UNIVERSITY of LIMERICK

OLLSCOIL LUIMNIGH

FINAL YEAR
PROJECT REPORT
2021

Programming Interface for Distribution of Ray Tracing Over Multiple Machines

Student Name:	Adam Aherne
Student ID Number:	12159603
Supervisor:	Patrick Healy
Date:	28/04/2021

CSIS Department of Computer Science & Information Systems

NOTE:

This report contains images produced as part of the Final Year Project. Converting the report to PDF format severely reduces the quality of these images. If viewing this report as a PDF, please bear in mind that the true quality of these images is not fairly represented.

Contents

Table of Figures

Section 1: Introduction

In this section I give a brief overview and summary of my final year project (FYP), covering the most important aspects. I also detail my motivations in choosing the project and provide an overview of the layout and structure of this report.

1.1: Project Overview

The goal of this project is to develop a simple library for the Julia programming language that allows users to create a simple, static, virtual scene. The library should allow the user to render this scene via an implementation of the ray tracing rendering technique, with the option of distributing the computation over multiple machines.

The library I aim to develop should be capable of performing a number of tasks. A general overview of the intended capabilities is given here, but the specific requirements are more precisely detailed in Section 3.

Users of the library should be able to create simple virtual scenes by importing the library into their program. The library should support, at least, the following actions:

- setting the size and colour of the scene environment;
- adding simple three-dimensional (3D) shapes to the scene;
- specifying the size and colours of added shapes;
- adding a light source to the scene;
- specifying the camera position, and viewing direction;
- specifying the viewing plane size and field of vision (FOV);
- rendering the scene locally, in a serial or concurrent manner;
- rendering the scene in a distributed manner, using multiple machines..

The library should be an abstraction of the ray tracing technique. Implementation details of the library should be hidden from users. Users need not be concerned about details such as the vector mathematics used to perform ray tracing, or the work allocation strategy for distributing computation. The library should provide simple datatypes and functions for users to create and render their scenes.

1.2: Motivation

My motivation for choosing this FYP topic can be divided into two areas: problem domains and programming language.

1.2.1: Problem Domains

My FYP topic is concerned with two primary problem domains. These are ray tracing implementation and distributed computing.

1.2.1.1: Ray Tracing

I have always had a strong interest in data structures and algorithms. Attempting to research, understand, and implement challenging algorithms is one of the most exciting elements of computer science. This is especially true when an algorithm is relatively new and cutting-edge. I wanted to undertake a project that would interest and challenge me. All of these factors informed my choice of problem domain. Ray tracing is a technique that I felt satisfied all of my criteria for an FYP topic.

Ray tracing is a challenging technique that is at the forefront of computer graphics. Ray tracing has all of the usual issues associated with graphics rendering. It involves vector mathematics, coordinate systems, viewing planes, and camera positioning. However, it also creates new challenges. This is evidenced by the fact that real-time ray tracing has only recently become possible on consumer hardware. This is despite the immense increases in computing power seen in the past decades. The challenge of ray tracing, and the possibility of gaining insight into such a cutting-edge family of algorithms, were major factors in my choice of FYP.

Another benefit of my FYP is the ability to create visual output. The interface I aim to create will be able to produce an image of a virtual scene, created using the ray tracing technique. The concept of a program is easier to demonstrate when it produces a visual output. This is particularly true when demonstrating it for non-technical individuals. They may not understand or appreciate the significance of numeric or textual output. Visual output can appear much more impressive.

1.2.1.2: Distributed Computing

My FYP is also relevant to the field of distributed computing. This is because its computational load can be distributed of over multiple machines. I decided to include this element in the project as it is more than just a general interest. Concurrency and distributed computing are important in many of the areas I would like to work in after graduation. Examples of this include creating concurrent or multi-threaded components for operating systems and developing infrastructure for cloud-based services. My FYP topic gives me a chance to implement distributed computing in a non-trivial algorithm. This could give me great insight into the field, and a chance to start developing my abilities in this area. At the very least, it should stand out to employers in the fields that interest me.

1.2.2: Programming Language

I chose the Julia programming language for my FYP for three reasons. The first is that Julia can be installed on many different computing environments and architectures. These include the Windows, macOS, Linux, and FreeBSD operating systems. 32-bit and 64-bit versions are available for all but FreeBSD. Julia even provides a version for Linux systems using ARM architecture processors. This is an important advantage, as my FYP should be able to create programs that can be distributed across multiple machines. By installing Julia on all of these machines, a homogeneous computing environment can be created. This allows for the creation of code that is architecture independent. This abstraction greatly simplifies the distributed aspect of the project.

The second reason I chose Julia is its speed and efficiency. Despite having to install a specific environment, Julia is orders of magnitude faster than other languages with similar requirements. This is because these other languages often make use of a virtual machine (VM) or interpreter. These VMs and interpreters often produce intermediate code, rather than machine code. An example of this is Java, which compiles to Java bytecode. This bytecode is then translated to machine code by the Java Virtual Machine (JVM) when the program is run. Julia avoids this approach. Instead, it compiles directly to machine code as the program is running. This is known as just-ahead-of-time (JAOT) compilation. The system Julia uses to optimise how it compiles its code is how it achieves its impressive performance. Once a segment of code has been compiled, its machine code is saved and it can be reused without the need for recompilation. Julia implementations of programs often perform within a factor of two relative to fully optimised C code (Kouatchou, 2018). If necessary, C functions can be called from directly within a Julia Program. The Julia environment also provides REPL (Read-Evaluate-Print-Loop) functionality. This means that users are able to use my interface dynamically from a terminal, without the need to create program files. This is useful for quick, once-off uses of the interface.

The final reason I chose to use Julia was to make my FYP stand out amongst all the others. Julia is a relatively young language. It has had great popularity in universities and academic settings. However, it has yet to come anywhere near the mainstream success of languages such as Java, Python and R. This is despite its speed and performance benefits, and its wide variety of powerful libraries. I believe that using Julia for my FYP is a good way to differentiate it from others and generate interest in the project.

1.3: Report Overview

The rest of this report is divided into five sections. These are:

- Research;
- Design;
- Implementation;
- Results;
- User Guide.

The first section details the research I have done in preparation for this project. This includes research papers, books, articles, and online tutorials. This section also describes a previous project I have completed using the Julia programming language. This project was undertaken to become familiar with various aspects of the language that are relevant to my FYP. The purpose of including this previous project in the report is not to detail its every aspect. I will focus on relevant parts and show how they helped me prepare for my FYP.

The second section describes the general design of the project. It also defines what the project should be able to do.

The third section deals with the implementation of the project design in actual code. It covers areas such as structure, datatypes and their relationships, and design decisions made during the project.

The penultimate section examines the final product that has been developed. It contains images rendered using the project ray tracing module. These images demonstrate the functionality of the module. It also examines the quality and usability of the module interface for users. This includes examining an example program that uses the module. This section also examines the effect of distributed computing on the time it takes to render an image. The functionality of the interface is also compared with the original goals of the FYP.

The final section provides a guide on how to create programs using the project interface. It provides details on how to create and render a virtual scene using the datatypes and functions provided by the module. It also provides details on how to correctly set up a computing cluster using Julia.

Section 2: Research

This section details the research I have conducted for my FYP topic and how it influenced the design and implementation decisions of the project. First, research into state-of-the-art technologies in ray tracing and distributed computing is examined. This mainly concerns real-time ray tracing and high-throughput supercomputers. Following this, research of a more suitable level for a university FYP is considered. Books, articles, and tutorials that have provided insight and instruction for various project components are discussed.

The second part of this section relates to the choice of programming language for the project. This involves research into the technical requirements of the language, and the benefits and disadvantages of various candidates. A technical evaluation of the chosen language, Julia, and how it satisfies the project requirements is also provided. To close this section, I examine a previous project I have completed using Julia. I give an overview of the project, focusing on the areas relevant to this FYP and how they have helped my preparation.

2.1: State-of-the-Art Technologies

Research into state-of-the-art technologies was conducted separately for both of the problem domains relevant to this FYP. In the field of ray tracing, state-of-the-art technology almost exclusively relates to real-time implementations for computer games. For distributed computing, the same can be said of globally distributed supercomputers. For this reason, these areas are the main focus of this subsection.

2.1.1: Real-Time Ray Tracing

The possibility of real-time ray tracing has been discussed in the field of computer graphics for at least two decades. Research with the goal of advancing ray tracing towards real-time rendering has been conducted since at least the mid-nineties. Researchers had described one of the main difficulties to be an absence of dedicated hardware (Humphreys & Ananian, 1996) (Schmittler, et al., 2002). Jörg Schmittler and his team were attempting to address this issue in 2002. They developed a hardware architecture known as the Ray Tracing Core. However, for complex scenes this architecture struggled to exceed 10 frames per second (FPS). It would take many years before adequate solutions would be developed.

Intel unveiled a prototype graphics processing unit (GPU) at its developer forums in 2009. This GPU went by the name “Larrabee”, and Intel claimed it could perform real-time ray tracing (Teglet, 2009) (Kobie, 2009). However, their efforts encountered difficulties and Larrabee was not released as a consumer GPU product as originally intended. Instead, it was released as a development platform for high-performance computing (Stokes, 2009) (Smith, 2010). In 2013 Caustic Graphics released their R2500 ray tracing acceleration boards. These plug-in graphics boards were not quite capable of real-time ray tracing but were seen as a huge leap towards this goal (Hruska, 2013).

Real-time ray tracing was still being attempted on non-specialised hardware during this time. However, it proved difficult to achieve significant results. Results were often comparable to home gaming consoles from years prior, which used traditional rasterization rendering techniques. Shih, et al. (2009) achieved similar performance to the original Xbox games console. Their implementation could render scenes with with as many as 871 thousand triangles at approximately 30 FPS. The Xbox could render over 970 thousand triangles at the same rate (IGN, 2001). However, the XBox had been commercialy released eight years previous and was considered one of the less powerful consoles of the time (IGN, 2001). The highest FPS reached on non-specialised hardware was between 95 and 180, by Aila & Laine (2009). However, this was achieved in three monochromatic scenes, the most complex of which contained almost three times fewer triangles than the XBox was capable of rendering. It also required the use of hand-optimised assembly code.

Only recently has specialised graphics hardware for real-time ray tracing been released. In 2018, NVIDIA launched its Qudro RTX range of GPUs. These were the first GPUs genuinely capable of real-time ray tracing (NVIDIA, 2018). They used a specialised architecture, known as the Turing architecture. NVIDIA and AMD have gone on to release more hardware specialised for real-time ray tracing since. As specialised hardware only recently became available, it has not seen widespread adoption. This is likely due to cost. In their most recent hardware and software survey, from December last year, Steam indicates that less than twenty percent of users have hardware capable of real-time ray tracing (Steam, 2020). Steam is a digital video game distribution service. It reports that it has over 95 million monthly active users (Steam, 2020). This shows that their hardware survey provides a significant sample size.

The research given thus far is to justify my assertion that real-time ray tracing is at the forefront of the computer graphics industry. It is state-of-the-art and cutting-edge technology. Although it has been researched for over two decades, only recently has it become feasible to provide real-time ray tracing solutions to consumers. The area is still the focus of much research, aimed at advancing capabilities and lowering the cost for consumers. Last year, NVIDIA's research department released no fewer than five research papers on the topic (NVIDIA Research, 2020). One of these papers details the most recent cutting-edge technique in the area. It forms the basis for much of the remainder of this subsection.

The most recent development in real-time ray tracing is the reservoir spatio-temporal importance resampling algorithm (ReSTIR). It is described by Bitterli, et al. (2020). This technique is cutting-edge and quite advanced. I do not pretend to understand every aspect of it, but a section on state-of-the-art graphics would be incomplete without some form of description.

The ReSTIR algorithm greatly increases the efficiency of how shadows can be rendered using ray tracing, especially with limited compute budgets (Wyman & Panteleev, 2020). In ray tracing, determining if a point in the environment is in shadow involves casting shadow rays. These are rays cast to lights in the environment. If the shadow ray intersects an object before it meets a light, that light is casting a shadow on the point. Shadow rays should be cast to as many lights as possible to ensure accuracy. Certain lights should be prioritised based on factors such as proximity and brightness. ReSTIR allows fewer shadow rays to be cast while still producing high-quality output. The increase in efficiency enables the creation of scenes with millions of dynamic emissive light sources (Bitterli, et al., 2020).

The key claim that Bitterli, et al. make is that the number of shadow rays cast should be based on human perception, and not scene or lighting complexity. There is a limited number of unique shadows that are likely to affect any single point in a scene. Humans are also not capable of perceiving every shadow that affects a point. It is wasteful to cast a shadow ray to every single light source in a scene. The increase in render quality is likely to be negligible and go unnoticed. Bitterli, et al. claim that three or four shadow rays should be sufficient for every pixel in a frame, provided you select the most appropriate rays. The ReSTIR algorithm is how they select which shadow rays to cast.

ReSTIR uses a mathematical approach called resampled importance sampling, often shortened to importance resampling. Basic importance sampling, without resampling, is used to focus samples to more important regions. This means that for each pixel it provides a weight for each light in the scene. Shadow rays are only cast to the lights with the largest weights. It does this by using Monte Carlo integration. This Monte Carlo formula can be rearranged to be far more efficient. However, this involves introducing an unknown constant. Importance resampling uses a function to approximate this constant. If the approximation is sufficiently accurate, the rearranged Monte Carlo Formula can be used to select the most important lights for each pixel. The success of ReSTIR depends on the accuracy of the function used to estimate the unknown constant. However, this function can be relatively simple to design and allows for the use of domain knowledge (Wyman & Panteleev, 2020).

The quality of ReSTIR's choice of important lights can be improved by using spatial and temporal data, hence the spatio-temporal part of its name. Adjacent pixels are likely to have similar important light sources to each other. The light source weights for adjacent pixels can be reused. To calculate the light source weights for a pixel, ReSTIR also considers the weights for the pixels in the surrounding five-by-five grid. The weight of a light source is also unlikely to change much in a single frame. For this reason, ReSTIR also considers the weights from multiple previous frames for these pixels. The algorithm is still under development by the NVIDIA research department and great progress is being made. NVIDIA were hoping to include a further optimised version of ReSTIR in their upcoming *RTX Direct Illumination* software development kit, that they aimed to release this spring, 2021 (Wyman & Panteleev, 2020). This goal was achieved.

2.1.2: Globally Distributed Supercomputers

The aim of a distributed system is to increase computing performance by sharing the workload among multiple machines. The increase in performance is an important point. If distributing computation does not aid efficiency, there is no reason to run a program on more than one machine. The increase in computational power provided by distributing the load is a primary benefit of a distributed system. For this reason, many supercomputers are actually distributed networks containing multiple machines.

Another major benefit of distributed computing is the ability run on hardware provided by many vendors, built with varying architectures (IBM, n.d.). The majority of major distributed projects make use many types of vastly different machines. This, generally, includes both machines with general-purpose hardware and machines with specialised hardware.

Finally, state-of-the-art distributed computing projects push the boundaries on just how distributed the system can be. Many projects have a network of machines that is globally distributed. Such projects, generally, allow volunteers to add and remove their machines to/from the network. This requires the network to be capable of dynamically adding and removing nodes. Machines in such a system must communicate using the internet. There is no private network large enough to facilitate this global communication. Also, a private network could not support the dynamic adding of machines from anywhere on the planet.

The previous paragraphs in this subsection outline the characteristics I contend are essential to distributed computing. The remainder of this subsection outlines research into distributed projects that exhibit these characteristics and show state-of-the-art capabilities in one or more. It would be misleading of me not to mention the rise of cloud computing in this subsection. Services such as Amazon Web Services (AWS) and Microsoft Azure provide high-performance cloud computing solutions to customers. However, these are private companies. As such, it is challenging to obtain sufficient information and data on their implementations and performance. This makes it difficult to compare these services to the distributed projects I have researched. For this reason, cloud computing remains conspicuously absent from the remainder of this report.

The most state-of-the-art distributed computing project at the moment is the Folding@home project (FAH). FAH simulates many aspects of protein dynamics. Its aim is to aid scientists develop new treatments for a variety of diseases. It uses volunteered computing resources from people around the world. It pushes the limits in all three of the areas I have mentioned previously.

The FAH software consists of three primary components: work units, cores, and a software client. Work units are how the software divides the computational load. A work unit is the protein data that the client is asked to process, and are a fraction of the simulation between the states in a Markov model (Beberg, et al., 2009). A single work unit is downloaded to a user’s machine. After the necessary processing has been completed, the relevant data is returned to FAH’s servers (Beberg, et al., 2009).

Cores, in the FAH software, are specialised molecular dynamics packages. They perform calculations on the downloaded work unit and are heavily based on the GROMACS software package (Beberg, et al., 2009). GROMACS is an open-source molecular dynamics package developed and maintained by the University of Groningen, the Royal Institute of Technology, and Uppsala University (GROMACS, 2018). GROMACS is highly efficient. It contains manually optimised assembly code and hardware optimisations, and scales well on parallel systems (Kutzner, et al., 2007) (Hess, et al., 2008). The GROMACS team provide a special closed-source license for the FAH project (Folding@home, 2012).

The FAH client is software users can download to contribute to the project. The client manages the other software components. The client runs continuously in the background at a very low priority. It uses idle processing power so that normal device use is unaffected (Folding@home, 2011). The client also decides which FAH core is suitable for the user’s machine (Folding@home, 2011).

FAH uses a client-server architecture. It has four types of server. A diagram that demonstrates the architecture is shown below. The following list was created using information from Beberg, et al. (2009). The four types of server in the FAH system architecture are:

- assignment servers: these servers act as the global scheduler and load-balancer in the FAH system. It decides which clients should receive which work units. This decision is based on a number of factors. Assignment servers can see what operating system a volunteered machine has, and its hardware specifications. It will assign the machine to a work server that contains work units suitable for that machine. Higher priority work units are assigned to machines with higher performance capabilities and faster network connections. Work unit priority is based on a number of factors. These include complexity, need for parallelisation, and publication deadlines;

- work servers: each worker server contains files and data related to its specific project(s). They generate work units for their project(s) and accept, store, log, and analyse completed work. They also analyse what work units to prioritise to best improve their project models. In February of 2009 FAH had a total of 70 work servers, which combined contained over 350 terabytes of data;
- collection servers: work servers can experience failures and outages. In 2004 collection servers were added to the FAH architecture to help with this issue. If a client cannot upload their results to a work server, they can instead send them to a collection server. When the relevant work server is back online, it can receive the completed work that it had missed. This allows clients to continue making progress when its work server is down, and returning work servers are not flooded with a backlog of requests from waiting clients;
- web/stats servers: these servers contain the public-facing FAH website and statistics about the project's completed work. They also contain the computational cores required to process work units. After downloading a work unit form a work server, it may be necessary for an FAH client to download the relevant core.

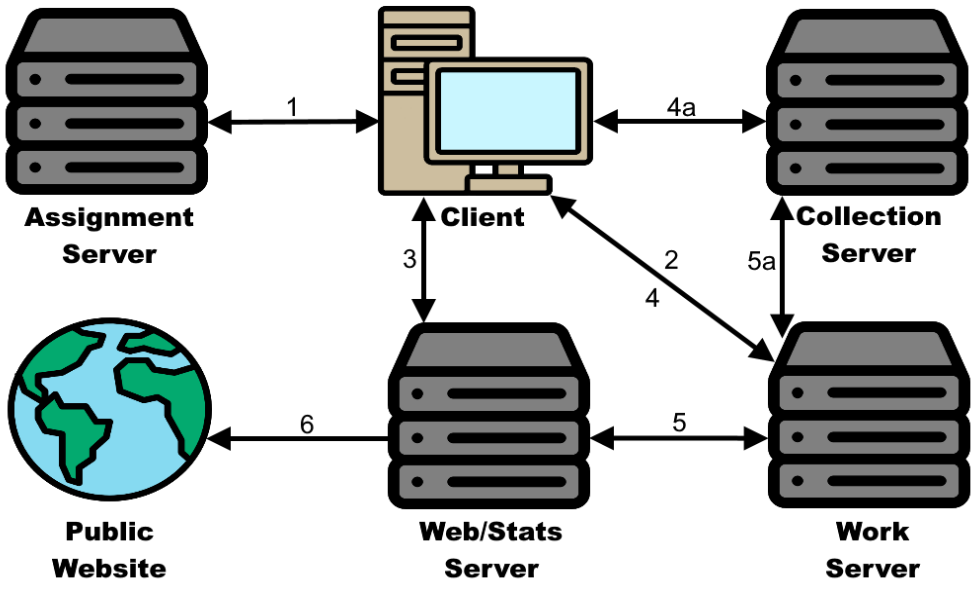

Figure 2.1: Dataflow through the FAH architecture. 1: Client assigned work server. 2: Client receives work unit. 3: Client may need to download core. 4: Results uploaded. 4a. If work server is down, results uploaded to collection server. 5: Results and statistics passed to web server. 5a: Work server may need to receive results from collection server after outage. 6:Results published on website.

Last year, FAH is estimated to have reached a peak performance that would make it the most powerful supercomputer on the planet. Performance is generally measured by the number of floating-point operations a system can perform per second (flops). In April of 2020, FAH was estimated to have performance of over 2.4 exaflops (Folding@home, 2020). That is over 2.4×10^{18} flops. According to TOP500 (2020), this performance would be over five times greater than the next best performing supercomputer, and make FAH the world's first exaflop system. TOP500 is an organisation that records and maintains available statistics for all known supercomputers. However, given the that FAH's resources are volunteered, this performance is not consistent. FAH's most recent statistics estimate performance to be approximately 2.2×10^{17} flops (Folding@home, 2021). Nonetheless, FAH reached cutting-edge performance levels, achieving a new milestone for computing in the process. It is also possible that it could reach these levels again.

FAH achieves its performance levels by having such a large network of machines. Their latest statistics claim that they have access to over 59 thousand GPUs and over 149 thousand central processing units (CPUs) (Folding@home, 2021). They are able to achieve this by making their software available for as many devices and architectures as possible. The FAH computer client is available on Windows, macOS, and six Linux distributions. There is also a version for Raspberry Pi computers and Docker containerization software and a specialised client for GPUs (Folding@home, n.d.). In 2007, FAH collaborated with Sony to release a client for the PlayStation 3 games console (Liao, 2007). FAH even went as far to release clients for the Chrome web browser and android mobile devices (Pande, 2014) (Pande, 2015). Unfortunately, these have both since been shut down, but there are plans to release an updated client for Android devices (Folding@home, 2019) (Thynell, 2018). FAH's commitment to making its software available on as many machines as possible is clear. They support both non-specialised hardware, and specialised GPUs. FAH have created a wide array of software clients, that have allowed people to volunteer using anything from web browsers and mobile devices to games consoles. The project has pushed the boundaries on just how many devices can be part of a distributed system.

2.2: Programming Languages

Based on my research into ray-tracing and distributed computing, I began to formulate requirements for the programming language I would use. Given the heavy computational load of ray tracing, I knew the language needed to be quick and efficient. When speed is a necessity, C and C++ are generally the languages used. They afford the programmer a lot of power and allow them to program "close to the metal". That is, they allow the programmer to optimise their code by using knowledge of the hardware on which the program will be run. However, I also wanted a language that would allow me to write device-independent code. This is necessary as the final product of this FYP should be able to be run in a distributed fashion. Having to create device-specific code for each machine on a network is an unwanted complication. For this reason, I decided to investigate higher-level languages.

Generally, high-level languages are device-independent because they run in some form of VM or use an interpreter. However, this causes a conflict with the performance requirements of the project. Programs that require an intermediate step to create executable instructions each time they are run will not be efficient. This is what happens with languages that require a VM or interpreter. Examples of this are Java, C#, and Python. These languages cannot achieve the performance of compiled languages such as C and C++. It was clear that I needed a compromise.

When I first approached my supervisor with ideas for an FYP topic, he mentioned that he would be very interested in supervising a project that used the Julia programming language. This is when I began researching Julia. I discovered that it was perfectly suited to the requirements I had developed for a project involving distributed ray tracing. The following are the technical requirements I had formulated, and how Julia satisfies them:

- speed: Julia is extremely fast. Pure Julia code often runs within a factor of two of what the corresponding fully optimised C code can achieve (Kouatchou, 2018). The Julia team benchmarked the language's performance against several others, including C, Fortran, Java, and Python. To do this they implemented several well-known complex functions, including matrix multiplication and recursive Fibonacci. Each language had identical algorithms with the same time complexity (The Julia Project, 2017). Julia's performance was comparable to C. It even outperformed C in the matrix multiplication function and a Mandelbrot function. The Julia Project have published the code used for each language online. A table comparing the results of Julia and C is shown below (The Julia Project, 2017);
- machine-independent: the Julia runtime environment must be installed for Julia programs to run. The runtime environment converts Julia code to machine code. This allows users to create machine-independent code. It is possible to create Julia code that is platform specific. However, the core Julia libraries support all operating systems for which Julia can be downloaded (The Julia Project, 2020);
- distributed computing libraries: the Julia standard library contains distributed computing modules. These libraries contain a `ClusterManager` datatype, modules for managing the distribution of arrays, and even modules for specifying network topology (The Julia Project, 2020).

Function	Execution Time (seconds)	
	C	Julia
pi summation (iterative)	27.37	27.67
matrix multiplication	72.01	70.25
matrix statistics	4.53	7.40
integer parsing	0.10	0.22
printing to file	9.93	10.87
Fibonacci (recursive)	0.02	0.03
quicksort (recursive)	0.26	0.26
Mandelbrot set	0.08	0.05

Figure 2.2: Performance benchmark comparison between C and Julia.

2.3: Design and Implementation Research

State-of-the-art ray tracing technology can render complex, interactive scenes at rates between 30 and 60 fps. I was aware from the start that anything approaching this performance would not be feasible for this project. My aim was only ever to have a project capable of rendering single, static scenes. I also knew that a globally distributed system was not a reasonable goal. This subsection details research that has directly impacted the design and implementation of my FYP.

The online book *Ray Tracing in One Weekend* by Peter Shirley (2020) is a fantastic introductory resource to implementing ray tracing. The author implements the technique using C++. Much like my FYP, his implementation produces images of static scenes. Many of the design decisions made are detailed and explained, and other possibilites are discussed. This provided great insight and helped me to analyse the options available to me when I had to make design decisions in my own project. Shirley provides C++ classes for vectors, rays, and spheres. These classes are rather basic, but are good starting points for developing more complex datatypes. The most beneficial part of *Ray Tracing* *in One Weekend* is its chapter on camera positioning. While the book mostly serves as an introductory reference point, this chapter has proved invaluable. Shirley provides an in-depth explanation of how to implement a moveable camera. He details his incremental development strategy and effective methods for debugging. Again, he justifies his design choices and offers alternatives. Examples of these design choices include whether to use a horizontal or vertical FOV, and whether to measure FOV in degrees or radians. The camera in my project is heavily based on Shirley's design.

Ray Tracing in One Weekend includes chapters on some more advanced topics. These include reflection and refraction. The issue is that it does not adequately explain the mathematics on which its implementation is based. It would be difficult for readers to alter the implementation to fit their own needs. Fortunately, Massachusetts Institute of Technology (MIT) have produced video lectures on ray tracing that cover these topics. The videos form part of MIT's *Introduction to Computational Thinking* lecture series (MIT, 2020). The lectures have the added benefit that their concepts are illustrated using Julia. They discuss in depth the mathematics behind reflection and refraction. They explore the concepts of colour transmission and a material's index of refraction, and how this might be emulated in Julia code. This involves in-depth discussion of mathematical formulas, such as Snell's law. By better understanding these mathematical concepts, I was capable of altering their implementation to suit the needs of my project.

Ray Tracing in One Weekend and *Introduction to Computational Thinking* are extremely valuable resources. Unfortunately, they only deal with the geometry of spheres. They also spend little or no time dealing with shadow. These gaps were filled by Jamis Buck's *The Ray Tracer Challenge* (2019). This book has three chapters dedicated to the geometry of planes. It details they can be combined to make cuboids and other more complex shapes. Buck also details the geometry of cylinders and triangles, and spends four chapters discussing the implementation of shadows. Buck provides information on many advanced topics such as various types of blurs, antialiasing, and normal perturbation. He has created a very complete resource.

Much of my research into designing a distributed system involved simply reading the Julia documentation. It outlines much of the provided functionality for setting up and managing a cluster. It details how to set up a master node and its slave nodes and how to set up communication channels between them. It also details how to set up a custom transport layer between nodes (The Julia Project, 2020). I also discovered a selection of algorithms implemented using Julia's built-in distributed computing libraries. The algorithms were part of a lecture given by Marc Moreno Maza (2014), published by the University of Western Ontario. Maza demonstrates how to distribute a large array across multiple machines and perform functions such as finding its maximum value. The lecture also shows a distributed implementation of merge sort, a matrix multiplication, and map reduce. Seeing distributed implementations of non-trivial algorithms was extremely helpful, and something that was absent from the Julia documentation.

This subsection is much shorter than the subsection that deals with state-of-the-art technologies. It references only five resources. This may seem like an unusually small amount of research. However, this FYP is not research based. The goal is to create a tangible product that can be used by programmers. It is not to contribute to the body research in a particular field. The resources detailed here have been more than sufficient to guide the design and implementation of what I aimed to develop. I believe the finished product reflects this.

2.4: Related Work

In preparation for my FYP, I began a project to help familiarise myself with the Julia language. It is a least-significant-bit steganography tool named J-DIS, short for Julia Digital Image Steganography. Digital image steganography is the hiding of files inside digital images. J-DIS accomplishes this by hiding the secret file data in the least significant bits of the image data. J-DIS can hide txt, doc, docx, pdf, and zip files inside png and jpg images. The difference between an original image and that image containing a hidden file should be imperceptible to humans. It can also extract hidden files from images, automatically determining the type of file that was originally hidden. J-DIS contains over 800 lines of pure Julia code. Two images are shown below. The image on the left is the original image. The image on the right is the same image, but with the entirety of J.R.R. Tolkien's *The Hobbit* hidden inside the image data. This was accomplished using J-DIS. If you view the image on the right at a shallow angle, you can notice some artefacts from the steganography process in the large blue section of the sky.

