study also identified a number of factors that could impact the final results and attempted to eliminate all of them. While many of these factors were present in the final experiments, the study shows that their effect is minor – perhaps minor enough not to worry about. Importantly, the study effectively provides a way to deal with the greatest variable identified during the course of the literature review, which is the variation of hardware and measurement techniques in the existing research.

While this study left many variables uncontrolled and experienced low correlation coefficients in one case, its results are still encouraging. The proposed benchmark achieved strong coefficients on the other test which was much more realistic.

These uncontrolled variables and the small flaws in the benchmark program are actually encouraging – if this technique could achieve high correlation coefficients with these handicaps, then it may be even more robust than the correlation coefficients have suggested.

Unfortunately, this study must conclude that it is currently impossible to compare ray tracing engines with no statistical or environmental bias whatsoever. For reasons stated in the data gathering chapter, there are several problems that are too complicated to deal with appropriately. Computers have become incredibly powerful and complex in the past couple of decades, and this has lead to a great deal of doubt when comparing the performance of different programs. It is very difficult to eliminate CPU contention, since modern operating systems rely so much on multi-tasking. Similarly, caching affects most results, since it would be impossible to turn off without significantly changing kernel code. The effect of thermal throttling on results is also worrisome – it would be foolish to switch this feature off, since this could result in damage to very expensive equipment. It is also very difficult to determine whether the CPU's clock speed has been lowered because of thermal throttling, since no other programs should execute during the tests.

More research could possibly eliminate these problems of CPU contention, caching and temperature fluctuations. This study's contribution is instead a technique that does not seem overly sensitive to these problems.

5.3 Recommendations

It is the recommendation of this study that the technique introduced here be refined and studied in more detail until the scientific community judges it to be an accurate tool for data gathering and comparison for future studies involving ray tracing.

While there are some limitations (as described below) to the technique discussed in this report, it is felt that it is superior, or has the potential to be superior, to the *ad-hoc* methods currently in use.

5.4 Limitations

As mentioned earlier, there were several limitations to the study.

First, not all the variables could be controlled for. Reasons for the omission of each control are given in the data gathering section of this report.

Briefly, the study could not eliminate data variability arising from overheating, disk caching and disk I/O. An attempt to eliminate CPU contention was made, but could not be entirely successful.

If one of the test systems overheated during the course of the experiments, its motherboard could have lowered the clock speed of the CPU in order to save the latter from burning out. This would have artifically lowered the performance of the test system at some time in the experiment, changing the performance profile of the test on that machine.

Similarly, if the operating system performed more context switches on one platform than on another, that system’s performance could be artificially lowered. If this behaviour consistently affects performance for a platform, it should not affect the results, since the technique described in this report is specifically designed to eliminate differences between platforms. It would be problematic if this performance difference fluctuated during the experiment, since that would introduce some noise into the performance profile for a specific system.

The problems of disk caching and I/O lag are much less severe. The programs read very little data from the disk (only the small scene files) and write out only 512x512 images. Against the run time for the benchmark which was a couple of minutes in most cases, the effects of this should be negligable and can be ignored. In some scenes, POVRay's run-time is under a second, where the effects of these factors could perhaps be seen.

These problems were not expected to be overly problematic in practise, but the reader should still be advised to take these issues into account when reading this report. Because of the high correlation coefficients achieved, it does not seem that these issues affected the results in any meaningful way.

Second, as was mentioned in the data analysis chapter, the benchmarking program developed for use in this study did not render each image 100% correctly. Small errors were therefore present in the program during the tests. However, since the program produces the correct output in general, these errors should not have affected the tests greatly. Since a high correlation coefficient was achieved, it would appear that this assumption was correct.

Third, the study focused only on CPU ray tracers that produce static scenes. No assertions can be made about the validity of the study for GPU ray tracers or animated ray tracers. This is unfortunate, since a lot of the recent research on ray tracers focuses on GPU ray tracers and real-time rendering of animated scenes.

Still, the results from this study are a good place to start and these discrepancies represent opportunities for future research.

Fourth, the study focused on only one ray tracer for its comparisons. It is quite possible that other ray tracers will have a different strength of correlation than the one used here. Again, this represents an opportunity for future researchers to apply a similar technique to the one discussed here to their own raytracers.

Lastly, there is an issue of sample size with the study. Since the author did not have access to many different platforms to test this technique on, it cannot be said to be valid for a wide range of hardware. The results gathered during the study are very encouraging, however. It is hoped that others will undertake to test the technique themselves and add to the body of evidence supporting it, or refuting it.

This also implies that the range of hardware tested was quite limited – only three systems were used in total and all had very similar hardware. For example, it would be interesting to see the results of this technique on older hardware or more specialized computers such as servers. The study should have made use of a wider range of technologies, to see if the results apply to more outlandish setups, unfortunately, this was not possible since the author did not have access to such systems.

5.5 Further Research

The flaws in the study discussed above and in previous chapters present a wide range of opportunities for future research. Since the benchmark application is still very simple there are many features that can be added to it in order to enable the comparison of a wider range of ray tracers than the current selection.

