TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Design and Implementation of Monte Carlo Path Tracing System on a Reconfigurable Logic-Based Platform

---

Author:
MICHAIL IASON CHATZAKIS

Thesis Committee:
Prof. Apostolos DOLLAS
Assoc. Prof. Sotirios IOANNIDIS
Dr. Euripides SOTIRIADIS

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer
in the
School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

July 25, 2022

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Design and Implementation of Monte Carlo Path Tracing System on a Reconfigurable Logic-Based Platform

by Michail Iason Chatzakis

In recent years the subject of ray tracing has caused great interest because it has many applications, ranging from the movie industry to flight simulators. Major companies develop specific hardware for ray tracing acceleration. Even though work has been done in the past, the interest of ray tracing on FPGAs has decreased as GPUs were deemed to be more suitable for the task at hand. In this thesis we re-visit the topic, albeit from the baseline of the implementation of tree structures which have not been implemented in hardware for ray tracing in the past. We aim to evaluate the performance of this ray tracing algorithm across 3 different platforms: CPU, GPU and FPGA. We also aim to introduce a ray tracing FPGA design to exploit the parallelism that the platform can provide. The first step in the process was the conversion of data structures and algorithms so that they are suitable to FPGA platforms, by converting recursive structures to iterative. Secondly, we create an architecture in which we used streaming pipeline logic in order to achieve good performance. With our introduced architecture, based on the given scene we managed to achieve speedup of up to 2x against an AMD 5600xt GPU, whereas both GPUs and FPGAs perform much better than CPUs even when multi-threading is used. To conclude, the use of FPGAs yielded a satisfactory speedup and further work on the introduced architecture can lead to better speedups.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Design and Implementation of Monte Carlo Path Tracing System on a Reconfigurable Logic-Based Platform

by MICHAIL IASON CHATZAKIS

Τα τελευταία χρόνια το θέμα του ray tracing, δηλαδή της εύρεσης με υπολογιστικές μεθόδους της όδευσης ακτινών σε κάποια σκηνή με αντικείμενα μελετάται ιδιαίτερα επειδή υπάρχει μεγάλο πεδίο εφαρμογών, από την βιομηχανία του κινηματογράφου έως τους προσομοιωτές πτήσεων. Μεγάλες εταιρίες αναπτύσσουν ειδικές μονάδες με σκοπό την επιτάχυνση της διαδικασίας του ray tracing. Αν και έχει γίνει δουλειά στο παρελθόν, το ενδιαφέρον για την διαδικασία σε πλατφόρμες FPGA έχει μειωθεί καθότι τεχνολογία GPU εθεωρείτο περισσότερο κατάλληλη. Σε αυτήν την διπλωματική εργασία ξαναπροσεγγίζουμε το πρόβλημα αυτό αλλά από διαφορετική σκοπιά, με υλοποίηση σε υλικό δενδρικών δομών που δεν έχει γίνει στο παρελθόν. Ο σκοπός μας ήταν να μελετήσουμε και να αξιολογήσουμε την διαδικασία μεταξύ τριών διαφορετικών πλατφορμών, CPU, GPU και FPGA. Επίσης, παρουσιάζουμε ένα σχέδιο ray tracing για FPGA που σκοπεύει να εκμεταλλευτεί τον παραλληλισμό που μπορεί να προσφέρει η πλατφόρμα. Πρώτο βήμα στην διαδικάσια ήταν η μετατροπή των δομών δεδομένων και αλγορίθμων σε κατάλληλη δομή για πλατφόρμες FPGA, με μετατροπή αναδρομικών δομών σε επαναληπτικές. Δεύτερο βήμα ήταν δημιουργία μιας αρχιτεκτονικής στην οποία χρησιμοποίουμε streaming pipeline λογική ώστε να έχουμε καλή απόδοση. Με την δική μας αρχιτεκτονική, ανάλογα με την κάθε σκηνή επιτυγχάνουμε επιτάχυνση έως και 2x σε σχέση με μια GPU AMD 5600xt, ενώ τόσο οι GPU όσο και οι FPGA αποδίδουν πολύ καλύτερα από CPU, ακόμη και σε πολυνηματική επεξεργασία. Συμπερασματικά, η χρήση FPGA έδωσε μια ικανοποιητική επιτάχυνση συγκριτικά με τις άλλες πλατφόρμες και περαιτέρω έρευνα πάνω στην αρχιτεκτονική που δημιουργήθηκε θα οδηγήσει σε ακόμη καλύτερα αποτελέσματα.

Acknowledgements

First of all I would like to thank my supervisor, Prof. Apostolos Dollas for trusting me with the the topic we worked with which was a fairly new for our lab. I would also like to thank Prof. Dollas for guiding through the process of creating and developing a document of academic level as well as his great inputs on the paths we should follow to complete the work.

I would also like to thank all the people in the MHL lab who helped when needed and provided the required hardware to work on. In particular I would like to thank Ph.D. Canditate Pavlos Malakonakis for the time he spent discussing with me the potential of the the software we worked on, suggesting techniques we can use to complete this project and finally for sharing his valuable knowledge regarding the Xilinx Tools. Moreover, I would like to thank Mr. Markos Kimionis for the help he provided to set up in the MHL lab when my personal setup was having issues.

Moreover, I would like to thank the members of my committee, Assoc. Prof. Sotirios Ioannidis and Dr. Euripides Sotiriades for their time in reviewing this work.

Last but not least, I would like thank my family and friends for their moral and emotional support through the demanding process of conducting this thesis, that became even harder due to the COVID-19 outbreak.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BRAM	**B**lock **R**andom **A**ccess **M**emory
BVH	**B**inding **V**olume **H**ierarchy
CPU	**C**entral **P**rocessor **U**nit
CS	**C**omputer **S**cience
DDR3-4-5	**D**ouble **D**ata **R**ate type **3-4-5** memory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
FF	**F**lip **F**lops
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GDDR5-6	**G**raphics **D**ouble **D**ata **R**ate type **5-6** memory
GPU	**G**raphic **P**rocessor **U**nit
HBM	**H**igh **B**andwidth **M**emory
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
HPC	**H**ight **P**erformance **C**omputing
LUT	**L**ook **U**p **T**able
RAM	**R**andom **A**ccess **M**emory
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SSE	**S**treaming **S**IMD **E**xtensions
TDP	**T**hermal **D**esign **P**ower
URAM	**U**ltra **R**andom **A**ccess **M**emory

Dedicated to my family and friends…

Chapter 1

Introduction

Computer graphics is a very important topic in computers. Many animation companies, architecture offices, video-game development companies, etc., rely on the creation of realistic and believable scenery to promote their work. As technology progresses and improves the demand and expectancy of a more realistic outcome is becoming more and more rudimentary. Rasterization is the default technique used for 3D illumination but it is becoming less and less appealing since its results can appear unrealistic in comparison to what is expected. Ray Tracing as a concept is an old technique that tries to replicate the natural process of light's energy transportation in the real world, thus providing visually incredible results but at the cost of a great computational cost. In the recent years hardware technology and software techniques have finally come to a point where creating scenes using ray tracing is becoming much more viable given its great outcome.

1.1 Motivation

Considerable work has been done to accelerate the Ray Tracing process by big GPU manufacturing companies. In the last few years GPUs with ray tracing specific hardware have been created, but their performance relies on extra hardware, which nonetheless do not address certain critical sections of the ray tracing algorithm itself and hence their performance is not the desirable one (Nvidia DLSS, AMD FidelityFx and more). FPGAs are platforms that can achieve great performance on heavily parallelizable workloads. The Ray Tracing algorithm falls into this category. Due to the recent interest around the topic, our goal was to evaluate the performance of a pure ray tracing system based on a modern FPGA platform.

1.2 Scientific Contributions

The scientific contribution is focused on two aspects. First of all, the conversion of the software version of the ray tracing algorithm to fit the FPGA and the GPU platforms. The algorithm is mainly recursive and written in an inheritance manner. Work needed to be done not only in the way data is stored but also on how data travels into the pipeline to solve the recursion issue. The second aspect involves the evaluation of the performance across different platforms. A method to import to 3D objects into the ray tracer needed to be introduced in order to evaluate the performance across different platforms in different situations. In the end an evaluation based on our results is made to determine weather or not modern FPGAs fit the workload of the ray tracing algorithm.

1.3 Thesis Outline

- **Chapter 2 - Theoretical Background:** Theoretical Concepts and Techniques regarding Ray Tracing.
- **Chapter 3 - CPU Implementation:** In depth coverage of math and procedures we included in our software Ray Tracer.
- **Chapter 4 - GPU Implementation:** Description of the steps followed to convert the software Ray Tracer into a GPU compatible version using OpenCL.
- **Chapter 5 - FPGA Implementation:** Our introduced architecture as well as description of the problems we had to deal with.
- **Chapter 6 - Results:** Comparison and Discussion regarding the above three implementations.
- **Chapter 7 - Conclusions and Future Work:** Conclusions and Future work.

Chapter 2

Theoretical Background

The first goal of this thesis was to understand and acquire knowledge regarding ray tracing and computer graphics in general. In this chapter we will present the basic ideas and concepts behind ray tracing.

2.1 Rasterization - The currently most dominant idea

Rasterization is a method which aims project 3D objects on a 2D computer screen. It works by projecting every vertex of every object on the screen while considering the angle between the object and the camera frame, connecting those vertices with lines and finally filling the space between the lines with the objects color. This process is repeated for every object.

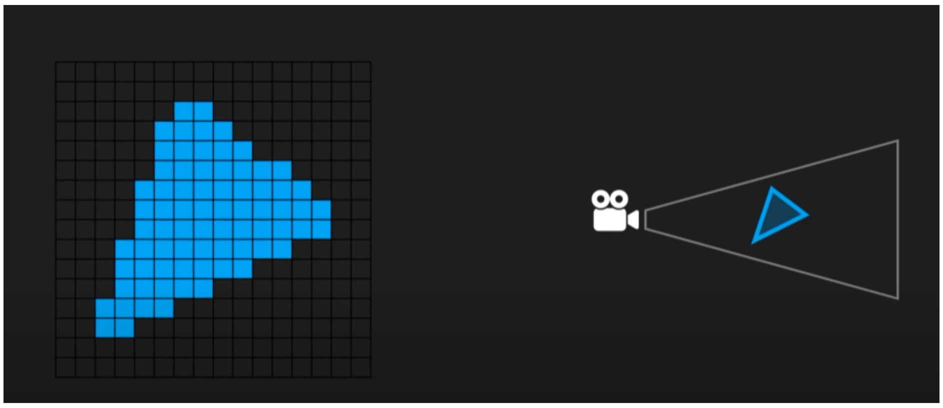


FIGURE 2.1: Rasterization. Image taken from [1]

The problem with this procedure is that it is not known in advance which objects are closer to the camera and which are further away. This is solved by keeping the distance data from the camera to every object in a buffer called the frame buffer or Z-buffer. The Z-axis location of each vertex determines the distance between the camera and the vertex so the Z-buffer holds this information. This way it can be determined which object should be projected in the front and which object should be projected in the back. This requires massive parallel computational power and this is why specific graphics processing units(GPU) are used to render 3D images. But in order to make a scene photo-realistic and "believable" more elements are needed, such as shadows, depth of field, anti-aliasing and many more effects, all of which require to be coded in and do not appear naturally. Since those effects do not appear naturally, they sometimes tend to not look very realistic in computer-generated scenes. This is the main benefit of Ray Tracing over the currently most dominant way of doing computer graphics.

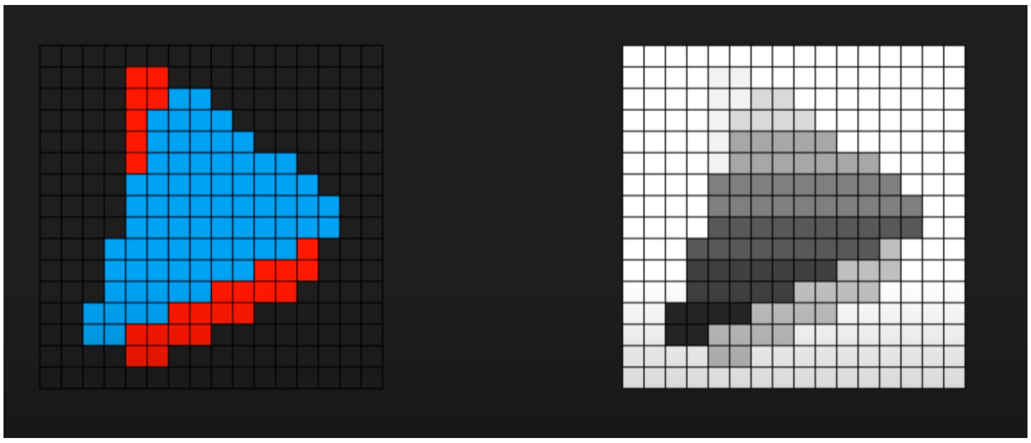


FIGURE 2.2: Z buffer simulation. Image taken from [1]

2.2 What is Ray Tracing

2.2.1 The idea

Ray tracing is a process whose main purpose is to illuminate a digital scene. The way in which ray tracing works is very similar to the way light behaves in the real world. When someone looks at a camera, the camera screen is a projection of the 3 dimensional world in the 2 dimensional camera screen. In a scene of an animated movie or a video game, the screen acts like the camera screen and the animated world acts like the real world from the previous example. The way a camera is able to produce an image is by collecting energy from light rays that are emitted from light sources such as the sun. The emitted rays bounce around in the environment and some of them eventually end up in the camera lens. The camera collects this light for every pixel of the screen and produces the image. Ray tracing attempts to solve the "illumination of the scene" problem in a very similar fashion.

2.2.2 Forward Ray Tracing

A digital 3 dimensional scene consists of a digital objects that absorb light, digital light sources that emit light and a digital camera that collects and prints the output to our computer screen. As described earlier the idea is that light rays are emitted from the light sources, they bounce around the scene between the objects and a small portion of them eventually ends up in the camera. This process is called forward ray tracing. There is a major problem with this approach. In order to determine whether or not a ray ends up in the scene, it is required that we fully trace the ray to its destination. That means calculating every bounce with every object in the scene for every ray, most of which will end up discarded since the majority of the rays will not hit the camera and therefore their color will not contribute to the image. This is a very chaotic and unproductive way that is way to costly to be practical.

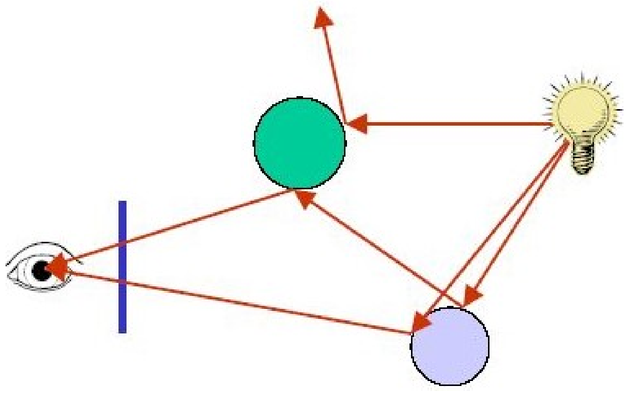


FIGURE 2.3: Forward ray tracing. Image taken from [2]

2.2.3 Backward Ray Tracing

To solve the aforementioned problem, a setup is created where all the light rays that are shot and processed are useful. The important rays are the ones that end up in the camera frame. The idea is that instead of shooting rays from the light sources and hoping that some of them will end up in the camera frame, a ray(primary ray) is shot for every pixel of the camera frame towards the scene. If the ray intersects an object, light from this objects color is gathered and a new ray(scattered ray) is spawned at the impact location.

The trajectory of the new ray is based on the objects material because different material scatter light differently. This process is repeated until we finally hit a light source. Its important to note that a bounce threshold needs to be set because if a ray takes too many bounces to finally hit a light source it will have lost so much energy to get to the camera that its impact on the pixel color is meaningless.