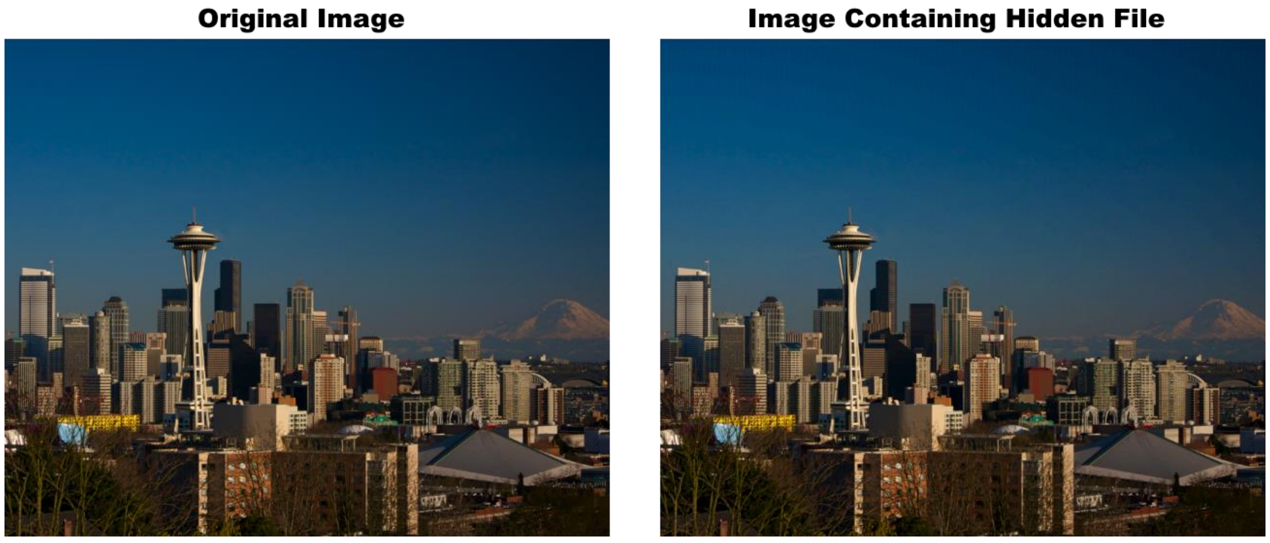

Figure 2.3: Julia steganography demo. The image on the right contains Tolkien's novel, "The Hobbit".

The J-DIS project helped me to learn many aspects of the Julia language. I became familiar with the general syntax and semantics of the language. I also became very comfortable with file handling and low-level bit manipulation. This was useful, but the J-DIS project helped me with three other areas that were directly relevant to my FYP. They are:

- working with images: this is a vital area as my FYP should be able to render images of a virtual scene. I learned to open, create, alter, and save images. I became familiar with the various datatypes Julia offers for storing colour data for pixels and images. From these datatypes, I have selected the most suitable for storing the colour information for the geometric objects in my FYP;
- mutability: composite datatypes in Julia can be defined as mutable or immutable. The datatypes that store colour information are all immutable. This has a large impact on the implementation of various elements in my FYP. I define many custom datatypes in the project code. Each must be defined as either mutable or immutable. Mutable datatypes are allocated on the heap and offer more flexibility. Immutable types can result in speed benefits as the compiler may be able to optimise operations on them. However, this can also cause other issues (The Julia Project, 2020). Being aware how mutability impacts implementation is essential;

- abstract types and inheritance: the colour datatypes in Julia form part of a type hierarchy. The purpose of type hierarchies is to group related objects. The input parameter of a function can be an abstract type. This function can then be called with any concrete subclass of this type. The function can check which concrete subclass has been passed and act accordingly (The Julia Project, 2020). Creating a type hierarchy in a project can help create cleaner interfaces. The various geometric types can inherit from the same abstract class or classes. This allows for the creation functions that can be called with multiple types of concrete objects as parameters. This removes the need to rewrite functions multiple times for different types of object. Users would not have to remember which version of a function to call for each type of object. In Julia, objects cannot inherit from concrete types. However, abstract types cannot have member variables. This is a design decision, as Julia was never meant to be an object-oriented language (The Julia Project, 2020).

Section 3: Design

The goal of this FYP is to create a programming interface. The interface is how programmers can use and interact with the underlying ray tracing implementation. A user should be able to use the interface in their own programs without the need for extensive research and learning. Users need not be concerned with the underlying implementation. To ensure that the interface is understandable and accessible for users, it must be carefully planned and designed. The second part of this section details the design of the project's interface.

The third part of this section details the implementation design of the FYP. It describes the design of the ray tracing algorithm itself. This subsection covers the general structure of the project. It also discusses the design of the functions in the project and their responsibilities, the objects and concepts that need to be represented in the module, creating an entry point for users to start the algorithm, and the methodology behind the design of constructors for custom datatypes.

Before starting the design phase, it is important to define what a product should be able to do. Therefore, I begin by defining what users should be able to accomplish by using my interface.

3.1: Defining the Product

Before I began designing and developing the software for my FYP, I defined what exactly the product should be able to do. First, it was necessary to define the minimum functionality needed to constitute a successful product. This is often called the minimum viable product (MVP). Subsequently, I defined some extra functionality. These are features I wanted to add to the project that would improve the product quality, but are not completely necessary. The first list below describes the functionality that would be considered part of the MVP. The second list describes the extra features. The inclusion of these features was based on time and resource constraints. All MVP functionality for the project was achieved, while only some of the extra functionality was achieved. A more detailed description of this is given in Section 5. The functionality of the MVP was defined as follows:

- environment definition: users should be able to define the dimensions of a cuboid shaped environment, and define its background colour using the red-green-blue colour model (RGB);
- addition of shapes: users should be able to add a number of spheres and cuboids to an environment. This includes defining the radius of spheres, and the length, width and height of cuboids, and setting the position of a shape's centre. Users should also be able to define the colour of a shape using RGB;
- shape removal: users should be able to remove added shapes from an environment;
- light source: users should be able to add a single light source to a scene and specify its position. Light sources have will have no shape and will consist of a single point. Users should also be able to set the ambient light of the environment;
- camera positioning and direction: users should be able to place a single camera in the environment and specify its position and the direction it faces;

- viewing plane: users should be able to define the height and width of the viewing plane and the vertical FOV;
- component removal: users should be able to remove any of the components they have added to an environment;
- rendering: users should be able to call a rendering function to produce an image of their environment as seen from a defined camera and viewing plane. Rendering should be implemented using the ray tracing technique. It should support shadows and object shading based on the objects and light source in the environment. The user should be able to view this image and save it to their machine;
- distributed rendering: the rendering function should be capable of running in a distributed manner over at least two machines.

The non-essential functionality I aimed to include is as follows:

- multiple light sources;
- coloured light sources;
- definable light source brightness;
- reflective objects;
- transparent objects;
- refractive objects.

3.2: Interface Design

The interface controls how someone can use the ray tracing module. The interface should allow the module to be used in a straightforward, powerful, and safe way. To achieve this, I focused on four key areas.

3.2.1: Controlling Interactions

A programming interface hides the underlying implementation and abstracts many responsibilities away from the user. It controls how the underlying code is used. To do this, we need to control how users can interact with the code. Providing simple, safe functions and helpful datatypes gives users powerful ways to interact with a module. However, it does not stop them from misusing the module by utilising other parts of the code incorrectly. One of the simplest ways to address this issue is to restrict the parts of the module users can access. This prevents the misuse of components that users should not interact with.

When using a module in Julia, users only have access to the data and functions that the module exports. A function or piece of data is exported by using the `export` keyword, followed by the name of the function, datatype, or variable we wish to export. This functionality allows for a simple way to control how users can interact with the module we are developing. A module in any language is likely to have “helper” functions and global variables that users should not use. By excluding these from our exports, we have restricted the components available to users. They can only be accessed via the components we have exported. This allows us to control when and how non-exported code can be run, and under what conditions. The user’s interaction with the module is controlled by how we design the exported components, which act as entry points to the hidden implementation that has been abstracted away.

One of the key benefits we get from controlling user interactions is the ability to better handle errors. It is often infeasible, or impossible, to create code that is robust to every situation. For example, in Julia it is impossible to create a function that accepts only positive real numbers as parameters. Julia does not provide an unsigned real datatype. The only way to deal with this problem is to check, inside the function, if the value of the number is below zero. Performing similar checks in every function that requires one would lead to a larger, less readable code base. It can also make finding the origin of an error extremely difficult. A piece of code could be called deep into a chain of functions calls. If the problematic data was passed by the user at the start of the call chain, it can be difficult to inform them of the exact part of their program that is causing the issue. This is especially true if data derived from earlier passed arguments causes the problem, as opposed to the argument itself, or if the data is part of a composite datatype. Such problems can be solved by carefully choosing our exports, and strategically designing the components we make accessible. For example, we can keep the intermediate functions in a call chain inaccessible, unless they are completely necessary for users. The exported functions, that act as entry points to our module, can perform error checking on given arguments before they cause problems. This allows us to raise errors and specify exactly where the user must alter their program. An example of this in the project is the constructor for my custom camera datatype. The user must specify the location of the camera and the point it is focused on. If these points are the same, the vector mathematics involved in ray tracing will cause errors. However, the error will not occur until the user tries to render a scene containing the camera. To prevent this, the Camera datatype is not directly exported, so users cannot access its constructor. They instead must use a function, `make_camera`, which will raise an error if the points are equal. This prevents the error from occurring later in the program, and we can specify exactly where it takes place. It also allows us to prevent behaviour that seems strange to the user, the cause of which can be hard to identify. For example, setting the ambient light of a scene to be greater than the brightness of the light source can cause a scene to look strange. A similar approach to the one used to prevent the camera issue is taken to solve this problem.

Controlling users' interactions with a module can also make using it much easier. Users do not have to remember each part of the module and how it works. The best example of this in the project is my custom cuboid datatype. A cuboid consists of six planes, which are also custom datatypes in the module. This means that the cuboid constructor requires the creation of six planes, which are passed to the function. This is an unnecessary level of complexity for users. Instead, we hide the cuboid and plane datatypes. A function, `make_cuboid`, is provided, that requires only four parameters to create a cuboid. The required parameters are the centre of the cuboid, its length, its width, and its height. The function uses this information to construct the planes of the cuboid. This is much simpler for users. It also means that no method that returns an instance of a plane is required. This means that there is less code to confuse users.

3.2.2: Object Representations

Ray tracing algorithms involve many components and complex mathematics. To simplify things and abstract away much of the complexity, we have to consider how we these components are represented in the module interface. The representations must seem logical and sensible to users. They must allow users to clearly understand how the different parts work together. To keep things simple, I based the interface around the representation one of one object: a scene. A scene acts as a container for all the other components available to users. A renderable scene contains zero or many geometric objects, a camera, and a light source. It can also act as a cuboid-shaped environment for its components, if the user chooses. Basing the rendering process on one simple object greatly simplifies things for users. They simply need to construct a scene object, before passing it to the rendering function. Users can construct a scene all at once, or create an empty scene and add components later. The number of components that make up a scene has been kept as small as possible. A minimal scene can be created with only two or three components.

The components of the scene also have simple base representations. Geometric objects can be created by simply specifying their position and dimensions. A light requires only a position, a brightness level, and an ambient brightness level. A camera requires three vectors. Positions in a scene are represented by a custom vector datatype. This vector representation simply consists of three real numbers. Each number represents a coordinate along the x, y, and z-axes. In the manner they are accessible to users, they primarily represent points within the scene. All of the objects available to users have a minimal design. They each require only a few components. I also believe it is clear what they represent and why they are needed.

While it was important to create simple object representations, a decent ray tracer should be capable of handling a lot more. The basic requirements for each object are detailed in the previous paragraphs. However, they can also contain much more information if the user chooses to provide it. Users can specify many extra parameters. This is achieved through the use of optional, keyword arguments. For scene objects users can also specify:

- its colour;
- the level of diffuse and specular reflection of its surface, if it is given dimensions;
- the shininess of its surface, if it is given dimensions.

For geometric objects users can also specify:

- its colour;
- its rotation around each of the x, y and z-axes. Otherwise they are assumed to be axis-aligned;
- the level of diffuse and specular reflection of its surface;
- the shininess of its surface;
- its refractive index;
- its level of transparency, and which faces to apply it to if it is a cuboid.

Colours are represented by Julia's built-in datatype, `RGBA`. This datatype stores a value for the red, green, and blue components of a colour, as well as for its opacity.

3.2.3: Package Constants

Some functions within the ray tracing module require integer IDs to represent information. For example, sometimes it is necessary to specify the side of a cuboid. Users can do this to specify which sides of a cuboid should be reflective. To make the interface more intuitive, an integer constant is provided for each side of a cuboid: left, right, front, back, top, and bottom. Reflection is assigned before object rotation, and it is assumed the front face is parallel with the x-axis and has a greater z-axis position value than the back face.

Common and useful configurations of some composite datatypes are also provided. This removes the burden of having to repeatedly instantiate some of the most common object representations. Ten constants representing various colours are provided. A vector representing the origin of the scene (point(0,0,0)) is also provided. The camera datatype requires a vector to specify the orientation and angle of its screen. Three vectors are provided to represent to most commonly used orientations: vertical, horizontal facing downwards, and horizontal facing upwards.

3.2.4: Quality-of-Life

Quality-of-life features of software are those that are not necessary to provide the intended experience or functionality, but make the software more easily usable, particularly when used for long periods. The functionality described in the previous subsection could be thought of as quality-of-life features. This subsection, however, focuses on functions provided to users.

The module provides simple, but extremely helpful functions. The scene datatype has many internal components. Functions are provided to set and edit the state of these components. By using these functions, users avoiding having to access the components directly.

A function is also provided to help users calculate an "up-vector". This is the vector that specifies the orientation and angle of the camera. The most commonly used up-vector is one that is perpendicular to the direction of the camera. This up-vector can be difficult to evaluate if the camera direction is not parallel with an axis of the coordinate system. The function provided requires the user to pass the camera's origin and focus point and the rotation they wish the camera to have. A vector has an infinite number of possible perpendicular vectors. This is why the rotation is required. It allows the position of the top side of the camera to be determined. Zero degrees of rotation will result in the top of the camera having the greatest possible value for its y-axis position. One hundred and eighty degrees of rotation will result in the top of the camera having the smallest possible value for its y-axis position. One hundred and eighty degrees of rotation will create a camera that is, essentially, upside-down.

3.3: Module Design

This subsection details the design of the actual code base for the project. It details the module structure, the various module components, and how the components interact with each other.

3.3.1: General Structure

First, I will detail the various composite datatypes in the module, their relationships with each other, and their visibility to users. I will describe why each datatype is required and the kinds of information they hold. More specific implementation details are detailed in the Section 4. At the end of this subsection there is an entity relationship diagram that visualizes the relationships between each of the datatypes.

The most important datatype in the module is the `Vector_3D` struct. Ray tracing, and graphics in general, relies heavily on vector mathematics. Vectors are used to represent directions, magnitudes, and points in the coordinate system. Many other datatypes in the module have a vector as one of their internal fields. I have chosen to implement my own custom datatype as it allows me to overload Julia's base operators without creating conflicts. This simplifies vector operations such as addition and multiplication, both with scalars and other vectors.

The next most important datatype is the `Ray`. It represents rays of light that travel through the scene the user has created. Rays can intersect objects, be reflected, and be refracted. Sending rays from the camera location, through "pixels" in the viewing plane, is the basis of how ray tracing works. The `Ray` datatype contains two vectors. One represents the origin of the ray. The other represents its direction. Rays are not accessible to users.

When calculating if a ray intersects an object, there is much information that needs to be saved. This information is required by other functions in the module. Some of this information is mathematical data about the intersecting vector. Some information is about the intersected shape. To hold this data, an `IntersectionValues` datatype was created. When vector mathematics are used to find if a ray and a shape intersect, a value known as the "t-value" is produced. This represents what fraction of the vector's magnitude is used to reach the intersection point. Detailed information on the mathematics behind these calculations is given in Section 4.5. The t-value is stored in an `IntersectionValues` struct. The struct also stores information on the transparency level of the intersected shape. This is used to check if a secondary ray needs to be produced. If a vector intersects a cuboid or a bounding box, the face that is intersected is stored. It is possible for a ray two intersect either one or two faces of a cuboid. Therefore, the `IntersectionValues` datatype can hold up to two t-values.

Users are able to create spheres and cuboids, and place them in their scenes. These are represented by the datatypes `Sphere` and `Cuboid`. Both of these are subtypes of the abstract `Shape` datatype. Spheres and cuboids both contain information about their colour and their centre point. Spheres also contain information on its reflectivity, transparency, refractive index, and texture map. This information is not contained directly within the `Cuboid` datatype. It contains six instances of the `Plane` datatype to represent each of its faces. This is where the extra information is held. The `Plane` datatype is also a subtype of `Shape`. It contains vectors to represent each of its four vertices. Spheres also contain information on the shapes rotation. This is only used for texture mapping and is not necessary for cuboids. More details on this are given in Section 4.6.

A very similar datatype to Cuboid is also provided to represent bounding boxes. These are also known as sky boxes. A user can place a bounding box in their scene to represent the boundary of the environment. While the `BoundingBox` datatype is very similar to `Cuboid`, it does not inherit from `Shape`. The reasons for this are discussed in the implementation section.

The module also contains datatypes to represent object textures, light sources, and cameras. Textures allow shapes to show external images on their surface, instead of a plain, solid colour. A camera contains three vectors. These represent the position of the camera, a point the camera is facing, and the orientation of the camera. Lights contain their position, the level of direct light they provide, and the level ambient light they provide.

A datatype is provided to represent the entire scene the user wishes to render. This datatype stores a camera, a light, the shapes in the scene, if any, a bounding box, if one is provided, and a colour to use as a background colour, if no bounding box is provided. The `Scene` datatype is passed to the rendering function that produces a ray traced image. All other scene information is accessed through it.

All colour information is represented by the `RGBA` datatype from the Julia module, `Images`. This is the only composite datatype in the module that I have not created and implemented myself.

In the diagram below, datatypes that are not accessible to users are indicated by the dimmed eye icon. Those that are accessible are indicated by the brighter eye icon. The element representing shapes is coloured differently. This is to represent that it is an abstract type. In Julia, type inheritance is signified using the "`<:`" operator.

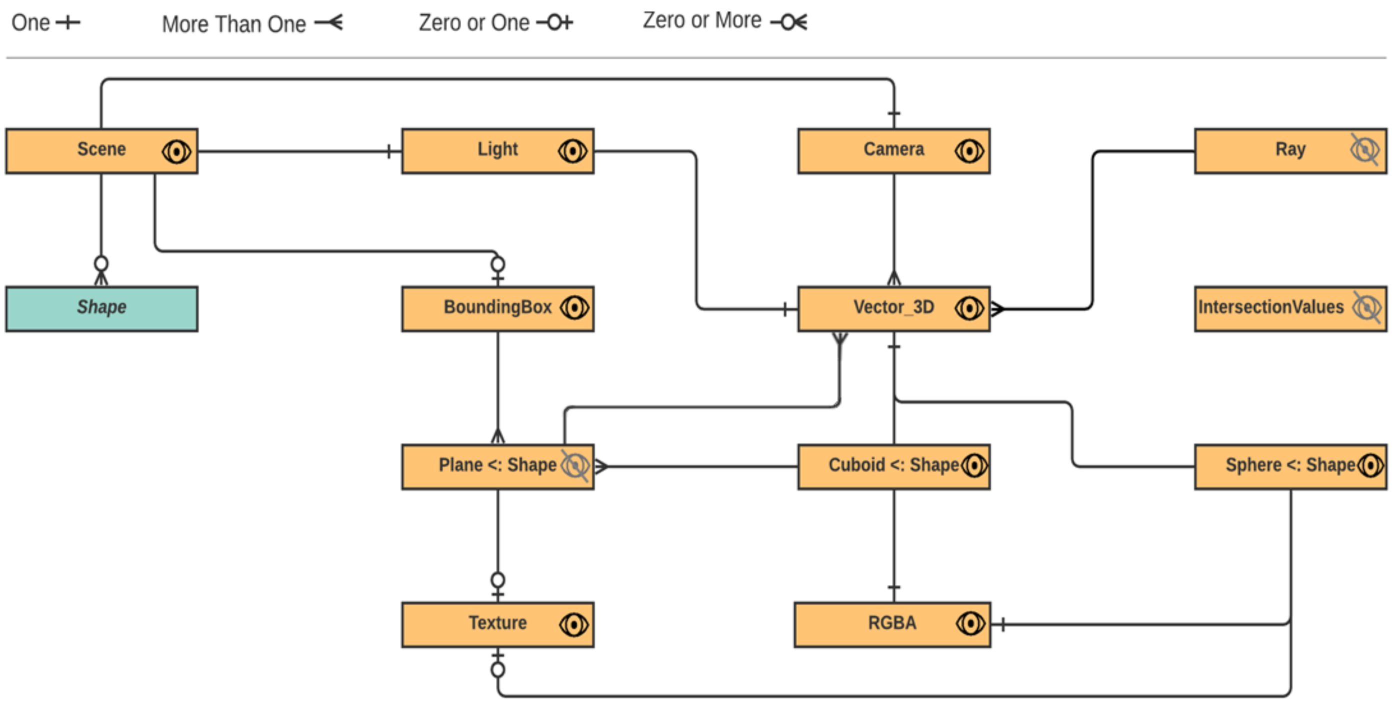

Figure 3.1: Module datatype entity relationship diagram. This diagram shows the composition of each datatype in terms of the other datatypes.

3.3.2: Phong Reflection Model

The model used to combine the various lighting effects in a scene is the Phong reflection model. This model is used to find the illumination level at any point in a scene. It does this by interpolating ambient light, diffuse light, and specular light levels at that point (Phong, 1975). The Phong model requires that shapes have values to represent their specular, diffuse, and ambient reflection levels, as well as how shiny they are. This is why they are present in the shape datatypes. The Phong model also requires multiple vectors to use in its calculations. Figure 3.2, below, is a visual representation of the vectors required. *S* represents the shadow ray, a ray cast from the intersection point to the light source. This is used to determine if the point is in shadow. *N* represents the normal vector to the intersected shape at the intersection point. *C* represents the original ray cast from the camera, and *R* represents the reflection of *S* through *N*. Below this is an image that shows the various types of reflection and how they are combined by the Phong model.

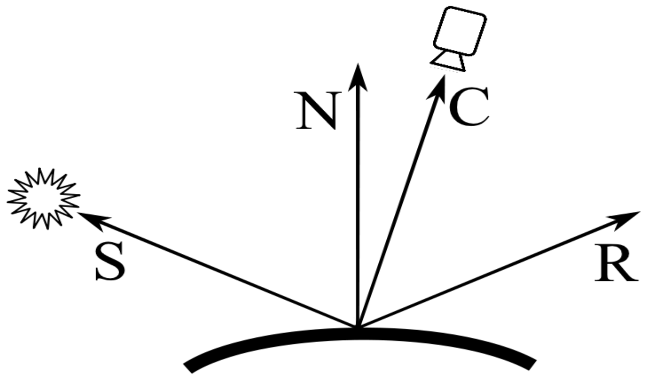

Figure 3.2: Vectors required for Phong reflection model.

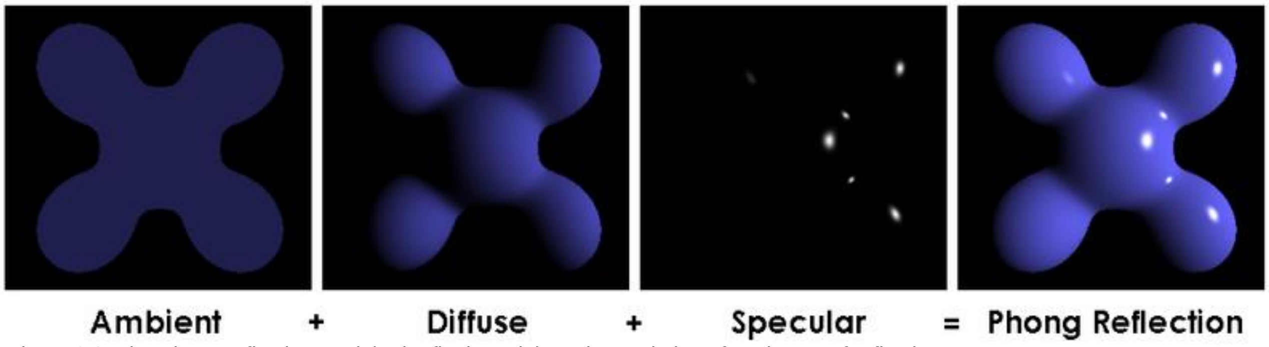

Figure 3.3: The Phong reflection model. The final result is an interpolation of each type of reflection.

3.3.3: Function Responsibilities

In designing this ray tracing module, I have tried to split the rendering process into components that make sense logically and that allow for maximum reuse. The key to this process was deciding which parts of the ray tracing pipeline to extract into their own functions. Each function is then responsible only for its own, smaller, portion of the rendering process.

There are thirteen rendering functions used to perform ray tracing recursively in the module. There are also many other functions that perform tasks that would not be specifically classified as rendering functions. An example is the function that calculates the normal of a plane. In total, there are over eighty functions in the module. It is not feasible to describe the role each plays in the rendering pipeline. I focus only on the most important functions, that I believe best illustrate the ray tracing process within the module. All but one of these functions are found the file *render.jl*. The main functions involved in the ray tracing process are:

- `render_scene`: this is the driver function of the rendering pipeline and the only function in this list that is accessible to users. The user must pass a scene as a parameter. Other information, such as the dimensions of the image produced, can also be passed. Otherwise, default values are used. `render_scene` first calculates the position of the viewing plane in the scene coordinate system. It allocates an array of the `RGBA` datatype. It loops through the array and uses the indices to calculate the location on the viewing plane through which to cast a ray. The path this ray takes determines the colour of that “pixel”. An instance of the

RGBA datatype which stores this colour information is assigned to the relevant index of the allocated array. The array is returned once the loop has finished;

- closest_intersection: this function calculates information about the first intersection a ray makes as it is cast into the scene. This includes the t-value, which object it has intersected, and, if the object is a cuboid, which side it has intersected. If no intersection takes place, nothing is returned. This is the equivalent of null in languages such as C and Java;
- colour_pixel: this is where the more interesting parts of the process happen. It is also where the recursion begins. Functions can be called from within this function that can call colour_pixel themselves. First, it uses a t-value to calculate the coordinates of the intersection point of a vector and an object. This point can become the origin of a new vector. Next, the colour of the intersected object is calculated. This can involve simply retrieving the colour information from an instance of a shape. It could also involve calculating which part of a texture map decides the shape colour at the point of intersection. If the shape is transparent or reflective, new rays will need to be created. The level of shadow being cast on the intersection point. Finally, the colour of the shape is used with information about reflectivity, transparency, shadow, the light levels in the scene, and the distance from the light source to determine the colour of the relevant pixel. These components are combined using the Phong reflection model. This information is passed back to whichever function called colour_pixel. Because colour_pixel can be called recursively, the calling function is not always the driver function;
- texture_colour: this function is called from within colour_pixel. It determines the colour of a shape at a given point on its surface. If the shape contains a texture, the function uses it to calculate the colour at the given point. Otherwise, it returns the colour information contained in the shape;
- transparency_colour: if a transparent shape is intersected, a new ray must be created. It should start from the end point of the intersecting vector. transparency_colour is called from within colour_pixel. It calculates the direction of the new ray, based on the refractive index of the intersected shape. It then calls colour_pixel, passing the new ray as a parameter;
- reflection_colour: if a shape is reflective, this function is called from within colour_pixel. It uses the normal at the point of intersection to calculate a reflected ray. This new ray is then passed as a parameter to a new call to colour_pixel;
- shadow_level: this function calculates the level of shadow that falls on a given point in a scene. It returns a value between zero and one, inclusive. It does this by casting a new ray from the point to the light source. If this ray intersects any objects, the point is in shadow. If any of the intersected objects are non-transparent, the value one is returned. Otherwise, the value is based on the combined transparency levels of all intersected shapes. If combined transparency levels are greater than one, one is returned.

The diagram on the following page visualizes a possible call path through the functions listed above. At the end of the path, an instance of the RGBA datatype is returned. Each call to colour_pixel is coloured in yellow to illustrate where recursion can take place.

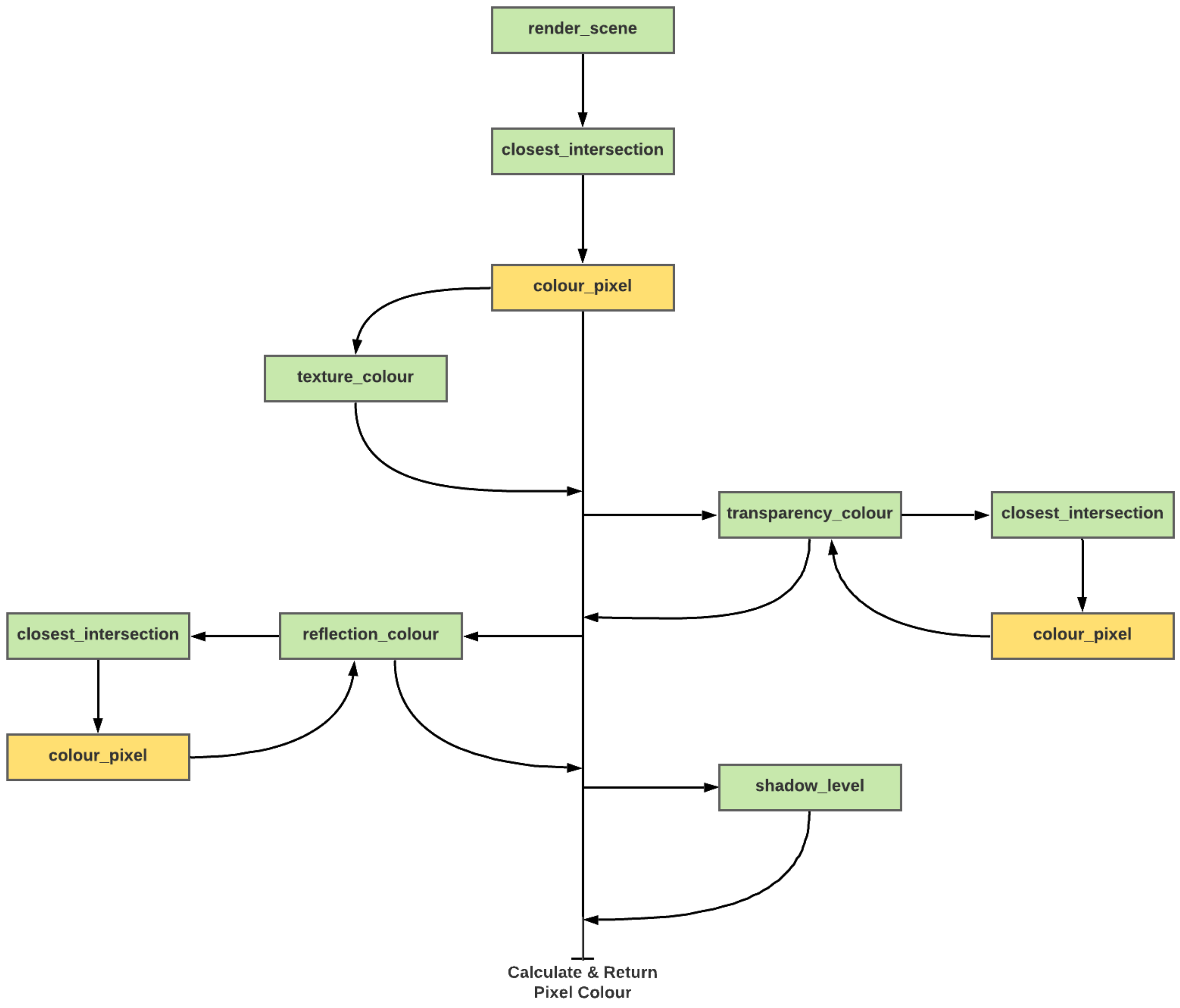

Figure 3.4:Function call path. This shows at what point in the ray tracing process the most important functions are called.