For example, the benchmark program should be extended to support animated ray tracing, loading additional scene description file formats and execution on a GPU.

Adding support for animated scenes will allow comprehensive benchmarking of ray tracers that have this feature. Most recent ray tracers focus on animated scenes, hoping to eventually replace, or augment, the rasterization algorithm. For this reason, this feature will be a very significant and useful addition. However, the benchmark ray tracer will have to be significantly optimized. The program in its current form is much too slow to render animated scenes in a reasonable time. A relatively simple scene such as the ones in the SPD suite can take hours to render. An animated scene with many frames could take a week or more.

Supporting additional scene description file formats will allow the benchmark program to be useful for ray tracers that either don't use the NFF file format, or make use of a different test battery than the SPD suite used in this study. Since there is such great variety in the scenes used for testing by the ray tracing research community, this feature will allow interesting research on the benchmarking of a wide variety of ray tracers.

Modifying the program to execute on the GPU will have a similar effect to the suggestions above. This feature will allow researchers to gauge whether GPU ray tracers can be benchmarked in the same way that CPU ray tracers can. Since the GPU's execution model and architecture is so different from the CPU, the answer to that question may not be obvious.

While the benchmark program was deliberately created to be a baseline ray tracer with very embellishments to obscure the results, it would be interesting to see the predictive capabilities of a more advanced or outlandish ray tracing algorithm.

The addition of more features will also make the benchmarking program comparable with a wider variety of real-world ray tracers. The benchmarking program's ability to seperate the performance of other ray tracers from the hardware on which they execute will be a good test for the conclusion of this study.

Given the study's small sample size and low hardware variability, it would also be benificial to continue the study on a multitude of different systems. Because the experiments supporting this study were only run on three different machines, there is some doubt about whether the results can be extended to the proliferation of different hardware setups that can be found in use today. Extending the study to as many different hardware configurations as possible will be a great way to test the conclusion discussed in this report. It is left up to the scientific community to judge the quality of the research conducted in this study and decide whether there is any possible value in continuing the evaluation of its central premise.

Since this study is focused on providing the capability to fairly compare multiple ray tracers, it would be interesting to see it used to compare some of the older and more inefficient approaches with contemporary ones. Such a study could produce a very clear picture of how far ray tracing has come in the past decades in terms of performance. It would also be interesting to see how much of the advance has been due to the increase in hardware capabilities since the inception of the algorithm to its modern day form. Since this study focuses on eliminating hardware as a variable, this should be easy to do in practise. The results of such a study will be eye opening – if hardware is more of a factor than previously thought, then researchers should be focusing on developing new hardware, or on GPU ray tracers. If it is less of a factor, then more thought should go into the algorithm itself.

Some of the variables identified in the data gathering chapter were not eliminated from the experiments conducted for this study. Future research could determine how to eliminate the effects of multitasking, caching and temperature fluctuations from experiments dealing specifically with ray tracers. Addressing these issues will do much for the scientific rigour of ray tracing experiments, where these concerns are seldom mentioned.

5.6 Final Notes

In the interest of scientific openness, all the material used for this study are available via anonymous FTP at ftp://philosoraptor.co.za/. Readers are welcome to browse the benchmark program's source code to check for any bugs that may have caused spurious results. All the data collected for the study is available for perusal in both raw and processed forms. The spreadsheets used for calculating the correlation coefficients have also been uploaded.

APPENDICES

6.1 Appendix A

This appendix shows Whitted's derivation of the reflected and refracted ray equations. Note that in the derivation below, $\vec{V}$ is not assumed to be a unit vector, but $\vec{N}$ is (Heckbert, 1989). The derivation does not guarantee that any vectors produced will be unit vectors.

This appendix is based heavily on work done by Heckbert (1989).

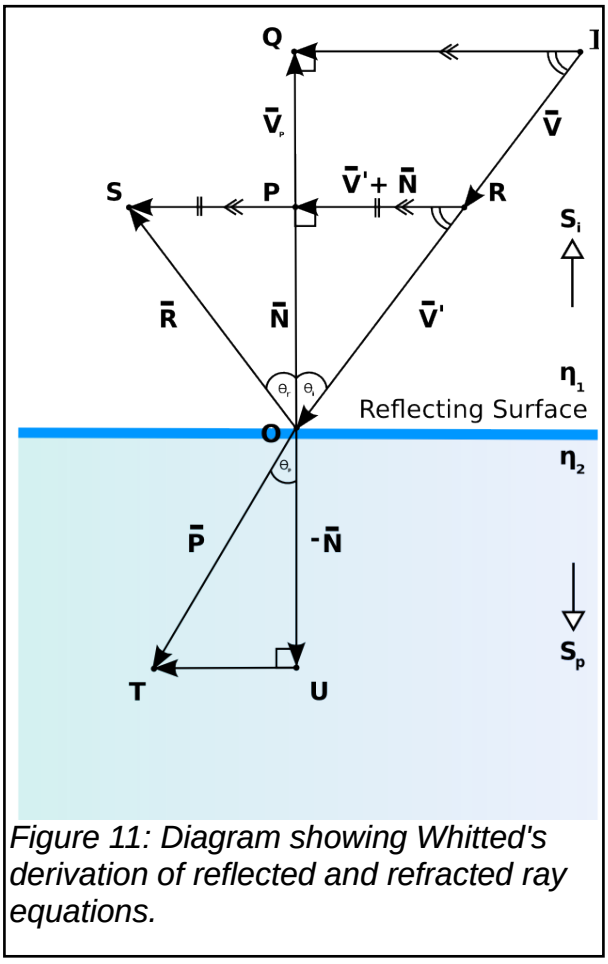

Figure 11: Diagram showing Whitted's derivation of reflected and refracted ray equations.