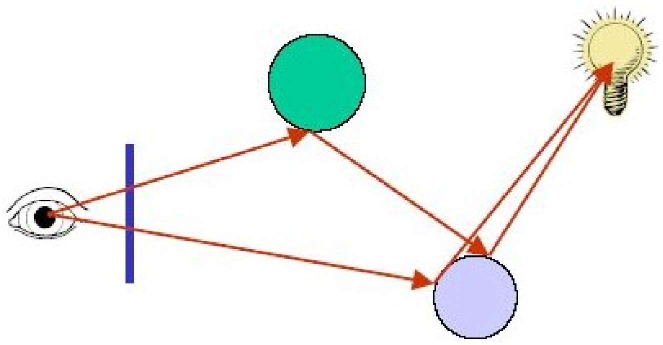


FIGURE 2.4: Backward ray tracing. Image taken from [2]

2.2.4 Monte Carlo integration

In the real world a camera's pixel can collect color from many different rays. Additionally, since objects are not perfect mirrors, an emitted ray at an intersection point could be the derivative of more than one ray and slightly different angles. In order to effectively replicate the outcome of forward ray tracing, monte carlo integration is used for the color of every pixel. Multiple rays originating the same pixel are shot into the scene at slightly different angles in a hemisphere, the paths and the bounces they make are slightly different from one another, and eventually the final color of each pixel is the average of the colors previously calculated(2.1). The amount of times that this process is repeated is directly linked to the noisiness of the image and its clarity, but at a toll of extra computational time.

$$FinalPixelColor = \frac{\sum_{1}^{samples} color}{samples} \tag{2.1}$$

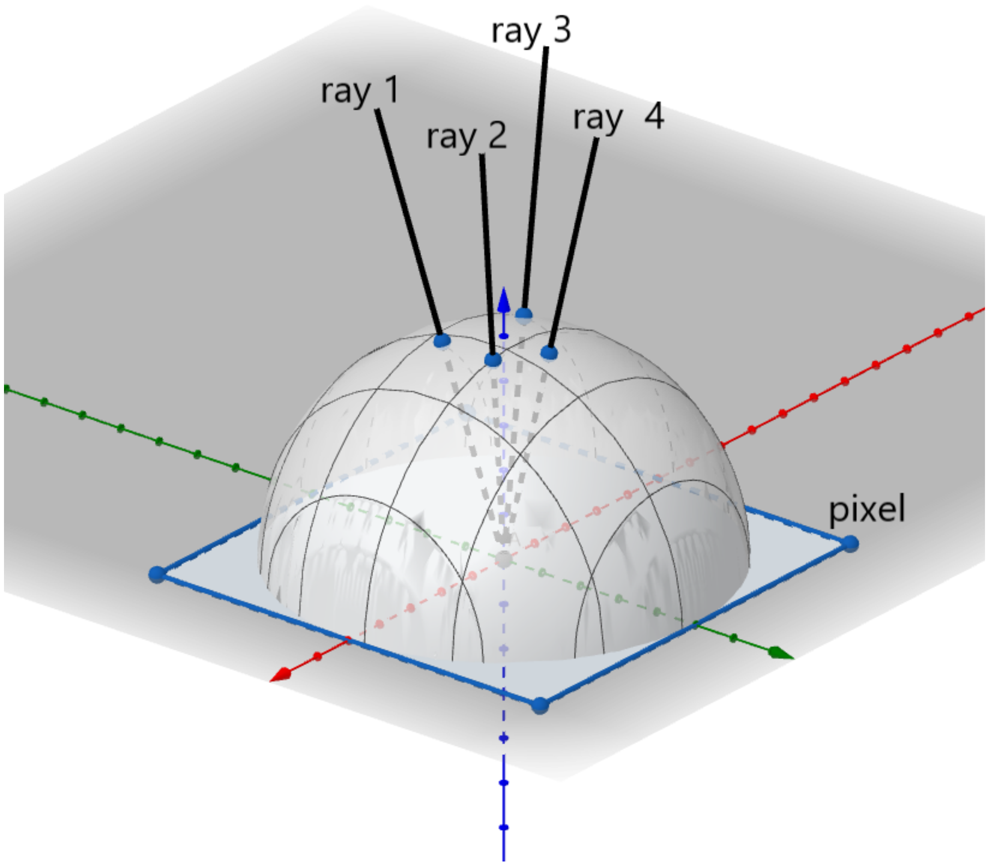


FIGURE 2.5: Rays cast from the same pixel at different angles in a hemisphere

2.2.5 Rays

A ray is a simple vector defined by 2 points, its origin and its direction. A ray in its nature is a line so its function can be defined as:

$$P(t) = O + tD \tag{2.2}$$

where O and D are the origin and direction respectively and $P(t)$ is the position on the "line". t could be considered as the distance because the higher the t, the longer the vector would be. Even though so simple, this equation is quite important and will be used later to define the object intersection equations.

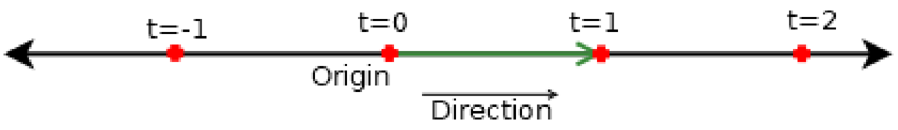


FIGURE 2.6: Simple Ray vector

Shadow Rays

When a ray intersects an object a intersection point is found. In order to determine weather or not that point in directly illuminated by a light source or indirectly illuminated by ray bounces, shadow rays are a method that makes this possible. Apart from the bounced ray that is spawned at that location and is traversed further, a shadow ray is also spawned but isn't traversed further. The shadow ray points towards the light source of the image. If there are no other objects in between the light source and the intersection point, then the point is directly illuminated, otherwise there is something in between, which casts a shadow on the point. Casting shadow rays is not the

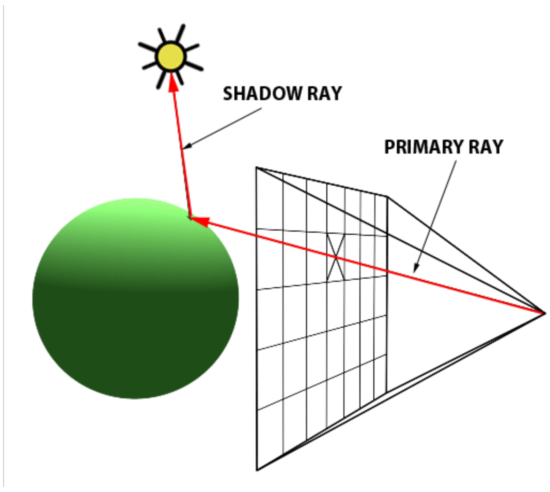


FIGURE 2.7: Shadow Rays. Image was taken from the Scrachapixel book.[3]

only way to perform this operation, but its the most common one. There are other less common ways that could provide similar results. Instead of scattering rays uniformly around a point when creating the bounced rays, some bias can be introduced to send some rays directly towards the light sources.

2.2.6 Space Partitioning

The "brute force" method

At this point a scene can potentially be rendered. Since every object has its parameters and its intersection test defined, its possible to render a scene by testing intersection with every object and shading a pixel based on the closest object. A very computationally expensive flaw can be noticed here. In a very possible scenario where the scene contains triangle messes the scene can be made of thousands of objects and if the resolution is scaled up(eg. 1920x1080 = 2073600 pixels) the problem becomes very apparent. Many intersection tests will need to be performed for every ray, only end up with 1 or 0 objects that intersects the ray. An algorithm like this has a time complexity of $O(n)$ where n is the number of objects. In addition the intersection test of objects such as spheres and triangles, are not a particularly fast intersection tests.

What is Space Partitioning

A lot of work has been done to speedup the previous process. In order to accelerate this procedure special structures are used to recursively divide and categorize the objects in the scene into 2 sub-volumes until a single or a small amount of objects are left in each sub-volume. After that instead of checking every object from the scene, recursively checking weather or not a ray intersects a sub-volume will lead to the same result but much faster. These sub-volumes need to fulfill some characteristics:

- Nodes need to be close to each other.
- Each node need to be of minimum volume.
- The sum of all bounding volumes should be minimal.
- Volume of overlapping nodes should be minimal.
- Greater attention should be paid to nodes near the root as creating better bounding boxes near the root could remove more branches from further consideration.
- The tree should be balanced as this would reduce the computations needed.

Isnert foto here
The most common ways to do this division is either by spacial division or object division.

Spacial Division

In spacial division the goal is to contain the objects of the scene inside equal boxes defined by imaginary planes. Recursively repeat the process until only a few or even single objects are left inside each box. The result will be a tree data structure The 2 most commonly used structures of that type are k-d trees and bounding volume hierarchies(BVH). Our ray tracer uses the later, so we will be explaining that one later.

AABB-Axis Aligned Bounding Box

Bounding Box is a box that very tightly surrounds a set of objects. A Bounding Box either contains objects, at the bottom of the tree, or other smaller bounding boxes on other levels of the tree. Since bounding boxes are not real objects in a scene, their intersection needs to be fast and since the hole tree hierarchy consists of a lot of them, they need to have a relatively small memory cost. Unlike other objects, the only information needed when an intersection is tested against a AABB is weather or not it was actually hit. Other information such us color, normals etc. are not needed since the AABB is not a real object. It only exists to accelerate the intersection algorithm.

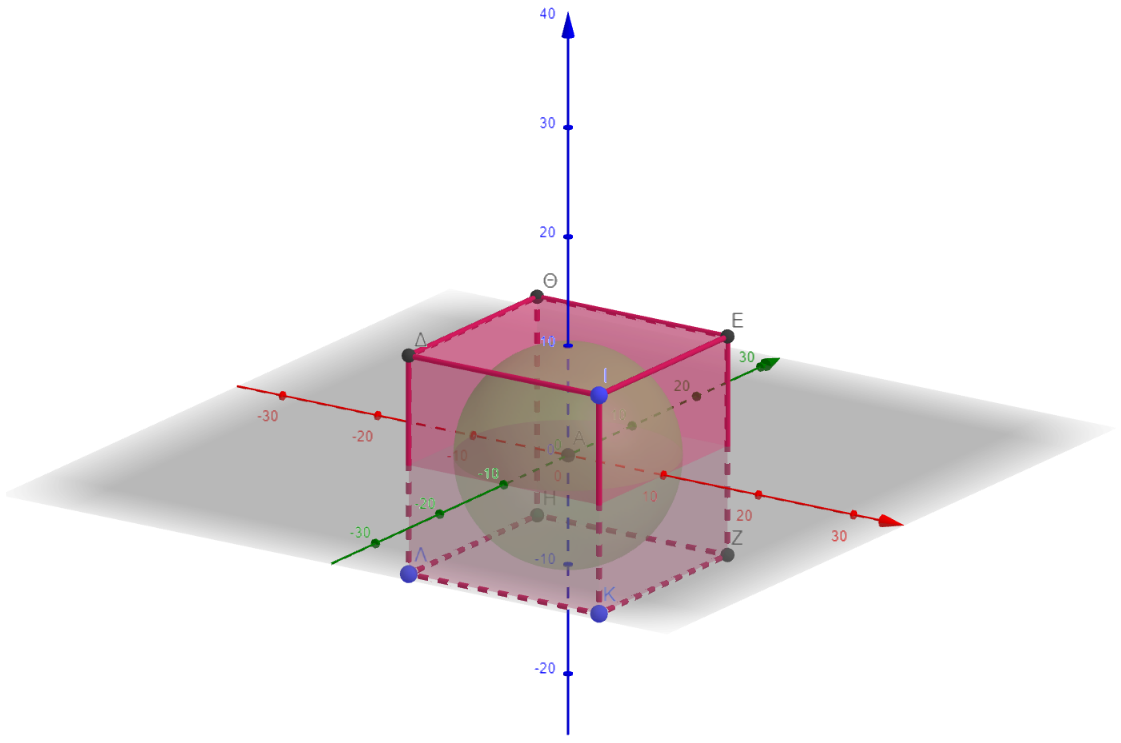


FIGURE 2.8: Bounding Box of a sphere

2.2.7 BVH - Bounding Volume Hierarchy

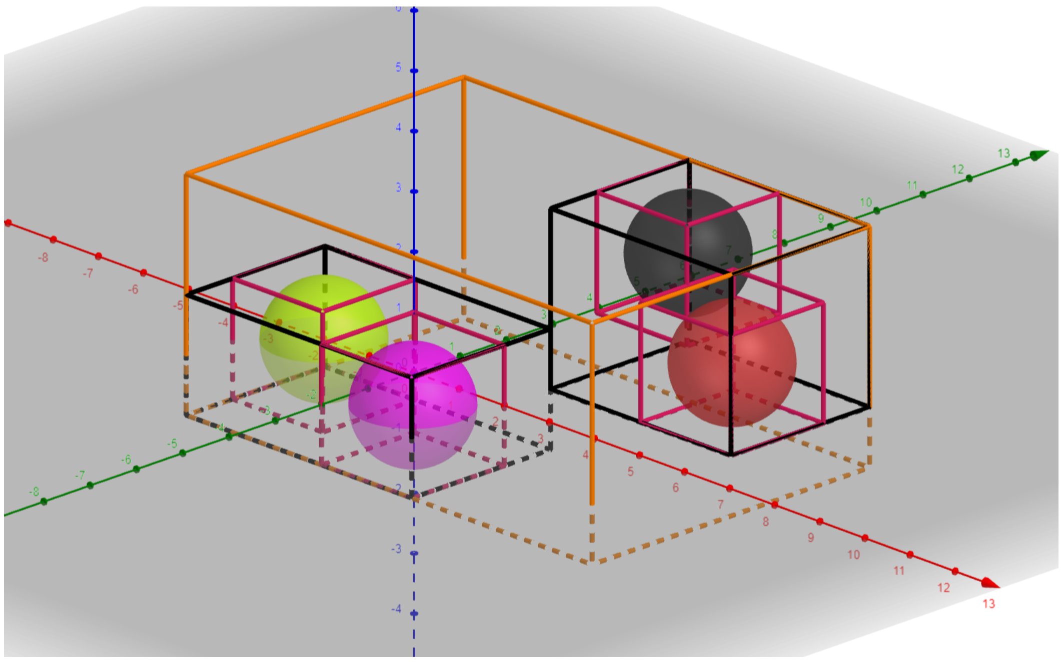
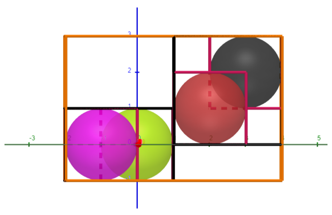
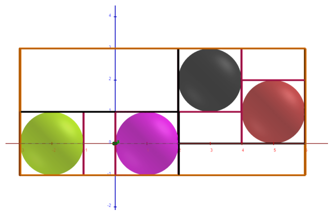


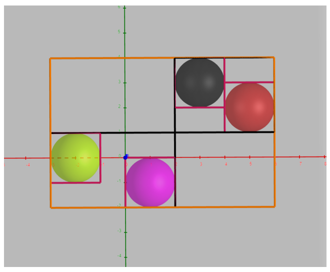
FIGURE 2.9: Visual representation of 3D objects with bounding boxes



(A) Y-Z Axis



(B) X-Z Axis



(C) X-Y Axis

FIGURE 2.10: View of previous image 2.9 around the different Axis

A bounding volume hierarchy is a object division structure. It categorizes the objects of a scene relative to their position with each other on a single axis and divides them into 2 sub-categories. A bounding box is put around each of the sub-categories and the process is repeated. Eventually when there are 2 or 1 objects left in each sub-category the process is complete. The final product will be a binary tree whose inner nodes will contain bounding box information. At the bottom of the tree, on its leaves, the objects are found. Instead of checking all the objects of the tree in order to find the closest intersection, traversing the tree, testing bounding box intersections on the inner nodes and finally testing object intersection on the leaves.

Since the tree is balanced, the number of levels of the tree is given by

$$levels = log_2 n \tag{2.3}$$

where n is number or objects. Theoretically we should traverse each of those levels once to find our desired object. A scene of $1,048,576$ objects would produce a tree of $log_2 1,048,576 = 20$ levels. This means that instead of testing all $1,048,576$ we would only need to test box intersection twice for each level and finally test object intersection for a total of around 40 tests instead of $1,048,576$. This is a pretty remarkable speedup.

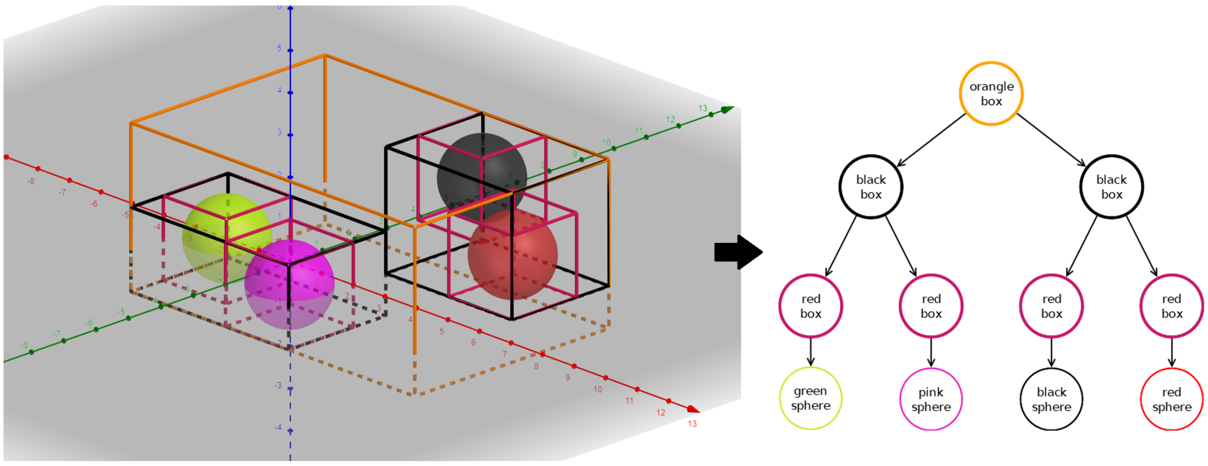


FIGURE 2.11: BVH tree equivalent of previous figure2.9

The major problem of the BVH

As described in [4] an important problem that BVH structures face, is the fact that boxes which overlap may be produced(2.12). This means that the check for box intersection on the same level of the tree can be true which would produce 2 different paths going down. Most frequently one of those 2 paths will be eliminated on the coming levels but this is not guaranteed. Sometimes the different paths can lead to different objects. In that case the distance to the camera is still the best way to determine which one is closest if more than one objects pass the intersection check.