Section 4: Implementation

This section details the concrete implementation of ray tracing in the module, including the mathematics required. The first three subsections describe the datatypes used in the module, and the custom functions and operators that were developed to facilitate the ray tracing implementation. The remaining subsections describe the implementations of various components of the module.

4.1: Datatypes

The first part of this subsection discusses the datatypes which represent a virtual scene, and the components a user may wish to add a scene, such as a camera or a light source. Essentially, these datatypes represent the state of a scene. The data they contain includes things such as colour information, coordinates, and attributes such as brightness and specular reflection.

The second part details the datatypes that are used primarily to store mathematical information required to perform ray tracing. This includes information such as t-values, mentioned in the previous section, and the direction vectors of the rays that are cast into the scene.

All the datatypes discussed here, apart from `RGBA`, are defined in the file *types.jl*.

4.1.1: Scene Components

The first datatypes we deal with are those that represent geometric objects which can be placed in a scene. Each of these are subtypes of the abstract datatype, `Shape`.

In the Julia language, there is no inheritance as such. There are no classes, only structs. Structs can be subtypes of abstract types, but not concrete types. Abstract types cannot contain any data or attributes. As such, there is nothing for a subtype to inherit. Many people, particularly those with a background in object-oriented languages, would likely assume that the use of subtypes would be to facilitate shared attributes and behaviour. However, in Julia this is not possible. Attributes common to multiple subtypes must still be declared in each. As inheritance is a very common feature in modern languages, I think it is important to point out that this is not why I decided to use an abstract type in this module.

Julia also supports multiple dispatch. This is the ability to use functions with the same name, but a different call signature. This is similar to overriding functions from a superclass in languages such as Java and C++. This means that a function can be defined multiple times, as long as each definition can be distinguished by its parameters. This allows the same function to be called with different kinds of shapes, adjusting its behaviour based on the shape it received. This also can be implemented without the use of abstract types.

The reason it was necessary to use an abstract datatype was to allow the creation of shape arrays that have a definite type associated with them. Julia allows arrays of mixed types. However, I thought it was necessary to enforce restrictions on the types that can be present in arrays. This allows for better type safety within the module. It is particularly important for the `Scene` datatype, which is accessible to users. This type contains an array of shapes. By defining the type of this array as `Shape`, Julia will ensure that the array only contains instances of the correct subtypes.

There was another method which also could have been used to enforce type safety. The type associated with shape arrays could have been declared as a union of all shapes provided by the module. However, choosing to use an abstract type was an important design decision. There are many discussions online that seem to indicate there is a slight performance penalty associated with arrays of unions. But the main factor behind my decision was the impact on the development cycle of the module. If I had chosen to use unions, expanding the number of shapes supported by the module would have been extremely time-consuming and laborious. For example, if I were to add support for cylinders, every reference to a shape array would have to be altered. The cylinder datatype would have to be added to the union. This is far less efficient than using an abstract type and has a larger code footprint. Any new shapes simply have to be declared as a subtype of `Shape`. They can then form part of any shape array without the need for modification. This theoretical situation is illustrated by the code samples below. The code below the dividing line is all that would have to be added to the module because of the design decision to use an abstract type. If I had chosen to use unions, the code alteration shown above the line would also have to be implemented in many locations in multiple files.

Adding a new shape with the Union Array implementation:

```
struct Cylinder
    ## datatype attributes here
end
```

This function signature would change to the new one below. Similar changes would need to be made throughout the code base:

```
function set_shapes(scene::Scene, shape_array::Array{Union{Sphere,Cuboid,Plane},1})
    scene.shapes = shape_array
end

function set_shapes(scene::Scene, shape_array::Array{Union{Sphere,Cuboid,Plane,Cylinder},1})
    scene.shapes = shape_array
end
```

Adding a new shape with the Abstract Type implementation:

```
struct Cylinder <: Shape
    ## datatype attributes here
end
```

The function signature below has not changed. However, the new cylinder type is still supported:

```
function set_shapes(scene::Scene, shape_array::Array{<:Shape,1})
    scene.shapes = shape_array
end
```

Figure 4.1:Implementation Comparison: Abstract Type vs. Union Arrays.

The `Sphere` and `Cuboid` datatypes are both accessible to users via custom constructor functions. Both contain vectors to represent their centre points and an instance of the `RGBA` type represent their colours. Both also contain 64-bit floating-point numbers to represent the level of diffuse and specular reflection given off by their surfaces, as well as how shiny their surfaces are. Each of these values must be between zero and one, inclusive. This restriction is enforced by the custom construction functions.

There are also many differences between the two datatypes. A sphere stores its radius. It also stores its level of transparency and reflection, its refractive index, any rotation applied to the shape, and texture data. The texture data is stored in another custom datatype. Spheres contain and union of this type and `nothing`. If the sphere does not have a texture associated with it, this field is `nothing`. Rotation is represented by three floating-point numbers, which represent the degrees of rotation around each of the Cartesian axes. This is only required for texture mapping, and cuboids do not need to store this information. This is discussed in more detail in Section 4.6.

Cuboids do not directly store information on their transparency, reflection, or index of refraction. Neither do they directly store any texture data. Instead, cuboids contain six instances of the `Plane` datatype. Each plane represents a face of the cuboid and stores that face's attributes. This allows for different attributes for each cuboid face. Only one texture can be applied to a cuboid. If one has been applied, each plane stores a reference to all, or part, of that texture. This is also discussed in Section 4.6. A plane is defined by the locations of its four corners[1]. Each corner is represented by a vector within the `Plane` datatype. These vectors are extremely important and are used in many of the mathematical calculations performed in ray tracing.

The module interface allows users to apply rotation to a cuboid, around any of the three Cartesian axes. However, unlike a sphere, it does not need to store this information, either directly or within one of its planes. To understand why this is, we will examine the custom cuboid constructor that is accessible to users. The ray tracing module exports a custom constructor; `make_cuboid`. This function must, at a minimum, be provided with the centre point of the cuboid, its length, width, and height. Users can also provide a variety of keyword arguments, as shown in the code sample below. The three we will focus on here are `x_rot`, `y_rot`, and `z_rot`. These represent the cuboids rotation around the x, y and z axes respectively. Other arguments are discussed in subsequent sections.

```
function make_cuboid( centre::Vector_3D, len::Real, wdh::Real, hgt::Real ;
        x_rot::Real=0, y_rot::Real=0, z_rot::Real=0, colour::RGBA=GREEN,
        diffuse::Real=0.5, specular::Real=0, shine::Real=500, transparency::Real=0,
        refraction::Real=0,reflection::Real=0,
        sides::Union{Int,Tuple{Int,Vararg{Int}}}=( LEFT, RIGHT, FRONT, BACK, TOP, BOTTOM ),
        texture::Union{Texture, Nothing}=nothing, map_mode::Int=TILE )
```

Figure 4.2:Custom cuboid constructor function signature.

The shape returned by the function is first constructed as an axis-aligned cuboid, with its centre at the origin. To do this, we must find the position of each of its eight vertices. To calculate the x-coordinate of a vertex, we must add or subtract half of the cuboid's width to/from the x-coordinate of the origin. To find the y-coordinate, half of the height is added or subtracted to/from the origin's y-coordinate. To find the z-coordinate, half of the length is added or subtracted to/from the origin's z-coordinate. As the origin has zero as each of its coordinate values, we can avoid performing any addition or subtraction. We can simply use half of the cuboid's width, height, and length, or their negations, as the components of the vertices' positions.

It is possible to determine the position of an axis-aligned cuboid's vertices in relation to

[1]: In mathematical terms, a plane is infinite and is defined by two vectors. Technically speaking, the faces of a cube can be seen as a section of a plane, or the overlapping area of two planes.

its centre point. The same mathematical procedure that is detailed in the previous paragraph can be used. However, rather than using subtraction and addition with the point of origin, the mathematics is performed using the centre point of the cuboid. This method is perfectly suitable if the cuboid does not need to be rotated. The formula to rotate a point in three dimensions will rotate it around one of the axes. So that this formula can be used correctly, we must ensure that each axis passes through the origin. This is why the cuboid is first constructed at the origin. Once any rotations have been applied, the cuboid can be moved to its correct location by adding its centre point to each vertex. All vertices are represented by vectors. Once this has been completed, six planes are constructed. These planes will form part of the `Cuboid` instance.

The user should supply the rotational measurements in degrees. First, the modulo operator is applied to the angles. This will give the remainder after the angle has been divided by 360. The angles are then converted to radians, as is required by the rotation formula. The sin and cosine of the angle are then calculated. The formulas for rotating about each axis is shown in the figure below. The implementation of these formulas is in the function `rotate_on_axis`, which is shown on the right side of the figure. The axis used for rotation is determined by an ID. These IDs are integer constants and are defined in the file *constants.jl*. They can be seen in the code sample below.

$$Angle\ to\ rotate\ =\ \theta$$

$$\boldsymbol{Rotate\ about\ x-axis}:$$
$$x^{'} = x$$
$$y^{'} = cos(\theta) \times y - sin(\theta) \times z$$
$$z^{'} = sin(\theta) \times y + cos(\theta) \times z$$

$$\boldsymbol{Rotate\ about\ y-axis}:$$
$$x^{'} = cos(\theta) \times x + sin(\theta) \times z$$
$$y^{'} = y$$
$$z^{'} = -sin(\theta) \times x + cos(\theta) \times z$$

$$\boldsymbol{Rotate\ about\ z-axis}:$$
$$x^{'} = cos(\theta) \times x\ -\ sin(\theta) \times y$$
$$y^{'} = sin(\theta) \times x + cos(\theta) \times y$$
$$z^{'} = z$$

```
theta = deg2rad( angle % 360 )
    cos_theta = cos(theta)
    sin_theta = sin(theta)

    if axis_id == X_AXIS
        y_ = cos_theta * point.y - sin_theta * point.z
        z_ = sin_theta * point.y + cos_theta * point.z
        return Vector_3D(point.x, y_, z_)
    end
    if axis_id == Y_AXIS
        x_ = cos_theta * point.x + sin_theta * point.z
        z_ = sin_theta * point.x + cos_theta * point.z
        return Vector_3D(x_, point.y, z_)
    end
    if axis_id == Z_AXIS
        x_ = cos_theta * point.x - sin_theta * point.y
        y_ = sin_theta * point.x + cos_theta * point.y
        return Vector_3D(x_, y_, point.z)
    end
```

Figure 4.3:Axial rotation formula & implementation.

Users can choose to add a bounding box to their scenes. This is represented by the `BoundingBox` datatype. This is the final datatype which represents an object that users can actually observe in their scenes. The composition of a bounding box is very similar to a cuboid. It contains six planes and colour information about its surface. However, it cannot be rotated, and its centre point is always at the origin. Currently the module does not ensure all scene components are within the bounding box.

Texture data is stored in the `Texture` datatype. This is simply a wrapper for an `RGBA` array, which is the most prominent way to store image data in Julia. The array inside the `Texture` struct is union of a one-dimensional and two-dimensional array. This allows users to use an image that is a single column or row of pixels. Admittedly, this is an edge case. However, support for this was simple to implement, and I decided to include the functionality. Users can create a texture using the supplied custom constructors. These constructors load the image data using the `load` function from the `Images` Julia module. The texture struct can then form part of shape. How texture data is mapped to points on a shapes surface is discussed in Section 4.6.

The final two scene components are the `Light` and `Camera` datatypes. `Light` is a fairly simple struct. It contains a vector to represent its position, and two floating-point numbers to represent the light's brightness and the level of ambient light it produces. The exported constructor ensures that both values are between zero and one, inclusive, and that the ambient value is not greater than the light source's brightness. As one can imagine, a scene in which the ambient light is brighter than the direct lighting would look strange and unnatural.

A camera contains three vectors; the position of the camera, a point the camera is facing, and an up-vector to represent its orientation and rotation. Rather than storing a point the camera is facing, a direction vector could be stored to represent the direction the camera is facing. This would require the user to calculate the direction vector for their camera. This would make things more complicated for users. The direction can be calculated by subtracting the point the camera is facing from the position of the camera. Rather than place this responsibility on users, the module calculates the direction itself. The responsibility is abstracted away from users. I believe this creates a simpler and more streamlined interface.

The final datatype discussed here is the `Scene` struct. This struct contains all the necessary elements of a user's scene and must be passed to the rendering function. Users can create an "empty" scene and use helper functions to add components later. However, a user must add at least a light source and a camera before passing a scene to the rendering function. Otherwise, an error will be raised. It must also contain an array of shapes and a background colour represented by an `RGBA` struct. However, these are set to an empty array and a default background colour respectively, unless otherwise specified. A scene can optionally contain a bounding box.

To allow an "empty" scene to be created, some of the datatype attributes are unions of the relevant scene component and `nothing`. The exact structire of the `Scene` datatype is shown in the figure below.

```
mutable struct Scene
    boundary::Union{ BoundingBox, Nothing }
    shapes::Array{ <:Shape }
    light::Union{ Light, Nothing }
    camera::Union{ Camera, Nothing }
    colour::RGBA
end
```

Figure 4.4:The `Scene` datatype.

4.1.2: Mathematical Components

There are three custom datatypes in the ray tracing module that do not represent a scene, or components in a scene. They, instead, represent mathematical information required to perform ray tracing. The first of these datatypes we will discuss is `Vector_3D`. This datatype contains three 64-bit floating-point numbers, which represent the x, y, and z-coordinates of the Cartesian coordinate system. In mathematical terms, vectors have no position. They have only a magnitude and a direction. They are used to represent directions in many parts of the code base. However, as they contain a set of coordinates, they can also be used to represent positions in a scene. Many examples of this are given in the previous subsection. Containing only three non-composite pieces of data, they are one of the simplest custom types developed for this project. However, they are also one of the most important. Ray tracing is heavily dependent on vector mathematics. As such, `Vector_3D` is the most used datatype in the module.

The `Ray` datatype consists of two vectors. One represents the origin of the ray. The other represents its direction. Vectors can be used to represent either a position or a direction, but not both. As such, a ray requires two vectors. Rays are cast into a scene. We can calculate what they intersect, and at which point. This is a fundamental part of ray tracing.

The most complex of the datatypes discussed in this subsection is the `IntersectionValues` struct. This type stores information about the intersections between a ray and an object, such as a sphere. The most important piece of data it contains are the t-values associated with an intersection. It is possible for a ray to intersect a three-dimensional object in either one or two locations. The `IntersectionValues` struct must support both of these cases. The situations where a ray will only intersect a shape at one point are as follows:

- the ray origin is inside the shape;
- when intersecting a cuboid, the ray is parallel to the intersected face[2];
- when intersecting a sphere, the ray is a tangent to that sphere.

A visualisation of these situations is shown in Figure 4.6. If the ray intersects a cuboid, the `IntersectionValues` struct must also store the faces that have been intersected. These are stored as integer IDs. These IDs are constants and are found in the file *constants.jl*. The transparency level of the intersected object must also be stored. This is to support transparent and refractive shapes. The `IntersectionValues` datatype must be capable of storing the information of a variable number of intersections. It must also support different shapes, with varying levels of transparency. As such, most of its attributes are unions with `nothing`. If an attribute is not required, it can simply be assigned the value `nothing`. How these attributes are used is discussed in subsequent sections. The implementation of the `IntersectionValues` struct is shown in Figure 4.5 below. The integer constants used to represent the faces of a cuboid are also shown.

```
const LEFT = 1
const RIGHT= 2
const FRONT = 3
const BACK = 4
const TOP = 5
const BOTTOM = 6
```

```
struct IntersectionValues
    t1::Float64
    t2::Union{Float64, Nothing}
    side1::Union{Int, Nothing}
    side2::Union{Int, Nothing}
    transparency1::Union{Float64, Nothing}
    transparency2::Union{Float64, Nothing}
end
```

Figure 4.5:Cuboid face IDs and the `IntersectionValues` datatype.

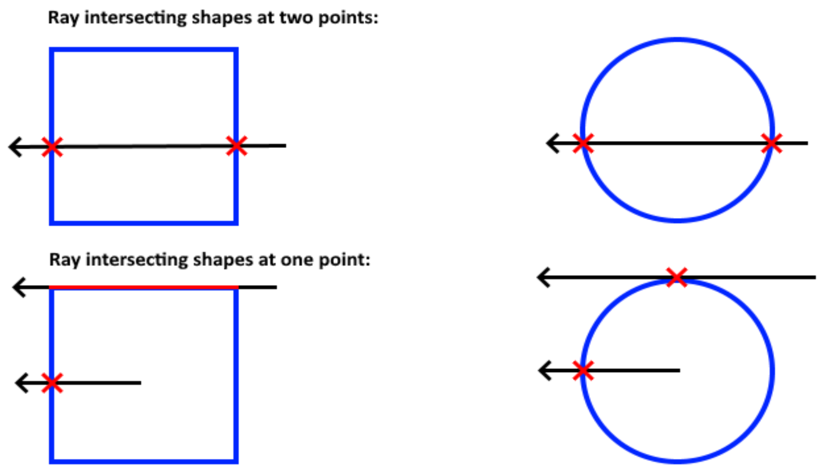
Ray intersecting shapes at two points:
Ray intersecting shapes at one point:

Figure 4.6:Ray-object intersections. This figure illustrates how different rays can produce a different number of intersections. Intersections are marked in red.

4.2: Vector Operations

It is not sufficient to have a vector datatype. The module must also implement a number of mathematical operations that can be performed on vectors. But before detailing these operations, I believe it is worthwhile to explain why I chose to implement my own custom vector datatype.

Julia contains no vector datatype. Instead, they are represented by arrays. This creates a number of issues. Arrays in Julia support a number of the necessary vector operations, but not all of them. The unsupported operations would have to be implemented by overloading mathematical operators. Overloading components of the base Julia library is not ideal. Users may expect operators to behave as defined in the Julia documentation, or may wish to overload them themselves. Other packages may also behave unexpectedly if base operators are overloaded for arrays. Creating a custom datatype avoids this issue. It also avoids the problem of variable length arrays. Vectors represented by Julia arrays can be of varying length. By creating a datatype with only three fields, we can ensure the user only passes the correct size vectors with less overhead. The name `Vector_3D` also clearly signifies the purpose of the type. Using Julia's base arrays does not make clear what the array is representing.

Six separate operators have been overloaded, to handle twelve different situations. The four basic binary mathematical operators have been overloaded. These operators are addition (+), subtraction (-), multiplication (*), and division (/). Each of these can operate on either two vectors, or one vector and one scalar. The implementations of commutive operators (+, *) produce the same results, regardless of the order of the operands. The unary negation operator (-) has also been implemented. This negates each field of a vector.

The final overloaded operator is the equality operator (==). This compares two vectors and returns true if they are equal. Vectors are equal if their x values are equal, their y values are equal, and their z values are equal. As vectors store floating-point numbers, this is actually more involved than one might first think. Floating-point numbers have limited precision. In Julia, they can have a maximum of 64 bits of precision. This makes it quite dangerous to use the equality operator to compare floats. Two numbers that one might think should be equal, may actually be seen as unequal by the computer. This is

[2]: Technically, when an intersecting ray is parallel to the cuboid face, there are an infinite number of intersections. For the purposes of this project, it is sufficient to think of this as either one or zero intersections.

due to their limited precision. The ray tracing module deals with this in two ways. The first is by using the maximum precision available. Any floating-point numbers used in the module are implemented using the `Float64` type. The second makes use of a Julia function named `eps`. This takes the name of a floating-point datatype as a parameter. It returns the smallest absolute value that the datatype can accurately represent, also known as epsilon. To compare two floats, we take the absolute value of their difference. This is then compared against the epsilon value of the relevant datatype. If it is less then this value, the values are equal. This is the method used to compare the x, y and z values of the `Vector_3D` datatype. The implementation of the overloaded equality operator is shown in the figure below. `EPSILON` is a constant that stores the value that is returned by the function call `eps(Float64)`.

```
function Base.:(==)(a::Vector_3D, b::Vector_3D)
	return (
		( EPSILON > abs(a.x - b.x) ) &&
		( EPSILON > abs(a.y - b.y) ) &&
		( EPSILON > abs(a.z - b.z) )
	)
end
```

Figure 4.7:Comparing vectors with the overloaded equality operator.

The ray tracing module also contains a number of functions that perform vector operations that cannot be represented by a single operator in Julia. These functions allow us to calculate the dot product and cross product of two vectors, the magnitude of a vector, unit vectors, and reflecting a vector about another vector. These functions and the overloaded operators are found in the file *operations.jl*.

4.3: Colour Operations

When a user calls the module rendering function, an image is returned. The colour of each pixel in this image is calculated using the Phong shading model. Reflective and transparent surfaces may also produce colours that are combinations if various colours in a scene. This functionality requires the ability to combine and aggregate colours. To accomplish this, I have overloaded a number of the base mathematical operators to take instances of the `RGBA` as operands. The module allows `RGBA` instances to be multiplied by or added to another `RGBA` instance. They can also be multiplied and divided by real numbers.

The `RGBA` datatype contains four attributes. The first three of these represent the level of red, green, and blue in a colour. When applying a binary operator to two colours, the regular operator is applied to the two red components, the two green components, and the two blue components. These form the colour components of a new `RGBA` instance. When applying a binary operator to a colour and a scalar, the regular operator is applied to the scalar and each colour component of the `RGBA` instance. Figure 4.8 shows the overloading of the multiplication operator to operate on the `RGBA` datatype.

The fourth attribute of the `RGBA` datatype is known as alpha, or the alpha channel. This represents the transparency level of a colour. Some might wonder if this can be used to represent the transparency of a shape in a virtual scene, or of a pixel in the image produced by the rendering function. The alpha channel could be used to represent the transparency of shape. There are four reasons I chose not to do this. The first is that having an explicit transparency property in the shape datatypes is more informative for users. It also adds to clarity to the code base. The second reason is to avoid passing references to entire `RGBA` structs between functions, when passing a single value would be more efficient. The third reason is that the `RGBA` datatype is immutable. If a user wished to change the transparency level of an object, they would have to create a new `RGBA` instance to replace the one stored by the object, rather than simply altering a single attribute of the object. The final reason relates to the representation of cuboids in the module. Each face of a cuboid can have a different level of transparency. This would require each face to store all of its colour information, even though they can only differ in transparency. To avoid this, the `Cuboid` datatype stores the shape's colour information. Its faces are represented by the `Plane` datatype. Each plane stores its own transparency data.

```
function Base.:*(num::Real, colour::RGBA{T})  where T<:AbstractFloat
    return RGBA(num * colour.r, num * colour.g, num * colour.b, colour.alpha)
end

function Base.:*(colour::RGBA{T}, num::Real) where T<:AbstractFloat
    return num*colour
end

function Base.:*(colour::RGBA{T}, colour2::RGBA{T}) where T<:AbstractFloat
    return RGBA(colour.r * colour2.r, colour.g * colour2.g, colour.b * colour2.b, colour.alpha)
end
```

Figure 4.8:Overloaded `RGBA` multiplication operator.

4.4: The Camera

4.4.1: The Viewing Plane

The camera is the point from which users view a scene. It is also the origin of each ray that is cast into the scene. Users set the location of the camera in the scene, as well as the pixel dimensions of the image to be rendered, and the field-of-view of the camera. Rays are cast from the camera, through a viewing plane and into the scene. Each point on the viewing plane through which a ray is cast represents a pixel in the final image. Before this process can start, the position of the viewing plane in scene coordinates must be calculated, and the dimensions of the plane must be scaled to match the scene. All of these calculations are performed at start of the rendering function, before any rays are cast into the scene.

The first step is to use trigonometry with the field-of-view to scale the size of the viewing plane. Then, vector mathematics can be used to find the bottom-left corner of the viewing plane, and the direction vectors of its left and bottom edges. Using this information, we can calculate which point on the viewing plane represents any specific pixel in an image. Each aspect of this process is described in detail.

First, the scaled dimensions of the viewing plane must be calculated. This can be calculated without knowing anything about the orientation of the plane. It requires only simple trigonometry. Users are asked to supply the field-of-view measurement in degrees.

Before any calculations begin, the field-of-view must be converted to radians. In Figure 4.9, this angle is represented by the Greek letter theta (θ). We can then imagine an isosceles triangle constructed with the camera as a vertex. The angle at this vertex is θ radians. The side opposite the camera represents the height of the viewing plane. By bisecting the angle at the camera, two congruent right triangles are created. Creating right triangles allows us to use Pythagorean theorems. The side representing the viewing plane has now also been bisected. As this side is opposite the angle θ, we can calculate its length using the tangent function ($\tan$). This would require us to know the length of the hypotenuse. However, if we simply set the hypotenuse length to one, the length of the bisected side becomes $\tan(\frac{\theta}{2})$. We must remember to divide θ by two in this equation. This is because the field-of-view angle has been bisected to create the right triangles. By keeping the length of the hypotenuse constant, the height of the viewing plane is based on the field-of-view. However, we must remember that we have actually worked out half of the viewing plane's height. To find its true height, we must double the calculated value. By making the camera's direction vector a unit vector, we ensure that the camera is always the same distance from the viewing plane. This process is how the field-of-view is controlled. The user provides the dimensions of the image they want to be produced. Using this information, we can easily calculate the aspect ratio. To calculate the width of the viewing plane, we simply multiple the height by the aspect ratio. This entire process is actual quite simple to implement and requires only five Julia statements. A visualisation of the process is shown in Figure 4.9. Figure 4.10 shows the implementation. Some may wonder why we cannot use $tan(\theta)$ to calculate the entire height of the viewing plane. This is because Pythagorean theorems only apply to right triangles. If θ is not bisected, we cannot use the tan function. Simply put, $tan\left(\frac{\theta}{2}\right) \neq \frac{tan(\theta)}{2}$.

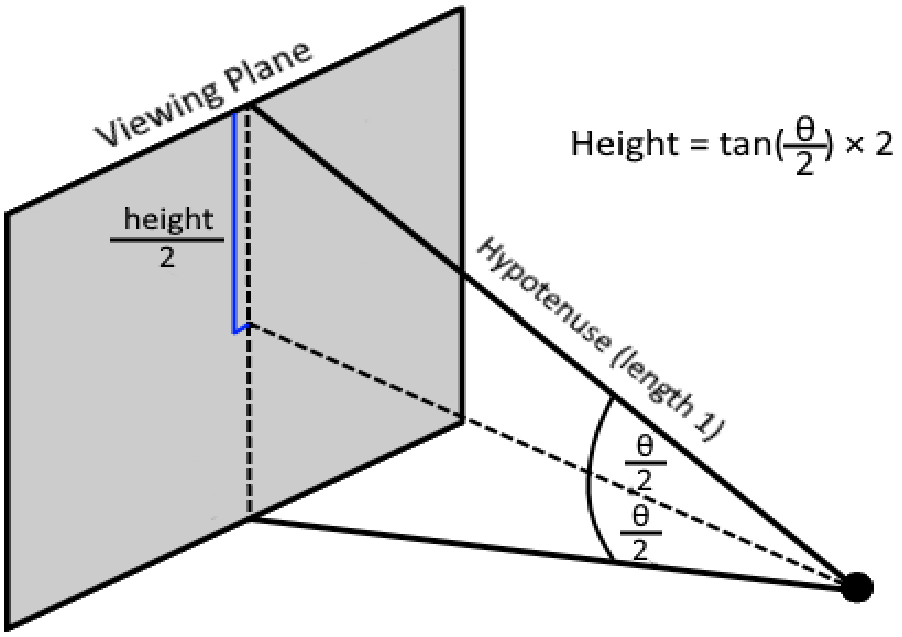

Figure 4.9:Calculating viewing plane dimensions with trigonometry.

```
aspect_ratio = image_width / image_height
fov_rads = deg2rad(vertical_fov)
h = tan( fov_rads / 2 )
viewport_height = 2.0 * h
viewport_width = aspect_ratio * viewport_height
```

Figure 4.10:Calculating viewing plane dimensions in code.

Once the size of the viewing plane has been determined, we must find its position and orientation in the scene. To this, the position of its lower-left corner is calculated. Then, the vectors that represent the direction and magnitude of its left and bottom edges from the corner are calculated.

The render function must be supplied with the position of the camera and a point it is facing. This point is called the focus of the camera. Subtracting the focus from the camera's position gives the direction of the camera. As mentioned previously, we convert this to a unit vector. This gives us the centre point of the viewing plane. To find the bottom corner of the plane we need to know its orientation. This is why users must provide an up-vector for cameras. An up-vector represents the direction from the centre point of a viewing plane to the middle of its top edge. The up-vector is converted to a unit vector. This is important as it allows the up-vector to be used as an "identity vector" for that direction. Any scalar multiplied by this unit vector will create a vector with that value as its magnitude, with the same direction as the unit vector. This important for later steps. This vector represents the direction of the viewing plane's left and right edges. We need to create another "identity vector" to represent the direction of the plane's bottom and top edges. We know that this vector must be perpendicular to the direction of the left and right edges. By enforcing the constraint that it must also be perpendicular to the camera's direction, we can easily calculate the vector. It is simply the cross product of the camera's direction and the direction of the left and right edges. Again, we must convert this to a unit vector. Now we have created "identity vectors" representing the vertical and horizontal directions from the viewing plane's bottom corner. We can multiply these vectors by the height and width of the viewing plane, respectively. This creates vectors that represent both the direction and magnitude of the left and bottom edges of the viewing plane, taking the bottom left corner as their origin.

We can now calculate the position of the viewing plane's bottom left corner. We can easily calculate the centre point of the viewing plane by subtracting the camera's focus from its origin. From this point, only simple geometry is required. We can move half of the width of the viewing plane to the left, staying parallel with the viewing plane's bottom edge. This will give us the centre point of the left edge. By moving half of the viewing plane height down the left edge, we find the position of the bottom left corner. To achieve this traversal across the viewing plane, we divide the vectors that represent the bottom and left edges by two. We then subtract them from the centre of the viewing plane. While this whole process may sound somewhat confusing, it is actually quite simple to implement. It requires only six lines of Julia code. This is shown in Figure 4.11. A simple visualisation of the traversal process is shown in Figure 4.12.

```
w = unit_vector( scene.camera.origin - scene.camera.focus )
u = unit_vector( cross( scene.camera.up_vector, w ) )
v = unit_vector( scene.camera.up_vector )

horizontal = viewport_width * u
vertical = viewport_height * v
lower_left_corner = scene.camera.origin - horizontal / 2 - vertical / 2 - w
```

Figure 4.11:Finding the bottom corner of the viewing plane.