Statements	Reasons
Given: IO is a ray incoming from the viewer / camera that is incident on a reflecting surface. The direction and magnitude of the ray IO is given by	

$\vec{IO} = \vec{V}$.	
Given: OP is the surface normal of the reflecting surface. The direction and magnitude of the ray OP is given by $\vec{OP} = \vec{N}$, $\left\|\vec{N}\right\| = 1$.	
Given: The angle between IO and OQ is θ_i, the angle of incidence.	
Project $\vec{V}$ onto $\vec{N}$ orthogonally, and label this vector $\vec{V}_p = \vec{OQ}$	
$\vec{V}_p = \vec{N}\frac{\vec{V}\cdot\vec{N}}{\vec{N}\cdot\vec{N}}$	Orthographic projection
$\vec{V}_p = N\left(\vec{V}\cdot\vec{N}\right)$	$\vec{N}\cdot\vec{N} = \left\|\vec{N}\right\|^2$; $\left\|\vec{N}\right\| = 1$
Construct $\vec{V'}$ so that its projection onto $\vec{N} = \vec{N}$ and that $\vec{V'}$ and $\vec{V}$ are co-linear. $Let \vec{V'} = RO$.	
$RP \perp OQ$	Orthographic projection
Similarly, $IQ \perp OQ$	
$QI \parallel PR$	Both QI and PR are $\perp OQ$
For triangles OPR, OQI	
$\hat{O} = \hat{O}$	Tautology
$O\hat{P}R = O\hat{Q}I = 90°$	Proven
$O\hat{R}P = O\hat{I}Q$	$QI \parallel PR$, corresponding angles
$\therefore$ triangles OPR, OQI are similar.	
$\therefore \frac{OR}{OI} = \frac{OP}{OQ}$	
$\therefore \frac{\left\|-\vec{V'}\right\|}{\left\|-\vec{V}\right\|} = \frac{\left\|\vec{N}\right\|}{\left\|\vec{V}_p\right\|}$	
$\frac{\left\|\vec{V'}\right\|}{\left\|\vec{V}\right\|} = \frac{\left\|\vec{N}\right\|}{\left\|\vec{N}\left(\vec{V}\cdot\vec{N}\right)\right\|}$	
$\frac{\left\|\vec{V'}\right\|}{\left\|\vec{V}\right\|} = \frac{\left\|\vec{N}\right\|}{\left\|\vec{N}\right\|\left\|\vec{V}\cdot\vec{N}\right\|}$	See Appendix C, theorem 1.
$\frac{\left\|\vec{V'}\right\|}{\left\|\vec{V}\right\|} = \frac{1}{\left\|\vec{V}\cdot\vec{N}\right\|}$	
$\left\|\vec{V'}\right\| = \frac{\left\|\vec{V}\right\|}{\left\|\vec{V}\cdot\vec{N}\right\|}$	

$\therefore \vec{V'} = \frac{\vec{V}}{\lvert\vec{V}\cdot\vec{N}\rvert}$	Since $\vec{V'}$ has the same direction as $\vec{V}$.
$\therefore \vec{RP} = \vec{V'}+\vec{N}$	Vector addition
Construct triangle OPS such that: • RP & PS are co-linear • $P\hat{O}S = P\hat{O}R$	
$\tan\left(P\hat{O}R\right) = \tan\left(P\hat{O}S\right)$	$P\hat{O}S = P\hat{O}R$
$\therefore \frac{RP}{OP} = \frac{PS}{OP}$	
$\therefore RP = PS$	
Since $\vec{PS}$ has the same direction as $\vec{RP}$: $\vec{PS} = \vec{RP} = \vec{V'}+\vec{N}$	Definition of a vector as direction and magnitude.
$\vec{OS} = \vec{OP} + \vec{PS}$	
$\vec{OS} = \vec{N} + \vec{V'}+\vec{N}$	
$\vec{OS} = \vec{V'}+2\vec{N}$	
$P\hat{O}S$ is the reflection angle θ_r	$P\hat{O}R = \theta_i$, $P\hat{O}S = P\hat{O}R$
$\therefore \vec{OS}$ is the reflection vector $\vec{R}$	
$\therefore \vec{R} = \vec{OS} = \vec{V'}+2\vec{N}$	
Construct triangle OUT such that: • OU is an extension of line QO represented by $-\vec{N}$. • $TU \perp OU$ • $T\left(\hat{O}\right)U = \theta_p$, where θ_p is the angle of refraction.	
$sec\theta_i = \left\lVert\vec{V'}\right\rVert$	Trigonometry; $\lVert N\rVert = 1$
Let $k_f = \frac{\tan\theta_p}{\tan\theta_i}$	
$k_f = \frac{\sin\theta_p}{\sin\theta_i}\times\frac{\cos\theta_i}{\cos\theta_p}$	$\tan\alpha = \frac{\sin\alpha}{\cos\alpha}$
$k_f = \frac{\sin\theta_p}{\sin\theta_i}\times\frac{\cos\theta_i}{\pm\sqrt{1-\sin^2\theta_p}}$	$\cos\alpha = \sqrt{1-\sin^2\alpha}$