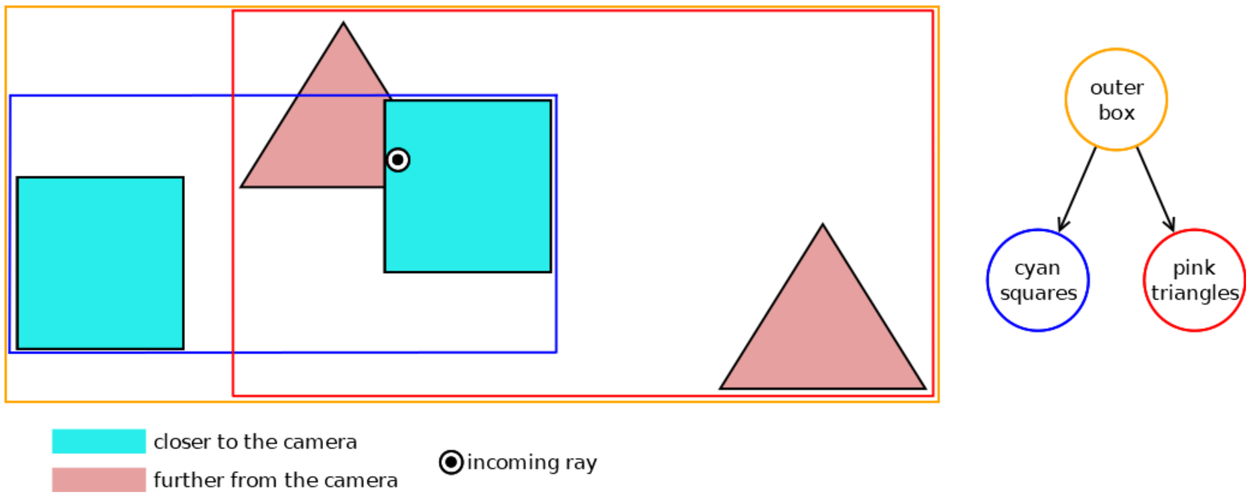


FIGURE 2.12: Overlapping objects problem.

In the above image(2.12), the axis that was chosen when creating the BVH structure was the z axis. This resulted in cyan squares and the pink triangles to be grouped together. When a ray is shot at the location specified on the image both blue and red children of outer orange node test true, thus both paths should be traversed to determine the answer. Moreover, this specific example will lead to both the right cyan square and the left pink triangle to be tested for intersection and finally comparing the distance that the ray traveled to hit them will result in the cyan square being the closest one.
The problem in this situation originated from the choosing of the z axis as our split axis. Had we chosen a different axis as the split axis, different objects would have been grouped together resulting in different bounding boxes, thus solving this issue. But the trajectory of the rays is unknown at tree construction time. Unfortunately, this reduces our worst case time complexity from an $O(log_2 n)$ to an $O(n^2)$[1] where n is number or real objects in the scene. The best case complexity still remains $O(log_2 n)$. Even though this seems to be a step down, the worst case or anything close to it, are highly unlikely to occur. After testing it was calculated that depending on the amount of objects in the scene, using the algorithm was going to be beneficial. When there were very few objects in the scene, less than 10, its better to use the "brute force" method of testing every object in the scene, and when there were more than 10 the benefits scale logarithmically. Any scene containing a polygon mesh which consists of thousands of triangles is getting a major speedup. Some test-benches such as the Cornell box, which contains only a few objects trapped in a box, can potentially see better performance using the

[1]there are n objects and n nodes for those objects to be put in a tree, thus $O(n^2)$ complexity.

"brute force" method.

2.3 Tools used for this thesis

2.3.1 GeoGebra

GeoGebra is a online geometry tool that can be used to create objects in 3D space. The tool takes math function inputs in 2D or 3D space and creates the corresponding 3D objects. In this thesis many of the figures and photos were created using this tool.

2.3.2 Dia

Dia is another tool that was used to create shapes in this thesis. This tool mainly used to create block diagrams and flowcharts.

2.4 Theoretical knowledge sources

Knowledge for this process was gathered from 3 online books regarding ray tracing and computer graphics.

- Ray Tracing in one weekend book series [5]
- Scratchpixel book [6]
- Physical Based Rendering book [7]

Chapter 3

Related Work

Using ray tracing acceleration structures is one of the most essential parts of a ray tracer.
In [8] many techniques are explained and some new are introduced. In their paper they note that when performing splits to create object intersection acceleration structures(BVH, kd-tree and more) the position of the origin(the camera) should be taken into account to create a better quality tree that will perform better. Instead of using the most commonly used BVH and kd-tree acceleration structures they mention another technique, the perspective grid. They also propose the use of 2 different types of shadow rays, hard and soft. In their testing they estimated to achieve 1.5x greater performance using their techniques.

In [9] the method used to render the movie "Cars" is explained. For spatial acceleration, they used a BVH(bounding volume hierarchy) structure. Ray tracing is only used to render specific effects such as reflections, shadows, ambient occlusion and more, and the REYES(Renders Everything You Ever Saw), algorithm is used to render directly visible objects. Multi-resolution geometry and textures are used and the best geometry resolution or texture is selected by comparing ray differentials. Finally, SIMD instructions are used to speed up the tree traversal part of the process. This method is also used by other movies as well.

The idea of accelerating the Ray Tracing Algorithm using FPGAs has been been explored in the past.

One of the earliest works regarding Ray Tracing and FPGAs was the SaarCor[10]. The SaarCor uses k-d trees to accelerate the intersection process and traces packets or rays. its pipeline consists of f a ray generation/shading unit, a 4-wide SIMD traversal unit, a list unit, a transformation unit, and an intersection test unit. The project aimed at real-time rendering of computer games. Unfortunately, it wasn't fully implemented on any VLSI boards and all the benchmarks and results are based on simulations. The core was later used and tested by[11].

RayCore[12] is another implementation by NAH. It also supports path tracing and was tested with it, but its primary aim was the use of real-time recursive ray tracing for mobile devices with low power consumption. An interesting comparison was done by not using only an FPGA for benchmarking, but also an ASIC, which gives a good approximation of how much performance gain is possible with an ASIC compared to an FPGA.

The T&I-Engine[13] was the first approach using Single Instruction, Single Data (SISD) to compute each ray independently. Similarly to SaarCOR, it only relied on a simulation for evaluating the design and accelerated the intersections using a k-d tree. Its core introduces three concepts: an ordered depth-first layout, a three-phase intersection test unit, and a ray accumulation buffer for latency hiding. Although its aim was real-time rendering based on recursive ray tracing, the design can be used for basic path tracing too. To support those class of algorithms, techniques such as sampling and path termination are additionally needed.

One of the most computationally intensive operations regarding ray tracing is the traversal of the object hierarchy. Work has been done specifically target to accelerate that without fully implementing a ray tracer. In [14] the k-means clustering algorithm was used to accelerate the tree traversal. They used a pipelined k-d tree implementation using multiple memory banks.

Another more recent work from the Sichuan University [15] involved using uniform spacial partition as the scene management method and using openCL to design the FPGA architecture. It uses a hard processor system(HPS) in combination with an FPGA. The FPGA was used to accelerate the intersection process. Shading and other related work for were dealt with by HPS.

At last, its important to note all the resent released consumer GPU cards from NVIDIA and, more recently, AMD that have ASIC ray tracing cores printed onto them. These cards are not scientific research projects so not a lot is known about their architecture. We know that NVIDIA's RT cores aim at accelerating the intersection process and leave the shading process for the normal GPU pipeline. Also in order to achieve real-time ray tracing in high resolutions, more image manipulation is required to achieve high frames per second(DLSS, Denoiser modules, etc.). Benchmarks and other comparisons are omitted here as at the writing of this thesis, the RTX cards were too new and expensive.

Chapter 4

CPU Implementation

Our next goal was to set up a basic ray tracer system in software. In this chapter we will discuss and dive further into the functionality and the math behind the ray tracer's operations and algorithms.

4.1 The Basic Ray Tracer that we used

The Ray Tracer that we started with was a simple ray tracer which was provided by the book series "Ray Tracing In One Weekend" by Peter Shirley. We didn't used the most advanced version of the code as a lot of the features that it provided weren't what this thesis was aiming to achieve, even though they are important for a fast ray tracer. We sticked to the version that is described in book 2 of the series(Ray Tracing: The Next Week) and the Ray Tracer was still able to take advantage of:

- Ray-Sphere Intercession
- Bounding Volume Hierarchies(BVH)
- Inti-aliasing effect
- Depth of filed effect
- 3 types of different materials(diffuse, metal, glass)
- Monte Carlo Integration

All of this was a good base to begin our setup. In addition to all those features we also implemented a few extra as well.

- Parallelize the workload as a multi-thread application
- Ray-Triangle Intercession

- Polygon Mesh readability

4.2 Parallelizing the work - multi thread

The first thing that we did to improve the code execution was to figure out if and where if it would be possible to parallelize the workload. If we assume that a pixel's position on the image frame is described as `pixel(x,y)` the basic loop of our algorithm is as follow:

```
for(y in y axis){
    for (x in x axis){
        for(every sample){
            calculate_pixel_color;
        }
    }
}
```

From this we can easily deduct that every pixel is completely autonomous and does not require any extra information from other pixels. We used the OpenMP libary to parallelize the workload on the `y axis loop`. We were basically giving a smaller part of the screen to each thread to render. So the result was:

```
#pragma omp parallel
{
    for(y in (y axis/num_threads)){
        for (x in x axis){
            for(every sample){
                calculate_pixel_color;
            }
        }
    }
}
```

4.3 Intersection Tests

An intersection occurs when a ray hits an object. Our raty tracer supports 3 different type of polygon intersections, triangle intersection, sphere intersection and Axis Alligned Bounding Box intersection.

4.3.1 Triangle Intersection - Möller–Trumbore intersection

Since our original code didn't include any triangle intersection algorithm we decided to use the Möller–Trumbore triangle intersection algorithm[16][17]. Even though this algorithm was created back in 1997 it is still considered very fast today. The main advantage that it has, is that it uses the barycentric coordinates[18] to describe the intersection point.

Barycentric coordinates

Barycentric coordinates are especially important in computer graphics in general. Barycentric coordinates are used to express the position of any point $P(w, u, v)$ located inside a triangle ABC with three scalars. In other words, for each triangle ABC there is a unique sequence of numbers $u, v, w \geq 0$, such that $u + v + w = 1$ and:

$$P(w, u, v) = wA + uB + vC \tag{4.1}$$

where P is the intersection point, and A, B, C are the vertices of the triangle. u, v, w are the areas defined by the sub-triangles ABP, BCP, APC as shown in figure 4.1. This system is also refereed to as areal coordinates system due to this property. Note that by knowing two of the coordinates we can find the third one, e.g $w = 1 - u - v$ and from the previous two we can deduct $u + v \leq 1$. so the equation can also be expressed as:

$$P(u, v) = (1 - u - v)A + uB + vC \tag{4.2}$$

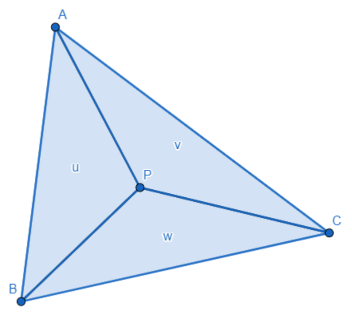


FIGURE 4.1: Barycentric Coordinate System - u,v,w are the areas defined by the sub-triangles

The coordinates of a point are always in regard to the specific triangle ABC so for example, the triangle centroid[1] point will always have homogeneous barycentric coordinates of $(1,1,1)$ since its a specific point on a triangle that has the same properties for all triangles.

Calculating the Areas w,u,v

The goal on this entire process is to find the intersection point of the ray with the triangle if that intersection point exists. The key idea here is that by finding the barycentric coordinates of that point, if the equations $u+v+w = 1$ and $u,v,w \geq 0$ are not fulfilled, we can conclude that the points exists outside the triangle, thus no intersection.
Lets continue with equation 4.2.

$$P(u,v) = (1-u-v)A + uB + vC \tag{4.3}$$

We know from previous chapters that the parametric equation of the Ray is $P(t) = O + tD$ where O is the origin, D is the direction and t is the distance. The intersection occurs when $P(t) = P(u,v)$. By using this we get:

$$\begin{gathered} P(t) = P(u,v) \Leftrightarrow \\ O + tD = (1-u-v)A + uB + vC \Leftrightarrow \\ \Leftrightarrow (\text{move terms around}) \Leftrightarrow \\ O - A = -tD + u(B-A) + v(C-A) \end{gathered} \tag{4.4}$$

Expressing the above equation in terms of matrices yields:

$$\begin{bmatrix} (-D) & (B-A) & (C-A) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - A \tag{4.5}$$

To make everything more clean and simple, denoting $E_1 = B - A$, $E_2 = C - A$ and $T = O - A$ gives:

$$\begin{bmatrix} (-D) & (E_1) & (E_2) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = T \tag{4.6}$$

[1]The point in which the three medians of the triangle intersect is known as the centroid of a triangle.

To elaborate further Cramer's Rule[19] is used. The equation now becomes:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\det \begin{vmatrix} -D & E_1 & E_2 \end{vmatrix}} \begin{bmatrix} \det \begin{vmatrix} T & E_1 & E_2 \end{vmatrix} \\ \det \begin{vmatrix} -D & T & E_2 \end{vmatrix} \\ \det \begin{vmatrix} -D & E_1 & T \end{vmatrix} \end{bmatrix} \tag{4.7}$$

Remember that all constants are either points or vectors defined in 3D space so they are actually 3x1 matrices themselves eg. $O = \begin{bmatrix} O_x \\ O_y \\ O_z \end{bmatrix}$. So what seems to be a 1x3 matrix in the equations is actually a 3x3 or a 1x3 vector matrix. The determinant of a 1x3 vector matrix is called **scalar triple product** [20] for which:

$$\mathbf{det} \begin{vmatrix} \boldsymbol{A} & \boldsymbol{B} & \boldsymbol{C} \end{vmatrix} = \det \begin{vmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ A_z & B_z & C_z \end{vmatrix} = \boldsymbol{(A \times B) * C} \tag{4.8}$$

The scalar triple product has(among others) the following three properties:

- Stays unchanged under a circular shift. $A * (B \times C) = B * (C \times A) = C * (A \times B)$
- Swapping any two of the three operands negates the triple product eg. $A * (B \times C) = -B(A \times C)$
- Swapping the positions of the operators without re-ordering the operands leaves the triple product unchanged. $A * (B \times C) = (A \times B) * C$

Applying 4.8 to 4.7 gives:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) * E_1} \begin{bmatrix} (T \times E_1) * E_2 \\ (D \times E_2) * T \\ (T \times E_1) * D \end{bmatrix} \tag{4.9}$$

Finally denoting again, $F = D \times E_2$ and $Q = T \times E_1$ the final equation becomes:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{F * E_1} \begin{bmatrix} Q * E_2 \\ F * T \\ Q * D \end{bmatrix} \tag{4.10}$$

Notice the properties of the scalar triple product were used to create the minimum amount of different cross products to be calculated. F and Q appear twice so instead of calculating 4 different cross products, only 2 are needed

and they are reused.
This way we can finally calculate the barycentric coordinates u, v and w is given by $w = 1 - u - v$. The distance t is also calculated during the process. As we discussed earlier $u + v \leq 1$ and each of the u, v, w need be positive numbers. If any of the coordinates is negative this tells us that the intersection point was outside the triangle and thus its a no hit. By doing some simple checks we can determine weather or not our ray did in fact intersect with the triangle.

4.3.2 Sphere Intersection

A ray-spheres intersection occurs to determine whether or not a ray intersects a sphere. The math for this intersection is fairly straight forward. First of all a sphere is defined by one point, its center and its radius. The sphere equation in 3D space is:

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = R^2 \tag{4.11}$$

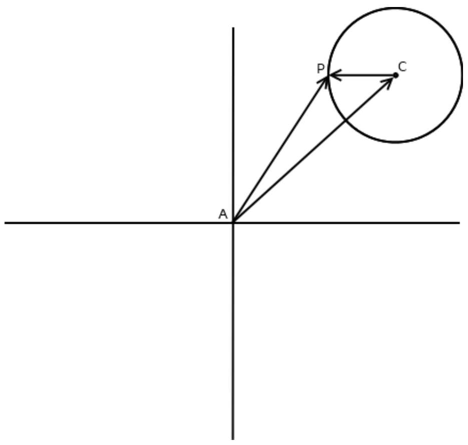
where (C_x, C_y, C_z) is the center, and R^2 is the radius of the sphere.
Its preferable to have this equation in vector form to keep x, y, z more tidy. So the equation becomes:

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = (P - C)^2 \tag{4.12}$$

where point P is (x, y, z) and C is the center (C_x, C_y, C_z) so our final sphere equation becomes:

$$\left.\begin{aligned}\vec{CP} &= \vec{AP} - \vec{AC}\\ \vec{CP} &= R\end{aligned}\right\} R = P - C \Leftrightarrow$$

$$R^2 = (P - C)^2$$



(A) ray-sphere roots

The above equation means that any point P that satisfies the equation is on the sphere. Our goal is to decide whether or not our ray $P(t) = O + tD$ intersects the sphere. If it does, then there must exist some t for which $P(t)$ satisfies the sphere equation. We are looking for a t that this becomes true:

$$R^2 = (P(t) - C)^2 \Leftrightarrow R^2 = (O + tD - C)^2 \tag{4.13}$$

Finally, if we expand this a little bit more and solve for t we are left with:

$$\boldsymbol{t^2 * D^2 + t * 2D(O - C) + (O - C)^2 - R^2 = 0} \tag{4.14}$$

This is second degree polynomial equation or quadratic equation. Depending on the solution we find there are 3 possible outcomes:

- 0 roots. If 0 roots are the result then it is decided that the ray did not intersect the sphere.
- 1 root. If 1 root is the result, it is decided that the ray did intersect the sphere at its periphery and its a tangent.
- 2 roots. If 2 roots are the result it is decided that the 2 roots are 1 in front of the sphere and 1 in the back of the sphere, so the closest root to the camera is chosen as the intersection point.