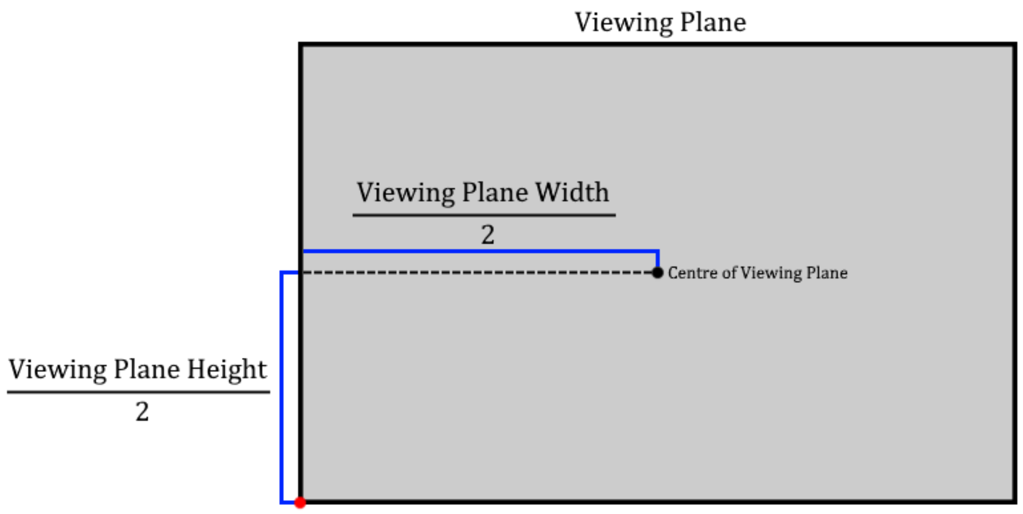

Figure 4.12:Traversing the viewing plane.

4.4.2: Pixel Positions

The final step before ray casting can begin is to implement a system that can transform a pixel's position in an image to a location on the viewing plane. This can be achieved by first calculating how far a pixel is, horizontally, from the left edge of the image, and how far it is, vertically, from the bottom edge of the image. These distances are then expressed as fractions of the image's total width and height, respectively. These fractions allow us to calculate how far into the viewing plane we should move to represent that pixel's location. They tell us what fraction to move along the vectors that represent the left and bottom edges of the viewing plane. This is, essentially, the reverse of the process shown in Figure 4.12. We take the fraction that represents how far a pixel is from the left edge, and we multiply it by the viewing plane's horizontal vector. We take the fraction that represents how far a pixel is from the bottom edge, and we multiply it by the viewing plane's vertical vector. This will produce two new vectors. These vectors are added to the vector that represents the bottom corner of the viewing plane. We now have the location on the plane through which to cast a ray that will represent the pixel's colour. We also have the origin of the ray, which is the position of the camera. By subtracting the cameras position from the relevant location on the viewing plane, we can calculate the direction of the ray. We now have the correct information to create an instance of the `Ray` datatype. This ray can be cast into the scene.

The code to implement this process is shown in Figure 4.13. `hrzt_factor` and `vert_factor` represent how far into the viewing play we should go. The relevant pixel can be on the left or bottom edge of the image. In this case, we should multiply the relevant vector by zero. This means that `hrzt_factor` and/or `vert_factor` should have zero as their numerators. However, Julia array indexing begins at one. This is why one is subtracted from the loop indices, `i` and `j`, when we assign values to `hrzt_factor` and `vert_factor`.

```
hrzt_factor = ( i - 1 ) / ( image_width - 1 )
vert_factor = ( j - 1 ) / ( image_height - 1 )
ray = Ray( scene.camera.origin, ( lower_left_corner + (hrzt_factor * horizontal) +
              (vert_factor * vertical) - scene.camera.origin ) )
pixel_colour = ray_cast(ray, scene)
```

Figure 4.13:Transforming a pixel's image position to the viewing plane.

There is a fundamental difference between pixels and points in a coordinate system. We can take advantage of this to implement a form of anti-aliasing. This allows users to create sharper images with better defined object edges. A point in space has a position, but no area. Moving even an infinitesimally small distance in any direction will result in a completely different point. A pixel in an image represents the colour to display on a pixel on a screen. Screens can be thought of as a grid of pixels. Each pixel has a finite area. The viewing plane coordinates we calculate merely represent the position of a pixel's bottom left corner. In theory, this means that we can cast an infinite number of rays through a pixel. In practice, this is limited by the precision of computers and the relevant datatypes. The ray tracing module allows users to specify how many rays they would like cast through each pixel. The final colour of a pixel is an average of each colour calculated using these rays.

There are two approaches to this technique. One is to cast rays through points that are as evenly spaced as possible. This ensures that the pixel is evenly sampled and the colour is as accurate as possible. This method requires a relatively complex system to calculate the position of the points, particularly as the number of points can vary. A simpler method is to pick the points at random from within the area of the pixel. This is a faster technique but is less accurate. The two methods offer a trade-off between performance and accuracy. I have implemented and tested both methods. Both methods produce the same quality of results. The extra accuracy achieved through even sampling is unnoticeable. For this reason, anti-aliasing is achieved by using random points from within the area of a pixel. Users can choose how many rays to cast through each pixel. They choose any value greater than or equal to one. If the value is not set by the user, it defaults to five.

4.5: Intersections

In mathematical terms, lines have infinite length. Rays can be viewed as a segment of line. If a ray intersects an object, the intersection point will be somewhere on the infinite line that contains the ray. In vector mathematics, there is a function to represent a point along a line in terms of any segment of the line that can be represented using two vectors. Rays use two vectors to represent a line segment. This function is often called $P(t)$, and is the origin of the term t-value. The $P(t)$ function is defined as:

$$P(t) = A + tB$$

A is the origin of the ray, and B is the direction vector of the ray. If t is equal to one, the point is at the end of the ray. If t is greater than one, the point is past the end of the ray. If t is less than one, but greater than zero, the point is between the ray origin and the end of the ray. If t is negative, the point is behind the ray origin. That is, its direction from the ray origin is $-B$. This is shown in the figure below.

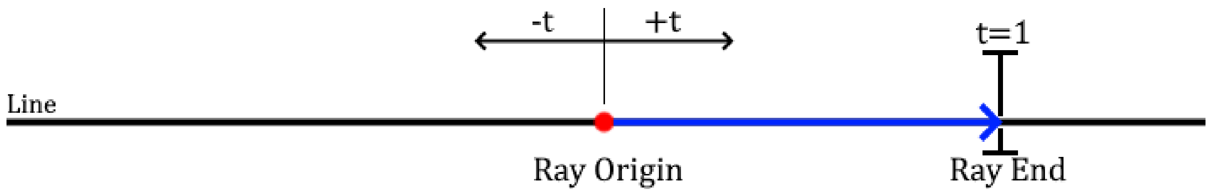

Figure 4.14:Visualisation of $P(t)$.

There are formulas for calculating the t-values of the intersection points between rays and circles, and rays and planes. With these values, we can use the $P(t)$ formula to calculate the position of the intersection point in the scene coordinate system. First, we will examine how the implementation handles sphere intersections. Then we will examine how it handles plane intersections.

The geometric equation of a sphere centred at the origin is $x^2 + y^2 + z^2 = r^2$. r is the radius of the sphere. x, y and z represent the three-dimensional coordinates of a point in space. If a point, (x, y, z) is on the surface of a sphere, then equality $x^2 + y^2 + z^2 = r^2$ will hold true. If $x^2 + y^2 + z^2$ is less than r^2, the point is inside the sphere. If $x^2 + y^2 + z^2$ is greater than r^2, the point is outside the sphere. This equation can be generalised to include spheres not centred at the origin. The general formula is $(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$. The point $C = (C_x, C_y, C_z)$ represents the centre of the sphere. This formula can be represented using vectors. This allows us to implement it in the module.

The line segment from the sphere centre C to a point P can be represented by the vector $(P - C)$. This is equivalent to the vector $(P_x - C_x, P_y - C_y, P_z - C_z)$. This is the square root of the left-hand side of the generalised equation of a sphere. This means that the equation in terms of vectors is:

$$(P - C) \cdot (P - C) = r^2$$

The left-hand side is now the dot product of $(P - C)$. Any point that satisfies this equation is on the surface of the sphere. We want to know if there is any point on the line that contains our ray that intersects the sphere. If so, there will be at least one value of t for the equation $P(t) = A + tB$, such that $P(t)$ satisfies the sphere equation. Therefore, we can substitute the point P for the equation $P(t)$. P is an entirely unknown quantity. However, we do have some information from $P(t)$. We know the origin of the ray, and its direction vector and magnitude. These are the variables A and B from $P(t)$. If we can solve for t we can calculate the exact position of the intersection point. If there is no value of t that solves the equation of the sphere, the ray and the sphere will never intersect.

To solve the equation of the sphere, we first substitute $P(t)$ for P. This leaves us with the equation:

$$(A + tB - C) \cdot (A + tB - C) = r^2$$

We can then subtract r^2 from both sides of the equation. Next, we can expand the brackets on the left-hand side. This gives the equation:

$$t^2 B \cdot B + t2B \cdot (A - C) + (A - C) \cdot (A - C) - \ r^2 = 0$$

This is a quadratic equation. It can be solved using the “minus b” formula:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In this equation, $a = B \cdot B$, $b = 2(B \cdot (A - C))$, and $c = (A - C) \cdot (A - C) - \ r^2$. The section of the equation, $\sqrt{b^2 - 4ac}$, is known as the determinant. This is because it determines how many solutions there are to the equation. If the determinant is greater than zero, there is two solutions for t. This is because a positive number always has two square roots, a positive number and a negative number. This means the ray intersects the sphere at two points. Negative “t values” represent points that are behind the ray origin. If the camera is the ray origin, these intersections cannot be seen by a camera, as they are behind it. This is why they are absent from Figure 4.6. Positive t-values are in front of the ray origin, and can be seen by the camera. If the determinant equals zero, there will be only one solution for t. This is because the only square root of zero is zero itself. Negative numbers do not have square roots. This means that if the determinant is less than zero, there is no real number solution for t. This means that the ray does not intersect the sphere at any point, either in front of or behind the ray origin. It is important to mention that the sign of the determinant does not decide the sign of t. If a ray intersects a sphere at two points, both can be behind or in front of its origin.

The implementation of this calculation is found in the function `intersect_values`. This function makes use of Julia's multiple dispatch functionality. It is overloaded to work with spheres, cuboids, and planes. The code for sphere intersections is shown in Figure 4.15. The function takes a sphere and a ray as parameters. First, we subtract the sphere centre from the ray origin. This vector is stored in `sphere_to_ray_origin`, which represents $A - C$. We then calculate a, b and c, as required by the "minus b" formula. Before evaluating the result of the formula, we must calculate the discriminant. It is important that this is calculated beforehand. If its value is less than zero, we should stop execution of the function and return `nothing`. This is because a negative discriminant means that there are no intersections, and there is no solution for t. There is no reason to continue the calculation and doing so will cause an error. The remainder of the calculation involves finding the square root of the discriminant. If we pass a negative value to `sqrt`, the square root function, Julia will raise an error and rendering will stop. If the discriminant is not negative, we calculate two solutions for t. If the discriminant is zero, both of these values will be the same. Finally, this information is used to create and return an instance of the `IntersectionValues` datatype. Two of the constructor parameters are `nothing`. These parameters are required only for cuboid intersections. They represent which faces of the cuboid were intersected. The actual coordinates of the intersection point are not calculated in this function. The t-values in the `IntersectionValues` instance are used in other parts of the code to determine the point's coordinates.

```
function intersect_values(sphere::Sphere, ray::Ray)
    sphere_to_ray_origin = ray.origin - sphere.centre

    a = dot( ray.direction, ray.direction )
    b = 2.0 * dot( sphere_to_ray_origin, ray.direction )
    c = dot( sphere_to_ray_origin, sphere_to_ray_origin ) - ( sphere.radius * sphere.radius )
    discriminant = ( b * b ) - 4( a * c )

    if discriminant < 0
        return nothing
    end

    t1 = ( -b - sqrt(discriminant) ) / ( 2 * a )
    t2 = ( -b + sqrt(discriminant) ) / ( 2 * a )

    return IntersectionValues(t1,t2,nothing,nothing,sphere.transparency,sphere.transparency)
end
```

Figure 4.15:Calculating a sphere intersection's "t-values".

The `intersect_values` function that is overloaded to work with planes is the only version of the function which does not return an instance of the `IntersectionValues` datatype. This because a plane cannot have two intersections with a ray. It can have zero, one, or infinitely many intersections. For the implementation, infinitely many intersections are thought of as zero intersections. This is explained in subsequent paragraphs. The plane intersection function returns a single t-value, or `nothing`. The function is called from within the cuboid intersection function. We will examine how this function uses the returned t-value. However, it is logical to look at the plane intersection function first, as this will explain how the t-values are calculated.

The vector notation for a plane is the set of points p, for which $(p - p_0) \cdot n = 0$. p_0 is a point on the plane, and n is a normal to the plane. We already know $P(t)$, the vector equation for a line segment or ray. We can substitute $P(t)$ into the equation of a plane. This gives us:

$$((A + tB) - p_0) \cdot n = 0$$

We can then expand the brackets to get the equation:

$$(b \cdot n)t + (A - p_0) \cdot n = 0$$

We can rearrange this equation to represent the value of t. This gives us:

$$t = \frac{(p_0 - A) \cdot n}{B \cdot n}$$

If $B \cdot n$ is equal to zero, the value t of is undefined. This means that the ray is parallel to plane. This can mean that there is no intersection point, or that the ray is contained in the plane and there are infinitely many intersections. If $B \cdot n$ is not equal to zero, there is a single point of intersection between the ray and the plane.

This value could be returned, and later substituted into $P(t)$ to find the intersection point. However, we cannot yet guarantee that this is correct. In mathematical terms, planes are infinite. The face of a cuboid is only a section of a true plane. Now that we know the true plane is intersected by the ray, we must check that the intersection is within the area of the cuboid face. To do this, we calculate the direction vectors of two edges of the face. This can be any two edges, as long as they are orthogonal. This means we cannot select opposite edges of the face. This is done by subtracting one edge vertex from its other vertex. We do not convert these vectors to unit vectors. This ensures we retain information about their magnitudes, which represent the lengths of the edges. We can calculate how far along the directions of these vectors the intersection point is. This is, essentially, measuring the perpendicular distance from an edge to the intersection point.

To see how far along a vector the intersection point is, a new vector must be created. This vector represents the distance and direction from the intersection point to the relevant edge vertex. This vertex should be the one that was subtracted from the other edge vertex in the previous steps. This vertex is then also subtracted from the intersection point. This ensures that both of the vectors created have had their directions calculated using the same vertex. This is important for the next step. This next step is to calculate the dot product of the vectors. If this value is negative, the vector representing the direction to the intersection point is travelling away from the inside of the cuboid face. This means there is no intersection between the face and our ray. If the dot product is positive, we must next ensure that its value is not too large. If the value is too large, the intersection point is past the end of the edge and is not on the face. This also means that there is no intersection between the ray and the cuboid face. It may seem logical to compare the dot product we have just calculated to the magnitude of the vector representing the edge of the face. However, this would be incorrect. The calculated dot product represents the square of the distance the intersection point is along the face edge. To accurately determine if the intersection point is too far from the start of the edge, we must compare the dot product to the square magnitude of the vector representing the edge of the face. The square magnitude of a vector is the dot product of that vector with itself. If the square magnitude is larger, the point could still be within the limits of the cuboid face. To be certain, we must repeat the process with the next edge vector. This involves creating a vector from the start of the edge to the intersection point. The previous steps are then repeated, using this vector and the second edge vector. If the dot product of these vectors is also within the correct range, the intersection point is within the limits of the cuboid face. We can then return the previously calculated t-value. Otherwise, the value `nothing` is returned.

This explanation may seem rather abstract and hard to visualise. To aid understanding, the process is visualised in Figure 4.16. A and B are the vectors that represent the direction and magnitude of the face edges. t and u are the other vectors required to see if point one is within the boundary of the face. Point one is in front of the origin of A and is the correct side of the top edge. This means that the $t \cdot A$ will be positive. The next step would be to evaluate $A \cdot A$. If this is larger than $t \cdot A$, the point is not past the end of A. As we can see, this is the case. Next, these steps are repeated with u and B. Point one is in front of B's origin. It also is not beyond the end of B. This means that $u \cdot B$ is both greater than zero and less than $B \cdot B$. In this case, `intersect_values` returns the t-value for point one.

To check if point two is within the boundary of the face, the same process is followed. First, we check that $v \cdot A$ is greater than zero and less than $A \cdot A$. Both conditions would be satisfied, as point two is in front A's origin, but not beyond the end of the vector. The next step is to check if $w \cdot B$ is greater than zero. As we can see from the diagram, point two is behind the origin of B. This means that $w \cdot B$ would be negative. In this situation, `intersect_values` would return the value `nothing`, indicating that point two is not within the face boundary.

Point 1 is on the face if:

$t \bullet A > 0$ and $t \bullet A < A \bullet A$ and $u \bullet B > 0$ and $u \bullet B < B \bullet B$

Point 2 is on the face if:

$v \bullet A > 0$ and $v \bullet A < A \bullet A$ and $w \bullet B > 0$ and $w \bullet B < B \bullet B$

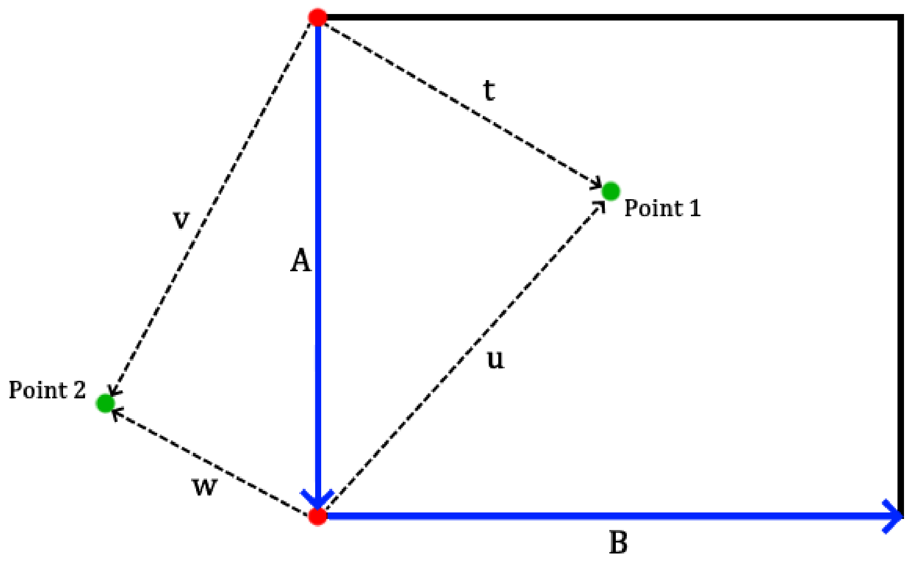

Figure 4.16:Verifying that a point is within the boundaries of a cuboid face.

The entirety of the code to find if a ray intersects a cuboid face is not shown here, as it would requires two full pages. All the code discussed in this section is found in the file *intersections.jl*. First, a normal vector to the face is calculated using the function `normal`. Next, the dot product of the plane normal and the ray's direction vector is calculated. If the dot product is zero, the vectors are orthogonal. This means the ray is parallel to the plane. This means that there is no intersection. In this case, the first if-statement fails, and `nothing` is returned. The equality operator (==) cannot be used to check if the dot product is equal to zero. The equality operator can be inaccurate comparing floats. This is due to the limited precision of floating-point numbers. Instead, the absolute value of the dot product is compared against `EPSILON`, the smallest value a 64-bit floating-point number can accurately represent. If the dot product is greater than `EPSILON`, the vector and the plane are not parallel. All values less than `EPSILON` are considered equivalent to zero. The dot product of the normal and the ray direction represents the $B \cdot n$ component of the formula to find the t-value. This is the denominator of the formula. This is why `nothing` must be returned if it is equal to zero.

If the dot product is greater than `EPSILON`, the function continues. As per the formula for t given previously, we create a vector from the origin of the ray to a point on the plane. This can be any of the vertices of the cuboid face, all of which are on the plane. The implementation uses the top-left vertex. This value represents the $p_0 - A$ component of the formula. We now have sufficient information to calculate the t-value. This value cannot be zero. However, it could still be less than `EPSILON`. Therefore, we must also compare the t-value to `EPSILON`, as values less than this are considered equivalent to zero.

```
n = normal(plane, ray)
dot_prod = dot(ray.direction, n)
if abs(dot_prod) > 0
	w = ray.origin - plane.upper_left
	t = -( ( dot(n, w) ) / dot_prod )
	if abs(t) < eps(Float64)
		return nothing
	end
	bounds_test_vector = ray.origin + ( t * ray.direction )
	a = dot(plane.lower_left - plane.upper_left, bounds_test_vector - plane.upper_left)
	b = dot(plane.lower_left - plane.upper_left, plane.lower_left - plane.upper_left)
	x = dot(plane.lower_right - plane.lower_left, bounds_test_vector - plane.lower_left)
	y = dot(plane.lower_right - plane.lower_left, plane.lower_right - plane.lower_left)
	if ( a >= -EPSILON && (b - a) > 0 ) && ( x >= -EPSILON && (y - x) > 0 )
		return t
	end
	return nothing
end
return nothing
```

Figure 4.17:Implementation of the cuboid face intersection tests.

Next, the intersection point is calculated, and vectors are created to represent two orthogonal edges. They are not assigned to variables, but are calculated within the call to the dot product function. The expression `plane.lower_left – plane.upper_left` is the vector A in figure 4.16. The expression `plane.lower_right – plane.lower_left` is the vector B. The expression `bounds_test_vector –`

plane.upper_left can represent both t or v. The expression bounds_test_vector - plane.lower_left can represent both u or w. Then the relevant dot products are calculated and compared to check if the intersection is within the boundary of the cuboid face. If it is, the t-value is returned. Otherwise, nothing is returned.

The t-value is returned to the cuboid intersection function. We know that a line will intersect two of the faces of a cuboid. We simply need to determine which faces it intersects. To do this, we simply iterate through each face of the cuboid, passing the face to the intersect_values function. Once two t-values have been returned, we use these and information about the surface of the face to create an IntersectionValues instance. This is then returned by the cuboid intersection implementation of intersect_values. Figure 4.18 shows the implementation of the cuboid intersection function. We first calculate the t-values of the intersections between the ray and the left and right faces of the cuboid. The integer IDs of the faces and their transparency values are also stored. If both t-values are defined and not equal to nothing, an IntersectionValues instance is created and returned. Otherwise, we continue checking the other faces. The remaining faces are checked in the order: front, back, top, and bottom. If the face's t-value is nothing, we continue through the rest of the faces. If it is a numeric value, we replace one of the saved nothing values with this number. We also update the relevant face ID and transparency information. We then check if both stored t-values are defined. If so, we use the stored data to create and return an IntersectionValues instance. If we have checked the t-values of each face and have not found two defined values, we return nothing. This indicates that our ray does not intersect the cuboid.

The intersect_values function is also overloaded to work with bounding boxes. This method differs only slightly from the cuboid intersection function. It takes a bounding box as a parameter, rather than a cuboid. As a design decision, bounding boxes cannot be reflective or transparent. As such, the function does not store any transparency information. The IntersectionValues instance returned contains nothing as the value for both of its transparency attributes.

The final function we discuss in this subsection is closest_intersection. This function determines which shapes a ray intersects, and which of the intersected shapes is nearest to the origin of the ray. The shape nearest the ray origin is the shape which should provide the colour information for the pixel relevant to the cast ray. All other intersection points are behind the closest intersected object. As such, they are obscured when looking from the ray origin. Depending on the reflectivity, transparency, and refractive index of the closest shape, shapes farther from the ray origin, and in different directions, may be visible. If this is the case, a new ray will be created. Its origin will be the closest intersection point.

First, a variable is declared to store the t-value of the closest intersection. It is initialised with the value of positive infinity. Variables are also declared to hold the index of the closest intersected shape in the scene's shape array, and the face ID if the shape is a cuboid or a bounding box. The function then iterates over each shape in the scene. It calculates the t-value of each shape with the ray passed as a function argument. If a value is undefined, it is ignored. If both values are undefined, it moves to the next shape. If a value is defined, it must be analysed to check if it might be the t-value of the closest intersection. If the value is negative, the intersection point is behind the ray origin. This means we can ignore the value. If it is greater than zero and less than the currently saved closest t-value, it is used to replace this t-value. The t-value is initialised to infinity. This means that any positive t-value found will replace its initial value. The shape array index will also be updated, as will the face ID if the shape is a cuboid. `intersect_values` cannot be called with a plane as a parameter from inside the `closest_intersection` function. This is because users do not have direct access to the `Plane` datatype. Its constructor is not exported. As such, users cannot add a plane to the shape array of a scene. This means that calls to intersection functions from within `closest_intersection` will always return an instance of the `IntersectionValues` datatype, or the value `nothing`. This means that we can make use of Julia's multiple dispatch feature to avoid having to check the type of the shape we are checking for intersections. If, after iterating through the shapes, we have found an intersection, we return the relevant t-value, the shape's index in the shape array, and its face ID if it is a cuboid.

If, after iterating through the shapes, the variable for storing the array index of the closest intersected shape is `nothing`, we can be certain that our ray has not intersected any shapes. If this is the case, and the user has added a bounding box to the scene, we should call the bounding box intersection function. We return the smallest positive t-value. Assuming the user has correctly constructed their scene, we can guarantee exactly one of the t-values will be positive. The relevant face ID is also returned. To indicate that the scene boundary was intersected, the array index returned is zero, as indexing in Julia begins at one. If the user has not added a bounding box to the scene, the value `nothing` is returned. The information returned by `closest_intersection` is used later in the module to determine information about the relevant intersected shapes.

4.6: Colours & Textures

4.6.1: Coordinate Mapping

The function `closest_intersection` returns enough information for us to determine the colour of the intersected shape. This information is passed to the function `colour_pixel`. This is where we decide which function to use to determine the shape's colour. If the given index is greater than zero, we know that a shape has been intersected. We can access the colour of this shape with the function `texture_colour`. This function is overloaded to operate on both spheres and cuboids. However, the version that operates on cuboids requires an extra parameter. The colour information may be different for every face of a cuboid if it has a texture applied. For this reason, we also need to pass the integer ID of the relevant cuboid face to the function. To know which version of the function to call, we must first check to see if the face ID is a number, or the value `nothing`. Then we can check the value of the shape's index.

If the face ID is a numeric value and the shape index is greater than zero, we call the cuboid texture colour function. If the cuboid does not have a texture, its colour is returned. For extra safety, we also check that the face ID is within the valid range. If the cuboid has a texture applied, more calculations must be performed to determine the colour that should be returned. We must convert the intersection point on the face to indices that can be used to access the colour data in the texture. This is accomplished with a call to the function `coords_to_x_y`. This function uses the bottom edge of the cuboid face as the x-axis of the face. It uses the left edge as the y-axis. The intersection position is then projected onto both axes to find its coordinates, relative to the cuboid face. Next, the coordinates are expressed as a fraction of the lengths of the axes. This yields two values that are between zero and one. These values are returned to `texture_colour`. The mathematical process to find these coordinates is very similar to that used to check if a point is within the boundaries of a cuboid face. The differences are due to the fact that now we are not just interested in whether or not there is an intersection. We have already determined this. Now, we are also interested in the exact location of the intersection point.

The bottom-left vertex of the cuboid face will serve as the origin. The left and bottom edges are the axes. We must represent them as vectors. Subtracting the bottom-left vertex from the top-left vertex gives the y-axis vector. Subtracting the bottom-left vertex from the bottom-right vertex gives the x-axis vector. Subtracting the bottom-left vertex from the intersection point gives the vector representing the line from the origin to the intersection point. Next, we calculate the dot product of the x-axis with itself, and the dot product of the x-axis with the vector to the intersection point. If we subtract these values from each other we, can find if the intersection point is behind or in front of the x-axis. We have done this when checking if a point is on a cuboid face. This time, however, we want to know the exact position along the x-axis of the intersection point. Therefore, we use division instead of subtraction. We divide the dot product of the x-axis and the vector to the intersection point by the dot product of the x-axis with itself. We must remember that the dot product produces square magnitudes. We have squared the intersection point vector's magnitude in the direction of the x-axis. We have also squared the magnitude of the x-axis itself. By dividing them, we find the fraction along the squared x-axis direction that the squared intersection vector lies. We then multiply this fraction by the magnitude of the original x-axis. This gives us the x-coordinate of the intersection point, relative to the cuboid face. We repeat this process using the y-axis vector in place of the x-axis vector. This gives us the y-coordinate. However, we must use these coordinates to index into a texture that could have any width or height. To use these coordinates, we must scale them to the range zero to one inclusive. To this, we divide the magnitude of the x-coordinate by the magnitude of the x-axis, and we divide the magnitude of the y-coordinate by the magnitude of the y-axis. These values are then returned to the calling function. Figure 4.18 shows a code fragment from the function `coords_to_x_y`. This is the implementation of the process described in this paragraph.

```
x_axis = plane.lower_right - plane.lower_left
y_axis = plane.upper_left - plane.lower_left
axes_intersection = intersection_point - plane.lower_left

x_projection = ( dot(axes_intersection , x_axis) / dot(x_axis , x_axis) ) * x_axis
y_projection = ( dot(axes_intersection , y_axis) / dot(y_axis , y_axis) ) * y_axis

x_idx = magnitude(x_projection) / magnitude(x_axis)
y_idx = magnitude(y_projection) / magnitude(y_axis)
```

Figure 4.18:Transforming global point coordinates to relative coordinates on a cuboid face.

The cuboid texture colour function receives the scaled coordinates from `coords_to_x_y`. It uses them to calculate the indices used to access the colour data in the cuboid's texture. A texture is an array. To calculate the column index, we multiple the scaled x-axis coordinate by the array width. To calculate the row index, we multiple the scaled y-axis coordinate by the array height. These values can be floating-point numbers. To convert these values to integers we use the `floor` function. This removes the decimal component. This may bring the indices value to zero. As array indexing starts from one, we must convert any zero values to one. To do this, we use the `max` function. This guarantees that indices will be at least one. However, we cannot use these indices yet. The coordinate system used by images and textures is not the same as the Cartesian system we used to calculate our coordinates. The origin in images is the top left corner. Row indices get larger as we move down the y-axis. This is the opposite to the Cartesian y-axis. We must transform our y-coordinate to be compatible with our texture. To do this, we subtract the index from the height of the image. We must then add one back onto the index. This is because the index could equal the height of the texture. This means that subtracting the coordinate produces a value in the range of zero to the image height minus one. We must add one to the value to shift the range to be compatible with Julia's indexing system. We then use the indices to access the relevant colour data in the texture.

There is one more step to complete before we can return the colour data to the calling function, `colour_pixel`. Textures are images that are supplied by the user. Pixels in these images can have different alpha channel values. This represents a colour's transparency, and is the fourth attribute in the `RGBA` datatype. The value zero represents a completely transparent pixel. A value of one represents a completely opaque pixel with no transparency. If the alpha channel of the colour data we accessed is less than one, the user should be able to see some of the shape's colour underneath. If the pixel's alpha channel has a value of less than one, we alter the colour data before it is returned. The texture colour is multiplied by pixel's alpha value. The shape's colour is multiplied by one minus the alpha value. These two new colours are added together. All of this is accomplished using the overloaded mathematical operators. This new colour is a combination of the texture colour and the colour of the shape underneath. The amount of each of the original colours present is based on the pixel's transparency value. If the alpha value is one, the colour of the shape underneath cannot be seen. Figure 4.19, below, shows a code fragment from the cuboid texture colour function. This function is found in the file *render.jl*. The function `coords_to_x_y` is found in the file *intersections.jl*.

```
x_axis_scaled, y_axis_scaled = coords_to_x_y(cuboid, face, intersection_point)
width_idx = floor( Int, min( (width(plane.texture) * x_axis_scaled),width(plane.texture) ) )
height_idx = floor( Int, min( (height(plane.texture) * y_axis_scaled),height(plane.texture) ) )
width_idx = max(width_idx, 1)
height_idx = max(height_idx, 1)

return_colour = plane.texture.map[ (height(plane.texture) + 1) - height_idx , width_idx ]
if return_colour.alpha < 1.0
    return_colour = ( return_colour * return_colour.alpha ) +
                    ( cuboid.colour * (1 - return_colour.alpha) )
end

return RGBA( return_colour.r, return_colour.g, return_colour.b, 1.0 )
```

Figure 4.19:Calculating texture indices.