$$k_f = \frac{\eta_1}{\eta_2} \times \frac{\cos\theta_i}{\pm\sqrt{1-\sin^2\theta_p}}$$

$$k_f = \frac{(\eta_1/\eta_2)\cos\theta_i}{\pm\sqrt{1-\left(\frac{\eta_1}{\eta_2}\right)^2 \sin^2\theta_i}}$$

$$k_f = \frac{1}{\pm\sqrt{\left(\frac{\eta_2}{\eta_1}\right)^2 sec^2\theta_i\left(1-\left(\frac{\eta_1}{\eta_2}\right)^2 \sin^2\theta_i\right)}}$$

$$k_f = \frac{1}{\pm\sqrt{\left(\frac{n_2}{n_1}\right)^2 sec^2\theta_i - \sin^2\theta_i sec^2\theta_i}}$$

$$k_f = \frac{1}{\pm\sqrt{\left(\frac{n_2}{n_1}\right)^2 sec^2\theta_i - \tan^2\theta_i}}$$

$$k_f = \frac{1}{\pm\sqrt{\left(\frac{n_2}{n_1}\right)^2 \left\|\vec{V}'\right\|^2 - \left\|\vec{V}'+\vec{N}\right\|^2}}$$

$$k_f = \pm\left[k_n^2 \left\|\vec{V}'\right\|^2 - \left\|\vec{V}'+\vec{N}\right\|^2\right]^{-1/2}$$

where $k_n = \frac{\eta_2}{\eta_1}$

$$\tan\theta_p = UT$$

$$\tan\theta_i = RP$$

$$k_f = \frac{\tan\theta_p}{\tan\theta_i} = \frac{UT}{RP}$$

$$\therefore UT = k_f \times RP$$

$$\therefore \vec{UT} = k_f \times \vec{RP}$$

$$\therefore \vec{UT} = k_f\left(\vec{V}'+\vec{N}\right)$$

$$\eta_1\sin\theta_i = \eta_2\sin\theta_p\left(Snell's\ law\right)$$

$$\therefore \frac{\eta_1}{\eta_2} = \frac{\sin\theta_p}{\sin\theta_i}$$

$$\frac{\eta_1}{\eta_2} = \frac{\sin\theta_p}{\sin\theta_i} \quad (See\ above)$$

$$\therefore \left(\frac{\eta_1}{\eta_2}\right)^2 \sin^2\theta_i = \sin^2\theta_p$$

$$\cos\theta = \frac{1}{sec}\theta$$

$$\sin^2\alpha sec^2\alpha = \tan^2\alpha$$

$$sec\theta_i = \left\|\vec{V}'\right\|(Proven)$$

$$\tan\theta_i = \left\|\vec{V}'+\vec{N}\right\|(Proven)$$

$$\tan\theta p = \frac{UT}{OU}, OU = \left\|-\vec{N}\right\| = 1$$

$$\tan\theta_i = \frac{RP}{OP}, OP = \left\|\vec{N}\right\| = 1$$

$$\tan\theta_i = \left\|\vec{V}'+\vec{N}\right\|(Proven)$$

$$UT \square RP\left(TU \perp QU, RP \perp QU\right)$$

$\vec{UT}, \vec{RP}$ have the same direction

$$\vec{RP} = \vec{V}'+\vec{N}$$

$\vec{OU} = -\vec{N}$ $\vec{OT} = \vec{P} = \vec{OU} + \vec{UT}$	Given. Vector addition
$\therefore \vec{P} = k_f\left(\vec{V}' + \vec{N}\right) - \vec{N}$	