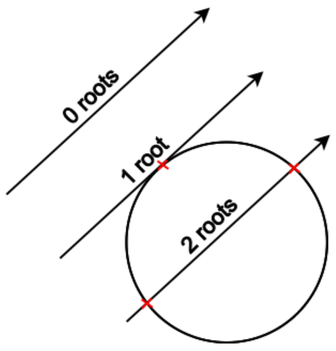


FIGURE 4.3: ray-sphere roots

4.3.3 Axis Aligned Bounding Box intersection

The most common way to test for intersection between an AABB and a ray is by using the "Slab" method[21]. A "slab" is the interval between two end points. In 4.4 we can see that a 2D box can be defined by 2 slabs vertical to each other(one being the slab between red lines x1-x2 and the other the slab between blue lines y1-y2).

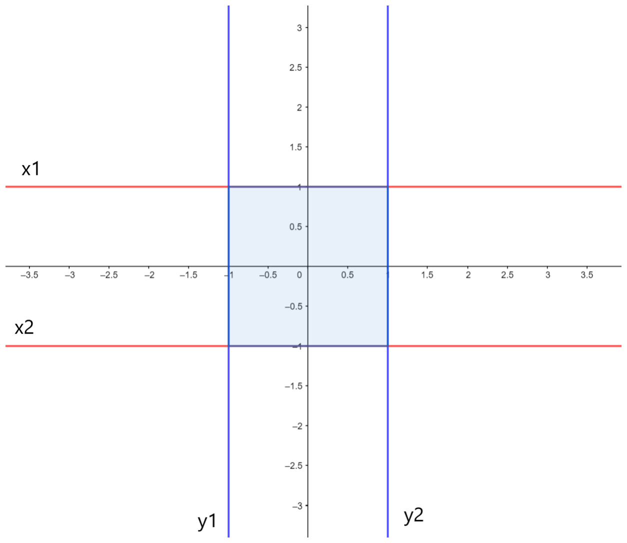


FIGURE 4.4: 2D Axis Aligned Bounding Box defined by slabs

The ray intersection between a ray and a slab can be seen below(4.5).

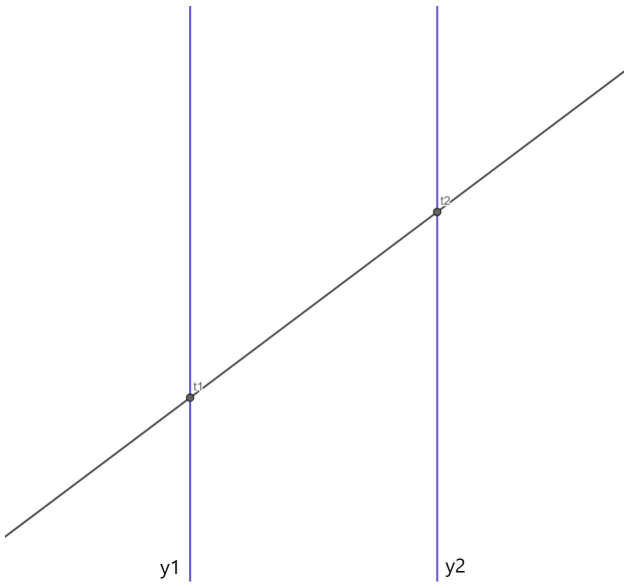


FIGURE 4.5: Ray-Slab Intersection

The equations for the planes y_1 and y_2 are $y = y_1$ and $y = y_2$. In order to test whether or not the ray intersects this slab, the values of the points t_1 and t_2

are needed. Earlier we discussed that the ray equation is $P(t) = O + tD$. This equation describes all 3 x,y,z coordinates. For a single plane eg. y, it would be $P_y(t) = O_y + tD_y$. So a ray will intersect plane y_1 at the point where the ray equation is satisfied for $P_y(t) = y_1$.

$$P_y(t) = O_y + tD_y \Leftrightarrow y_1 = O_y + t_1 D_y \tag{4.15}$$

And the hit point t_1 is:

$$t_1 = \frac{y_1 - O_y}{D_y} \tag{4.16}$$

Similarly, the equation for t_2 would be:

$$t_2 = \frac{y_2 - O_y}{D_y} \tag{4.17}$$

The key observation of the slab method, is that the above $[t_1 - t_2]$ intervals need to overlap on different axis for the same box. When they don't overlap, the rays doesn't pass through the box, and when they do overlap the ray passes through the box as shown in 4.6.

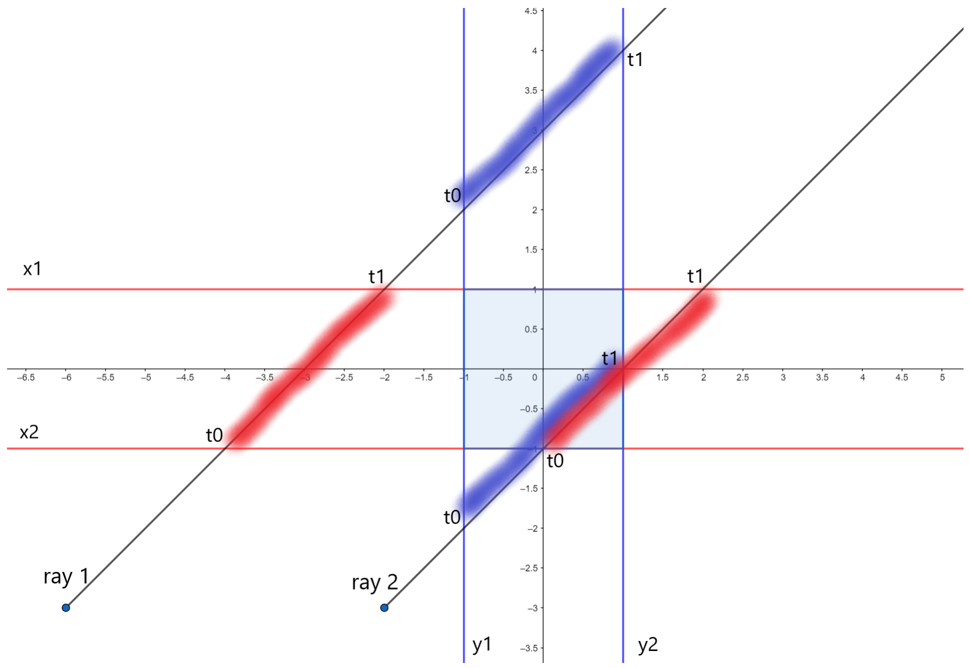


FIGURE 4.6: 2D Ray-Box Intersection
Ray 1 misses the bounding box and the blue-red areas don't overlap.
Ray 2 hits the bounding and the blue-red areas overlap.

By finding $[t_0 - t_1]$ intervals on both axis for a ray, and then checking weather or not their regions overlap we can decide on the intersection test.

The above method scales up to 3 dimensions the same way, by testing for all 3 axis. It is a great method since the math for it are simple and fast.

4.4 Polygon Meshes

One very important feature that our Ray Tracer was lacking, was the ability to render triangles. This is very important because most modern modeling programs are able to create complex objects that are described as polygon meshes. These polygon meshes are constructed by smaller and smaller structures the base of which are a simple triangle. The triangle meshes that we used are in the .obj format.
So in order to be able to add those complex triangle meshes to our ray tracer we needed 2 things. A way to read the .obj file and a way to render the triangles that it would add to our algorithm. In order to accomplish this we advised a GitHub project from the user bicknyers[22].

4.4.1 Mesh

A polygon mesh is a collection of triangles stored in a file which define a more complex shape when added together. These polygon meshes are created using 3d software such as Cinema 4D and Autodesk 3ds Max. Our object format of choice was the .obj file format. Reading data from an .obj file was a fairly simple process. The file contains lines of the following type:

```
v v1 v2 v3
v v4 v5 v6
.
.
.
f n1 n2 n3
f n41 n5 n6
.
.
.
```

Lines that start with the letter `v` describe vertices and `f` describe faces. Based on bicknyers project, we created a tokenaziation setup were the input is tokenized and combined for the correct polygons to be added. Since the .obj format is flexible so that an `f` line(face line) can describe multiple faces, support was added. Some of the faces are also grouped so that specific materials are used for them.

4.5 Main Functions

4.5.1 The World_Hit function

This function is responsible for traversing the BVH tree and returns true when an object is intersected, along the object's data.The function reads node data from the BVH structure and performs intersection tests to find the closest object. This is were the ray tracer will be spending most of its time. In our tests we calculated that about 90% of the total rendering time is spent in this function. This function will be a major topic for discussion in the upcoming sections so we will briefly be looking into its functionality and provide some pseudo-code.

Algorithm 1 Software BVH traversal function

```
1: procedure WORLDHIT($ray, output$)
2:     if not $box.hit(ray)$ then
3:         return $false$
4:     $hit\_left \leftarrow$ WORLDHIT $for\ left\ child$
5:     $hit\_right \leftarrow$ WORLDHIT $for\ right\ child$
6:     return ($hit\_left$ OR $hit\_right$)
```

This is a heavily recursive class function. Left and right children as well as the box are part of the class' data. A depth-first search on the binary BVH tree is performed to find the closest object. It is possible for both hit_left and hit_right to test positive which creates multiple possible results, in which case the closest of those to the camera.

Chapter 5

GPU Implementation

After adding all the features we wanted to the ray tracer our goal was to create a framework in order to run the Ray Tracing algorithm on a GPU(Graphics Processing Unit). Our framework of choice was the openCL programming framework.

5.1 What is OpenCL

OpenCL is a programming framework set from the Khronos group for writing programs that execute across heterogeneous platforms including GPUs. A CPU based "Host" is responsible for controlling multiple compute devices,

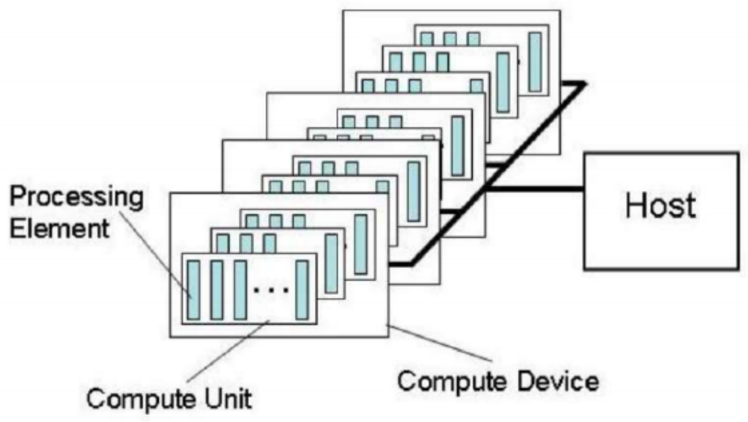


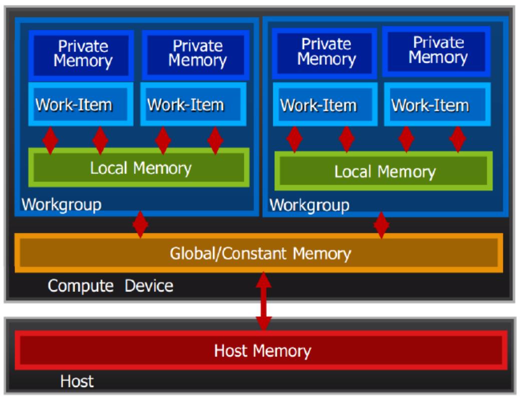
FIGURE 5.1: OpenCL architecture. Image taken from [23]

such as CPUs, GPUs and even FPGAs. These compute devices consist of multiple "compute units", which, on their end are made of multiple "Processing Elements". At their core, these "Processing Elements" execute openCL "kernels". The aforementioned "compute units" vary from device to device. We could say that in FPGAs these "compute units" are our custom made IPs.

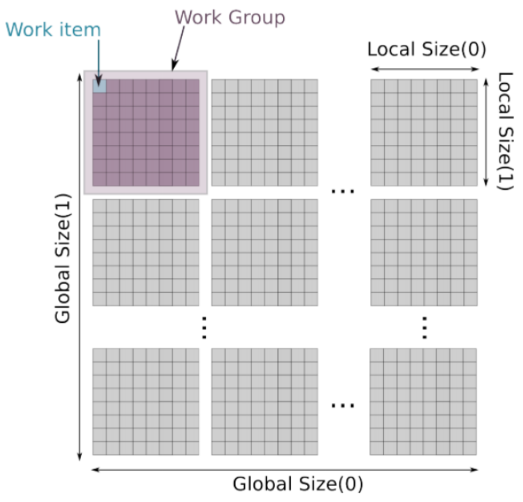
Nvidia and AMD call their "compute units" "stream multiprocessors" and "stream cores" respectively. They both have fairly complex hardware architectures, but at the bottom line its important that they can execute SIMD and VLIW instructions.

5.1.1 OpenCL execution

At the top level, an OpenCL host uses the OpenCL API to select compute devices, submit work to them. At the other side of the execution hierarchy lie the OpenCL kernels, running on the processing units. The code for those kernels is written in openCL C language and execute in parallel in a predefined N-dimensional work domain. Each element of in this domain is called a "work item". These "work item" can be grouped together to form "work groups".



(A) OpenCL memory model. Image taken from [23]



(B) OpenCL work groups. Image taken from [24]

To sum everything up, some basic pipeline can be outlined for an OpenCL application.

1. A host defines an N-dimensional computation domain, an array. Every work item will be assigned to one of the array's index. Eventually, each work item will run the same kernel code to compute his part.
2. Host defines grouping of those item into work groups. Work item in the same work groups can share the same memory which can have major benefits in the overall throughput.
3. Host "sends" data to the device RAM.

4. Work items start their work in parallel to compute the entire result of the domain.

5.1.2 The openCL API

The openCL API exists to control the platform and execute code on the device. The same API is used by the xilinx Vitis tool to communicate with different FPGAs. For GPUs the API follows the following steps to connect to a device and run some code:

1. Create context
2. Create Command Queue
3. Create buffers holding the data that the device need to process
4. Create program providing the kernel code
5. Build the program
6. Create Kernel by denoting the main function
7. Set arguments for the kernel including the aforementioned buffers
8. Define the N-D range of the kernel
9. Execute the kernel
10. Deallocate the buffers

The 9th step, "Execute the kernel", works by calling the `clEnqueueNDRangeKernel(...)` command that the API provides. This command takes an NDRange as an input and divides the workload into separate threads to provide parallelism. In our case, the workload is the pixels of the image multiplied by the number of samples we want to acquire per pixel. A thread is spawned for each pixel in the workload and the device performs the computation.
A design choice was made here. Calling the kernel one time for the entire workload, which means $\#pixels \times samples\_per\_pixel$, was causing major stability issues and sometimes was hitting some internal timeout since the GPU is also used to provide picture. To provide some relaxation to the system, we decided to call the kernel multiple times for a smaller workload, in fact $samples\_per\_pixel$ times. This means that instead of calculation the integrated color of each pixel and then move to the next, we make a pass one the entire image once, and then start again. This resulted in $samples\_per\_pixel$

different images which were combined together to ensure monte-carlo integration.The major potential bottleneck in this case would be I/O between CPU memory and GPU memory. If our BVH structure, which carries the biggest cost in memory, is stored in device memory only once, then the I/O would be greatly reduced. After testing, it was assured that the time difference between calling the kernel multiple times for smaller workloads or calling it just once for a bigger workload, was negligible.

5.1.3 OpenCL C limitations

In our attempt to run the code in OpenCL on an external device(GPU) we faced three major problems.
The first problem we faced when trying to run our code in OpenCL is the fact that our entire C++ code was constructed in an inheritance manner. OpenCL C doesn't support any type of inheritance so our entire project needed to be converted to a more basic C style.
Another problem that we had to deal with is that the basic loop of our algorithm was recursive. OpenCL C does not support any kind of recursion either. We needed to figure out a way to transform our code so that it could run.
Our final problem was a combination of the above two problems. Our implementation uses a BVH structure which, in its nature, is a binary tree used to divide the objects and accelerate the search of the closest object. There was no way to either traverse the tree in a recursive way nor send its data to the device if a new way to store data wasn't introduced. So we needed to figure out a way to combat those problems.

5.2 Converting data Objects

For our basic objects such as spheres or triangles instead of having classes describing the object and its functions, we created structures that held the variables of each class and functions with different names for each class function. This is an example of what a conversion looks like:

```
C++:                          OpenCL:
example_class(){              struct example_struct{
var1;                         var1;
var2;                         var2;
}                             }
func1::example_class(){};     func1_of_example_struct(){};
```

More information regarding these structures and how they reference each other will be given in later chapters.
Another important thing to note here is that we took advantage of the types that OpenCL provides. We mostly used the `double3` type instead of the custom class, `vec3`, that we had in C++ which describes a 3D vector. A `double3` element can contain the coordinates of a vertex `(x,y,z)` or the color of a pixel `(r,g,b)`.