The sphere texture function works in precisely the same way as the cuboid texture function. The only difference is the function it calls to calculate the scaled coordinates used to determine the texture indices. Spheres do not have a flat, two-dimensional face. We cannot construct an x and y-axis as we did for the cuboid face. Instead, we must calculate the latitude and longitude of the intersection point, relative to the sphere.

These values can then be used to calculate the texture indices. This is accomplished by the function `coords_to_lat_lng`.

The mathematics used to calculate the latitude and longitude of the intersection point requires the sphere to be located at the origin. Similar to the cuboid texture function, the sphere must be scaled to unit length. However, for spheres we must consider rotation. A cuboid can be defined by the positions of its vertices. These points can be rotated around the centre point of the cuboid, if the user wishes. The new positions of the vertices are saved, and we need not be concerned with the cuboid's rotation again. However, a sphere has no vertices. It has only a centre point. If the sphere is rotated, the centre point is unaltered. We have no way of determining the sphere's rotation unless we store it. If we were to calculate the latitude and longitude of the intersection point at this stage, that would give us the texture information for an unrotated sphere. First, we must check the sphere's rotation data. We must rotate the intersection point the correct amount, around the relevant axes. The rotation point must actually be moved in the opposite direction to the sphere's rotation. For example, if the sphere is rotated clockwise around the y-axis, the intersection point has moved anticlockwise, relative to the texture. Therefore, instead of rotating the sphere clockwise, we can rotate the intersection point anticlockwise, around the centre of the sphere. We must store this new position without overwriting the original position of the intersection point. The point's position has changed relative to the sphere texture. However, its position in the scene had not changed. The point's original position is still the one used when calculating reflection, refraction, and lighting information. Figure 4.20 shows the implementation of `coords_to_lat_lng`. Subtracting the sphere centre from the intersection point gives us the point if the sphere was centred at the origin. This vector is divided by the sphere radius. This moves the point to the position it would have on a unit sphere. Next, we check the rotation information stored in the `Sphere` instance. If any of the rotation values are greater than zero, we apply the relevant rotation to the intersection point. We apply the negated rotation values. This is achieved with a call to `rotate_on_axis`. It is important to note that spheres store their rotation data in radians. This means that only one conversion from degrees to radians is required for a rotated sphere with a texture. Otherwise, a conversion would be required every time `coords_to_lat_lng` is called for that sphere. It is also important to note that shapes are rotated clockwise around an axis, looking in the axis' positive direction.

Once the final coordinates have been calculated, we can transform them to a scaled latitude and longitude. As we have transformed the intersection point so it is on the surface of a unit sphere, we can use inverse trigonometric functions to calculate these values. Latitude is the angle between the horizontal plane and the intersection point vector. This is shown in Figure 4.20. It is denoted by the Greek letter theta (θ). As the intersection point is on the surface of a unit sphere, the vector to that point is guaranteed to have a magnitude of one. This vector represents the hypotenuse on a right triangle constructed using the intersection point, the origin, and a point on the horizontal plane, as the vertices. As the hypotenuse has a magnitude of one, we can use the sine and cosine functions to directly calculate the magnitudes of the other sides of the triangle. This is similar to the mathematics used to calculate information about the viewing plane in Section 4.4.1. The difference here is that we know the length of the side of the triangle. We are trying to calculate the latitude angle. Therefore, in this situation we use an inverse trigonometric function. We can calculate the angle using the arcsine function. Figure 4.20 also demonstrates this. The length of the side opposite θ is simple to calculate. It is the y-coordinate of the intersection point, as it represents the distance from the horizontal plane. We are not taking its absolute value. We are allowing the side to have a negative length. We now need to scale this angle to the range zero to one, inclusive. The arcsine function produces a result in the range $-\frac{\pi}{2}$ to $\frac{\pi}{2}$, inclusive. To scale this to our desired range, we add $\frac{\pi}{2}$ to θ, then divide by π. The value of θ can be no greater than $\frac{\pi}{2}$. If we add $\frac{\pi}{2}$ to $\frac{\pi}{2}$, we get $\frac{2\pi}{2}$, which simplifies to π. And, of course, $\frac{\pi}{\pi}$ has a value of one. The value of θ can be no less than $-\frac{\pi}{2}$. If we add $\frac{\pi}{2}$ to $-\frac{\pi}{2}$, we get zero. Of course, $\frac{0}{\pi}$ has a value of zero. This is why our equation for scaling the latitude angle works. Once returned, the scaled value is used by `texture_colour` to calculate to row index value required to access the correct texture data.

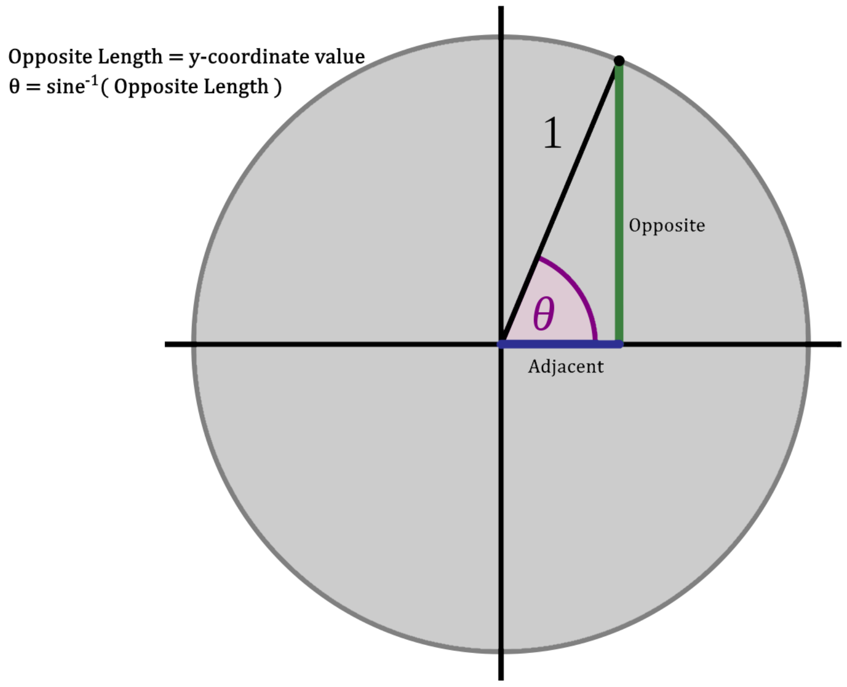
Opposite Length = y-coordinate value
θ = sine^{-1}(Opposite Length)
1
Opposite
θ
Adjacent

Figure 4.20:Using trigonometry and the unit sphere to calculate latitude.

The next step is to calculate the longitude of the intersection point. Latitude is the angle between the horizontal plane along the x and y-axes, and the vector from the origin to the intersection point. This can then be used to calculate to column index value required to access the correct texture data. To do this, we use a variant of the arctangent function that takes two arguments. In Julia, the `atan` function is overloaded to support the two-argument variant. It calculates the rotation of the intersection point around the y-axis. To do this, we must project the intersection point onto the horizontal plane. We can then create a two-dimensional, right triangle. We use the projection of the intersection point, the origin, and a point along the x-axis to construct the triangle. Our goal is to calculate the angle between the x-axis and the vector to the projection point. In the implementation, this angle is denoted by the Greek letter phi (ϕ). However, as the intersection point has been projected onto the horizontal plane, we cannot guarantee that the hypotenuse has a length of one. This will only be true when the intersection point's y-coordinate is zero. This means that the intersection point is already on the horizontal plane. However, we can calculate the lengths of the adjacent and opposite sides. The length of the adjacent side is simply the value of the projection point's x-coordinate. The length of the opposite side is the value of the point's z-coordinate. The angle ϕ is simply the arctangent of the length of the opposite side divided by the length of the adjacent side. However, this approach loses some vital information. The sign associated with the tangent function depends on the quadrant of the unit circle that the right triangle is in. This can easily be calculated by using the signs of the x and z-coordinates. This information is lost when the values are divided. For example, dividing two negative numbers produces a value with the same sign as dividing two positive numbers. This is why the two-argument version of the arctangent function is used. It first uses the sign information of its arguments to determine which quadrant the right triangle is in. It then divides the divides the first argument by the second argument and calculates the arctangent of the result. It then adjusts the sign of the result as necessary before returning the value. The values will be in the range $-\pi$ to π, inclusive. As before, we must scale this value to the range zero to one, inclusive. This is done by adding π to ϕ, then dividing by 2π. The maximum value of ϕ is π. This means that the maximum value of $\phi + \pi$ is 2π. The value of $\frac{2\pi}{2\pi}$ is one. The minimum value of ϕ is $-\pi$. This means that the minimum value of $\phi + \pi$ is zero. The value of $\frac{0}{2\pi}$ is zero. This is why our equation for scaling the longitude angle works. This value can be used to calculate the column index for accessing the sphere's texture data. However, the texture image will be flipped horizontally. To avoid this, we subtract the value from one. This reflips the texture. We have now calculated the scaled latitude and longitude. The values are returned to `texture_colour`, which uses them to calculate the texture indices. Figure 4.21 shows the implementation of `coords_to_lat_lng`.

```
point_translated_scaled = (intersection_point - sphere.centre) / sphere.radius

if abs(sphere.x_rot) > EPSILON
    point_translated_scaled = rotate_on_axis( point_translated_scaled, -sphere.x_rot, X_AXIS )
end
if abs(sphere.y_rot) > EPSILON
    point_translated_scaled = rotate_on_axis( point_translated_scaled, -sphere.y_rot, Y_AXIS )
end
if abs(sphere.z_rot) > EPSILON
    point_translated_scaled = rotate_on_axis( point_translated_scaled, -sphere.z_rot, Z_AXIS )
end

φ = atan( point_translated_scaled.z, point_translated_scaled.x )
θ = asin( point_translated_scaled.y )
lat = ( θ + π/2 ) / π
lng = 1 - (φ + π) / (2π)

return (lat, lng)
```

Figure 4.21:Implementation of latitude and longitude calculation.

4.6.2: Texture Modes & Storage

The ray tracing module provides two methods to apply textures to cuboid objects. The first is tile mode. This mode applies the same image data to each cuboid face. The second mode is cube-map. This mode applies data from different parts of a texture to each cuboid face. When creating a cuboid with a texture, an integer ID can be passed to indicate the intended texture mapping mode. Two integer constants are provided to allow users to choose their preferred method: `CUBE_MAP` and `TILE`. If no integer ID is provided, tile mode is used.

4.6.2.1: Tile Mode

Tile mode is the simpler of the two texture mapping methods. Images can be used in their regular layouts as textures in tile mode. The image is loaded into an array, using the `load` function from the Julia module, `Images`. In tile mode, the texture for each face of a cuboid is a reference to this image. This means that every face has the exact same texture data. Depending on the dimensions of the cuboid, the texture may seem more stretched or squashed on different faces. It is the user's responsibility to use appropriate dimensions when creating the cuboid.

4.6.2.2: Cube-Map Mode

Using cube-map mode is a more involved process for users. They must ensure that the layout of the image is appropriate. Cube-map mode must determine which parts of the texture should be applied to each face. To do this, it uses the dimensions of the cuboid. As such, the user should make sure the dimensions of the texture image are appropriate. For example, if the cuboid's front face is twice as long as its sides, this should be reflected in the texture image. Some edges will be duplicated in the texture image. Duplicated edges should have the same length. If any of these constraints are broken, the image will not be correctly applied to the cuboid.

Figure 4.22 shows how texture images should be laid out to be used in cube-map mode. By the definition of a cuboid, some edges will have the same length. Some edges will also be duplicated in the texture image. All edges are labelled either x, y, or z. Edges with the same label should have the same length. The diagram also shows the lengths of various points along the edge of the image. These lengths are expressed in terms of the side lengths, x, y, and z. This diagram was made using image editing software. It is based on a hand drawn diagram I created on graph paper. The diagram was extremely useful when implementing the calculations necessary for cube-map mode. The length markings along the side of the diagram contain all the information we need to create scaled coordinates for the texture. For example, Figure 4.22 shows that the texture data for the left face starts at the point $(\mathrm{z}, 0)$. It continues to the point $(\mathrm{z}+\mathrm{y}, \mathrm{z})$. To scale these coordinates to the range zero to one, we simply need to divide by the dimensions of the image. In scaled coordinates, the texture for the left face of the cuboid extends from point $(\frac{z}{2z+y}, 0)$ to $(\frac{z+y}{2z+y}, \frac{z}{2z+2x})$. We can then multiply the relevant indices by the height and width of the texture image. This gives us the indices for the segment of the texture that should be used for the left face. However, before we can evaluate the values of the indices, we must determine the values of x, y, and z. These values are simply the lengths of the cuboid edges. The user must provide these values when using a cuboid constructor.

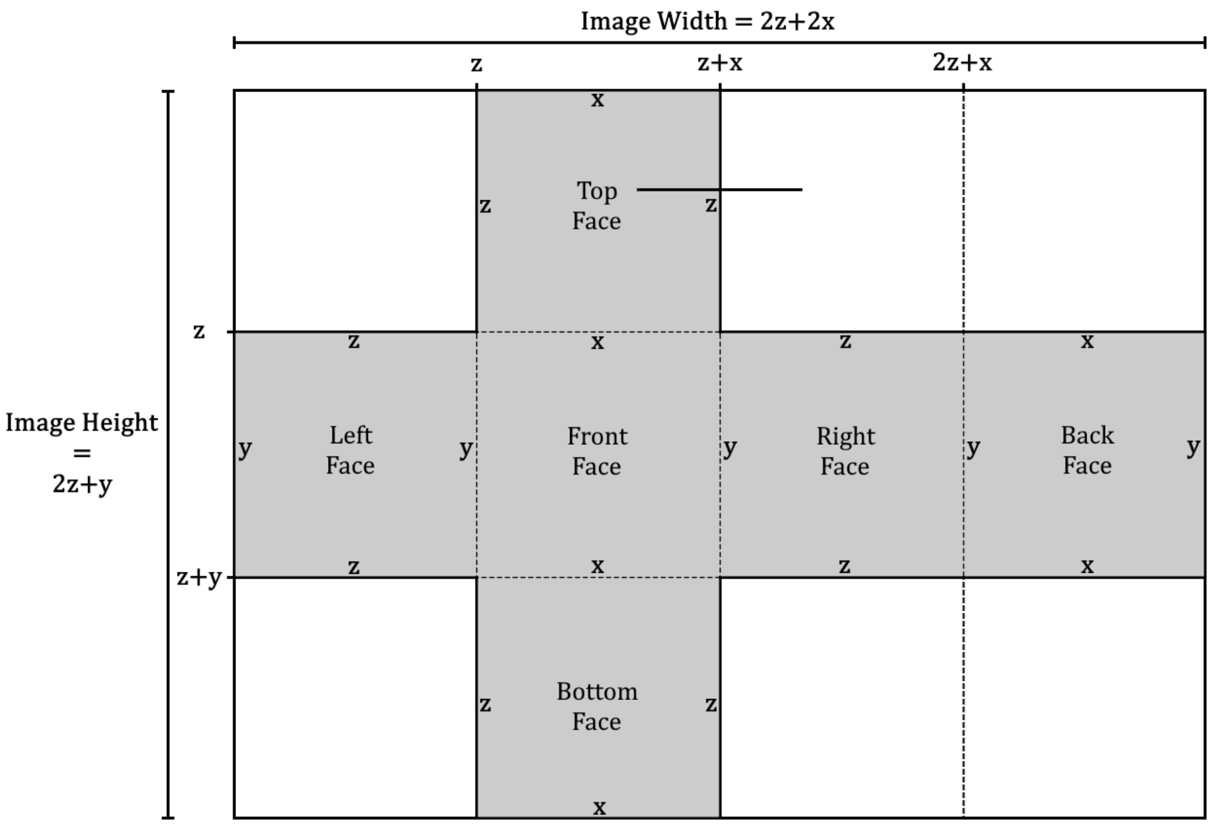

Figure 4.22:Template for cube-map texture images.

Figure 4.24 shows a code fragment from the cuboid constructor. This fragment creates textures for each cuboid face, according to the process detailed above. The variables `len`, `wdh`, and `hgt` contain the dimensions of the cuboid. We use these variables to calculate the dimensions of the texture in terms `len`, `wdh`, and `hgt`. These represent the lengths z, x, and y , respectively, from Figure 4.22. Next, we calculate the scaled positions of all the marked points along the edges of the diagram. The points along the vertical edge are represented by the variables `y_scale_1` and `y_scale_2` . The points along the horizontal edge are represented by the variables `x_scale_1`, `x_scale_2`, and `x_scale_3`. These values are then multiplied by the dimensions of the image to get their corresponding positions in the texture image array. These positions are used to create textures that reference the relevant part of the image array. Figure 4.23, below, shows an example of an image saved in the cube-map format. It is designed to look like a storage chest. Inspiration was taken from a Minecraft texture.

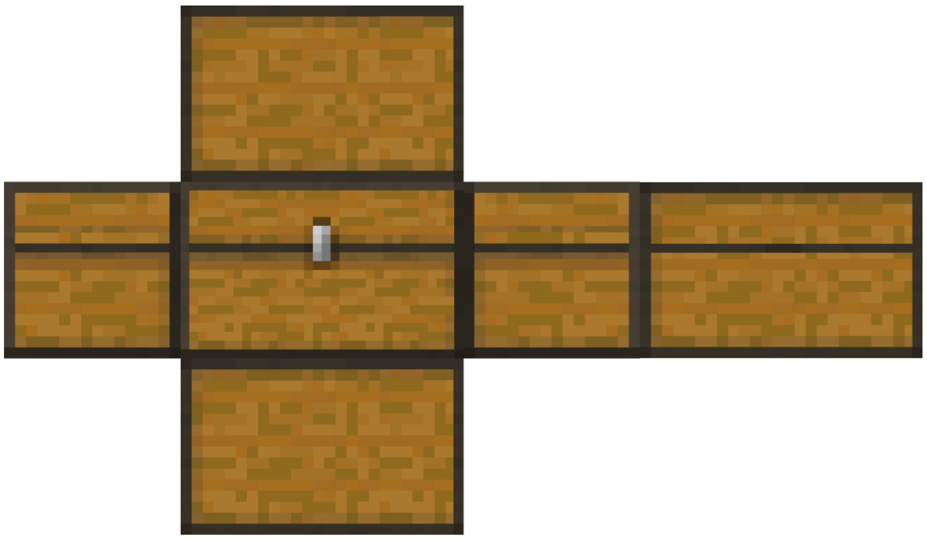

Figure 4.23:Texture image stored in the cube-map format.

```
cube_map_height = 2 * len + hgt
cube_map_width = 2 * len + 2 * wdh

y_scale_1 = len / cube_map_height
y_scale_2 = ( len + hgt ) / cube_map_height

x_scale_1 = len / cube_map_width
x_scale_2 = ( len + wdh ) / cube_map_width
x_scale_3 = ( 2 * len + wdh ) / cube_map_width

y_pos_1 = floor( Int, height(texture) * y_scale_1 )
y_pos_2 = floor( Int, height(texture) * y_scale_2 )

x_pos_1 = floor( Int, width(texture) * x_scale_1 )
x_pos_2 = floor( Int, width(texture) * x_scale_2 )
x_pos_3 = floor( Int, width(texture) * x_scale_3 )

texture_list[LEFT] = Texture( texture.map[ y_pos_1+1:y_pos_2 , 1:x_pos_1 ] )
texture_list[RIGHT] = Texture( texture.map[ y_pos_1+1:y_pos_2 , x_pos_2+1:x_pos_3 ] )
texture_list[FRONT] = Texture( texture.map[ y_pos_1+1:y_pos_2 , x_pos_1+1:x_pos_2 ] )
texture_list[BACK] = Texture( texture.map[ y_pos_1+1:y_pos_2 , x_pos_3+1:width(texture) ] )
texture_list[TOP] = Texture( texture.map[ 1:y_pos_1 , x_pos_1+1:x_pos_2 ] )
texture_list[BOTTOM] = Texture( texture.map[ y_pos_2+1:height(texture) , x_pos_1+1:x_pos_2 ] )
```

Figure 4.24:Creating cube-map textures from the main texture image.

4.7: Transparency & Refraction

This section details the implementation of transparent and refractive surfaces. First, it focuses on the general implementation, which covers the majority of interactions with rays. The second subsection deals with the more complex phenomena of total internal refraction (TIR) and the Fresnel Effect.

4.7.1: General Implementation

When light is cast on a transparent surface, a portion of the light ray will pass through the it. This means that the ray might also interact with whatever is behind the transparent surface. Any other objects the ray interacts with will affect the colour that can be seen from the light ray. To simulate this, when a transparent shape has been intersected, a new ray is created. This new ray then continues into the scene, where it can intersect other shapes and surfaces. Another approach would be to store and sort all of the various intersections a ray makes. Then, if it were to intersect a transparent surface, there would be no need to create a new ray. We could simply examine the colour and transparency information stored for in the next nearest intersection. This approach works well for shapes that are transparent. However, the shapes in this module can also be refractive. This means that the path of the ray can be altered and bent to travel in a different direction. In such cases, storing and sorting intersection data is no longer appropriate. It also wastes computation cycles. This is why a new ray must be spawned.

When an object is intersected, it is first necessary to check if it is transparent. To do this, we simply check if the object’s transparency value is greater than zero. If so, a new ray is created and is cast into the scene via a recursive call to `colour_pixel`. However, before the ray is cast, we must perform some calculations. The first step is to calculate the origin of the new ray. At first glance, it may seem logical to use the intersection point of the original ray as the origin. However, there is a familiar issue that makes this approach inadequate. Again, we must deal with the limited precision of floating-point numbers. When testing for intersections, we are only interested in those that occur in front of the ray origin. These are the intersections with positive t-values. Creating a ray with its origin directly on a surface can interfere with this process. Due to the limited accuracy of floating-point numbers, such a ray could intersect the surface from which it starts. This means that if a ray intersects a transparent surface, the newly created ray could intersect this surface again multiple times. This does not accurately simulate the passage of light through a transparent surface. It can also cause an effect known as “acne”. Acne is when a surface has blemishes that are the incorrect shade. The surface looks “fuzzy” instead of having a smooth solid colour. An example of this is shown in Figure 4.31 in Section 4.9. The `closest_intersection` function does have the option to ignore intersections with a given shape. However, we cannot use this functionality, as the new ray can still intersect the shape at a different point on its surface.

The solution is to move the origin of the new ray a very small amount. The amount should be as close to the lowest possible value that can be used to represent the position of an intersection. Through experimentation, I have found this lowest value that completely avoids the acne effect to be ten times the epsilon value of the `Float64` datatype. The next step is to calculate the direction in which the ray origin should be moved. We do not want to move the point away from the surface at an unusual angle. We want to move it in a direction perpendicular to the intersection point. To achieve this, we can move the point in the direction of the normal to the intersection point. Of course, every intersection point will have two normals, both having opposite directions. We know that the new ray should move away from the camera. This is because it is passing through the transparent surface, and there is no refractive index that could bend a light ray to travel back towards the origin of the original ray. This means that we must move the point in the direction of the normal that points away from the origin of the original ray. If we moved the point back towards the origin of the original ray, we would actually be guaranteeing the new ray would intersect the transparent surface again. The normal of the original intersection point is calculated within the `colour_pixel` function. It does this via a call to the function `normal`. This function always returns the normal that points back towards the side the origin of the intersecting ray is on. We can simply negate this normal vector to find the direction we must move the intersection point to find the origin of the new ray. To do this, we multiply the vector by ten times the epsilon value of the `Float64` datatype. This is then subtracted from the intersection point. This is done in the function `transparency_colour`. This function also determines how the ray is refracted and alters its direction appropriately.

Next, we must determine how great an effect the new ray has on the colour of the intersected surface. This depends how transparent the surface is. A shape can have a transparency value between zero and one, inclusive. The greater this value, the more the newly created ray affects the colour of the shape’s surface. For example, a shape may have a transparency value of 0.7. The function `transparency_colour` will create a new ray and recursively cast it into the scene and return a colour. This colour will then be multiplied by 0.7. The colour will later be combined with other information, such as light levels and reflection information, to return a final colour for the relevant pixel.

The `transparency_colour` function is where most of the computation occurs. The function first checks the refractive index of the intersected shape. It also checks how many times the original ray that was cast from the camera has been refracted. Users can set a limit on how many times a ray can be refracted. This is important, as a ray could interact with many refractive objects. Users can also set refractive indices to values not found in the real world. This means that light rays can get trapped inside a refractive object, unable to escape, due to TIR. Rays can be refracted recursively. This means that these situations could cause a stack overflow to occur. To avoid this, users should set a limit on the number of times a ray can be refracted. This limit has a default value of ten. If the shape is not refractive, the intersection point is moved in the relevant direction to avoid acne. This new point is used as the origin for a new ray. The direction of the ray is the same as the original ray. This new ray is cast into the scene via a recursive call to `colour_pixel`. The colour information determined by this ray is returned to the calling function.

If the intersected object is refractive, and the refraction limit has not been reached, the ray is refracted, and the refraction count is incremented by one. This is done via a call to the function `refract_ray`. This function returns a new ray, as well as information about the angle between the original ray and the intersected surface. This information can be used to determine if the Fresnel effect should occur. The `refract_ray` function will also determine if TIR has occurred. If it has, the ray it returns is equal to `nothing`, no new ray is cast by `transparency_colour`, and the base colour information of the intersected shape is returned. The `transparency_colour` function also returns the variable `schlick_value`. More details on this and the Fresnel effect are given in the next subsection. The implementation of `transparency_colour` is shown below.

```
function transparency_colour( scene::Scene, ray::Ray, normal::Vector_3D, intersection_point::Vector_3D,
                              idx::Int, side::Union{Plane, Nothing}, reflect_count::Int, refract_count::Int )

    refraction = side == nothing ? scene.shapes[idx].refraction : side.refraction
    schlick_value = 0.0
    if refraction > 0 && refract_count < REFRACT_LIMIT
        transparency_ray, cos_ = refract_ray(ray, normal, intersection_point, refraction)
        schlick_value = schlick(1, refraction, cos_)
        refract_count = refract_count + 1
    else
        transparency_ray = Ray( intersection_point - ( normal * (10 * EPSILON) ), ray.direction )
    end

    if transparency_ray == nothing
        return ( scene.shapes[idx].colour, schlick_value )
    end
    closest_intersection_value, closest_index, side = closest_intersection(scene, transparency_ray)

    if closest_index != nothing
        return_colour = ( colour_pixel( scene, transparency_ray, closest_intersection_value, closest_index, side,
                          reflect_count, refract_count ), schlick_value )
    else
        return_colour = ( background_colour( scene, transparency_ray.direction ), schlick_value )
    end

end
```
Figure 4.25: Implementation of the `transparency_colour` function.

While implementing the refraction system for the module, an important design decision had to made. Refraction occurs when light passes from one medium to another medium that has a different refractive index. To calculate how this affects rays of light, it is necessary to know both refractive indices. This means that if a ray originates inside a shape, we must know the refractive index of that shape. This can be difficult to determine. The module does not place restrictions on shapes being placed within other shapes. This means that the origin of a ray can be inside multiple shapes simultaneously. It would be necessary determine which shape is the innermost object. If an intersection were to occur, we would have to determine if it is leaving or entering a shape. The medium that is being entered and which is being exited affects the bending of the light ray. Calculating all of the necessary information can be extremely computationally expensive and difficult to implement. The other option is to assume that all shapes are hollow. Then we can set a base refractive index for the scene. Whenever refraction takes place, the refractive index of the medium being left is set to the base value of the scene. The refractive index of the medium being entered is the index of the intersected shape. This avoids many extra, complex calculations that would decrease the performance of the ray tracing implementation. The refractive index of air is approximately 1.0003. For the module implementation, I rounded this value to one. This is the base refractive index for scenes.

The direction of a refracted ray is calculated in the function `refract_ray`. To calculate the direction, we use Snell's law. Snell's law states that the ratio of the sines of the angles of incidence and refraction is equivalent to the reciprocal of the ratio of the indices of refraction. With certain combinations of refractive indices, Snell's law indicates that the sine of this angle is greater than one. This is not possible. Sine values must be between one and negative one, inclusive. When Snell's law indicates that the sine is greater than one, this means that TIR has occurred. This will become important in the next subsection. The equation for Snell's law is:

$$\frac{sin\theta_2}{sin\theta_1} = \frac{n_1}{n_2}$$

The angle of incidence created by the incoming ray is represent by θ_1. The angle of created by the refracted ray is represented by θ_2. The refractive index of the material being exited is represented by n_1. The refractive index of the material being entered is represented by n_2.

To calculate the angle at which a ray refracts, first we calculate the cosine of the angle created between the incoming ray and the surface normal. We can accomplish this using the dot product of the ray and the normal vector. The value of cosine must be between one and negative one, inclusive. However, because of the limited precision of floating-point numbers, value produced by the dot product operation can be very slightly outside this range. The `min` and `max` functions are used to correct this. We can rearrange Snell's law to show the sine of the angle of refraction is equal to $\frac{n_1}{n_2} \times sin\theta_1$. Using the Pythagorean identity of sine, we can rewrite this equation as to $\frac{n_1}{n_2} \times \sqrt{1 - cos^2\theta_1}$. This is helpful, as we have already calculated the cosine of the angle of incidence. Calculating square roots can be a computationally costly operation. As such, it is avoided where possible. We know that the sine of the angle should be less than or equal to one. This means that its square root must also be less than or equal to one. Any other value would result in a value for sine that is too large. Therefore, we can square both sides of the above equation to calculate $sin^2\theta$. We can then check if this value is greater than one. If it is, TIR has occurred. As such, no refraction takes place, and no ray is returned. The value `nothing` is returned instead, along with the cosine of the angle of incidence. If the $sin^2\theta$ is less than or equal to one, we calculate the direction of the refracted ray. To do this, we need to find the cosine of the angle of reflection. This is done using the cosine Pythagorean identity. This states that $cos\theta = \sqrt{1 - sin^2\theta}$. We have already calculated $sin^2\theta$ of the angle of refraction. We can use this to calculate its cosine. Unfortunately, in this instance the square root operation cannot be avoided. We use the cosine value to calculate the direction of the refracted ray. Its origin is moved away from the intersected surface to prevent any acne. This new ray is then returned, along with the cosine of the angle of incidence. The implementation of this process is shown in the figure below.

```
function refract_ray(ray::Ray, normal::Vector_3D, intersection_point::Vector_3D, refraction::Real)
    refractive_ratio = 1 / refraction
    cos_1 = dot( unit_vector(-ray.direction), normal )
    cos_1 = min( cos_1, 1.0 )
    cos_1 = max( cos_1, -1.0 )
    sin_2 = (refractive_ratio * refractive_ratio) * ( 1 - (cos_1 ^ 2) )
    if sin_2 > 1.0
        return ( nothing, cos_1 )
    else
        cos_2 = √( 1.0 - sin_2 )
        direction = normal * ( refractive_ratio * cos_1 - cos_2 ) - unit_vector(-ray.direction) * refractive_ratio
        return ( Ray( intersection_point - ( normal * (10 * EPSILON) ), direction ), cos_1 )
    end
end
```

Figure 4.26: Using Snell's law to construct a refracted ray.