6.2 Appendix B

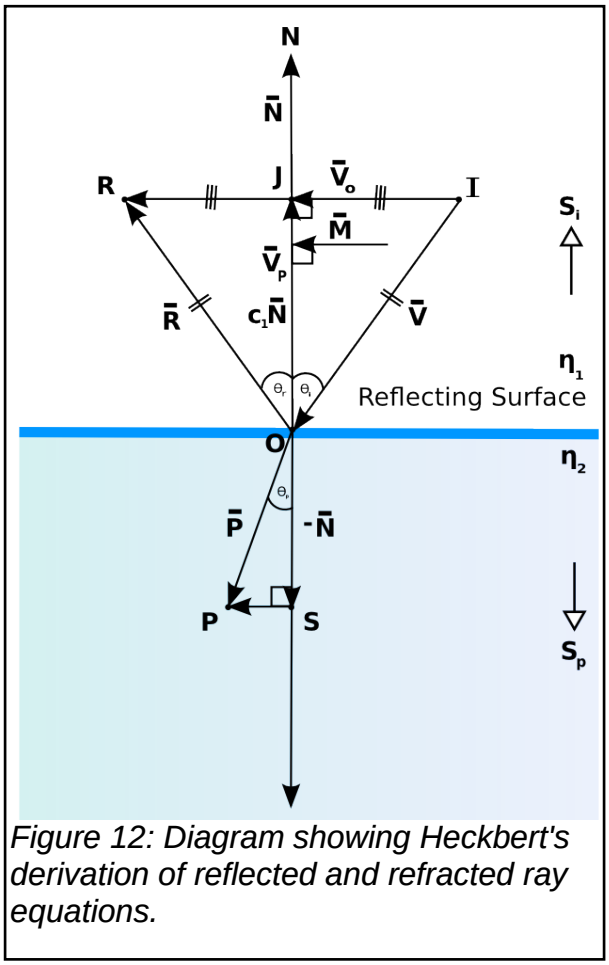

Figure 12: Diagram showing Heckbert's derivation of reflected and refracted ray equations.

This appendix shows the Heckbert derivation of $\vec{R}$ and $\vec{P}$. Note that Heckbert's derivation deviates from Whitted's. Heckbert assumes that both $\vec{N}$ and $\vec{V}$ are unit (Heckbert, 1989). In addition, the derivation guarantees that both $\vec{R}$ and $\vec{P}$ are unit vectors (Heckbert, 1989). The steps of the derivation shown here are likely different from Heckbert's even though they are based on his work (Heckbert, 1989), but they serve as proof of his equations anyway.

Statements	Reasons
Given: IO is a ray incoming from the viewer / camera that is incident on a reflecting surface. The direction and magnitude of the ray IO is given by $\vec{IO} = \vec{V}$, $\left\|\vec{V}\right\| = 1$.	

Given: ON is the surface normal of the reflecting surface. The direction and magnitude of the ray ON is given by $\vec{ON} = \vec{N}$, $\left\|\vec{N}\right\| = 1$. Given: The angle between IO and ON is θ_i, the angle of incidence. Construct $IJ \perp ON$ Let $\vec{OJ} = c_1\vec{N}$ $\cos\theta_i = \dfrac{\left\|c_1\vec{N}\right\|}{\left\|\vec{V}\right\|}$ $\cos\theta_i = \dfrac{c_1\left\|\vec{N}\right\|}{\left\|\vec{V}\right\|}$ $\therefore c_1 = \dfrac{\cos\theta_i\left\|\vec{V}\right\|}{\left\|\vec{N}\right\|}$	J is on ON, $\therefore OJ$ is a scaled version of ON. $O\hat{J}I = 90°$
$\therefore c_1 = \cos\theta_i$	$\left\|\vec{N}\right\| = \left\|\vec{V}\right\| = 1$
Decompose $\vec{V}$ into two components: one parallel to the surface normal ($\vec{V}_p = \vec{OJ}$), one orthogonal to it ($\vec{V}_o = \vec{IJ}$). Construct triangle OJR such that: • $J\hat{O}R = J\hat{O}I$. • IJR is the extension of IJ. $\vec{OR} = \vec{R}$ For triangles OJR and OJI: • $J\hat{O}R = J\hat{O}I$ • OJ is a communal side. • $O\hat{J}R = 90° = O\hat{J}I$ $\therefore OJR \equiv OJI$ And $J\hat{O}R = J\hat{O}I = \theta_i = \theta_r$ $\therefore JR = IJ$ $IJ = \left\|\vec{IJ}\right\| = \left\|\vec{V} + c_1\vec{N}\right\|$ $\therefore \vec{JR} = \vec{V} + c_1\vec{N}$	$J\hat{O}I = \theta_i = J\hat{O}R = \theta_r$ $\vec{OR}$ is the reflection vector Construction. $IJ \perp ON \therefore RJ \perp ON$ Angle-side-angle. • Construction. • Angle of incidence equals angle of reflection. $OJR \equiv OJI$, sides opposite equal triangles. Vector addition.
$\therefore \vec{R} = \vec{V} + 2c_1\vec{N}$	Vector addition.