5.3 Converting the BVH data stracture

Finding a way for our BVH structure to run on the external device was the hardest problem we had to solve and a few major changes needed to be implemented. First of all the data structure that was holding all the information consisted of 3 different elements, Axis-Aligned Bounding Boxes, Spheres and Triangles. A visual representation of the structure would be like:

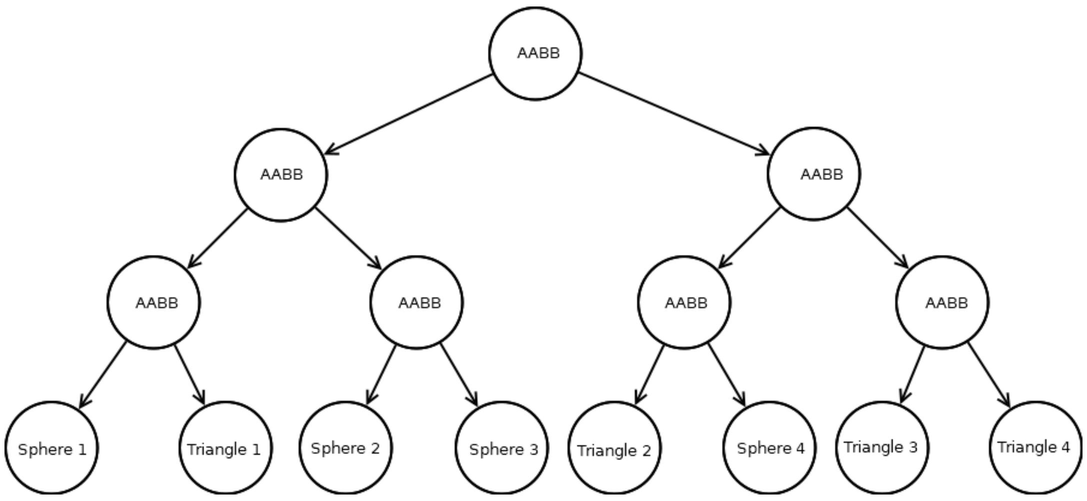


FIGURE 5.4: Visual Representation of BVH binary Tree

Every node of the tree contains an AABB object and every leaf of the tree contains an actual object of our scene. openCL as a language doesn't allow dynamic data structures, so a conversion from a dynamic to a static data structure was needed. Our idea was to fully traverse the already defined dynamic data structure and and to create static arrays with the corresponding data. For this job 3 major arrays were created:

- **Nodes array**: contains the node elements of the BVH.
- **Spheres array**: contains the sphere elements.
- **Triangles array**: contains the triangle elements.

5.3.1 Sphere and Triangle Arrays

The elements of the sphere and the triangle arrays are similar with each other. They both contain the data that describes each object and a material.

```
struct cl_sphere{
cl_double3 center;
cl_double radius;
cl_material mat;
}
```

```
struct cl_triangle{
cl_double3 v1;
cl_double3 v2;
cl_double3 v3;
cl_material mat;
}
```

The index of each of those element's position in their corresponding array acts as a unique key that will be used when trying to acquire the correct element from each array.

5.3.2 Nodes Arrays

The nodes array contains information about the each node of the BVH tree. Its elements consist of the bounding box that each node refers to and 4 important variables.

```
struct cl_node{
cl_aabb boudning_box;
cl_int left_node;
cl_int right_node;
cl_int obj_pointer;
cl_int obj_type;
}
```

Variable `obj_type` holds information about the node's nature. We have set that:

- `obj_type=0` the current node refers to a node.
- `obj_type=1` the current node refers to a sphere.
- `obj_type=2` the current node refers to a triangle.

Variables `left_node,right_node` are keys that contain the indexes of elements in the same array. Their information is only valid when `obj_type=0`. These elements represent the left and right pointers of the dynamic binary tree. When a node's children are not leaves, the keys are positive integers and for they are leaves the keys are set to `-1` because the information is not valid.

Finally, variable `obj_pointer` holds information about the object that the node points to if that points to one. This information is only valid when `obj_type!=0` thus it is set to `-1` for all the inner nodes of the tree and to a number for the leaves. This key is an index to an element of either the triangle or the sphere array. The appropriate element is obtained from one of those arrays when the traversal is finished based on this index.

5.3.3 Creating the Arrays

A function was created to perform the aforementioned conversion. The function recursively goes over every node of the tree in a Death First Search manner. Since it goes over every element of the tree it has a time complexity of `O(n)`. In our tests, compared to the time it took for the BVH dynamic data structure to be created, the time that the conversion took was negligible, about `1%` of the time. Pseudo-code of the function follows:

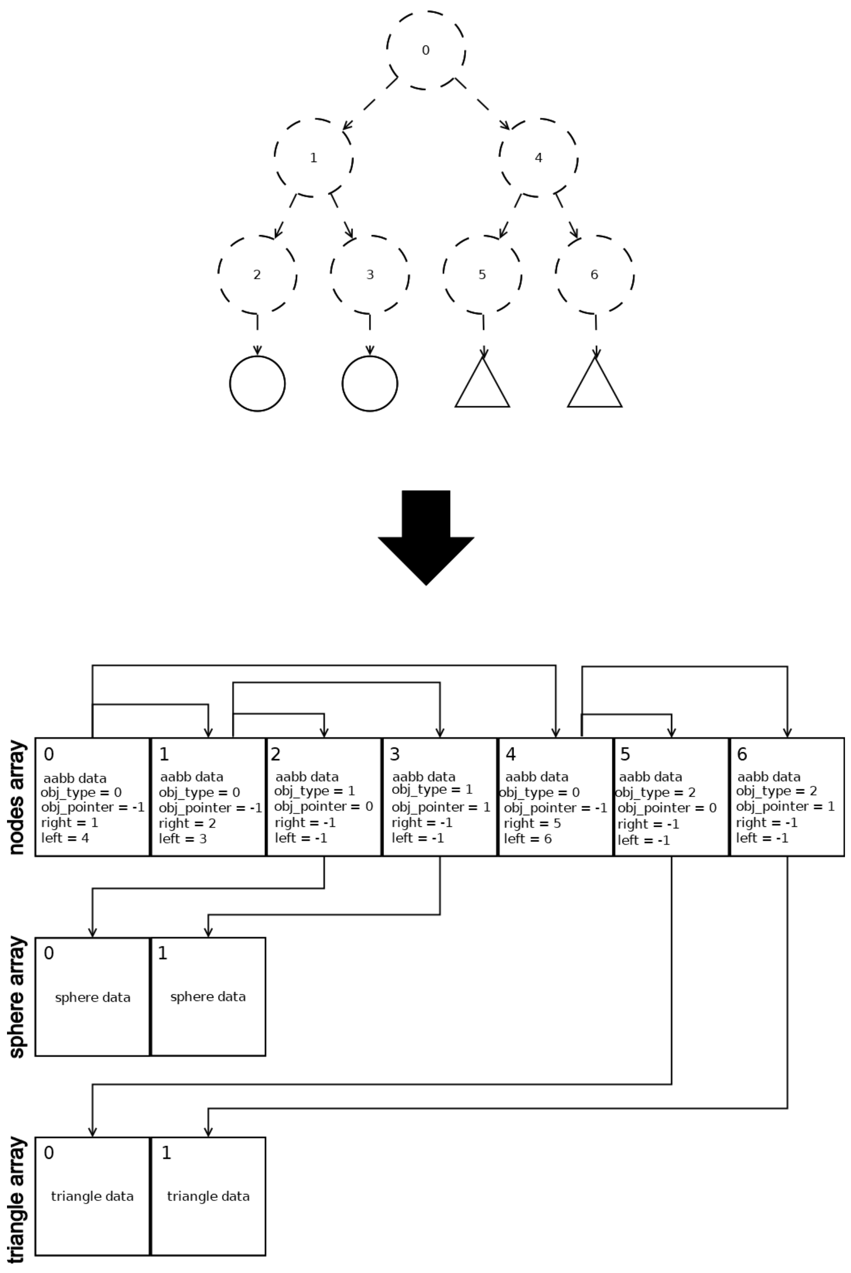

0
1
4
2
3
5
6
nodes array
0 aabb data obj_type = 0 obj_pointer = -1 right = 1 left = 4
1 aabb data obj_type = 0 obj_pointer = -1 right = 2 left = 3
2 aabb data obj_type = 1 obj_pointer = 0 right = -1 left = -1
3 aabb data obj_type = 1 obj_pointer = 1 right = -1 left = -1
4 aabb data obj_type = 0 obj_pointer = -1 right = 5 left = 6
5 aabb data obj_type = 2 obj_pointer = 0 right = -1 left = -1
6 aabb data obj_type = 2 obj_pointer = 1 right = -1 left = -1
sphere array
0 sphere data
1 sphere data
triangle array
0 triangle data
1 triangle data

FIGURE 5.6: BVH tree to Arrays conversion example. Note that the arrows in the picture are not pointers. They are there for better visual understanding.

Algorithm 2 Tree Conversion Algorithm

```
 1: procedure TREE_CONVERSION($ptr(tree\_node, nodes\_array, spheres\_array, triangles\_array)$)
 2:     $temp\_cl\_info\_pack \leftarrow$ GET_INFO($tree\_node$)
 3:     switch $temp\_cl\_info\_pack.obj\_type$ do
 4:         case 0                                   ▷ Stores node data and begins recursion
 5:             $nodes\_array \leftarrow temp\_cl\_info\_pack.node\_info$
 6:             if ($tree\_node \rightarrow left$ exists) then
 7:                 TREE_CONVERSION($tree\_node \rightarrow left, nodes\_array, spheres\_array, triangles...$)
 8:             if $tree\_node \rightarrow right$ then
 9:                 TREE_CONVERSION($tree\_node \rightarrow right, nodes\_array, spheres\_array, triangles...$)
10:             break;
11:         case 1                                   ▷ Stores sphere data
12:             $spheres\_array \leftarrow temp\_cl\_info\_pack.sphere\_info$
13:             break;
14:         case 2                                   ▷ Stores tiangle data
15:             $triangles\_array \leftarrow temp\_cl\_info\_pack.triangle\_info$
16:             break;
17:         case default                             ▷ Never goes here
18:             break;
        return
19: function GET_INFO($ptr(node)$)
20:     Returns information regrading the specific node.
21:     if node points to a node, it collects the aabb of the node.
22:     if node points to a sphere, it collects the sphere data.
23:     if node points to a triangle, it collects the triangle data.
24: return $info$
```

5.4 Converting Recursion logic into Iterative logic

Our original CPP algorithm was, for the most part, recursive. Since OpenCL doesn't allow recursive algorithms, a general re-construction of the dataflow was needed. The algorithm was broken down into its major functions and loops:

- Spawn_rays loop : This loop was part of the main function. It runs for every pixel of the screen and spawns a ray for it.
- World_hit function: This is a recursive function that returns true when a ray intersects an object. It also returns the color data of that object.
- Scatter function: Takes as input the color data of the previous function, shades the pixel and creates a new ray, the "scattered" ray, at the impact location.

- Object_hit functions: Takes a ray as an input and returns true when the ray intersects the object.

5.4.1 Top level

Similarly to the original project, a "top level" function was needed to connect everything together. In the original project this process was also done in a recursive manner and needed to be modified. Again, we decided to use a *while* loop to perform the iterations. This function performs a `world_hit` followed by a `Scatter` operation. A new ray is spawned and the process is repeated for the spawned ray. When a ray doesn't intersect any object on the scene, it is considered as an escaped ray, thus the process is complete. A threshold is set for potential "trapped" rays who bounce between objects indefinitely. This effectively means that up to threshold amount of bounces can be performed before the computation is complete for a pixel. Finally, a color that corresponds to a pixel is calculated and the process is finished. Pseudo-code for this process:

Algorithm 3 Color function in openCL

```
procedure COLOR($primary\_ray$, ALL ARRAYS, $background$)
    $temp\_ray \leftarrow primary\_ray$
    $temp\_color \leftarrow (0,0,0)$
    $final\_color \leftarrow (1,1,1)$
    $depth \leftarrow 0$
    while $depth < threshold$ do
        $[WH\_res, object] \leftarrow$ WORLD_HIT($temp\_ray$, ALL ARRAYS)
        if not $WH\_res$ then
            $final\_color \leftarrow final\_color * background$
            break
        $[Sc\_res, scattered\_ray, temp\_color] \leftarrow$ SCATTER($temp\_ray, object$)
        if not $Sc\_res$ then
            $final\_color \leftarrow (0,0,0)$
            break
        $temp\_ray \leftarrow scattered\_ray$
        $final\_color \leftarrow final\_color * temp\_color$
        $depth++$
    return $final\_color$
```

5.4.2 World_Hit Function

In 4.5.1 we discussed the purpose and usage of this function. Since its a recursive algorithm, we had to make some major changes for it to work in openCL. We know that its purpose was to traverse the BVH tree and return the closest intersected object. All rays start the traversal from the root element. Instead of using recursion to traverse the tree, we used a *while* loop. On every iteration of the loop, intersection with the current node's children is tested. When on of those checks true, the current node is set to that child and the next iteration begins. Eventually, the bottom of the tree is reached were a real object indicator lies. Polygon intersection is tested against that object and, if it is hit, the color is returned.

There is a catch in this process. Earlier we discussed that due to the BVH's nature of having overlapping AABBs it is possible for both the left and the right child intersection test, to test positive. In this case, both paths starting on those children need to be traversed, but only one can be tested per iteration. In order to deal with this issue, we used a cache array to store the index of the second child. It was calculated that in the highly unlikely case of this happening for every node, which is the worst case scenario, the size of this array will need to be equal to the depth of the BVH to store all the extra not yet traversed children. When this "double intersection occurs" an index to that second element is stored in the cache array. At the end of the algorithm, if the cache array is empty we can safely return. Otherwise, an element is poped from the array and the process continues at that node. This method looks more like a breadth first search rather than a depth first search which was used in software. A flowchart and pseudo-code for this process follow:

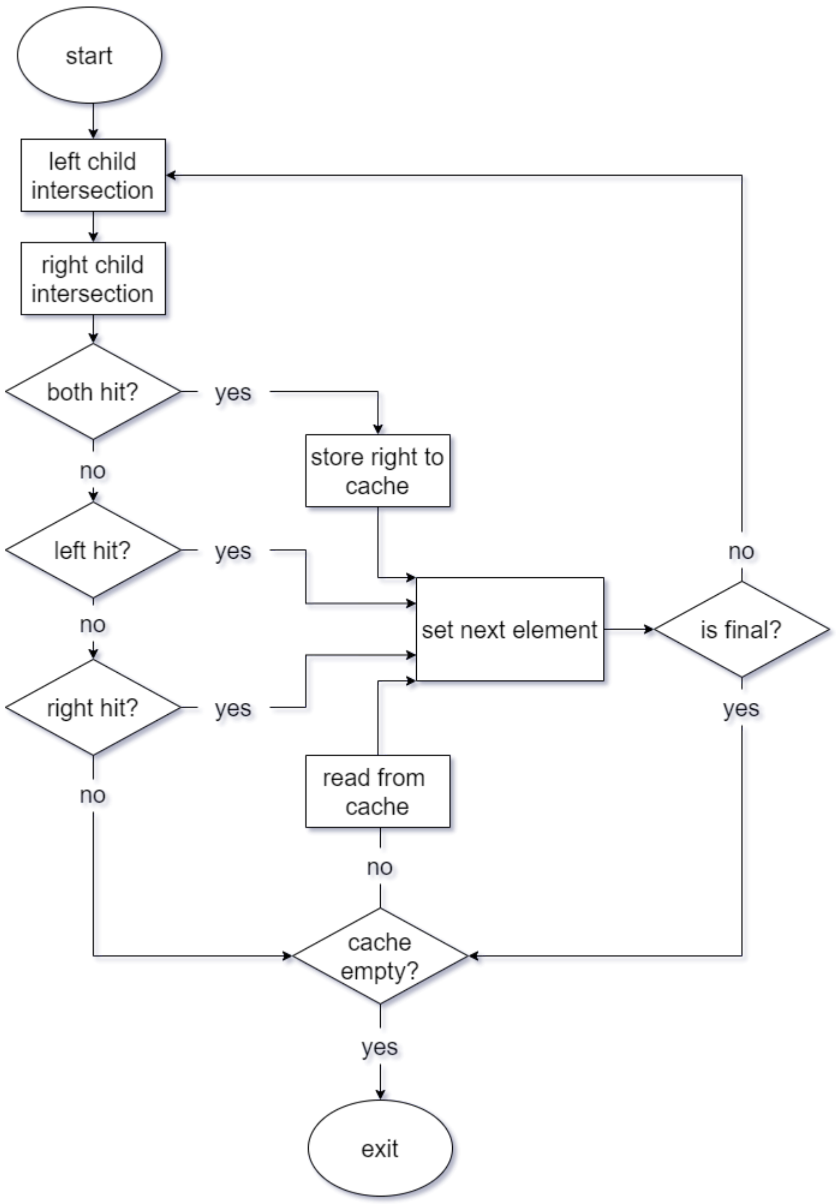


FIGURE 5.7: Flowchart describing the world hit process in openCL.