4.7.2: Total Internal Reflection & The Fresnel Effect

TIR and the Fresnel effect are phenomena seen in refractive objects. When light enters a refractive object at a sufficiently steep angle, it is reflected instead of refracted. This is known as TIR. The angle required to produce this effect depends on the refractive index of the object being entered. Light entering a refractive object can be both refracted and reflected simultaneously. The amount of light refracted compared to the amount reflected depends on the angle at which the light enters the object, as well as the object's refractive index. This is known as the Fresnel effect. This effect and TIR are both simulated by the ray tracing module.

The Fresnel effect is described by a set of equations, known as the Fresnel equations. These equations are not just used to describe the reflection and transmission of light. They are used to describe electromagnetic radiation in general. The equations involve many different components and are extremely complex. The ray tracing module does not implement the Fresnel equations. In 1994, Christophe Schlick developed a highly accurate, but efficient, approximation of the specular reflection component of the Fresnel equations (Schlick, 1994). The equation for the Schlick approximation is:

$$R_0 + (1 - R_0)(1 - cos\theta)^5$$

R_0 is equal to :

$$\left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$$

The symbol n_1 represents the refractive index of the medium being exited. The symbol n_2 represents the refractive index of the medium being entered. The angle between the incoming light ray and the normal to the intersected surface is represented by θ.The value produced by the equation will be between zero and one, inclusive. This represents the proportion of light that is reflected. However, before we use the Schlick approximation we must check for TIR. This can be done using Snell's law. On this occasion, we do not need to use the equation to find the angle of refraction. We only need to check if the sine of the angle of incidence is greater than one, which indicates that TIR is occurring.

The implementation of the Schlick approximation with Snell's law is shown in Figure 4.27. First, we examine the refractive index of the material from which the light ray is entering. If this is greater than the refractive index of the material being entered, it is possible that TIR will occur. As such, we must use Snell's law to check if this is the case. We calculate the ratio of the refractive indices. We then calculate the sine of the angle of incidence. If it is greater than one, we know that TIR should occur. As such, there is no need to evaluate the Schlick approximation equation. We simply return the value one. However, if `index_a` is greater than `index_b`, we must change how we use the approximation. We must use the cosine of the angle of refraction, rather than the angle of incidence. This is calculated using the cosine Pythagorean identity.

```
function schlick(index_a::Real, index_b::Real, angle_cos::Real)
    if index_a > index_b
        refractive_ratio = index_a / index_b
        sin2 = (refractive_ratio * refractive_ratio) * ( 1 - (angle_cos ^ 2) )
        if sin2 > 1.0
            return 1.0
        end

        angle_cos = √( 1.0 - sin2 )
    end

    r = ( (index_a - index_b) / (index_a + index_b) ) ^ 2
    return r + (1 -r) * ( ( 1 - abs(angle_cos) ) ^ 5 )
end
```

Figure 4.27: The Schlick approximation with Snell's law.

The Schlick value is returned to `transparency_colour`. It is then returned to `colour_pixel`, along with the colour of the newly created ray. The greater the Schlick value, the more light is reflected and the less that is refracted. Therefore, the colour returned is multiplied by the shape's transparency value minus the Schlick value. If the shape's reflectivity value is higher than the Schlick value, the colour is simply multiplied by the transparency value. This is because, in this case, the transparency colour will be combined with a reflected colour that will be stronger than the colour reflected due to the Fresnel effect. If, however, the Schlick reflection is stronger than the base reflection from the shape, the shape's reflection value is updated to match the Schlick value. The transparency colour is initialised as black. Therefore, if the shape is not transparent, when the various colours are combined later, the transparency colour will have no effect.

```
transparent_colour = BLACK

if transparency > 0 && idx > 0
    transparent_colour, schlick_value =  transparency_colour( scene, ray, n, intersect_point, idx,
                                                              side, reflect_count, refract_count )
    transparent_colour = transparent_colour * ( transparency - (schlick_value > reflection ?
                                                schlick_value : 0) )
    reflection = max( reflection, schlick_value )
end
```
Figure 4.28: Using the Schlick approximation value to determine colour intensity.

4.8: Reflection

When a ray intersects a reflective surface, a new ray is created and recursively cast into the scene. When a shape is intersected, its reflectivity value is stored. If it is greater than zero, the function `reflection_colour` is called. The value stored can be altered before this check is made. This can happen if the shape is refractive, as TIR or the Fresnel effect may take place. The function `reflection_colour` first checks that the reflection limit for the intersecting ray has not been reached. Similarly to the refraction limit, this can be set by users to avoid infinite reflection loops. It also has a default value of ten. If the limit has not been reached, the intersecting ray is reflected off the surface it has intersected. This is done by reflecting its direction vector about the normal to the surface. It is very important that the normal is facing back towards the side from which the ray was cast. This is done via a call to the function `reflect`. This uses a simple vector formula to reflect the direction vector. The intersection point of the original ray is used as the origin of the reflected ray. We must move this point away from the surface, in the direction of the normal vector. This is to avoid the acne effect. Again, it is important that the normal is facing back towards the side from which the original ray was cast. The new ray is then cast into the scene, and its colour information is returned to the calling function. Figure 4.29 shows the implementation of the functions, `reflection_colour` and `reflect`. When the colour information is returned, it is multiplied by the reflection value mentioned earlier. This colour will later be combined with other colour information, such as the shape's base colour and the colour of any refracted rays. The reflected colour is initialised to black. Therefore, if no reflection occurs, the colour will have no effect when colour information is combined.

4.9: Lighting & Shadows

The final colour of a pixel is a combination of the base colour of the shape it intersects, any refracted and reflected light rays, the ambient light in the scene, and the various components of the Phong reflection model discussed in Section 3.3.2. The first step is to determine the level of shadow falling on the ray's intersection point. This is determined by the function `shadow_level`. Shadow level checks the transparency level of all the shapes between the intersection point and the scene's light source. Before this process is started, the intersection point must be moved away from the intersected surface. We achieve this by moving it along the normal to the intersection point. We must make sure that correct normal is used. This process is similar to that used for refraction and reflection. The difference is how we decide which direction is correct. To do this, we first

```
function reflect(v::Vector_3D, axis::Vector_3D)
    n = unit_vector(axis)
    return v - ( ( 2 * dot(v, n) ) * n )
end

function reflection_colour(scene::Scene, ray::Ray, normal::Vector_3D, intersection_point::
                           Vector_3D, idx::Int, reflect_count::Int, refract_count::Int)
    if reflect_count >= REFLECT_LIMIT
        return scene.shapes[idx].colour
    end

    reflect_vector = reflect( unit_vector(ray.direction), normal )
    reflect_ray = Ray( intersection_point + ( normal * (10 * EPSILON) ), reflect_vector )

    closest_intersection_value, closest_index, side = closest_intersection(scene, reflect_ray)

    if closest_index != nothing
        return colour_pixel(scene, reflect_ray, closest_intersection_value,
                            closest_index, side, reflect_count+1, refract_count)
    end

    return background_colour(scene, reflect_vector)
end
```

Figure 4.29: Reflecting a vector and casting it into the scene.

create a vector from the intersection point to the light source. The dot product of this vector and the surface normal is calculated. The module function to calculate normal vectors always returns a normal that faces to the side of the incoming ray's origin. If the dot product value is negative, it means that the angle is greater than ninety degrees. This means that the light is on the opposite side of the intersected surface to the origin of the ray. The intersection point will be in shadow, as it is on a part of the shape that is facing away from the light. In this situation, we want to consider the transparency of the intersected surface. The ray cast by `shadow_level` should be able to intersect the surface. As such, we move the intersection point along the surface normal that brings it to the side of the surface that is in shadow. If the dot product is positive, the light source is on the same side of the surface as the ray origin. As such, any rays cast to the light source should not intersect the surface. Therefore, we move the intersection point in the direction of the negated normal vector. Rays from an intersection point to a light source are called shadow rays. The adjusted intersection point will be the origin of our shadow ray. The code fragment below shows the process of calculating the ray origin. The variable `light_side_n` is the vector that points in the direction we should move the intersection point. If the dot product is negative, the variable is assigned the value of the normal vector. Otherwise, it is assigned the value of the negated normal vector. The variable intersection point is moved very slightly in the direction of the normal. The result is stored in `epsilon_point`, which will later be used as the origin of the shadow ray.

```
light_side_n = dot(n, -light_ray) < 0 ? n : -n
epsilon_point = idx > 0 ? intersect_point + ( light_side_n * (10 * EPSILON) ) : intersect_point
```

Figure 4.30: Determining the origin of a shadow ray.

If the intersection point is not moved away from the shape's surface, the acne effect could occur and reduce the quality of the image. Below is an image rendered using an early version of the ray tracing module. It suffers from acne. The snooker balls appear fuzzy. Bands of varying shades of the correct colour can be seen on some of the balls. The dark, fuzzy patches are caused by the top of the balls casting a shadow on itself. Acne tends to be more severe with spheres. The flat surfaces of cuboids are less susceptible to the issue. The green surface, on which the balls are resting, is a cuboid. It is not suffering from any acne related blemishes. However, it can still occur, particularly when rays intersect a cuboid at an acute angle. The higher the precision the floating-point datatypes used, the less likely acne is to occur, and the less severe it will be when it does occur. Greater precision also decreases the distance by which intersection points must be moved. The floating-point datatype with the greatest accuracy in Julia is `Float64`. It offers 64 bits of precision.

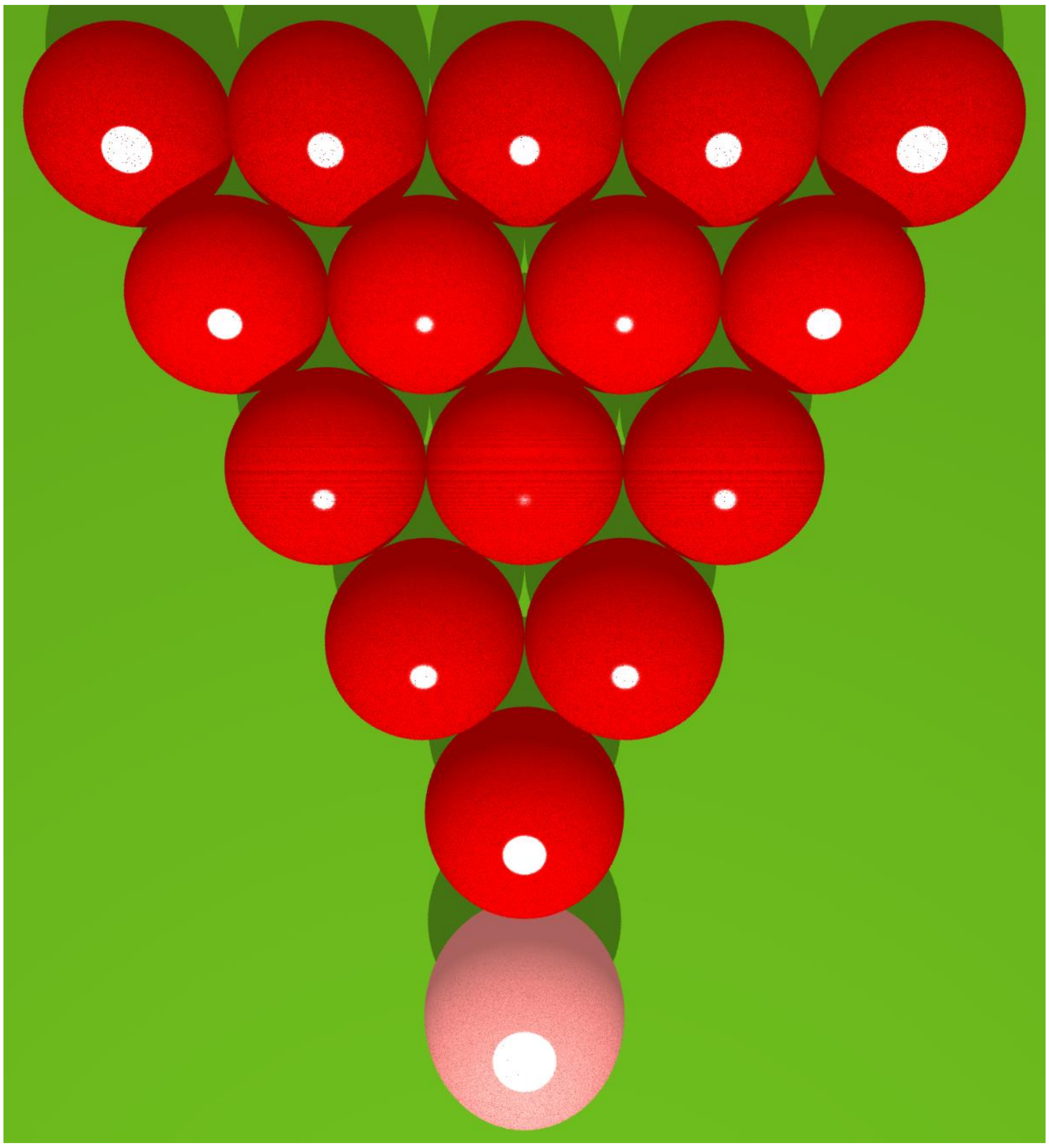
Figure 4.31: The acne effect.

Once we have determined the origin of the shadow ray, we can cast it into the scene. The origin point is passed to the function `shadow_level`. A vector to the light source is created by subtracting the ray origin from its position. This vector is converted to a unit vector. But first we calculate and store its magnitude. This very is important. The vector's magnitude will be used to determine if a surface casts a shadow on the intersection point. The shadow ray is then created. The variable `shadow` is also created and initialised to zero. This is used to combine the level of shadow cast by each shape. This is necessary as shapes can be transparent. Transparent shapes cast shadows that are less intense. We iterate through the shapes in the scene and check for intersections with the shadow ray. As usual, intersections with a negative t-values are ignored, as they are behind the shadow ray. We also compare the t-value to the magnitude of the light ray. If the value is less than the ray's magnitude, it occurs between the intersection point and the light. If it is greater than the ray's magnitude, the intersection happens on the opposite side of the light. In this case, the shape cannot cast a shadow on the intersection point. As such, it is ignored. The transparency of every intersected shape is subtracted from one, then added to the variable `shadow`. If the variable's value is ever greater than or equal to one, the intersection point is in complete shadow, and the value one is returned. Otherwise, `shadow` is returned. The implementation of the function `shadow_level` is shown below.

```
function shadow_level(scene::Scene, point::Vector_3D)
    if length(scene.shapes) == 0
        return 0
    end
    light_ray = scene.light.position - point
    mag = magnitude(light_ray)
    dir = unit_vector(light_ray)
    shadow_ray = Ray(point, dir)
    shadow = 0
    for i in 1:length(scene.shapes)
        intersection = intersect_values( scene.shapes[i], shadow_ray )
        if intersection == nothing
            continue
        end
        if intersection.t1 >=0 && intersection.t1 < mag
            if intersection.transparency1 < eps(Float64)
                return 1
            end
            shadow += (1 - intersection.transparency1)
            if shadow >= 1 return 1 end
        end
        if intersection.t2 >=0 && intersection.t2 < mag
            if intersection.transparency2 < eps(Float64)
                return 1
            end
            shadow += (1 - intersection.transparency2)
            if shadow >= 1 return 1 end
        end
    end
    return shadow
end
```

Figure 4.32: Calculating shadow intensity.

If the value returned by `shadow_level` is greater than zero, the intersection point is in some level of shadow. In this case, we ignore the diffuse and specular components of the Phong reflection model. We simply combine the base colour of the intersected shape, the colours returned by any reflected or refracted rays, and the shadow data for the intersection point. Before we combine this information, we compare the level of shadow cast on the intersection point to the ambient lighting in the scene. If the shadow level is darker than the ambient light, we instead use the ambient light when combining colour information. Ambient light is the base level of light in an environment. It has no single source and is often called background light. Shadows cannot be darker than the ambient light level. Combining the above information allows us to calculate a single colour value. This is returned by `colour_pixel` to the calling function. The calling function could be the driver function, `ray_cast`. It could also be a number of other functions, as `colour_pixel` is recursively called by functions within itself.

In the real world, if light could not reach a point in a straight line, it is not guaranteed that it is in shadow. Light can be reflected and refracted around obstacles to illuminate a point. However, this light could come from an infinite number of directions. To accurately simulate this effect, the number of rays that would need to be cast is prohibitive. It is not feasible to simulate this effect.

If the intersection point is not in shadow, we calculate the diffuse and specular components of the Phong model. First, we calculate the magnitude of the vector from the intersection point to the light source. The further from the light the intersection point is, the less intense the diffuse light will be. When light is shining perpendicularly on a point, the diffuse reflection will be at its maximum. When the light is parallel to the surface the point is on, there is in diffuse reflection. This corresponds with the values produced by the dot product operation. If two vectors are perpendicular, the dot product is one. If they are parallel, the dot product is zero. Angles between zero and ninety degrees will produce a dot product between zero and one. If we have reached this section of the code, we know that the light source is on the correct side of the intersected surface. Therefore, we do not have to be concerned with negative values for the dot product. The base colour of the intersected shape is multiplied by its diffuse value, which has been set by the user. The result is then multiplied by the dot product value. The distance to the light source should affect the colour's intensity. Therefore, we then divide by the magnitude of the vector from the intersection point to the light source. This provides us with the final colour of the diffuse component.

The next step is to calculate the specular component of the Phong reflection model. The model formula tells us that we must calculate the dot product of the vector from the intersection point to the camera and the reflection of the vector from the intersection point to the light source. These are the vectors *R* and *C* in Figure 3.2. These vectors should be normalised to unit vectors first. This value should then be raised to power of the intersected shape's shininess constant. The specular component is determined by multiplying the result of this calculation by the shape's specular reflection constant. This and the shape's shininess constant are both set by the user. The dot product of the vectors *R* and *C* will produce a value between zero and one. A value less than one will become smaller when raised to a power. The greater the power, the smaller the final value becomes. This means that the greater a shape's shininess constant, the smaller its specular highlights will be. However, this is only true if *R* and *C* are normalised first. Unfortunately, this is the basis of an error in the ray tracing module. The error was discovered after the deadline for the FYP product submission. As such, it was unable to be rectified in time. The vectors *R* and *C* are not normalised in my implementation. This causes specular highlights on surfaces to be larger than they should be. The implementation of the Phong model is shown in Figure 4.33, below. The code is shown with the error still present. All elements of the Phong model are combined to determine the colour of the intersection point. This colour is then returned by `colour_pixel`.

```
distance = magnitude(light_ray)
cos_light_ray_normal = dot(light_ray, n)
diffuse_light = ( base_colour * scene.shapes[idx].diffuse * cos_light_ray_normal ) / distance
reflect_ray = reflect(-light_ray, n)
cos_reflect_ray_cast_ray = dot( reflect_ray, -ray.direction )
if cos_reflect_ray_cast_ray < 0
    specular_light = RGBA(0, 0, 0, 1.0)
else
    specular_factor = cos_reflect_ray_cast_ray ^ scene.shapes[idx].shine
    specular_val = ( scene.light.brightness * scene.shapes[idx].specular *
                     specular_factor ) / distance
    specular_light = RGBA(specular_val, specular_val, specular_val, 1.0)
end
return ambient_colour + diffuse_light + specular_light + reflect_colour + transparent_colour
```

Figure 4.33: Combining the components of the Phong reflection model.

4.10: Concurrent & Distributed Execution

The ray tracing module can render images in a concurrent or distributed fashion. It does this by distributing the array of pixel data that represents the image over multiple worker processes. Each worker process is responsible for rendering its own allocated section of the image. The image sections are then returned to the main Julia process, where they are combined into a single array. Processes can be threads on a single machine, or they can be distributed among machines in a computing cluster. The module treats processes in the same way, regardless of whether they are on a local thread or a separate machine. The only difference is the setup required. It should be possible use both local threads and external cluster nodes in the rendering of an image. In the first subsection, I detail the how the rendering procedure uses processes to distribute its workload. The following two subsections then detail the setup required to add threads or cluster nodes to the worker process pool available to the render function.

4.10.1: Processes

The render function can be passed a vector of integers, via the keyword argument `worker_list`. These integers should represent the process IDs (PIDs) of the processes that the function should use to distribute its workload. The type of `worker_list` is a union of a vector and the value `nothing`. Its default value is `nothing`. The render function checks the value. If it is an integer vector, it will render its image using the processes specified. Otherwise, it will render the image in a serial fashion on the main Julia process. The Julia method, `workers`, returns the PIDs of all the available processes. Users can use this method to supply the PIDs to the render function. They do not have to use all available processes. They can use a subset of the processes. The easiest way to do this is to provide a slice of the vector returned by the `workers` function. For example, the statement, `workers()[1:3]`, will return the PIDs of the first three available worker processes. This is known as “slicing”. Using this method, users could run multiple instances of the render function asynchronously, providing each with a different subset of the available worker processes.

If a list of PIDs has been provided, the render function will iterate through them. It uses the `spawnat` macro from the `Distributed` Julia module to send instructions to each process. The instructions are to create local copies of all the necessary variables on that process. Complex compound datatypes are not copied. Rather, a distributed reference is copied to allow them to access the variable’s internal data that is on the main process. Each process also has its own copies of any global variables in the module. The copies on each process must be set to the relevant value. Users can set reflection and refraction limits. These are stored in global variables and must be set on each process. Calls to `spawnat` are run asynchronously. This means that subsequent code that uses variables on a worker process could be executed before those variables have finished being declared. This could cause non-deterministic runtime errors. To avoid this, we can use the `sync` macro. This macro will ensure that all calls to `spawnat` in a closed block of code are finished executing before the program continues. In this case, the closed block of code is a loop. The code used to copy variables to worker processes is shown in Figure 4.34, below.

```
@sync for id in worker_list
    @spawnat(id, Base.eval(Main, Expr(:(=), :scene, scene)))
    @spawnat(id, Base.eval(Main, Expr(:(=), :image_width, image_width)))
    @spawnat(id, Base.eval(Main, Expr(:(=), :image_height, image_height)))
    @spawnat(id, Base.eval(Main, Expr(:(=), :fov_rads, fov_rads)))
    @spawnat(id, Base.eval(Main, Expr(:(=), :h, h)))
    @spawnat(id, Base.eval(Main, Expr(:(=), :viewport_height, viewport_height)))
    @spawnat(id, Base.eval(Main, Expr(:(=), :viewport_width, viewport_width)))
    @spawnat(id, Base.eval(Main, Expr(:(=), :w, w)))
    @spawnat(id, Base.eval(Main, Expr(:(=), :u, u)))
    @spawnat(id, Base.eval(Main, Expr(:(=), :v, v)))
    @spawnat(id, Base.eval(Main, Expr(:(=), :horizontal, horizontal)))
    @spawnat(id, Base.eval(Main, Expr(:(=), :vertical, vertical)))
    @spawnat(id, Base.eval(Main, Expr(:(=), :lower_left_corner, lower_left_corner)))
    @spawnat id (global REFLECT_LIMIT = reflect_limit)
    @spawnat id (global REFRACT_LIMIT = refract_limit)
end
```

Figure 4.34: Copying variables to worker processes.

The next step is to distribute the image array among all of the processes specified by the user. The array is declared, and space allocated for it, earlier in the render function. Each member of the array is undefined when it is declared. This is perfectly adequate when rendering an image in a locally stored, regular array. However, distributed arrays are a different datatype to Julia’s base array. Distributed arrays are not compatible with undefined data. For that reason, we first fill each index of the image array with the colour black. The image array is then distributed among the relevant processes. We must remember to only use the processes specified by the user. They may wish to use the other processes for various purposes. The code to distribute the array is shown below, in Figure 4.35.

```
fill!(img, RGBA(0.0,0.0,0.0,1.0))
d_arr = distribute( img , procs=worker_list )
```

Figure 4.35: Creating a distributed array.

Once the array has been distributed, each process can begin to render its section of the image. An image section is rendered by iterating over each of its pixels in a nested loop. A ray is then cast through each pixel. We iterate over the list of processes with a loop. Instructions are sent to each worker process to begin looping over its pixels and perform the ray tracing procedure on its section of the image. Again, we must use the `sync` macro to ensure all processes have finished executing before the program can continue. Otherwise, the image array would be returned by the render function before it is complete.

To iterate over the image array, we need to know the range of indices in the array. This can easily be calculated using the width and height of each array section. However, these are the indices for the process' local section of the array. The indices of each pixel in the complete array are also required. This is because they are necessary to calculate the position of a pixel in the scene's coordinate system. If we use the local array indices, the pixels position will be calculated incorrectly. For example, each local array section will start with the position [1, 1]. This is the case regardless of whether that pixel corresponds to the position [1, 1] in the full array. This means that pixel data could be altered and overwritten multiple times. Some pixels will not be rendered at all. Fortunately, the function `localindices` returns the range of indices in an image section, relative to the complete image. We can use this information to calculate pixel positions in the scene's coordinate system. This is done slightly differently to the method used when an image is rendered in a serial fashion. The vertical axis in a Cartesian coordinate system has its values increase as it moves up the axis. The vertical axis in an image has its values increase as it moves down the axis. When the render function is executed serially, each pixel's position in the scene is calculated. Then its vertical image coordinate is subtracted from the image height. This has the effect of vertically flipping every pixel. This is why images are produced with the correct orientation, despite the differences in the coordinate systems. This method is not guaranteed to work correctly with a distributed array. The array sections could be split horizontally. This means that to flip the image vertically, we would need to assign pixel data to locations in the complete array to which the process does not have access. Pixels can only have their positions flipped relative to the process' local section of the array. If the local section is flipped, a result similar to the image in Figure 4.36, on the following page, will be produced.

The solution to this issue is to change the vertical offset we use to calculate a pixel's position in the scene's coordinate system. To do this, we simply need to subtract the offset from the image's height. However, we must make sure we use the height of the complete image, not of the local section. As Julia's indexing system begins with one, we must also add one the offset. We can then use this information to accurately calculate the pixel's coordinates. The code to determine a pixel's position relative to the complete array, calculate its position in the scene coordinate system, and cast a ray through that position is shown in Figure 4.37. Before returning the rendered image, we first need to convert it back to a regular array. This is done with the function `convert`, which is supplied by the module `DistributedArrays`. The image can then be returned to the user.

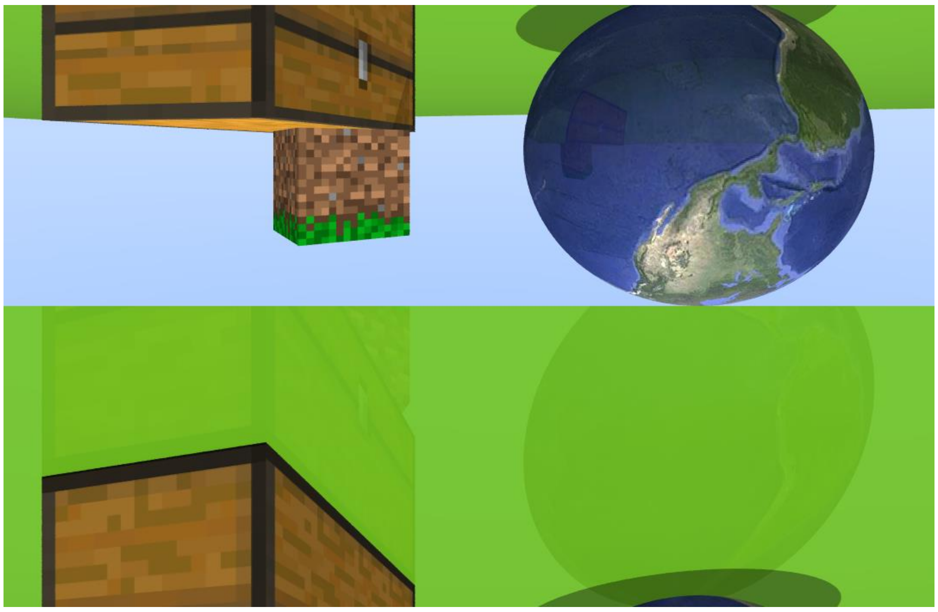

Figure 4.36: Result of confusing local array indices with global indices.

```
h_offset = localindices(d_arr)[2][1] + ( i - 1 )
v_offset = localindices(d_arr)[1][1] + ( j - 1 )
v_offset = abs(v_offset - image_height) + 1
hrzt_factor = h_offset / ( image_width - 1 )
vert_factor = v_offset / ( image_height - 1 )
ray = Ray( scene.camera.origin, ( lower_left_corner + (hrzt_factor * horizontal) +
                                  (vert_factor * vertical) - scene.camera.origin ) )
pixel_colour = ray_cast(ray, scene)
localpart(d_arr)[ j, i ] = RGBA( pixel_colour.r, pixel_colour.g, pixel_colour.b, pixel_colour.alpha )
```

Figure 4.37: Determining a pixel's true scene coordinates.

4.10.2: Processes On The Same Machine

Processes can be added by using the function `addprocs`. If we pass this function an integer greater than zero, the Julia environment will attempt to make that many processes available for use. If it cannot create as many processes as were requested, it will create as many as possible. It also returns a vector of the PIDs of the created processes. To use the `addprocs` function, the `Distributed` module must be imported. This must happen before any calls to `addprocs`. For the ray tracing module's rendering function to be able to use any of the processes, they must first be properly prepared. Each process must be instructed to import the module. This can be accomplished using the `everywhere` macro. This sends the specified instructions to each process. Once each process has imported the module, they can be passed to the rendering function. All processes used by the function must have the module imported. Otherwise, errors will be raised by Julia. The code to add local processes and import the ray tracing module on each process is shown in Figure 4.38. It is important that `addprocs` is used before importing the module using the `everywhere` macro. The code in Figure 4.38 should be placed at the beginning of any program that uses local processes with the ray tracing module. The file path in the include statement may need to be altered to be compatible from the current working directory. More information on details such as this is given in the Section 6.4. The environment variable `Sys.CPU_THREADS` contains the number of logical CPU cores on the machine.