Furthermore, $\left\|\vec{R}\right\| = \left\|\vec{V}\right\| = 1$	$OJR \equiv OJI$, $\left\|\vec{V}\right\| = 1$
Construct triangle OPS such that: • OP is of unit length. • $P\hat{S}O = 90°$ • $P\hat{O}S = \theta_p$	
$OP = \left\|\vec{P}\right\|$	The angle between OP and the extension of the surface normal OS is θ_p . $\therefore \vec{OP}$ is the the direction of refraction.
$\cos\theta_p = \frac{OS}{OP}$ $\cos\theta_p = \frac{OS}{\left\|\vec{P}\right\|}$ $\cos\theta_p = \frac{OS}{1}$ $\cos\theta_p = OS$ $\therefore OS = \cos\theta_p$	$\left\|\vec{P}\right\| = 1$
$\sin\theta_p = \frac{SP}{OP}$ $\sin\theta_p = \frac{SP}{\left\|\vec{P}\right\|}$ $\cos\theta_p = \frac{SP}{1}$ $\cos\theta_p = SP$ $\therefore SP = \sin\theta_p$	$\left\|\vec{P}\right\| = 1$
Let $\vec{M}$ be a unit surface tangent vector with the same direction as $\vec{V}_o$.	$\vec{V}_o$ is a tangent to the surface since it is $\perp \vec{N}$.
$\therefore \vec{M} = \frac{\vec{V}_o}{\left\|\vec{V}_o\right\|}$	Definition of unit vector.
$\therefore \vec{M} = \frac{\vec{V} - c_1\vec{N}}{\sin\theta_i}$	1. $\vec{V}_o = \vec{V} - c_1\vec{N}$ (Proven) 2. $\left\|\vec{I}_o\right\| = \sin\theta_i$ (Proven)
Decompose $\vec{P}$ into two components: one parallel to the surface normal ($\vec{P}_p = \vec{OS}$), one orthogonal to it ($\vec{P}_o = \vec{SP}$).	Since OPS is a right angled triangle, the components of OP (which corresponds to $\vec{P}$) are the other sides of the triangle. The components of the vectors that correspond to these sides have the same property.
$\vec{OS} = x\left(-\vec{N}\right)$	The refracted ray must pass through the surface, therefore it has a direction opposite to the surface normal. Therefore $\vec{OS}$ is a scaled version of $-\vec{N}$.

$\vec{OS} = \cos\theta_p\left(-\vec{N}\right)$	• $\left\|-\vec{N}\right\| = 1$ • $\left\|\vec{OS}\right\| = OS = \cos\theta_p$
$\vec{OS} = -\cos\theta_p\vec{N}$	Distributive property of multiplication.
$\vec{SP} = y\vec{M}$	$\vec{SP} \parallel \vec{M}$ and they have the same direction.
$\vec{SP} = \sin\theta_p\vec{M}$	• $\left\|\vec{M}\right\| = 1$ • $\left\|\vec{SP}\right\| = SP = \sin\theta_p$
$\vec{P} = \vec{OS} + \vec{SP}$	Decomposition (vector addition).
$\vec{P} = -\cos\theta_p\vec{N} + \sin\theta_p\vec{M}$	
$\vec{P} = -\cos\theta_p\vec{N} + \sin\theta_p\vec{M}$	
$\vec{P} = \sin\theta_p\vec{M} - \cos\theta_p\vec{N}$	
$\vec{P} = \sin\theta_p\dfrac{\left(\vec{V} + c_1\vec{N}\right)}{\sin\theta_i} - \cos\theta_p\vec{N}$	
$\vec{P} = \dfrac{\sin\theta_p}{\sin\theta_i}\left(\vec{V} + c_1\vec{N}\right) - \cos\theta_p\vec{N}$	
But $\eta_1\sin\theta_i = \eta_2\sin\theta_p$	Snell's law.
$\therefore \dfrac{\sin\theta_p}{\sin\theta_i} = \dfrac{\eta_1}{\eta_2}$	
Let $\eta = \dfrac{\sin\theta_p}{\sin\theta_i} = \dfrac{\eta_1}{\eta_2}$	
$\therefore \vec{P} = \eta\left(\vec{V} + c_1\vec{N}\right) - \cos\theta_p\vec{N}$	
$\therefore \vec{P} = \eta\vec{V} + \eta c_1\vec{N} - \cos\theta_p\vec{N}$	
Let $c_2 = \cos\theta p$	
$c_2 = \sqrt{1-\sin^2\theta_p}$	
$c_2 = \sqrt{1-\eta^2\sin^2\theta_i}$	
$c_2 = \sqrt{1-\eta^2\left(1-c_1^2\right)}$	$c_1 = \cos\theta_i$ (Proven).
$\therefore \vec{P} = \eta\vec{V} + \eta c_1\vec{N} - c_2\vec{N}$	
$\therefore \vec{P} = \eta\vec{V} + \left(\eta c_1 - c_2\right)\vec{N}$	

Heckbert also proposes an alternative to this derivation, by replacing η with $\frac{1}{n}$, where $n = \frac{\eta_2}{\eta_1}$, we can substitute in the previous equation:

$$\therefore \vec{P} = \eta\vec{V} + \left(\eta c_1 - \sqrt{1-\eta^2\left(1-c_1^2\right)}\right)\vec{N}$$

$$\therefore \vec{P} = \frac{\vec{V}}{n} + \frac{c_1 - n\sqrt{1-\frac{1-c_1^2}{n^2}}}{n}\vec{N}$$

$$\therefore \vec{P} = \frac{\vec{V}}{n} + \frac{c_1 - \sqrt{n^2-\left(1-c_1^2\right)}}{n}\vec{N}$$

$$\therefore \vec{P} = \frac{\vec{V}}{n} + \frac{c_1 - \sqrt{n^2-1+c_1^2}}{n}\vec{N}$$

$$\therefore \vec{P} = \frac{\vec{V}}{n} + \frac{\left(c_1 - \sqrt{n^2-1+c_1^2}\right)\vec{N}}{n}$$