Algorithm 4 World_Hit in OpenCL

```
procedure WORLD_HIT(ray, ALL ARRAYS)
    $cache\_array[tree\_depth]$
    $cur\_obj \leftarrow nodes\_array[0]$
    while $cur\_obj.type = 0$ do
        $left\_child \leftarrow nodes\_array[cur\_node.left]$
        $right\_child \leftarrow nodes\_array[cur\_node.right]$
        $hit\_left \leftarrow$ OBJECT_HIT($r, left\_child$, ALL ARRAYS)
        $hit\_right \leftarrow$ OBJECT_HIT($r, right\_child$, ALL ARRAYS)
        if $hit\_left$ and $hit\_right$ then                         ▷ Double hit
            $cur\_obj \leftarrow left\_child$
            $cache\_array \leftarrow right\_child$
        else if $hit\_left$ then                                     ▷ Left hit only
            $cur\_obj \leftarrow left\_child$
        else if $hit\_right$ then                                    ▷ Right hit only
            $cur\_obj \leftarrow right\_child$
        else                                                         ▷ No hit
            if $cache\_array \neq$ empty then                        ▷ get from cache
                $cur\_obj \leftarrow cache\_array.pop()$
            else
                break
        if $cur\_obj.type \neq 0$ and $cache\_array \neq$ empty then
            $cur\_obj \leftarrow cache\_array.pop()$
    return
function OBJECT_HIT($ray, cur\_obj, spheres\_array, triangles\_array$)
    switch $item.obj\_type$ do
        case 0
            return BOX_HIT($ray, cur\_obj.bounding\_box$)
        case 1
            return SPHERE_HIT($ray, spheres\_array[cur\_obj.obj\_pointer]$)
        case 2
            return TRIANGLE_HIT($ray, triangles\_array[cur\_obj.obj\_pointer]$)
        case default                                                 ▷ Never goes here
            return $false$
```

5.4.3 Scatter function

Scatter function's purpose was to perform scattering of light at the location of the intersection. Depending on the type of the object's material, light scatters differently. In he original code, inheritance was applied to the entire project. After world_hit (5.4.2) returns a object that is intersected, this function is called to create a new ray on the impact location(scattering) and calculate the color that should be projected on that pixel.
The material structure hold information about the material.

```
struct cl_material{
cl_double3 albedo;
cl_double fuzz;
cl_double refIdx;
cl_int type;
}
```

There are 3 different materials in our ray tracer. The `type` variable is a reference to the type of the material. The others are data regarding the material.

- type=0 Refers to a lambetian material.
- type=1 Refers to a metal material.
- type=2 Refers to a dielectric material.

Depending on the `type` variable the scattering is performed.

Algorithm 5 Scatter function in openCL

```
procedure SCATTER($ray, object$)
    switch $object.material.type$ do
        case 0
            return $[new\_ray, color] \leftarrow$ LAMBERTIAN($ray, object.material$)
        case 1
            return $[new\_ray, color] \leftarrow$ METAL($ray, object.material$)
        case 2
            return $[new\_ray, color] \leftarrow$ DIELECTRIC($ray, object.material$)
        case $others$
            $false$                                        ▷ Never goes here
```

Chapter 6

FPGA Implementation

The final part of this project was to once again, convert and create the appropriate units to fit in a FPGA board. Our FPGA of choice was one of the alveo family, the alveo U50. The alveo U50 is a modern but small reconfigurable logic board. This board is connected to the host computer via PCI which offers high datarates between host's RAM and the device's RAM. To program architectural units we used Xilinx's Vitis IDE and Vitis HLS.

6.1 Tools Used

6.1.1 Vitis IDE and Vitis HLS

The Vitis IDE is a high-level environment used to program architectures for FPGAs. There are 2 sides on every program created on the Vitis IDE. The way it is set up, is by having 2 sides on every program, the host side and the kernel side.

The host side is responsible for the creation and collection of data as well as to transfer them to the FPGA. The code used here is of the C++ programming language. Much like our GPU implementation, the host code uses the openCL API to create the required buffers and structures to ensure data transfer to the FPGA as well as to deploy kernels.

The kernel side is where the real work is being done. This is were we begin to imagine our architecture. The programming language used here resembles C++, but once again is limited. Directives are used to provide different mode of behaviour for numerous implementations and shape the design units.

In our implementation, we primarily used the following directives:

- `DATAFLOW`. Enables task level pipelining. This directives automatically attempts to pipeline the tasks themselves as well.
- `ARRAY_PARTITION`. Partitions an array into a smaller array of individual elements.
- `INTERFACE`. Creates the requires input/output operations by using the I/O protocols.
- `INLINE`. Inlines the code of a function into the caller function.

Finally, we used the Vitis HLS tool to perform synthesis of our kernel. The tool performs synthesis and reports back what it managed to achieve based on the directives we provided, as well as report the utilization on the chosen FPGA.

6.2 Parallelizing Data Transfer | Multiple Banks

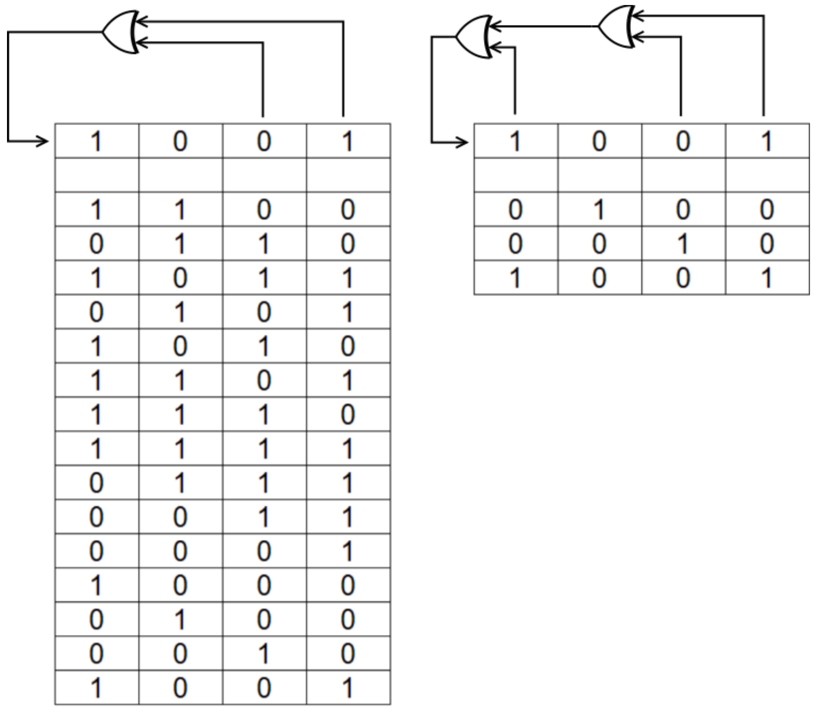
When it comes to data transfer, there are 2 parts that need to be considered. First of all, the data transfer between the host's RAM and the device's RAM. Many FPGAs use the USB interface to transfer data. In our project we aimed to use the Xilinx Alveo family FPGAs. These boards are connected to the host computer through the PCI-e 3.0 interface. The PCI-e interface is broken down into lanes. Every lane has a max bandwidth of 984.6 MB/s. The Alveo FPGA board is connected to the x16 connector. This provides a total bandwidth of 15.8 GB/s. There isn't much we can do to change this and its already fast enough compared to the process.

The second part that need to be considered is, the data transfer between the device RAM and the device core.The Alveo family boards provide 2x4GB HBM stacks with a total of 32 HBM AXI interfaces or "banks". These banks can transfer data in parallel from the device's memory to the IP cores. The data in the device HBM memory is in the form of arrays. In our application we have multiple arrays containing data so having the ability to read from and write to those arrays in parallel is important. Specifically, our application contains 3 input buffers and 2 output buffers. Those buffers were assigned to different banks so that level of parallelism can be achieved.

6.3 Random Number Generation

6.3.1 Linear Feedback Swift Register

One of the first problems we had to deal with when creating our hardware implementation was random number generation. So far, random number were generated using the "Mersenne Twister" pseudo-random number generator. After some research, we decided to use Linear Feedback Shift Registers(LFSR). The LFSR is a shift register for which the input bit is the combined outcome of some of its other bits. More specifically, the input bit is defined by selecting some of the LFSR's bits and driving them through an XOR gate together. The result of the XOR is then passed as an input to the most significant bit.
For the LFSR to start his process an initial state of the bits need to be defined called a "seed" state. After performing a number of shifts and generating different pseudo-random numbers, it is inevitable that the original "seed" state of the LFSR will eventually be met. At that point the LFSR will repeat generating the same numbers as before in the same order. The amount of different pseudo-numbers an LFSR can provide before reaching its initial state is called the "period" of the LFSR.
The bits which are chosen to be XORed together to provide the input for the LFSR are called "taps". In the figure above we can see that selecting the right taps for an LFSR is of major importance. On the left side of the figure we can see that bit 1 and 2 were selected as the taps which resulted in a 16 cycle period.
On the right side, we can see that the bits 1,2 and 4 were selected as the taps. Even though this looks like it should provide a greater period since more bits are combined and more hardware is used, that is not the case. After only 3 clock cycles the initial state showed up, resulting in a 3 cycle period.
What is also interesting about the above figure, is that it describes 4-bit register. With 4 bits it is possible to represent up to $(^4-1 = 15$ different numbers. Selecting bit 1 and 2 as the taps resulted in 16 different numbers, thus it provided the maximum potential period it could. Number 0 can't be encountered since XORing any number of 0 together, from any positions results in 0 which would lock the LFSR to only produce 0.



1	0	0	1
1	1	0	0
0	1	1	0
1	0	1	1
0	1	0	1
1	0	1	0
1	1	0	1
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1
1	0	0	0
0	1	0	0
0	0	1	0
1	0	0	1

1	0	0	1
0	1	0	0
0	0	1	0
1	0	0	1

FIGURE 6.1: Period difference when selecting different taps in a 4bit LFSR

6.3.2 32-bit LFSR

In our case, we needed to create 32 bit floating numbers, thus we constructed a 32-bit LFSR. As we discussed earlier it is important that the right taps need to be selected to provide the maximum period. In a document published by Xilinx in 2007 [25] it is stated that for a 32 bit LFSR, if bits 32,22,2 and 1 are selected, the maximum period will be encountered.

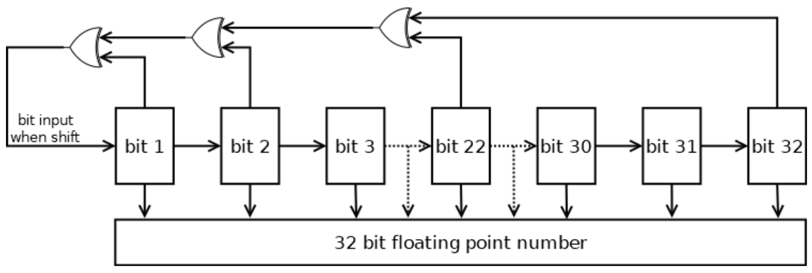


FIGURE 6.2: 32 bit Fibonacci LFSR with maximum period

This LFSR has the maximum potential period which is $2^{32} - 1 = 4,294,967,295$. Now we can appreciate the fact that in hardware, such a simple design, just a register and a few gates, can provide such a large amount of pseudo-random numbers. This is what makes the LFSR a great design for the job.
The LFSR design we were discussing so far is called the "Fibonacci" LFSR. The "Galois" LFSR is another variant which works a bit differently. XOR

gates are placed in between a few of the bits instead of them being combined and only going to the input.

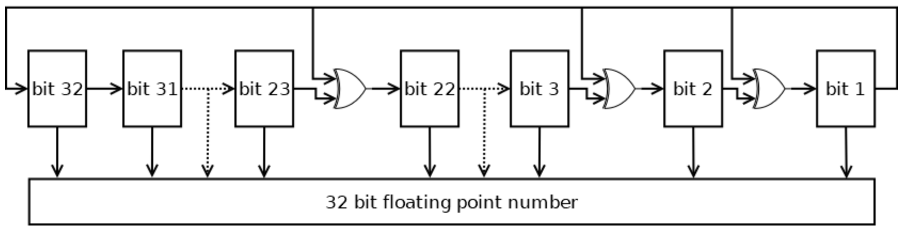


FIGURE 6.3: 32 bit Gallois LFSR

This can result in better frequency for the system since for the Fibonacci LFSR, the delay of the input is that of 3 cascaded XORs, whereas for the Galois LFSR the XORs are placed in between different bits resulting in 1 XOR delay. Since we only have 3 XORs and modern FPGAs use LUTs which typically support 6 logic functions, the result of those 2 shouldn't be different. We decided to stick with the Fibonacci design which is more simple to create.

6.4 The ultimate Goal

The ultimate goal for our kernel architecture is an architecture that will be able to produce, process and calculate the color of a ray every clock cycle.

6.5 First Generation

In our first Generation our goal was to create a setup without much change of the ray tracing algorithm as we already had it. Even though we weren't trying to achieve any considerable performance, a few changes still needed to be implemented. Even though the Vitis tool allows kernel to be programmed using openCL, directives for parallel data transfer between the host(CPU ram) and the kenrel(FPGA ram) only exist for C++ kernels. The Vitis tool also doesn't allow recursive functions because they are not synthesizable. Another thing that is a limitation for the Vitis tool are dynamic data structures. Xilinx in its user guide provides a technique to transform dynamic data structures into synthesizable counterpart using arrays and vectors.
All of this sounds a lot like the problems we needed to deal with when creating the openCL version of the kernel. But we couldn't use OpenCL because of the aforementioned reason. We decided to, once again, create another C++ kernel but with the elements of our OpenCL kernel. This meant that using

the very useful custom types of openCL(`float3,float4,float8 etc.`) was not an option. The vec3 class from the original C++ project was used once again to describe 3D vectors and RGB colors. The Tree Traversal and Array structure setup that we had created in OpenCL was used again in place of the recursive functions. We also used the 32-bit Fibonacci LFSR that we discussed in 6.3.1.

6.5.1 The problems

In the core of the Ray Tracing algorithm lies the tree traversal algorithm. Earlier it was mentioned that this stage is where most of the execution time is spent in software. At this point the tree traversal algorithm is very similar to the one that we used in openCL(5.7).

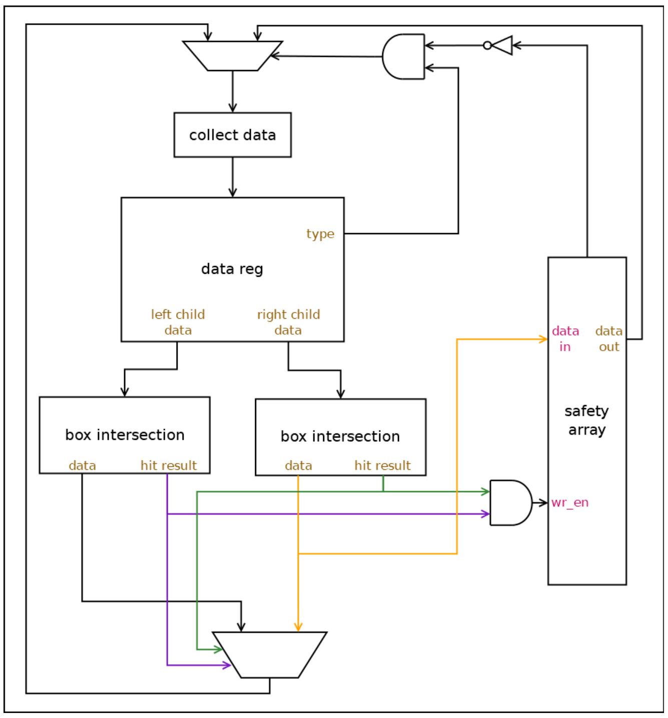


FIGURE 6.4: Traversal Unit Architecture. Some routes are colored for better clearance.

The Vitis tool performs pipelining on loops. The theoretical best performance we can achieve is 1 iteration of the loop per clock cycle. But even if we manage to achieve that, the core will not be processing 1 ray per cycle. It will be performing 1 step of the tree traversal at a time. This means that in the best case of 0 double box intersections, the latency of this core to process a ray will be #*TreeDepth* cycles. Since this is intended to be a pipelined algorithm it will be slowing down all other processing units.

6.6 Second Generation

As we know this is a repeating algorithm. The lifetime of a ray begins when it is produced and ends after its corresponding color is found. The ray produces a new ray which is fed back into the algorithm for that process to repeat. With this in mind, we outlined the spectrum of the kernel we intended to create. The first step is a unit related to creating rays, and the last part is a unit to write the color and send back the produced rays.
The algorithm is, once again, broken down into its major components and autonomous IP cores are created for each of those. The major components are:

- A ray creation unit.
- A tree traversal unit.
- An object intersection unit.
- A ray scatter unit.
- A write back unit.

A common way to send data between those cores and achieve pipelining is by connecting those IP core units using FIFO streams.

6.6.1 Random 3D Points and Vectors

Software

As we already discussed, the trajectory of the rays we create need to be slightly different to one another so that Monte-Carlo integration can be applied(2.5). These differences are performed by random slight adjustments to the trajectory of the ray. In the software implementation we had 2 major functions to provide those numbers.

- Random in Unit Circle. Creates a 3D vector in a unit circle.
- Random in Unit Sphere. Creates a 3D vector in a unit sphere.