4.10.3: Distributed Processes

Distributed processes can also be added using `addprocs`, as the function is overloaded. By passing it the correct parameters, it will attempt to add machines to a cluster, instead of creating local processes. The only required parameter is an array of tuples. Each tuple should contain either a string, or a string and an integer greater than zero. The strings represent the machines to be added to the cluster. They should contain the username and internet protocol address (IP) of the machine to be added. The format of the string can be seen in Figure 4.39. The variable `machines` contains the information of the machines that will be added to the cluster. The integer in the tuples indicates the number of local worker processes to be launched on that machine. If omitted, it will default to one. The keyword argument, `dir`, specifies the working directory to use on the added machines. We specify that the network topology is a master-worker architecture. Each worker node can independently render its own section of an image. There is no reason for them to communicate with each other. The keyword argument, `tunnel`, is set to true. This specifies that ssh tunnelling will be used to connect to the worker from the master process. Not all of these arguments are needed to add a node to a cluster. These flags were required to correctly set up the cluster I used to test the ray tracing module. General guidelines on the cluster requirements of the module and Julia are given in Section 6.4.

```
using Distributed
addprocs(Sys.CPU_THREADS)
@everywhere include("../src/trace.jl")
@everywhere using .Trace
using FileIO
```

Figure 4.38: Adding and preparing local worker processes.

```
using Distributed

machines = [ ("ubuntu@3.249.197.51",1),("ubuntu@34.248.8.28",1),
             ("ubuntu@34.240.33.89",1) ]
dir = "/home/ubuntu"
addprocs( machines; dir=dir, topology=:master_worker, tunnel=true )

@everywhere include("trace.jl")
@everywhere using .Trace
using FileIO
```

Figure 4.39: Adding and preparing cluster nodes.

Section 5: Results

This section considers the quality of the completed FYP product. It shows images that were rendered using the FYP ray tracing module. It examines the source code used to create these images to examine the usability of the interface. It also compares the effect of rendering images in a concurrent and distributed fashion on program runtimes. Finally, it compares the goals of the project against the final product.

5.1: Rendered Images

The first image I would like to show is a very basic scene, but uses much of the functionality of the module. It is shown in Figure 5.1, below. It is an image with a resolution of 3,840 pixels by 2,160 pixels (4K). Ten rays have been cast for each pixel. It is a reflective snooker table with twenty-two snooker balls. The yellow ball, however, is not visible from this angle. It also contains three opaque blocks and one transparent block. The pink snooker ball is also reflective. Subsequent images will focus on demonstrating specific functionalities of the module, but I would like to begin by showing a general image.

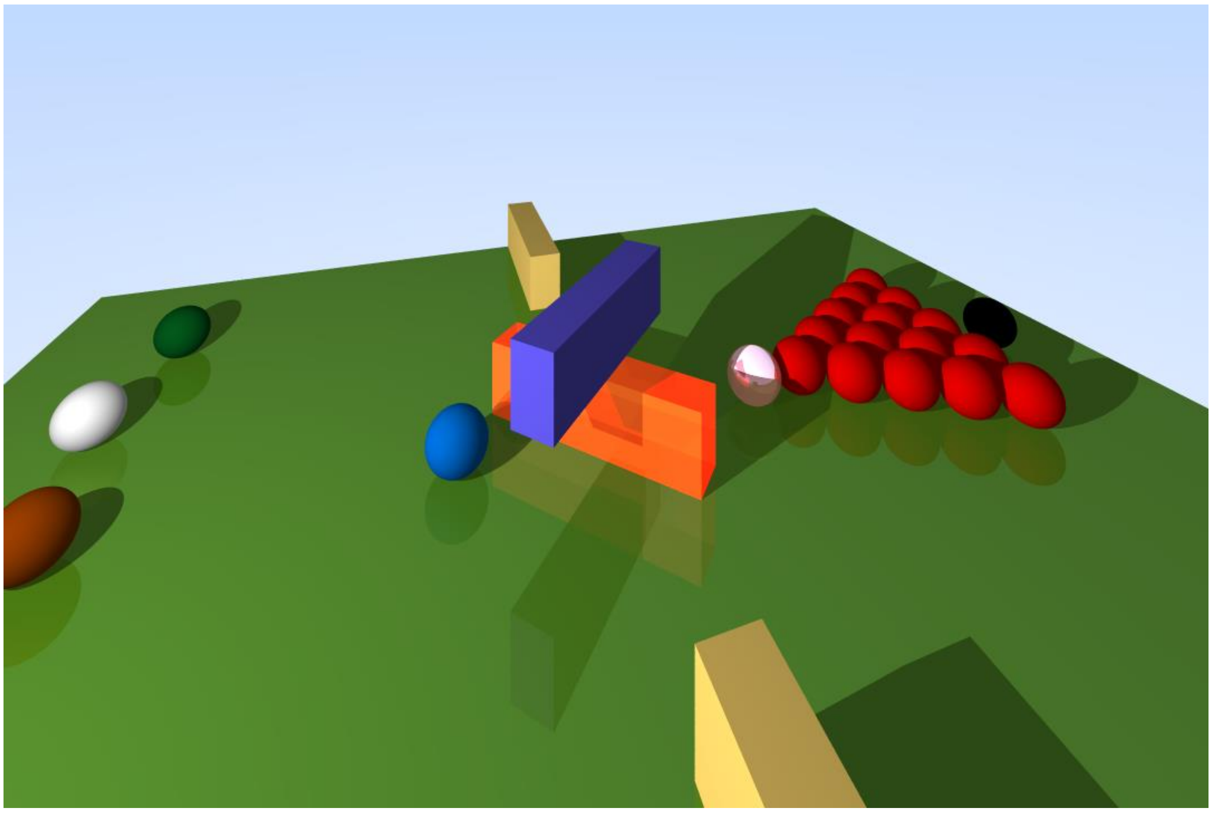
Figure 5.1: Render of a snooker table scene.

The next selection of images demonstrates the transparency and refraction functionality. Figure 5.2 shows four images. The upper left image is transparent red sphere that is also refractive. The upper right image shows a variation of this scene. The sphere is now completely transparent, and its index of refraction has been changed. The lower left image shows the same scene as the upper right image. However, the camera is now inside the refractive sphere. The final image shows the scene with a refractive cuboid in place of the sphere. Its colour is completely transparent. However, its transparency value is lower than that of the sphere. As such it appears darker. All four images have a resolution of 1,080 pixels by 720 pixels (720p), with ten rays cast for each pixel.

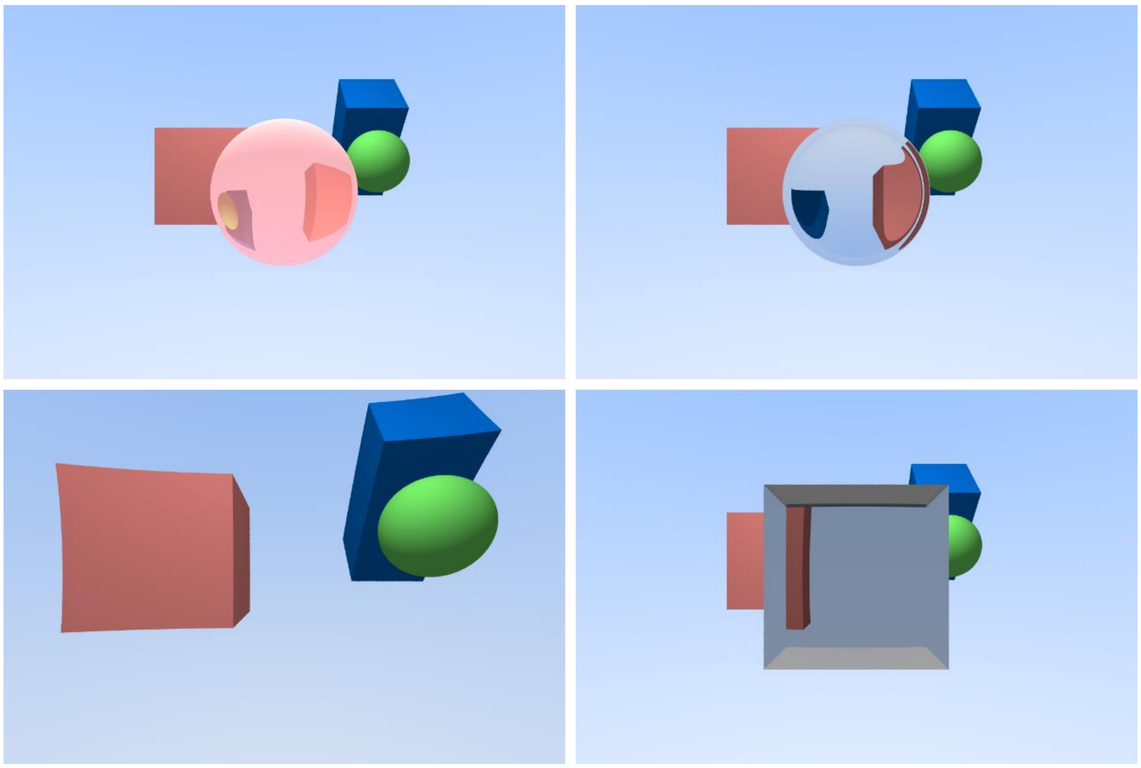

Figure 5.2: Transparency and refraction demonstration.

The next two images demonstrate the reflection functionality of the module. The left image shows a yellow block that reflects an image of the pink and red snooker balls. The pink ball is also reflective. The blocks reflections are also reflected by the ball, and vice versa. The right image shows a scene in which each ball is reflective. The pink ball is also transparent. The complex interaction of transparency and reflection is shown. This is discussed in more detail in later paragraphs. In each scene, only one face of the central block has been specified to be transparent. The other faces generate no reflections. The table is also reflective. You can observe that the intensity of its reflections is affected by shadows in the scene. The right image is in 720p. The left image is in 4K.

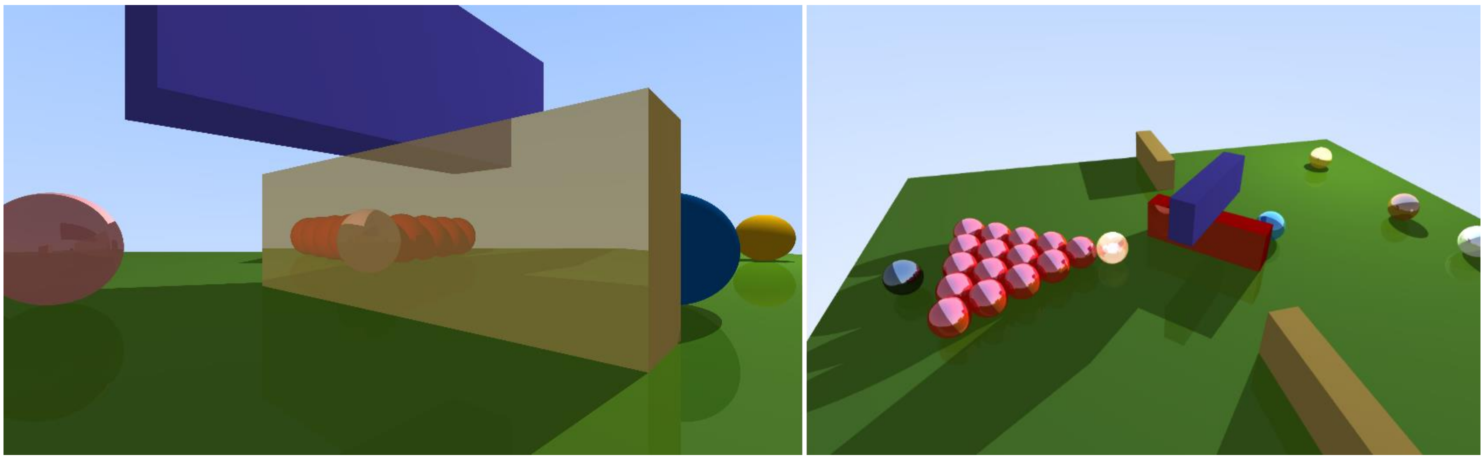

Figure 5.3: Reflection demonstration.

The two images in Figure 5.4 show the interaction of transparency and reflection. The scene is based on the scene shown in the left image of Figure 5.3, above. The colour of the bottom block has been changed to red. The block has also been made transparent. As a result, the blue ball is now entirely visible, as is another block that was hidden before. In the left image, the wide face, farther away from the camera is reflective. It shows the pink and red balls. The reflection is much fainter than before due to the transparency of the shape. In the right image, the block has an extra reflective face. The wide face, closest to the camera is also reflective. As a result, we can see two separate reflections of the pink and red balls. The effect of the extra reflective surfaces can also be seen in the reflections in the pink ball and the green table. The images also demonstrate how transparency affects shadows. The shadow cast by the opaque purple block is darker than the shadow cast by the transparent red block. The resolution of the images is 720 pixels by 405 pixels. This is the same aspect ratio as the other images, which is $\frac{16}{9}$. Ten rays were cast for each pixel.

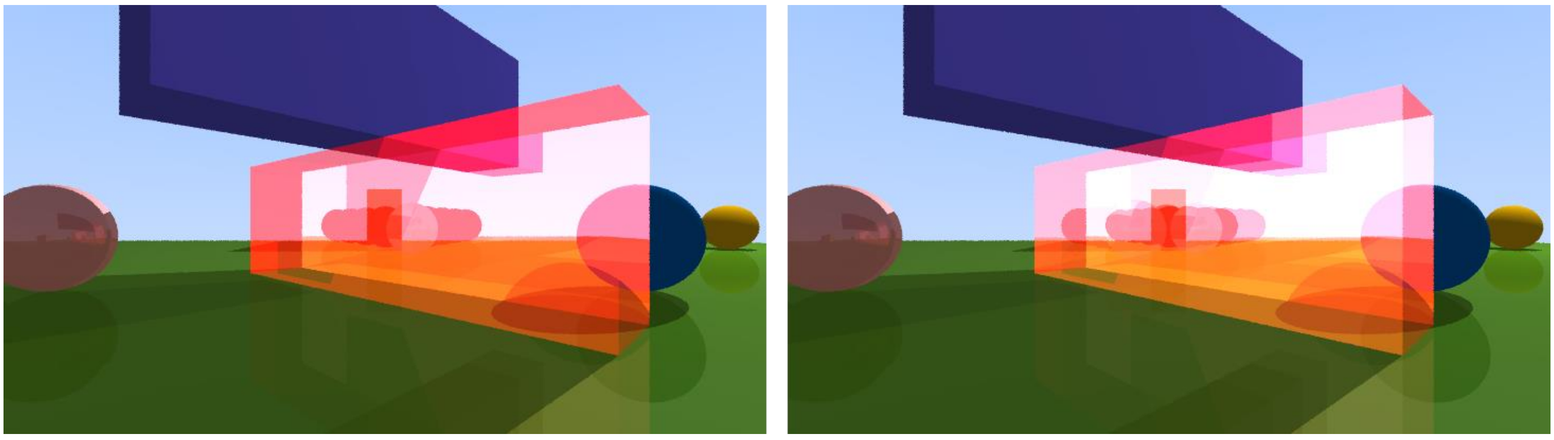

Figure 5.4: Interaction between transparency and reflection.

Figure 5.5 demonstrates the use of textures, and shows that they can be used with reflective, transparent, and refractive shapes. The right image shows a scene with two cuboids and a sphere resting atop a green table. The table is also a cuboid. The shapes on the table have all had textures applied. The cuboids have had their textures applied in cube map mode. The sphere is also reflective. The sphere and the smaller cuboid have been rotated about the y-axis. The left image is the same scene with some slight alterations. The sphere has not been rotated. A different part of it is visible. The larger cuboid has been made transparent and refractive. On its top face, we can see the reflection of the smaller cuboid. This is because of the Fresnel effect. This is discussed in later paragraphs. Both images are in 720p, with twenty rays cast for each pixel.

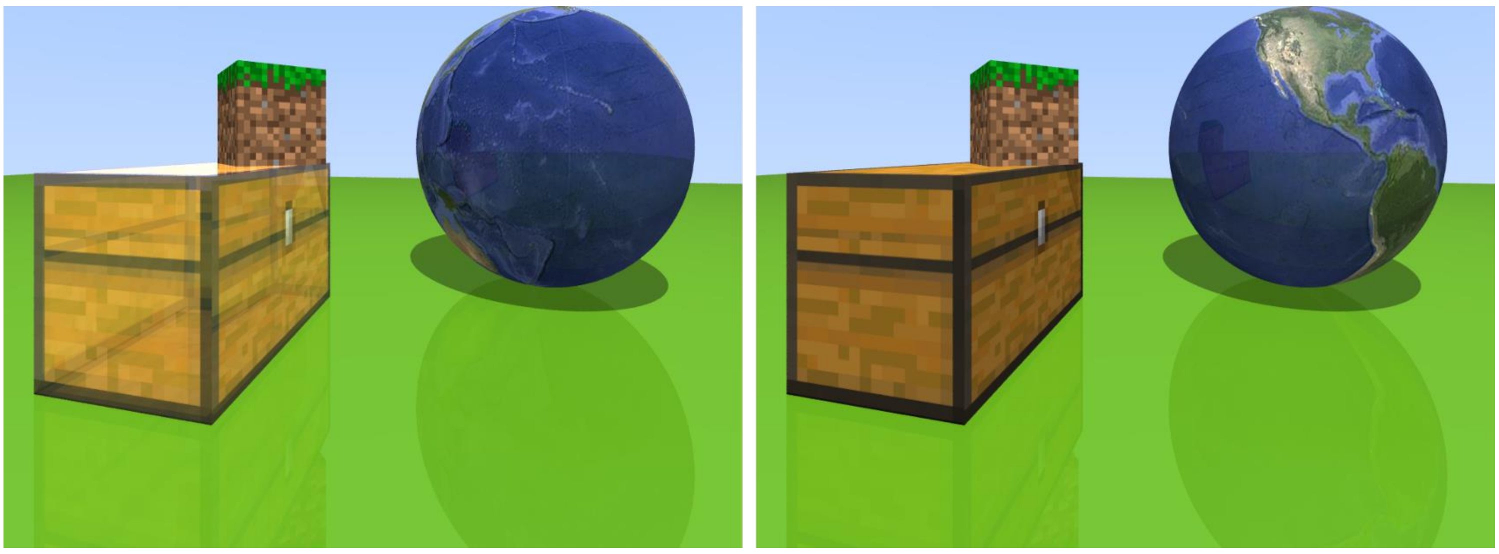

Figure 5.5: Textures are compatible with transparency, refraction, and reflection.

The final images in this subsection demonstrate the ray tracing module’s simulation of the Fresnel effect. This was described in Section 4.7.2. An image of the Fresnel effect occurring in the real world is shown below. This can be compared with the module’s simulation. The light that shines towards the far side of the lake enters the water at a more acute angle. This causes the water’s surface to reflect more light than it lets through. Light that enters the water closer to the camera does so at a greater angle. Therefore, there is less light reflected, and we can see the rocks beneath the surface more clearly.

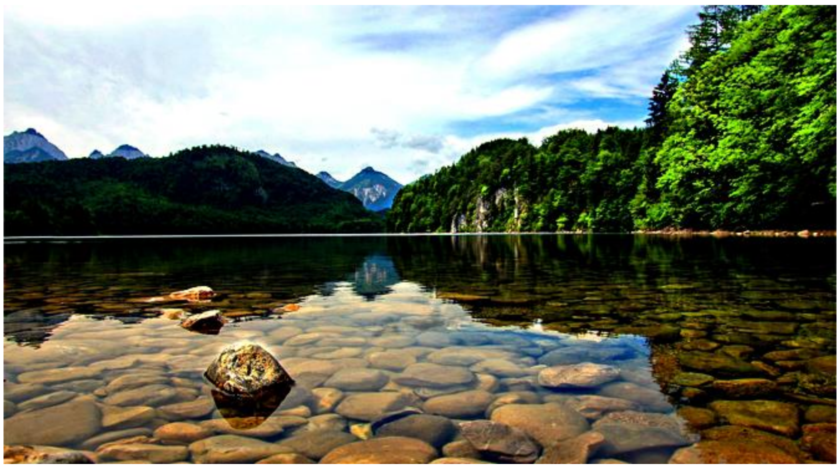
Figure 5.6: The Fresnel effect in the real world.

Figure 5.7 shows two rendered images that demonstrate the Fresnel effect. Both scenes contain the same cuboid, that has a texture of a mountain scene applied. Both scenes also contain a refractive cuboid. The refractive cuboid in the right image is less transparent. Because of this, it is darker, and its reflection are easier to see. The camera in the right image is at a steeper angle to the refractive shape. This also affects the intensity of the reflections. As can be seen in the images, the intensity of the reflections increases farther away from the camera. This is particularly noticeable around the edges of the reflected image. The closer to the camera the reflection is, the less defined its edge becomes, and the colours on either side start to blend. Figure 5.8 shows the same scene as in the right image of Figure 5.7. The camera’s field of view has been changed to hide the edges of the cuboids. I believe this image looks reasonably realistic, particularly as its resolution is only 927 pixels by 588 pixels and ten rays were cast for each pixel.

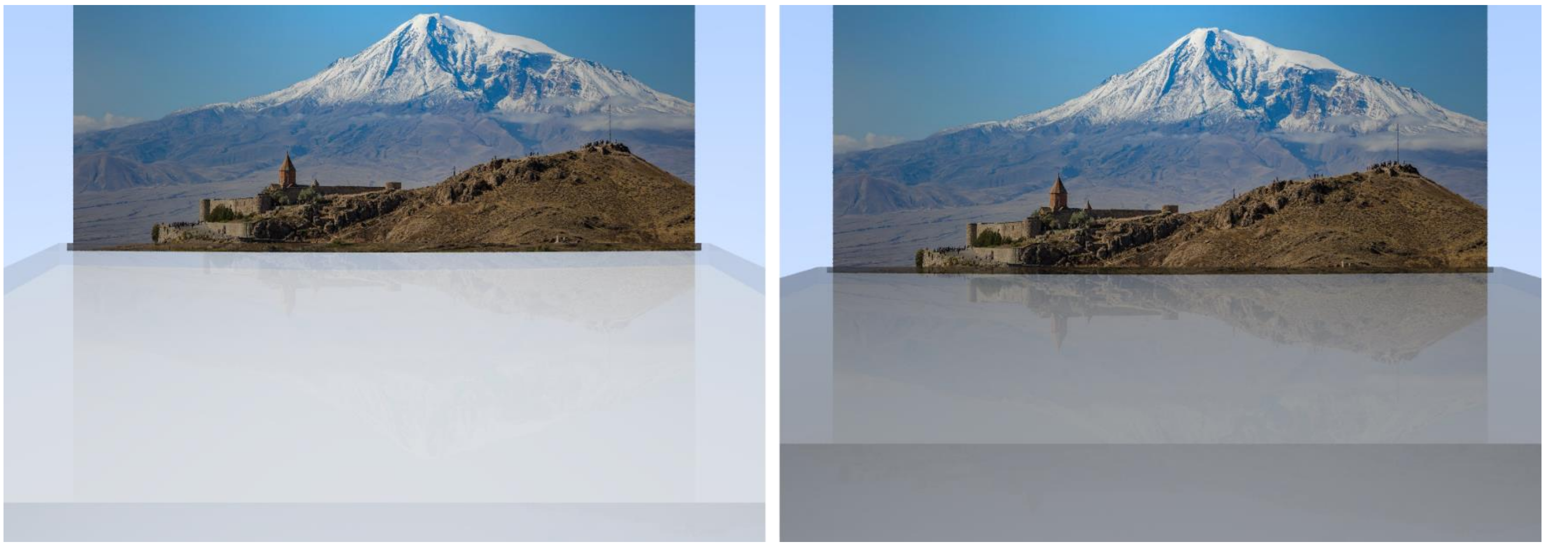
Figure 5.7: Simulation of the Fresnel effect.

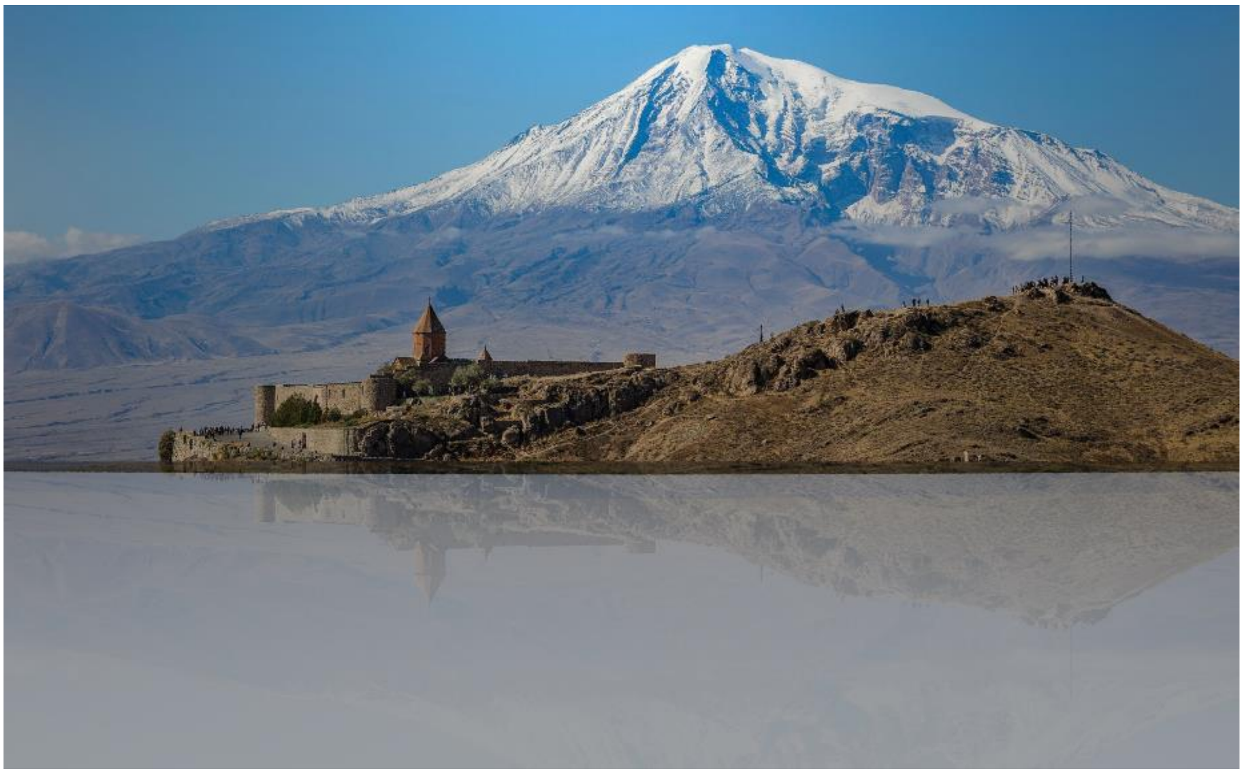
Figure 5.8:Realistic render of a mountain and lake scene.

The image below is the last in this subsection. The camera in this scene is at a much higher angle to the refractive cuboid below. Because of this, the Fresnel effect is far more subtle. It also allows us to see the shapes below the cuboid. Faint reflections of the coloured spheres become slightly visible towards the far end of the refractive cuboid. The shadows they cast on the cuboid are also visible.

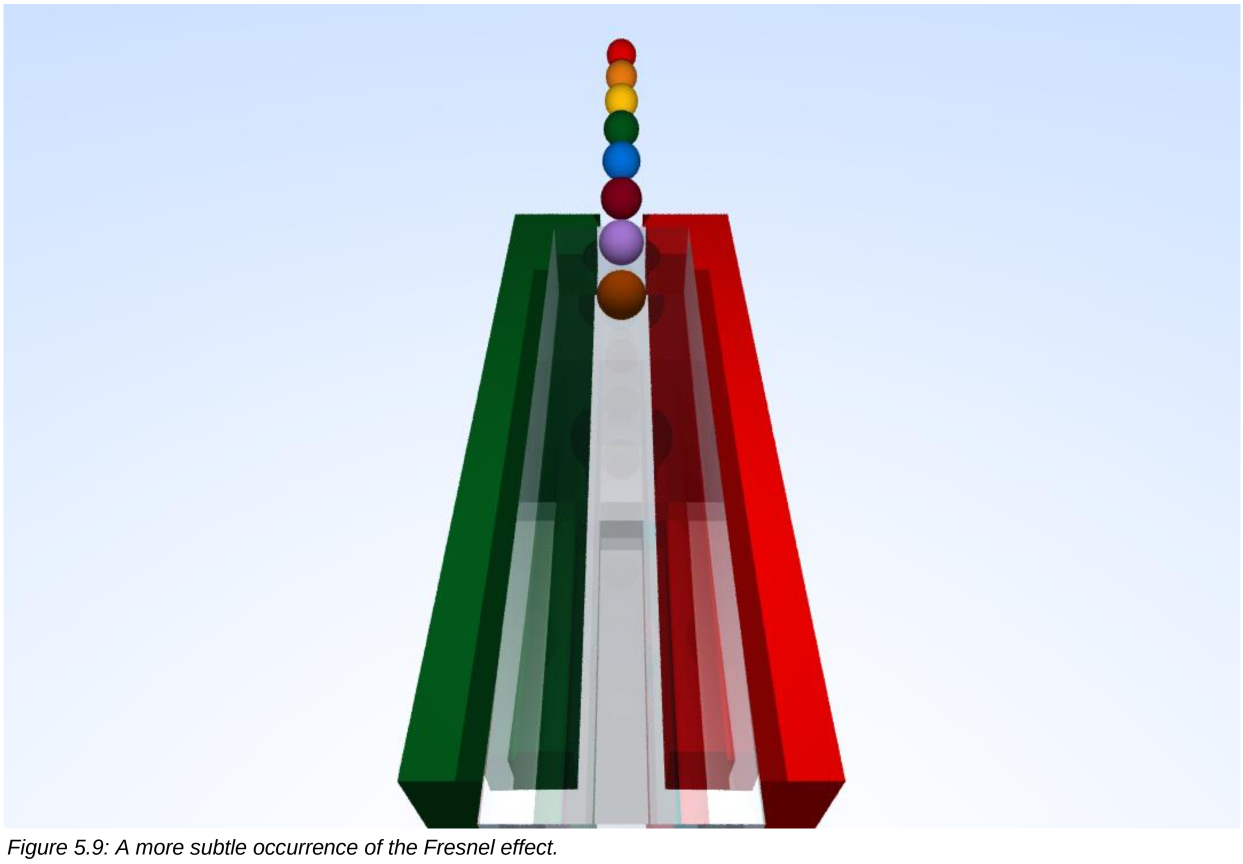
Figure 5.9: A more subtle occurrence of the Fresnel effect.

5.2: Interface Usability

In this subsection we examine to code used to produce the left-hand image in Figure 5.5. This image was rendered using a distributed computing cluster. The code base for the module totals over 1,500 lines. However, the programs that created the images in Figure 5.5, and saved them to the machine, both contain fewer than forty lines of code.