And finally, $\therefore \vec{P} = \frac{\vec{V} + \left(c_1 - \sqrt{n^2-1+c_1^2}\right)\vec{N}}{n}$ where $n = \frac{\eta_2}{\eta_1}$

6.3 Appendix C

6.3.1 Theorem 1

Let $\vec{V} = (v_1, v_2, v_3)$ be a vector and r be some scalar.

$r\vec{V} = (rv_1, rv_2, rv_3)$	Definition of scalar-vector multiplication.
$\|r\vec{V}\|$ is the magnitude of $r\vec{V}$	
$\therefore \|r\vec{V}\| = \left\|\sqrt{(rv_1)^2 + (rv_2)^2 + (rv_3)^2}\right\|$	Definition of vector magnitude. The absolute is added over the square root, since magnitude must be positive in terms of the definition of vectors.
$\therefore \|r\vec{V}\| = \left\|\sqrt{r^2v_1^2 + r^2v_2^2 + r^2v_3^2}\right\|$	
$\therefore \|r\vec{V}\| = \left\|\sqrt{r^2(v_1^2 + v_2^2 + v_3^2)}\right\|$	
$\therefore \|r\vec{V}\| = \left\|r\sqrt{v_1^2 + v_2^2 + v_3^2}\right\|$	$\sqrt{x^2y} = x\sqrt{y}$
$\therefore \|r\vec{V}\| = \|r\| \quad \left\|\sqrt{v_1^2 + v_2^2 + v_3^2}\right\|$	
$\therefore \|r\vec{V}\| = \|r\| \quad \|\vec{V}\|$	

$\therefore \|r\vec{V}\| = \|r\| \quad \|\vec{V}\|$

BIBLIOGRAPHY

AILA, T & LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. (*In* Spencer, S.N., McAllister, D., Pharr, M. & Wald, I., *eds*. Proceedings of the Conference on High Performance Graphics 2009. New York, NY: ACM. p. 145-149.)

APPEL, A. 1968. Some techniques for shading machine renderings of solids. (*In* American Federation of Information Processing Societies. Proceedings of the April 30-May 2, 1968, spring joint computer conference. Washington, D.C.: Thompson Book Company. p. 37-45.)

BENTLEY, J.L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509-517.

BUCK, I., FOLEY, T., HORN, D. SUGERMAN, J., FATAHALIAN, K., HOUSTON, M. & HANRAHAN, P. 2004. (*In* Marks, J., *ed*. ACM SIGGRAPH 2004 Papers. New York: ACM. p. 786.)

CARR, N.A., HALL, J.D. & HART, J.C. 2002. The ray engine. (*In* Ertl, T., Heidrich, W., Doggert, M. *eds*. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. Eurographics Association. p. 37-46).

CARR, N.A., HOBEROCK, J., CRANE, K. & HART, J.C. 2006. Fast GPU ray tracing of dynamic meshes using geometry images. (*In* Mann, S. & Gutwin, C., *eds.* Proceedings of Graphics Interface 2006.Mississauga, Canada: Canadian Information Processing Society. p. 203-209.)

CHEN, C.C. & LIU, D.S.M. 2007. Use of hardware z-buffered rasterization to accelerate ray tracing. (*In* Association for Computing Machinery. Proceedings of The 2007 ACM Symposium on Applied Computing. New York, NY. p. 1046-1050.)

CHRISTENSEN, P., FONG, J., LAUR, D.M. & BATALI, D. 2006. Ray tracing for the movie “Cars”. (*In* Wald, I. & Parker, S.G., *eds*. Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing. Los Alamitos, CA: IEEE Computer Society. p. 1-6.)

FOLEY, T. & SUGERMAN, J. 2005. Kd-tree acceleration structures for a GPU raytracer. (*In* Meissner, M. & Schneider, B.-O., *eds.* Proceedings of the ACM

SIGGRAPH / EUROGRAPHICS Conference on Graphics Hardware. New York: ACM. p. 15-22.)

FUSSEL, D. & SUBRAMANIAN, K.R. 1988. Fast ray tracing using k-d trees. Technical Report No. TR-88-07. Department of Computer Sciences, University of Texas, Austin. March 1998. p. 1-21.

GEORGIEV, I. & SLUSALLEK. 2008. P. RTfact: generic concepts for flexible and high performance ray tracing. (*In* IEEE Computer Society. Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing. Washington, D.C. p. 115-122.)

HAINES, E. 1987. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, 7(11):3-5, November.