These two functions work similarly. They create a random vector in the outer square square and the outer cube cube respectively. Then they check whether or not the created point solves the circle or sphere equation. If it does, that means the point is in the circle, if it doesn't, a new number is generated and it checked again until a correct number is found. We called this the "square" or "sphere" "sample method". The performance of this method can be evaluated

by comparing the volumes of the circle and the sphere relative to the square and the cube.
The volume or area of the circle is given by $D_{circle} = \pi * r^2$ and its a unit circle so $r = 1$. That means $D_{UnitCircle} = \pi = 3.14$.
The square around that circle has a side of size 2. Its area is given by $D_{square} = side_a * side_b$ and $side_a = side_b = 2$. That means $D_{square} = 4$.
The area of the circle covers about 78.54% of the area of the square. Since random points are generated in the range of $[-2, 2]$ and they land inside the square, we can say that there is probability of 78.54% or $P_{SquareCorrect} = 0.7854$ that they will also land inside the contained circle.
Using the same logic, the probability of a random point that is generated inside a cube to also land inside the contained sphere is 52.36% or $P_{SphereCorrect} = 0.5236$.
From this we can deduct that, statistically, about 1 out of 2 points generated will be correct for the circle, and about 1 out 3 will be correct for the sphere.
Another possible way to produce the same result is by using trigonometric functions. We tested both methods in software and it turns out that even though the "square-sphere sample method" might need to re-run a few times to produce the result, it is the fastest method. The also produced slightly different results. Based on that, we decided to use the square-sphere sample method for our kernel.

Hardware

In our implementation we are trying to achieve pipeline were 1 ray is processed per cycle, so the component that generates random points, needs to have an Interval of 1. In their current state, the random point generation functions can't provide this since they rerun until they find a fitting number and can potentially never end, even though that's highly unlikely. Since both of the functions work the same way but in different dimensions, we will analyze for 2 dimensions. Instead of performing the same operation sequentially, more units are added to perform the since cycle equivalent operation in parallel and when 1 of the results is correct, it is promoted. The more the units we add, the higher the probability of "hitting" a correct result. A compromise needed to be made. Since it is possible that none of our units will generate a point that land inside the circle, a final unit is added that behaves differently. Instead sampling the square around the circle(cyan in 6.5,[HGFE]), this unit samples the square inside the circle(blue in 6.5,[BADC]). This guarantees that the point will land inside the circle, but also introduces some bias since some

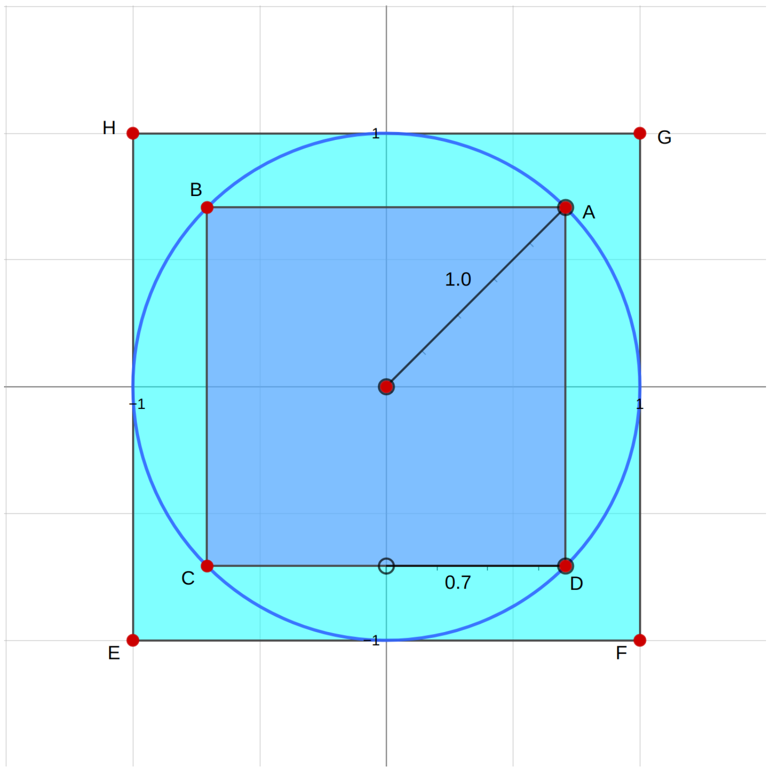


FIGURE 6.5: Square and Circle Geometric Representation

areas of the circle will never be hit.

We attempted to create a setup were at least 99% of the time the random point will be generated normally, from the outer square [HGFE] and the process can be pipelined to provide 1 random point per cycle. We know that $P_{SquareCorrect} = 0.7854 = 78.54\%$. Adding extra units that perform the same operation increases the probability of at least one of them being correct. The overall success probability of having 2 units for the random in unit circle function is:

$$\begin{aligned}
P_{OverallSuccess} &= (P1_{fail} * P2_{success}) + (P1_{success} * P2_{fail}) + (P1_{success} * P2_{success}) \leftrightarrow \\
P_{OverallSuccess} &= (0.2146 * 0.7854) + (0.7854 * 0.2146) + (0.7854 * 0.7854) \leftrightarrow \\
P_{OverallSuccess} &= 0.1685 + 0.1685 + 0.6168 \leftrightarrow \\
P_{OverallSuccess} &= 0.9538
\end{aligned} \tag{6.1}$$

This is not enough since we are aiming for 99% coverage, so a third unit to perform the same operation is added, which yields a probability of success:

$$\begin{aligned}
P_{OverallSuccess} &= 1 - P_{OverallFail} \leftrightarrow \\
P_{OverallSuccess} &= 1 - (P1_{fail} * P2_{fail} * P3_{fail}) \leftrightarrow \\
P_{OverallSuccess} &= 1 - (0.2146 * 0.2146 * 0.2146) \leftrightarrow \\
P_{OverallSuccess} &= 1 - 0.0098 \leftrightarrow P_{OverallSuccess} = 0.9901
\end{aligned} \tag{6.2}$$

Having 3 units covers our attempt for a 99% coverage for the Random in Unit Circle function.
Similarlly, for the 3 dimensional counterpart and 5 units:

$$\begin{aligned}
P_{OverallSuccess} &= 1 - P_{OverallFail} \leftrightarrow \\
P_{OverallSuccess} &= 1 - (P1_{fail} * P2_{fail} * P3_{fail} * P4_{fail} * P5_{fail}) \leftrightarrow \\
P_{OverallSuccess} &= 1 - (0.4764 * 0.4764 * 0.4764 * 0.4764 * 0.4764) \leftrightarrow \\
P_{OverallSuccess} &= 1 - 0.0245 \leftrightarrow P_{OverallSuccess} = 0.9754
\end{aligned} \tag{6.3}$$

This still wasn't 99% but adding extra units wasn't yielding much difference compared to the extra hardware loss, so we decided to settle for 5 units(we would need 2 extra units to gain 0.0146 and reach the 99% probability goal).

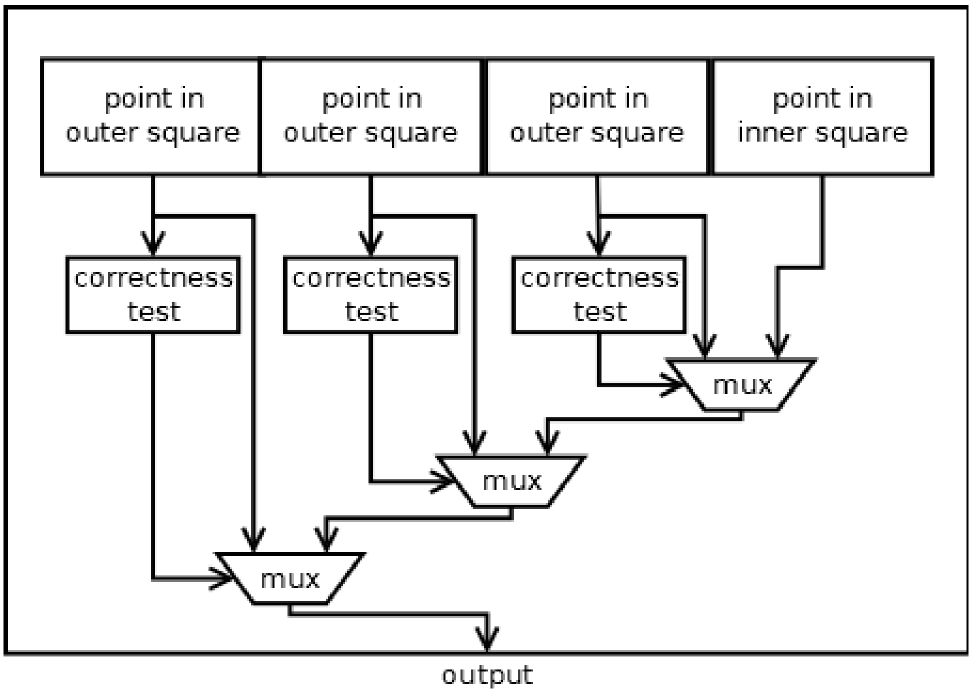


FIGURE 6.6: Random in Unit Circle hardware architecture

6.6.2 Streaming Dataflow/Pipeline technique

This is a common technique that is used to connect units and achieve pipeline. The algorithm is once again divided into its major components. A distinct and autonomous IP is created to perform the operations for each individual major component. Those IPs are connected together with FIFO streams, so every IP has an input stream and an output stream. When there is no data in the input stream or when the output stream is full, stalls are performed on each individual unit, untill all data is consumed. Every component of this chain is also pipelined by the Xilinx Vitis HLS tool. Two important values are indicated:

- Iteration Latency. Indicates the #cycles it takes for 1 piece of data travel from the input to the output stream for each IP.
- Interval. Indicates how many cycles it takes for data to move 1 step forward in the inner pipeline of each IP. In other words, how often(in term of cycles) a piece of data is written to the output stream.

For 1 piece of data, the overall latency of this technique is determined by adding together the Iteration Latency for all component. Its also important to note that since this is a pipe-lined process, the 2ed piece of data will not have the same latency but instead will have a latency equal the to interval. The unit with the worse Interval determines the overall interval of the chain. So overall the latency is given by:

$$T_{latency} = \sum T_{IterationLatency} + T_{WorseInterval} * TotalData \tag{6.4}$$

Now it becomes clear that when dealing with a large amount of data($TotalData$) Iteration Latency becomes somewhat irrelevant, and minimizing Interval should be the main focus.

6.6.3 Top level datapath

As discussed previously the algorithm was broken down to its 5 major units. Those units are connected together with the aforementioned FIFO streams. We managed to achieve an interval of 2 cycles for our worse case unit. That means when there is a large of data(rays in our case), a ray is processed approximately every other cycle.

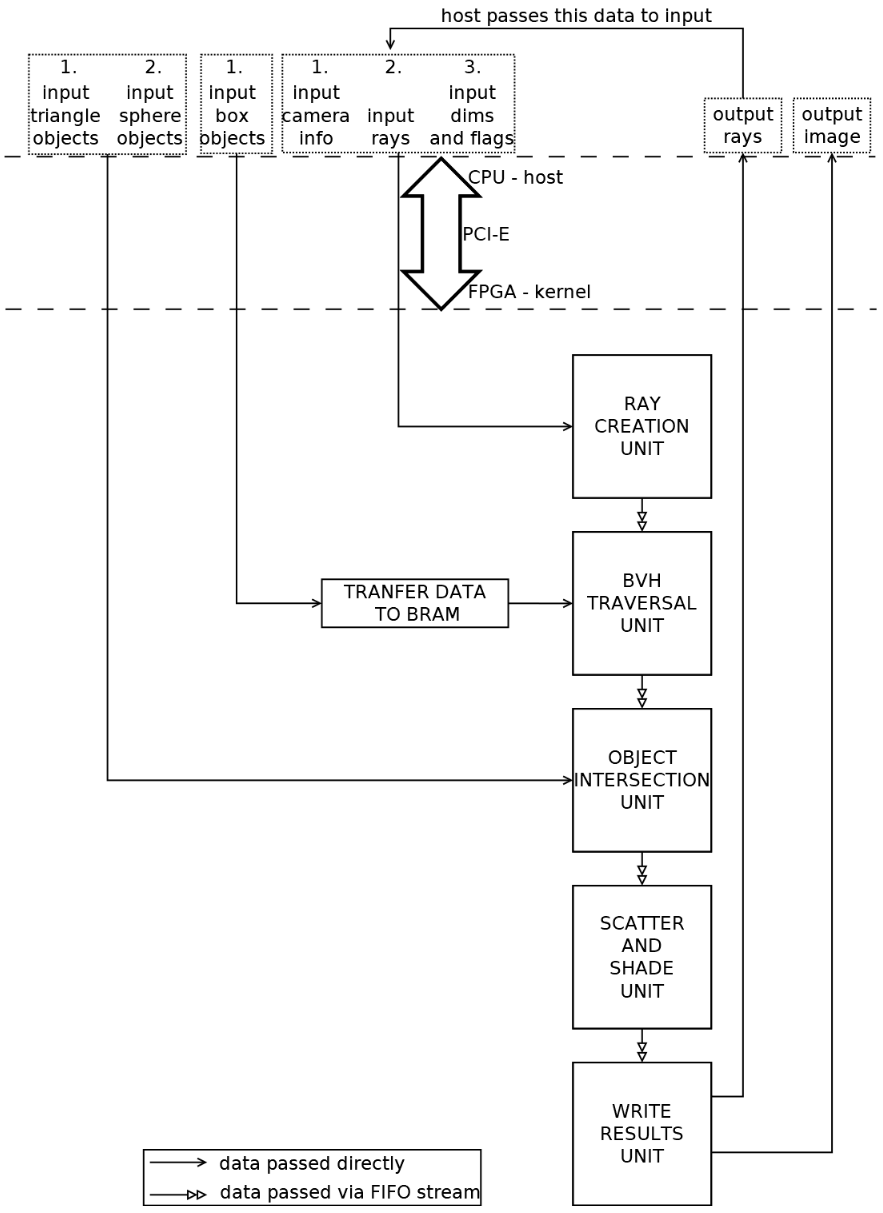


FIGURE 6.7: Top level architecture

6.6.4 Main Dataflow Components

Ray Creation Unit

This unit is responsible for creating new Rays, when rendering begins, or feed already processed and bounced rays to the pipeline. In the core of this unit lay some of our aforementioned random point creation units to create appropriately different rays.

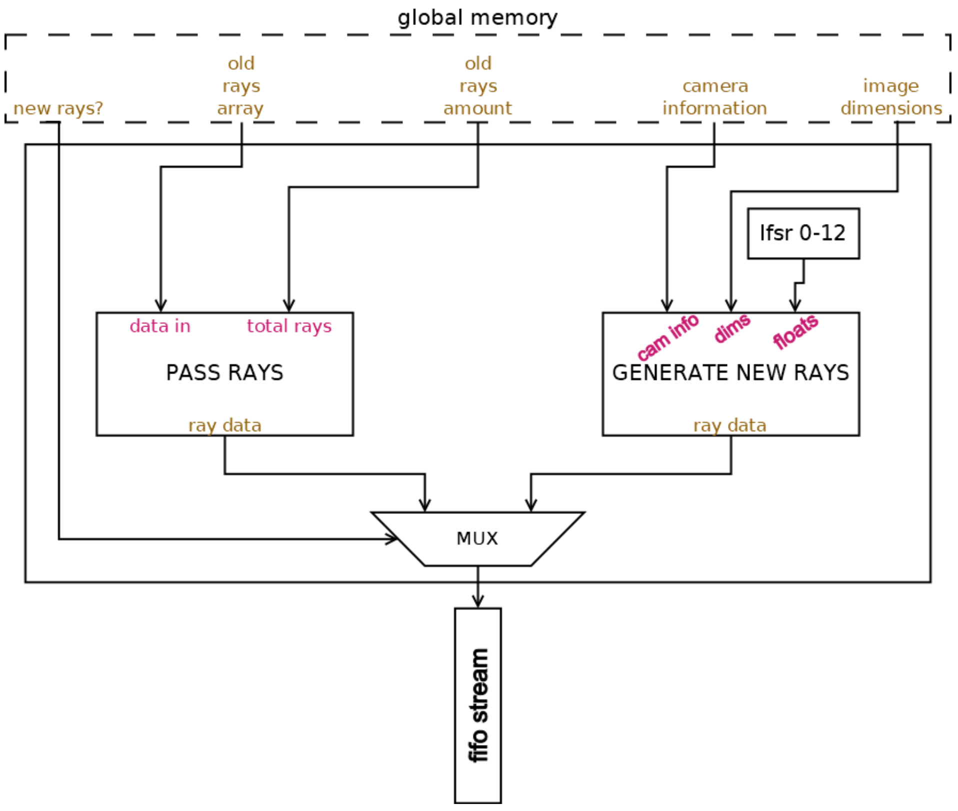


FIGURE 6.8: Ray Production Unit block diagram

The inner "Generate new Rays" component is responsible for creating new rays while the "Pass Rays" component just reads and writes bounced rays from global memory to the output stream. A top level value indicated as "new rays?" determines which of the two components will be used to feed data into the pipeline.