As the image is rendered using multiple machines, we must import the `Distributed` module. Next, an array is created which stores the information of each worker node we wish to use. We store the correct working directory in a string, then pass the relevant parameters to `addprocs`. This connects to the machines and adds them to our cluster. We then import the ray tracing module, called Trace, on each process. The program will save the image it renders. Therefore, we must import the `FileIO` module. Next, variables are initialised to store information about the shapes that will be created. The variable `table_centre` is assigned the value `ORIGIN`. This is a module constant that stores a vector representing the position (0, 0, 0). The first shape to be created is the green table. To do this, we must use the `make_cuboid` function, as cuboids are not directly exported. This lets us control how users create their shapes. The method must be passed a vector representing the shapes centre, and three positive real numbers. These numbers represent the shapes length, width, and height, respectively. We can also pass keyword arguments. We use the keyword arguments `colour` and `reflection` to specify the cuboid's colour and reflectivity value.

```
using Distributed

machines = [("ubuntu@3.249.197.51",1),("ubuntu@34.248.8.28",1),("ubuntu@34.240.33.89",1)]
dir = "/home/ubuntu"
addprocs( machines; dir=dir, topology=:master_worker, tunnel=true )

@everywhere include("../src/trace.jl")
@everywhere using .Trace
using FileIO

table_length = 22
table_width = 17
table_height = 1
table_centre = ORIGIN
table_colour = RGBA(0.387, 0.668, 0.105, 1)
table_reflection = 0.1
table = make_cuboid(table_centre, table_length, table_width, table_height,
                    colour=table_colour, reflection=table_reflection)
```

Figure 5.10: Examining the module interface – Part 1.

Next, we create variables to store information about the sphere. We then create the sphere using the function `make_sphere`. Here, we see a number of other keyword arguments. We also see an extra positional argument. A string can be passed after the position and dimensional arguments. The function will use this string as a path to an image. It will attempt to apply this image to the sphere as a texture. The path can be absolute. If not, the path is relative to the current working directory.

```
ball_radius = 0.7
ball_diffuse = 0.85
ball_specular = 0
ball_shine = 750
ball_reflection = 0.1
globe = make_sphere( Vector_3D(1, 1.2, 1.4), ball_radius,
                     "example_program_textures/globe_texture.png",
                     colour=TRANSPARENT, diffuse=ball_diffuse,
                     specular=ball_specular, shine=ball_shine, reflection=ball_reflection )
```

Figure 5.11: Examining the module interface – Part 2.

Next, the two texture mapped cuboids are created. The code fragment below shows another way to apply textures. If a string is passed to a shape constructor, the constructor will use it create a texture object. It will then apply it to the shape. The `make_texture` function creates a texture object that the user can save and apply to multiple shapes. It can be passed to a shape constructor via a keyword argument. If the user has loaded in an image themselves, they can also pass this to a shape constructor function. It will then be used as the texture for that shape. The shape constructor functions are overloaded to accept textures in any of these ways. The code fragment also shows how to specify the texture mapping mode for cuboids. The module constants `TILE` and `CUBE_MAP` are integers that represent the two available modes. They can be passed to the cuboid constructor functions via the `map_mode` keyword argument. The default value is `TILE`. If the user provides a value other than `TILE` or `CUBE_MAP`, the function will use the tile mode. The smaller cuboid is also rotated forty-five degrees about the y-axis. This is accomplished using the `y_rot` keyword argument.

```
block_reflection = 0
chest_texture = make_texture("example_program_textures/mc_chest_cube_map.png")
dirt_texture = make_texture("example_program_textures/mc_dirt_cube_map.png")
chest = make_cuboid( Vector_3D(0, 0.8125+eps(Float64), 0), 0.625, 1, 0.625, diffuse=0.95,
                     shine=200, reflection=block_reflection,
                     texture=chest_texture, map_mode=CUBE_MAP )
dirt = make_cuboid( Vector_3D(0.3, 1.325+eps(Float64), 0.078), 0.4, 0.4, 0.4, y_rot=45,
                    diffuse=0.95, shine=200, texture=dirt_texture, map_mode=CUBE_MAP )
```

Figure 5.12: Examining the module interface – Part 3.

The final section of the program creates all the necessary components for a scene. First, a light is created. Its position, brightness, and the ambient brightness it creates must be specified. Next, a camera is created. Its origin and a point it is looking directly at must be provided. An up-vector must also be provided. This specifies the direction of the top of the camera from its origin. Here, another package constant, `VERTICAL` is used. If the user wished to create an up-vector that is orthogonal to the direction the camera is facing, they can use the function `make_up_vector`. This takes two points and produces a unit vector orthogonal to the line between the points. The user can also specify how the vector should be rotated. If no rotation is specified, the vector will have the maximum possible value for its y coordinate. Next, the created shapes are added to an array. A scene is then created with the light, shape array, and camera. Finally, the render function is called, and the image is saved.

```
light_position = Vector_3D(-3, 7.5, 1.55)
light = make_light(light_position, 0.9, 0.7)
camera = make_camera( Vector_3D(-1.5, 1.2, 1), Vector_3D(0, 0.75, 0.65), VERTICAL )
shapes = [ table, globe, chest, dirt ]
scene = make_scene( light=light, camera=camera, shapes=shapes )

img = render_scene(scene, 1080, 720, 60, 20, worker_list=workers())
save( File(format"PNG", "example_image3.png"), img )
```

Figure 5.13: Examining the module interface – Part 4.

I am quite happy with the design of the module interface. There is much complicated code required to accomplish tasks such as creating shapes, applying textures, and actually rendering an image. However, these are all abstracted away from the user. The program shown in this subsection is quite simple and compact. I believe that the purpose of each function and module constant is reasonably self-explanatory. This is achieved thorough a descriptive naming scheme and the use of keyword arguments.

There is one element of the interface that was not implemented as originally designed. This was due to the limitations of Julia's `Distributed` module. My original goal was to remove the responsibility of creating a cluster from users. I wanted users to be able to pass the details of the machines to the render function. The function would then create the cluster for the user. The aim was to simplify the process, particularly for users that may have been given the details of the machines, but might not have the technical knowledge required to create the cluster. This was not possible in Julia. The `using` keyword imports modules. This keyword can only be used as a top-level statement. It cannot be used in a closed block of code. Closed blocks of code include functions and loops. Therefore, the necessary modules cannot be imported on the worker processes from within the render function. This is why users must add processes themselves, and pass the relevant PIDs to the render function.

Julia's limitations also create another inconvenient requirement. Users do not have to pass every available worker process to the render function. However, unless the module is being used from the Julia REPL, every process must import the ray tracing module. This can cause extra time to be spent importing the module on workers that do not require it. This is also because of the restriction that modules can only be imported from top-level statements. It is not possible to loop through a subset of the workers and only import a module on those processes. This means that selective imports cannot be done programmatically. It can be done manually. However, we cannot be sure how many processes will be added by `addprocs`. Their PIDs are also not guaranteed to be a contiguous sequence of numbers with a known starting point. Therefore, they only safe way to import modules on selected processes is manually via the Julia REPL.

5.3: Execution Methods

In this section the effect of concurrent and distributed execution of the rendering function is examined. The program that produces the left image from Figure 5.5 was used to compare concurrent, distributed, and serial execution of the rendering function. The scene from the image was prepared in the Julia REPL. The rendering function was executed thirty times. It was executed serially, concurrently, and in a distributed manner, ten times each. The execution times were recorded. Figure 5.14 is a table of the results.

The fastest execution time, the slowest execution time, and the mean execution time is shown for each method of execution. The times were recorded using Julia's built-in timing functionality, which can be accessed via the `time` macro.

The decision to carry out timing using the Julia REPL was deliberate. There are a number of factors that make accurate timing simpler when carried out using the REPL. Julia compiles code only when it is used. This means that the first call to a function will take longer than subsequent calls. All subsequent calls to the function will execute code that has already been compiled. For this reason, the first call to the render function is always ignored. The first call is also executed using the `time` macro. The code used to time execution must also be compiled. The first function call will compile the required code. The function is run in a distributed manner first. This execution method uses code that is not used during serial execution. To guarantee that this code is compiled, the ignored function call is run using the distributed execution method. The code used by this method is also used when executing the function with local threads. Therefore, it is not necessary to have a second, unused, timing of the render function.

It is important to keep the execution environment as stable as possible. It should change as little as possible between function calls. Other processes running on the same machine, web browsers and media players for example, can use computing resources. There is no way to ensure that the same amount of resources will be available to Julia during each function call. To avoid this issue as much as possible, no other programs were running during timing. Timing was carried out on AWS Elastic Compute Cloud (EC2) instances. The machines used for distributed execution had the same specifications as the main node used for serial and concurrent testing. This was to keep the comparisons as fair as possible. The number of threads used for concurrent execution was the same as the number of machines for distributed execution. Both execution methods added three worker processes. The timing results are shown in the table, below. The mean time for distributed execution is less than half of that for serial execution. This gap would likely grow the more cluster nodes are added. The difference between serial and concurrent execution is much smaller. This is likely because of the power of the EC2 instances. I, currently, only have a free tier AWS account. As such, the power of the machines available to me is very limited. I would have performed more testing and comparisons if it were possible. The AWS free tier provides a limited number of hours that machines can be active. The hours are also split among machines. At the time of writing, I have no hours left for the current month. Unfortunately, this means that I cannot perform further comparisons.

Execution Method	**Execution Time (seconds)**		
	Fastest	Slowest	Mean
Serial	117.420898	122.278178	119.275885
Concurrent	114.051263	118.572175	115.203891
Distributed	56.666133	60.124265	57.421226

Figure 5.14: Comparing serial, concurrent, and distributed runtimes.

5.4: Goals

The original goals for the project are detailed in Section 3.1. The minimum functionality of the project is defined, as is some extra, non-essential functionality. To constitute a successful project, all of the minimum functionality must be implemented. Any extra functionality is not necessary for a successful project. It simply brings added value. All of the minimum requirements for a successful project were satisfied. A selection of the extra functionality has also been implemented. The table below shows which goals declared in Section 3.1 have been achieved, and which have not. For more detailed descriptions of the functionality, please see Section 3.1.

Minimum Project Functionality	
Functionality	**Achieved**
environment definition	✓
addition of shapes	✓
shape removal	✓
light source	✓
camera positioning and direction	✓
viewing plane	✓
component removal	✓
rendering	✓
distributed rendering	✓
Non-Essential Functionality	
multiple light sources	✗
coloured light sources	✗
definable light source brightness	✓
reflective objects	✓
transparent objects	✓
refractive objects	✓

Figure 5.15: Project aims and outcomes.

Section 6: User Guide

This section provides a guide to using the ray tracing module. It contains information on how to create and render a scene. It also provides some information on Julia's requirements for setting up a computing cluster, and some of the restrictions of using the `Distributed` module.

6.1: Vectors & Colours

The ray tracing module provides a custom datatype to represent three-dimensional vectors. This type is called `Vector_3D`. A vector can be created by using its base constructor and passing it three real numbers. The base constructor is generated by Julia. Alternatively, the function `make_vector` can be used. This also takes three real numbers as parameters. The difference that if any of these parameters are omitted, their values will default to zero. Internally, the module uses vectors to represent both positions and directions. The interface provided to users only uses them to represent positions. The necessary base mathematical operators have been overloaded to operate on vectors. Vectors can be added and multiplied together. They can also be subtracted from another vector and divided by another vector. Vectors can be multiplied and divided by scalar numbers, and scalars can be added to, and subtracted from vectors. The unary negation operator can also be used with the `Vector_3D` datatype.

In the module, colours are represented by the `RGBA` datatype. This datatype contains four variables. These variables represent the red, green, and blue components of the colour, as well as its opacity. The `RGBA` type can use many kinds of real datatypes to store its colour information, but it will only accept positive numbers. The ray tracing module requires the data to be in the form of 64-bit floating-point numbers. The necessary base mathematical operators have also been overloaded to operate on the `RGBA` datatype. They can be multiplied and added together. A colour can also be divided by another colour. Colours can also be multiplied and divided by real numbers. The example code below shows how to create instances of the `Vector_3D` and `RGBA` datatypes.

```
shape_centre = Vector_3D(2.5, -6.75, 4)
orange = RGBA(0.961, 0.639, 0)
```

Figure 6.1: Instantiating vector and colour instances.

6.2: Scene Components

This subsection details how to create the various components of a virtual scene. For concrete examples of the creation of these components, see the example program in Section 5.2.

6.2.1: Shapes

There are three overloaded sphere constructor functions. Each takes at least two positional arguments. The positional arguments are required. The first two positional arguments are a vector to represent the shape's centre point, and a real number greater than zero to represent the shapes radius. If a third positional argument is provided, one of the two other overloaded constructors will be called. If a string is provided, the function will use it as the path to an image file. This image will be loaded and applied as a texture to the sphere. If the third positional argument is an image, this image will be applied as a texture to the sphere. Images must be an array of the `RGBA` datatype. The array can be one or two-dimensional. The constructors can also be passed a number of keyword arguments. These are shown in the table in Figure 6.2. The table also shows the range of values they can take, as well as their default values. It should be noted that the keyword argument, `texture`, is only available for the base constructor. The base constructor is the constructor that does not accept a string or an image. The sphere constructor is named `make_sphere`.

Keyword Argument	Description	Valid Values	Default Value
x_rot	degrees to rotate shape about x-axis	any real number	0
y_rot	degrees to rotate shape about y-axis	any real number	0
z_rot	degrees to rotate shape about z-axis	any real number	0
colour	base colour of the shape	RGBA	RED
***texture**	texture to apply	Texture or nothing	nothing
diffuse	level of diffuse reflection	real number >= 0 | <=1	0.5
specular	level of specular reflection	real number >= 0 | <=1	0
shine	shininess coefficient	real number >= 0	250
reflection	reflectivity level	real number >= 0 | <=1	0
transparency	transparency level	real number >= 0 | <=1	0
refraction	index of refraction	real number >= 0	0

*Figure 6.2: Keyword arguments for sphere constructors. *Not available to all constructors.*

As can be seen in the above table, the sphere constructor can be passed an instance of the `Texture` datatype. This is simply a wrapper type to store images as an array of `RGBA` instances. This array can be either one or two-dimensional. Having a wrapper type for textures allows us to do a number of things. It allows us to provide custom constructors for the type. We can create constructors that take simple datatypes and use them to create something more complex. The module provides an overloaded texture constructor, `make_texture`. If passed a string, the function will treat it as the path to an image file. It will load the image and use it to create a texture. It will then return the texture to the user. The function can also be passed a one or two-dimensional `RGBA` array. It will use this array to create a texture. It will then return the texture to the user.

Cuboids can be constructed using the function `make_cuboid`. There are three overloaded versions of the constructor. The reason for this is the same as for the sphere constructors. One takes a string as an extra positional argument. This will be used to load an image and apply it as a texture to the shape. One constructor takes an array of the `RGBA` type as an extra positional argument. This is also used to apply a texture to the shape. As with the sphere constructors, the `texture` keyword argument is not available to these two constructors. Each constructor takes at least four positional arguments. These are a vector to represent the shape's centre point, and three real numbers that are greater than zero. These numbers represent the length, width, and height of the shape, respectively. The table below describes the available keyword arguments.

Keyword Argument	Description	Valid Values	Default Value
x_rot	degrees to rotate shape about x-axis	any real number	0
y_rot	degrees to rotate shape about y-axis	any real number	0
z_rot	degrees to rotate shape about z-axis	any real number	0
colour	base colour of the shape	RGBA	GREEN
***texture**	texture to apply	Texture or nothing	nothing
diffuse	level of diffuse reflection	real number >= 0 | <=1	0.5
specular	level of specular reflection	real number >= 0 | <=1	0
shine	shininess coefficient	real number >= 0	250
reflection	reflectivity level	real number >= 0 | <=1	0
transparency	transparency level	real number >= 0 | <=1	0
refraction	index of refraction	real number >= 0	0

sides	cuboid face IDs to which the reflectivity level will be applied	integer or variable size tuple of integers	(LEFT, RIGHT, FRONT, BACK, TOP, BOTTOM)

*Figure 6.3: Keyword arguments for cuboid constructors. *Not available to all constructors.*

6.2.2: Lights

Lights can be constructed using the `make_light` function. This function takes three positional arguments. They are a vector to represent the light's position and two real numbers between zero and one, inclusive. These numbers represent the brightness of the light and the level of ambient light it produces. If the ambient light level is greater than the brightness of the light, an error is thrown. There is also a second, overloaded version of the light constructor. This version does not take a vector. Instead. it takes three real numbers as its first positional arguments. These are the x, y and z-coordinates of the light.

6.2.3: Cameras

A camera can be constructed using the function `make_camera`. This function takes three vectors as positional arguments. The first vector represents the camera's position. The second represents a focus point that the camera is facing. The final vector is the cameras up-vector. If a camera's origin and focus point have the same position, it is impossible to determine the direction the camera is facing. As a result, an error will be raised. If a camera's up-vector and direction are coplanar, a blank image will be produced. This is because it, essentially, reduces the viewing plane's height to zero, regardless of the other camera and image parameters. A visualisation of this situation is shown below. The black line is a side view of the viewing plane. The viewing plane can be thought of as a camera lens in the real word. The blue arrow represents the camera's direction. As can be seen in the visualisation, a camera's viewing plane cannot be orthogonal to the direction it is facing. This is what is happening when the up vector is coplanar to the camera direction.

Figure 6.4: Impossible camera configuration.

6.3: The Scene

A scene consists of a light, a camera, an array of shapes, and a background colour. It, optionally, can also contain a bounding box. A scene can be created with the constructor function, `make_scene`. This function can take up to three positional arguments. These represent the length, width and height of the scenes bounding box. These arguments can be omitted. If they are, a default value of zero will be used. If any of the bounding box's dimensions are less than or equal to zero, the scene is created without a bounding box. Keyword arguments can be used to construct a scene with a light, a camera, and an array of shapes. The function also provides keyword arguments to set the background colour of the scene, as well as its level of diffuse and specular reflection, and its shininess coefficient. These arguments are used when creating the bounding box. If the scene does not contain a bounding box, these arguments are not used. The full list of keyword arguments is shown in Figure 6.5.

A scene can be created without a camera or light source. However, they must be added before an image can be rendered. Passing a scene that does not have a camera or light to the render function will generate an error. A scene can be rendered with an empty shape array. The module provides functions to set, remove and alter the components of a scene. The camera can be set with the function `set_camera` and removed with function `clear_camera`. The light can be set with the function `set_light` and removed with function `clear_light`. The bounding box can be removed with the function `clear_boundary`. The scenes array of shapes can be set with the function `set_shapes` and emptied with the function `clear_shapes`. The array can have a single shape added using the function `push!`. It can have another shape array appended to it using the function `append!`. The functions `push!` and `append!` are overloaded versions of base Julia functions. The exclamation mark is required to indicate that its argument is altered.

Keyword Argument	Description	Valid Values	Default Value
light	light to add to the scene	Light or nothing	nothing
camera	camera to add to the scene	camera or nothing	nothing
shapes	shape array to add to the scene	Shape array	empty Shape array
colour	base colour of the background/bounding box	RGBA	RGBA(0.5, 0.7, 1.0, 1.0)
diffuse	level of diffuse reflection	real number >= 0 | <=1	0.5
specular	level of specular reflection	real number >= 0 | <=1	0.5
shine	shininess coefficient	real number >= 0	0.5

Figure 6.5: Keyword arguments for the scene constructor.

6.4: The Render Function

The function used to create images is `render_scene`. It takes five positional arguments. The first is a scene, which cannot be omitted from the function call. This scene must contain a light and a camera. Otherwise, an error will be raised. The next two positional arguments represent the width and height of the image that will be rendered. These must be integers greater than zero. The default image width is 1,920. The default image height is 1,080. The next positional argument is the image's vertical field of vision. This must be a real number between zero and one hundred and eighty, inclusive. The field of vision is measured in degrees. Its default value is ninety degrees. The final positional argument represents how many rays should be cast through each pixel to determine its colour. Multiple rays can be cast through each pixel to achieve an antialiasing effect. The more rays that are cast, the higher the quality of the image. However, the program runtime also increases with the number of rays. The default number of rays to cast for each pixel is five.

The render function also provides three keyword arguments. These are `reflect_limit`, `refract_limit` and `worker_list`. These represent the maximum number of times a ray can be reflected, the maximum number of times a ray can be refracted, and the PIDs of worker processes to use for rendering. The reflection and refraction limits must be positive integers. Both have a default value of ten. The argument, `worker_list`, can be an array of integers, or the value `nothing`. The default value is `nothing`. The integers should each be the PID of an active and available worker process. If no PIDs are provided, the render function is executed serially.

6.5: Cluster Setup

Julia requires machines to have a particular configuration before they can be added to cluster and sent instructions. The precise configuration of each machine can differ depending on the network topology specified. Each node that needs to communicate must be able to connect via passwordless ssh. In a master-slave topology, as is used in this project, this means that all slave nodes must be able to connect to the master without the use of a password. In an all-to-all topology, each node must be able to connect to every other node. Julia also allows users to set up a custom topology. The connections between different machines can use different ssh keys, but the connection must not be required to use a password. This means that connected machines must have each other on their list of authorised hosts. They must also have the relevant public and private keys stored. Setting up passwordless ssh in this way could be a security risk. If the ssh keys are stored in the global configuration folder, any person with access to a slave node could use ssh to gain access to other nodes, possibly including the master node. To avoid this, keys should be stored in a user's personal configuration folder. The username used by Julia to access a machine can be specified when adding a node to the cluster. This can be seen in the example program in Section 5.2. In my AWS cluster, each machine was set up identically. They each used to same public and private keys, which were stored in the root user ssh configuration folder. It would have been more secure to set up and use a separate account. However, as I would be the only one using the AWS machines, I deemed it acceptable to use the root account.

Machines are added to a Julia cluster using the `addprocs` function. The working directory to use for the machines can be specified, otherwise it will default to the current working directory of the host. Only one value can be specified as the working directory. This means that machines that use different directories must be added with separate calls to `addprocs`. If any files on the added machines must be accessed, the file path should be relevant to the machines working directory. Alternatively, absolute file paths can be used. The path to the Julia executable can also be specified. Again, machines that have different paths to the executable must be added using separate calls to `addprocs`. Machines often have an environment variable that stores the directories to check for executable names when a command is executed. If the Julia executable's folder is in this variable, its directory does not need to be specified.

A selection of Julia commands can only be used as top-level statements. They cannot be used on closed blocks of code. Closed code blocks include loops and functions. The `using` function is an important command that can only be used as a top-level statement. It is used to import modules into a process. This means that importing different selections of modules into different cluster nodes cannot be done programmatically. Worker nodes cannot be iterated over to have modules imported, as the `using` command will cause an error when used in a loop. It can be done manually using the Julia REPL. Otherwise, the only option is to use the `everywhere` macro to import modules to every process. The `everywhere` macro will run a specified set of statements on every process currently available. Processes added after using the macro will not have the module imported. The macro can then be used again to import a different set of modules on the newer processes. This offers some control to users. However, every new import statement used with the macro will also import the specified modules on all previously added processes. Therefore, users can not have total control without manually setting up processes at the Julia REPL.

References

Aila, T. & Laine, S., 2009. *Understanding the Efficiency of Ray Traversal on GPUs.* s.l., Association for Computing Machinery, Inc., pp. 145-149.

Beberg, A. L. et al., 2009. *Folding@home: Lessons From Eight Years of Volunteer Distributed Computing.* s.l., s.n., pp. 1-8.

Bitterli, B. et al., 2020. Spatiotemporal reservoir resampling for real-time ray tracing. *ACM Transactions on Graphics,* July.Volume 39.

Buck, J., 2019. *The Ray Tracer Challenge.* 1st ed. s.l.:The Pragmatic Programmers.

Folding@home, 2011. *FAQ - Folding@home.* [Online]
Available at: https://foldingathome.org/support/faq/
[Accessed 24 November 2020].

Folding@home, 2012. *Folding@home Gromacs FAQ.* [Online]
Available at: https://www.webcitation.org/6AqrEXY89?url=http://folding.stanford.edu/English/FAQ-gromacs
[Accessed 30 November 2020].

Folding@home, 2019. *NaCl Web Client Shutdown Notice.* [Online]
Available at: http://nacl.foldingathome.org/
[Accessed 28 November 2020].

Folding@home, 2020. *Folding@home stats report.* [Online]
Available at: https://archive.vn/20200412111010/https://stats.foldingathome.org/os
[Accessed 2 December 2020].

Folding@home, 2021. *Folding@home stats report.* [Online]
Available at: https://stats.foldingathome.org/os
[Accessed 2 December 2021].

Folding@home, n.d. *Alternative Downloads - Folding@home.* [Online]
Available at: https://foldingathome.org/alternative-downloads/
[Accessed 30 November 2020].

GROMACS, 2018. *About GROMACS.* [Online]
Available at: https://www.gromacs.org/About_Gromacs
[Accessed 30 November 2020].

Hess, B., Kutzner, C., van der Spoel, D. & Lindahl, E., 2008. GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation. *Journal of Chemical Theory and Computation,* 8 February, Volume 4, pp. 435-447.

Hruska, J., 2013. *The future of ray tracing, reviewed: Caustic's R2500 accelerator finally moves us towards real-time ray tracing.* [Online]
Available at: https://www.extremetech.com/extreme/161074-the-future-of-ray-tracing-reviewed-caustics-r2500-accelerator-finally-moves-us-towards-real-time-ray-tracing/2
[Accessed 20 December 2020].

Humphreys, G. & Ananian, C. S., 1996. *TigerSHARK A Hardware Accelerated Ray-tracing Engine,* Princeton: s.n.

IBM, n.d. *What is distributed computing.* [Online]
Available at:
https://www.ibm.com/support/knowledgecenter/en/SSAL2T_8.2.0/com.ibm.cics.tx.doc/concepts/c_wht_is_distd_comptg.html
[Accessed 1 December 2020].

IGN, 2001. [Online]
Available at: https://web.archive.org/web/20010331050522/http://cube.ign.com/news/32458.html
[Accessed 21 December 2020].

IGN, 2001. *XBox Specs.* [Online]
Available at: https://www.ign.com/articles/2001/10/02/xbox-specs
[Accessed 22 December 2020].

Kobie, N., 2009. *Intel's Larrabee makes public debut.* [Online]
Available at: https://www.itpro.co.uk/615448/intels-larrabee-makes-public-debut
[Accessed 19 December 2020].

Kouatchou, J., 2018. *NASA Modeling Guru: Basic Comparison of Python, Julia, Matlab, IDL and Java (2018 Edition).* [Online]
Available at: https://modelingguru.nasa.gov/docs/DOC-2676
[Accessed 4 October 2020].

Kutzner, C. et al., 2007. Speeding up parallel GROMACS on high-latency networks. *Journal of computational chemistry,* September, Volume 28, pp. 2075-84.

Liao, J., 2007. *The Home Cure: PlayStation 3 to Help Study Causes of Cancer.* [Online]
Available at: https://www.webcitation.org/6ApoTZ2lc?url=http://www.mb.com.ph/node/11811
[Accessed 29 November 2020].

Majercik, Z., Marrs, A., Spjut, J. & Morgan, M., 2020. *Scaling Probe-Based Real-Time DynamicGlobal Illumination for Production,* s.l.: s.n.

Maza, M. M., 2014. *Parallel and Distributed Ccomputing with Julia,* London: University of Western Ontario.

MIT, 2020. *Introduction to Computational Thinking,* Boston: s.n.

NVIDIA Research, 2020. *Real-Time Rendering.* [Online]
Available at: https://research.nvidia.com/research-area/real-time-rendering
[Accessed 8 November 2020].

NVIDIA, 2018. *NVIDIA Unveils Quadro RTX, World's First Ray-Tracing GPU,* s.l.: s.n.

Pande, V., 2014. *Adding a completely new way to fold, directly in the browser.* [Online]
Available at: https://foldingathome.org/2014/02/24/adding-a-completely-new-way-to-fold-directly-in-the-browser/
[Accessed 29 November 2020].

Pande, V., 2015. *First full version of our Folding@Home client for Android Mobile phones.* [Online]
Available at: https://foldingathome.org/2015/07/07/first-full-version-of-our-foldinghome-client-for-android-mobile-phones/
[Accessed 27 November 2020].

Phong, B. T., 1975. *Illumination for Computer Generated Pictures,* s.l.: s.n.

Schlick, C., 1994. An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum,* 1 August.pp. 233-246.

Schmittler, J., Wald, I. & Slusallek, P., 2002. SaarCOR —A Hardware Architecture for Ray Tracing. *Graphics Hardware,* Volume 17, pp. 1-11.

Shih, M., Chiu, Y.-F. & Chang, C.-F., 2009. *Real-Time Ray Tracing with CUDA.* Taipei, Springer, pp. 327-337.

Shirley, P., 2020. *Ray Tracing in One Weekend,* s.l.: s.n.

Smith, R., 2010. *Intel Kills Larrabee GPU, Will Not Bring a Discrete Graphics Product to Market.* [Online]
Available at: https://www.anandtech.com/show/3738/intel-kills-larrabee-gpu-will-not-bring-a-discrete-graphics-product-to-market
[Accessed 20 December 2020].

Steam, 2020. *Steam - 2019 Year in Review.* [Online]
Available at:
https://steamcommunity.com/groups/steamworks/announcements/detail/1697229969000435735
[Accessed 29 December 2020].

Steam, 2020. *Steam Hardware & Software Survey: December 2020.* [Online]
Available at: https://store.steampowered.com/hwsurvey/videocard/
[Accessed 1 December 2020].

Stokes, J., 2009. *Intel's Larrabee GPU put on ice, more news to come in 2010.* [Online]
Available at: https://arstechnica.com/gadgets/2009/12/intels-larrabee-gpu-put-on-ice-more-news-to-come-in-2010/
[Accessed 19 December 2020].

Teglet, T., 2009. *Intel Showcases Larrabee Wafer at IDF 2009 in Beijing.* [Online]
Available at: https://news.softpedia.com/news/Intel-Showcases-Larrabee-Wafer-at-IDF-2009-in-Bejing-109181.shtml
[Accessed 21 December 2020].

The Julia Project, 2017. *Julia Micro-Benchmarks.* [Online]
Available at: https://julialang.org/benchmarks/
[Accessed 1 December 2020].

The Julia Project, 2020. [Online]
Available at: https://docs.julialang.org/en/v1/manual/distributed-computing
[Accessed 12 October 2020].

The Julia Project, 2020. *Handling Operating System Variation.* [Online]
Available at: https://docs.julialang.org/en/v1/manual/handling-operating-system-variation
[Accessed 1 December 2020].

The Julia Project, 2020. *Multi-processing and Distributed Computing.* [Online]
Available at: https://docs.julialang.org/en/v1/manual/distributed-computing
[Accessed 6 December 2021].

The Julia Project, 2020. *Types.* [Online]
Available at: https://docs.julialang.org/en/v1/manual/types
[Accessed 1 August 2020].

Thynell, A., 2018. *Android client overhaul.* [Online]
Available at: https://foldingathome.org/2018/02/02/android-client-overhaul/
[Accessed 28 November 2020].

TOP500, 2020. *TOP500 List - November 2020 | TOP500.* [Online]
Available at: https://www.top500.org/lists/top500/list/2020/11/
[Accessed 2 December 2020].

van der Spool, D. et al., 2005. GROMACS: Fast, flexible, and free. *Journal of computational chemistry,* December, Volume 26, pp. 1701-1718.

Wyman, C. & Panteleev, A., 2020. *Rendering Games With Millions of Ray-Traced Lights,* s.l.: s.n.