HAVRAN, V., HERZOG, R. & SEIDEL, H.P. 2006. On the fast construction of spatial hierarchies for ray tracing. (*In* Wald, I. & Parker, S.G., *eds*. Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing. Los Alamitos, CA: IEEE Computer Society. p. 71-80)

HECKBERT, P.S. 1989. Derivation of Refraction Formulas. (*In* Glassner, A, *ed*. Introduction to Ray Tracing. London: Academic Press. p. 263-293.)

HORN, D.R., SUGERMAN, J., HOUSTON, M. & HANRAHAN, P. 2007. Interactive k-D GPU raytracing. (*In* Association for Computing Machinery. Proceedings of the 2007 Symposium on Interactive Ray Tracing. New York, NY. p. 167-174)

HUANG, P., WANG, W., YANG, G. & WU, E. 2006. Traversal fields for ray tracing dynamic scenes. (*In* Slater, M. & Tal, A., *eds*. Proceedings of The ACM Symposium On Virtual Reality Software And Technology. New York, NY: ACM. p. 65-74.)

KALOJANOV, J. & SLUSALLEK, P. 2009. A parallel algorithm for construction of uniform grids. (*In* Spencer, S.N., McAllister, D., Pharr, M. & Wald, I., *eds*. Proceedings of the Conference on High Performance Graphics 2009. New York, NY: ACM. p. 23-28.)

KROEZE, J.C.W., JORDAAN, D.B. & PRETORIUS, P.D. 2010a. Benchmarking for ray tracing performance measurement (full paper). *Proceedings of the 15th International Business Information Management Association Conference (15th IBIMA),* 6 - 7 November 2010, Cairo, Egypt, pp. 427-437. (Knowledge Management

and Innovation: A Business Competitive Edge Perspective, edited by Khalid S. Soliman. On CD: ISBN: 978-0-9821489-4-5).

KROEZE, J.C.W., JORDAAN, D.B. & PRETORIUS, P.D. 2010b. The advent of GPU ray tracers (full paper). *Proceedings of the 15th International Business Information Management Association Conference (15th IBIMA),* 6 - 7 November 2010, Cairo, Egypt, pp. 438-439. (Knowledge Management and Innovation: A Business Competitive Edge Perspective, edited by Khalid S. Soliman. On CD: ISBN: 978-0-9821489-4-5).

LEXT, J., ASSARSSON, U., MÖLLER, T. 2001. A benchmark for animated ray tracing. *IEEE Computer Graphics and Applications*, 21(2):22-31, March.

OVERBECK, R., RAMAMOORTHI, R. & MARK, W.R. 2008. Large ray packets for real-time Whitted ray tracing. (*In* IEEE Computer Society. Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing. Washington, D.C. p. 41-48.)

POPOV, S., GÜNTHER, J., SEIDEL, H.P. & SLUSALLEK, P. 2007. Stackless KD-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum,* 26(3):415-424, September.

PURCELL, T.J., BUCK, I., MARK, W.R. & HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):268-277, July.

SUBRAMANIAN, K.R. & FUSSEL, D.S. 1990a. Factors affecting performance of ray tracing hierarchies. Technical Report No. TR-90-21. Department of Computer Sciences, University of Texas. Austin. July 1990. p. 1-27.

SUBRAMANIAN, K.R. & FUSSEL, D.S. 1990b. Applying space subdivision techniques to volume rendering. (*In* IEEE Computer Society. VIS '90: Proceedings of the 1st conference on Visualization '90. San Francisco, California. p. 150-159.)

SUBRAMANIAN, K.R. & FUSSEL, D.S. 1991. Automatic termination criteria for ray tracing hierarchies. (*In* Canadian Information Processing Society. Proceedings of Graphics Interface ’91. Toronto. p. 93-100.)

WÄCHTER, C. & KELLER, A. 2006. Instant ray tracing: the bounding interval hierarchy. (*In* Akenine-Möller, T. & Heidrich, W., *eds.* 2006. Eurographics Symposium / Workshop on Rendering. Nicosia, Cyprus. Eurographics Association.)

WALD, I. & SLUSALLEK, P. 2001. State of the art in interactive ray tracing. (*In* Duke, D. & Scopingo, R., *eds.* STAR Proceedings of Eurographics 2001. Budapest, Hungary: Eurographics Association.)

WALD, I., MARK, W.R., GÜNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S.G. & SHIRLEY, P. 2007. State of the art in ray tracing animated scenes. (*In* Schmalsteig, D. & Bittner, J., *eds.* STAR Proceedings of Eurographics 2007. Budapest, Hungary: Eurographics Association. p. 89–116)

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343-349, June.

WOOP, S., MARMIT, G., SLUSALLEK, P. 2006. B-KD trees for hardware accelerated ray tracing of dynamic scenes. (*In* Association of Computing Machinery. Proceedings of The 21st ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware. New York, NY. p. 67-77)

ZHOU, K., HOU, Q., WANG, R. & GUO, B. 2008. Real-time KD-tree construction on graphics hardware. (*In* Hart, J.C., *ed*. ACM SIGGRAPH Asia 2008 Papers. New York, NY: ACM. Article #126.)