BVH traversal Unit - Converting Recursion to fit the FPGA

This Unit is responsible for traversing the BVH tree. This was a heavily recursive algorithm in software. In OpenCL we solved the recursion issue, but using the same technique was impossible since it could not provide the appropriate interval. Upon closer inspection of the software algorithm's operation, we observed that in each iteration of the recursive process an intersection test is made, but every time the tests involves one level of the tree. So for example on the first iteration only the test is made on some data from the first level of the tree, on the second iteration the test is made on some data from the second level and so on, until the bottom is reached and an object is determined. Our idea was to create a cascaded set of similar units that perform intersections but each of them has its unique set of BVH objects corresponding to the BVH tree level objects as shown in 6.9. These smaller

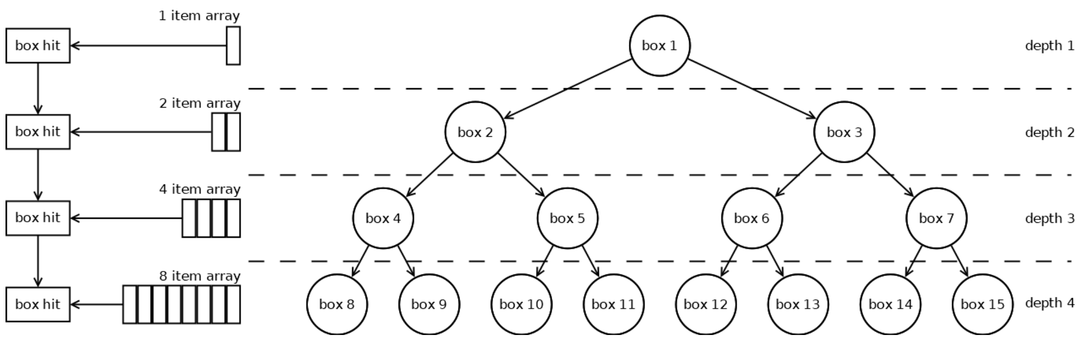


FIGURE 6.9: BVH conversion for a BVH with 16 objects

portions of the BVH are stored in different BRAMs and URAMs so they can all be accessed individually without creating any issues. There is an issue that needs to be noted with this design choice. This process is bound by the total BRAM and URAM of the FPGA since each level requires exponentially more object space.

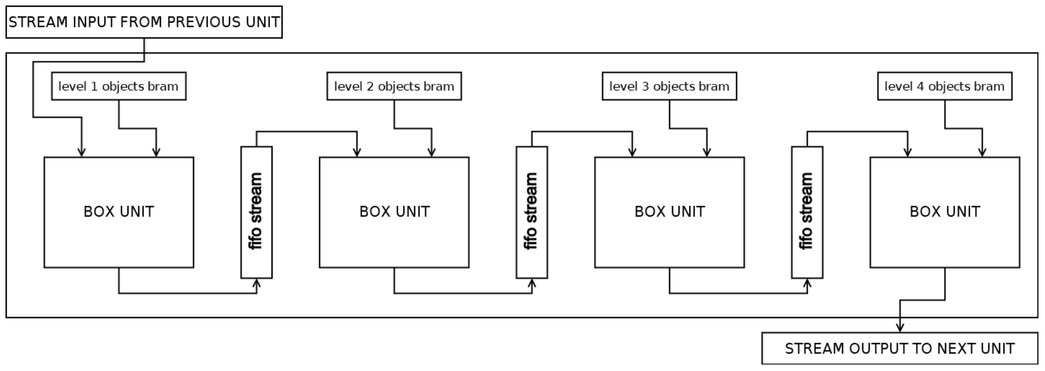


FIGURE 6.10: BVH traversal Unit with 4 box intersection units

As shown in 6.10 data flows from one box unit to the next performing intersection tests on the appropriate level of the tree, until the tree is fully traversed.

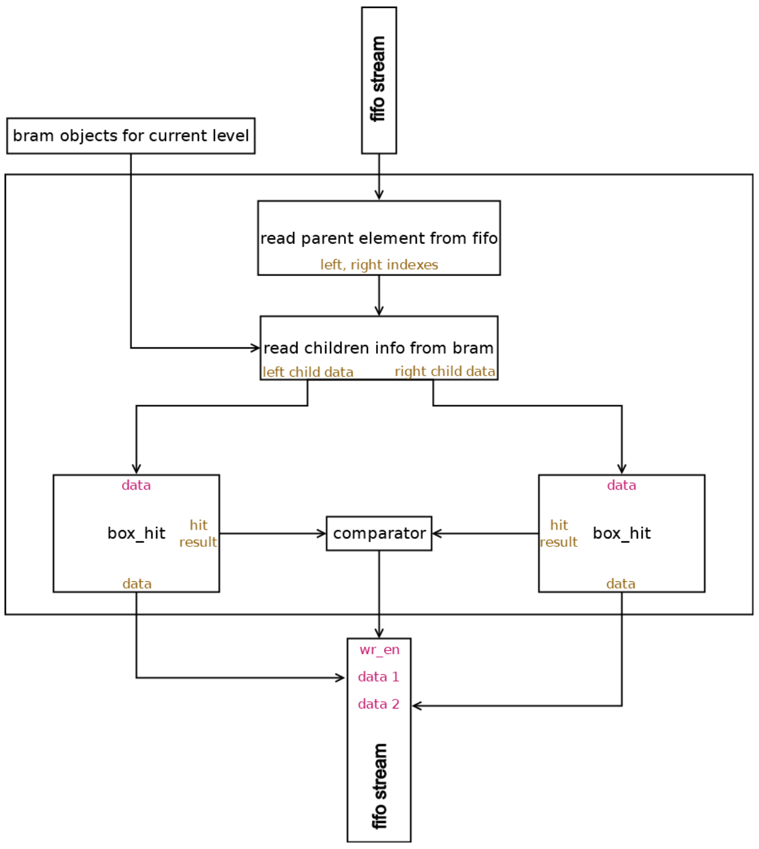


FIGURE 6.11: Box unit block diagram

Each of the box units actually performs two intersection tests(6.11). One for the left and one for the right child of the input node. If one of those is correct, it is promoted to the output of the box unit and to the input of the next box unit. It is possible for both of those tests to be correct, in which case both need to be promoted and this is were the limit to a pipeline interval of 2 originates, since FIFO streams only have 1 write port and thus, it would take

2 cycles to write both data. This can also lead to a single ray intersecting with multiple objects, in which case the closest of those object should be selected. This problem will be dealt with later.

In order to allow our model to accept more scenes with different amounts of triangles, we went for a design that allows a max BVH depth of 16(including 0, the root), corresponding to 17 cascaded box units. This design allows up to $2^{depth} = 2^{16} = 65536$ polygons. This was the maximum amount we managed to fit based on the bram and uram constraints that this design has and the available BRAM and URAM of our FPGA. If it is detected that the traversal has completed early(in case of a smaller scene with a smaller BVH that requires less stages), data is simply passed from one box unit to the next until it finally goes to the next major unit. This design choice introduces some extra delay for a single piece data to be calculated in case of smaller scenes, since data unnecessarily needs to flow through all the units, but also allows for a much larger variety of different scenes to be rendered without the need for different build architectures which cost a large amount of time to complete.

Object Intersection Unit

The object intersection unit is the unit responsible for actual polygon intersection. In the core of this unit lay the primary polygon intersection algorithms. After a ray finishes traversing the BVH tree and a candidate object is found, this unit checks if the ray did in fact intersect with that object.

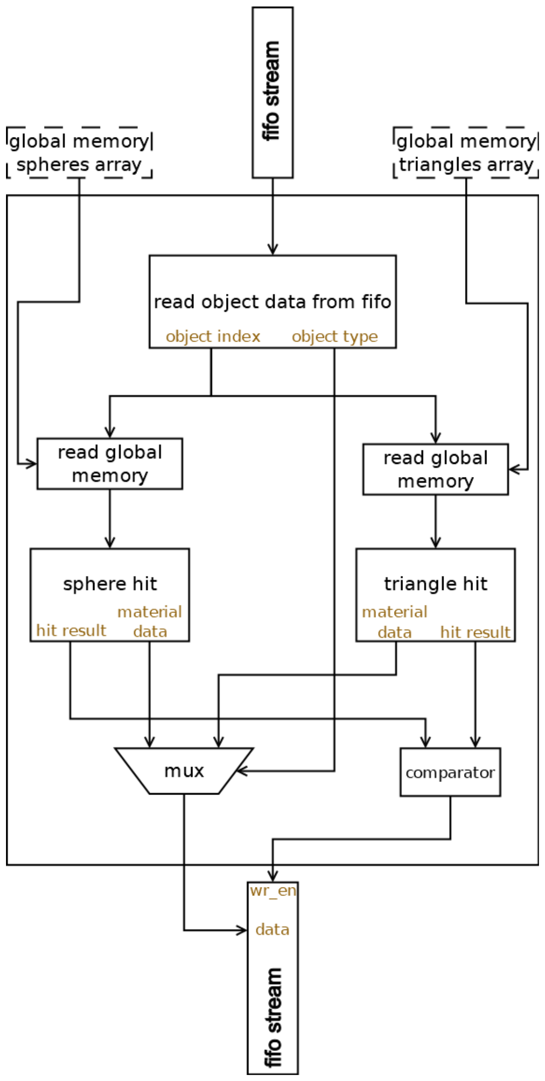


FIGURE 6.12: Object intersection Unit block diagram

If it is detected that the ray intersected with the object, color and material data of the object is gathered for the corresponding pixel to be shaded.

Scattering Unit

Scattering Unit is responsible for shading the corresponding pixel, as well as creating a new bounced ray originating at the intersection point. The new ray also corresponds to the same pixel. At the core of this unit, once again, lay some more LFSR random point generator units and a shading unit for each of the three acceptable materials. We noticed a small optimization here. Since only one of those three shading units is used on each iteration, it was unproductive to generate random points for each of them. Instead we only generate 1 random point which is then used as an input to the units. Even though not a major component optimization, it does save a few LFSR point generation units.

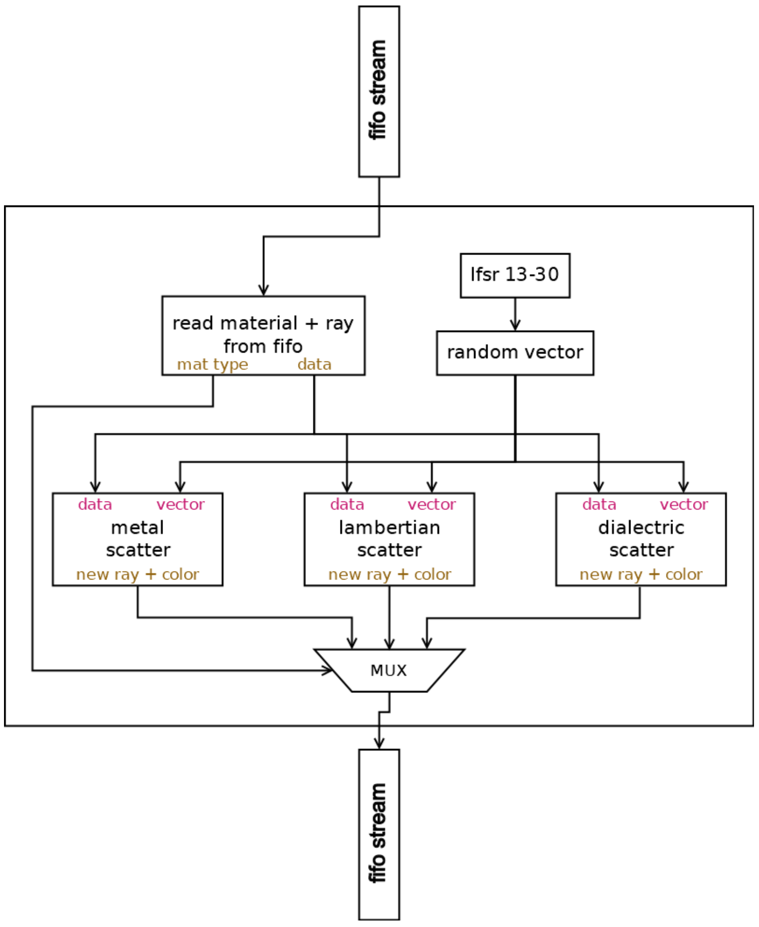


FIGURE 6.13: Scatter and Shade unit block diagram

The output of this unit includes color for the pixel and information about the new bounced ray.

Writeback Unit

The final unit of the dataflow is the writeback unit. This unit is of course responsible for writing data back to the interface or global memory, but also solve the final problem, of duplicate rays intersecting different objects. In this case, the closest object should be selected. The original idea was to keep distance data in a global array and compare the distance before overwriting a ray to the closest one. That would require global memory reads on each iteration so we decided to optimize this as we could.
We confirmed that duplicate rays can in fact show up at this stage corresponding to the same pixel with different object and distance data, but we noticed they always show up in order. We used 2 registers to compare 2 results on 2 inputs. For explanation purposes we'll call them the "current" register and the "previous" register, referring to the data each is holding. A piece of data travels from the current register to the previous register on each iteration. So the current register always has a new piece of data from the FIFO stream while the previous register has an older "state" of the current register.
The pixel_ids of the data in those registers are compared and if they are different to one another, then the color data inside the previous register is safe to be written to the interface since its pixel_id will not show up again as they come in order.
If the pixel_ids are the same then the closest needs to be picked. The distance of the two is compared and the closest one is written to the previous register instead of the interface, in case another piece of data will come with the same pixel_id which will also need to be compared. The process repeats for all rays until the image is formed.

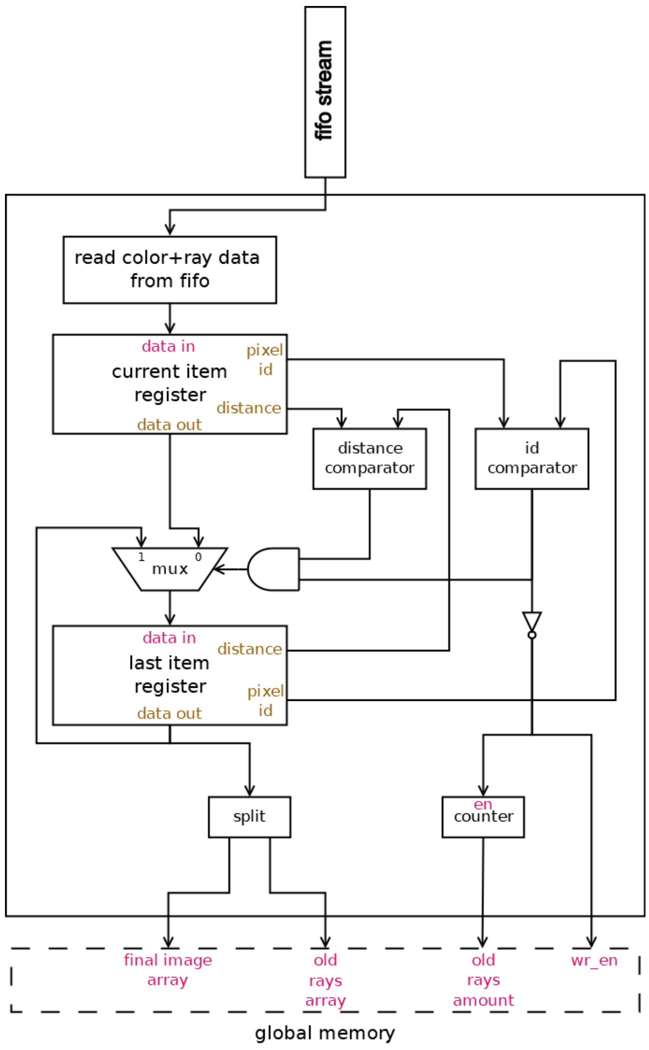


FIGURE 6.14: Write Unit block diagram

Chapter 7

Results

7.1 Hardware used

For testing in software a personal computer was used. The processor used was a AMD Ryzen 5-3600 6 core/12 thread chip, with a frequency of 3.6GHz and a single core boost of up to 4.2GHz. The computer also has 16GB of 3000MHz DDR4 memory.
The GPU that we used for the GPU testing was a AMD RX 5600XT with 6GB of GDDR6 VRAM. This GPU has a base clock of 1420MHz with a boost of up to 1750 MHz. The core consists of 36 compute units and 2304 stream processors.
Finally as we already mentioned, the FPGA we used was the Alveo U50 Data Center Accelerator Card. This card consists of 872k LUTs, 1743K registers and 5952 DSP slices. It also has 8GB of RAM, 28MB of SRAM and 2MB of BRAM.

Power Consumption

Unfortunately we were unable to monitor the exact power consumption during testing. Nevertheless bellow are presented the TDPs(Thermal Design Power) of our hardware.

Ryzen 5 3600	AMD RX 5600 XT	Alveo U50
65-80W	150-170W	75W

7.2 FPGA utilization and timings

The timing outcome that Vitis HLS tool generated was:

target	estimated	uncertainty
3.33ns	2.433ns	0.9ns

For the Alveo U50 FPGA card the following utilization estimates were generated:

BRAM	DSP	FF	LUT	URAM
899(33%)	1183(19%)	905243(51%)	358422(41%)	120(18%)

Finally, our processing units displayed the following timing results in terms of cycles.

	iteration latency	interval
ray production unit	61	1
box intersection unit	61	2
object intersection unit	300	2
scatter and shade unit	183	2
writeback unit	73	2

There are 17 instances of the box intersection unit so the overall latency of a ray to go through the pipeline is:

$$T_{latency} = \sum T_{UnitLatency} = 61 + 17 * 61 + 300 + 183 + 73 = 1654 cycles \tag{7.1}$$

As we already discussed the process is pipelined and since the worse interval case in our pipeline is 2 cycles, then we can assume that a ray is processed every other cycle.

7.3 The scenes

To evaluate the performance of our work across different platforms 4 different scenes were created. All our scenes were rendered in 1920x1080 image resolution with a threshold of 50 ray bounces for each ray.

Cornell Box

This is a fairly small scene. The Cornell box is a very famous benchmark to evaluate a ray tracer's performance. Since our ray tracer doesn't include light sources and our scenes are illuminated as if it was a sunny day, we had to remove the ceiling of the Cornell box instead of adding a light source on it.

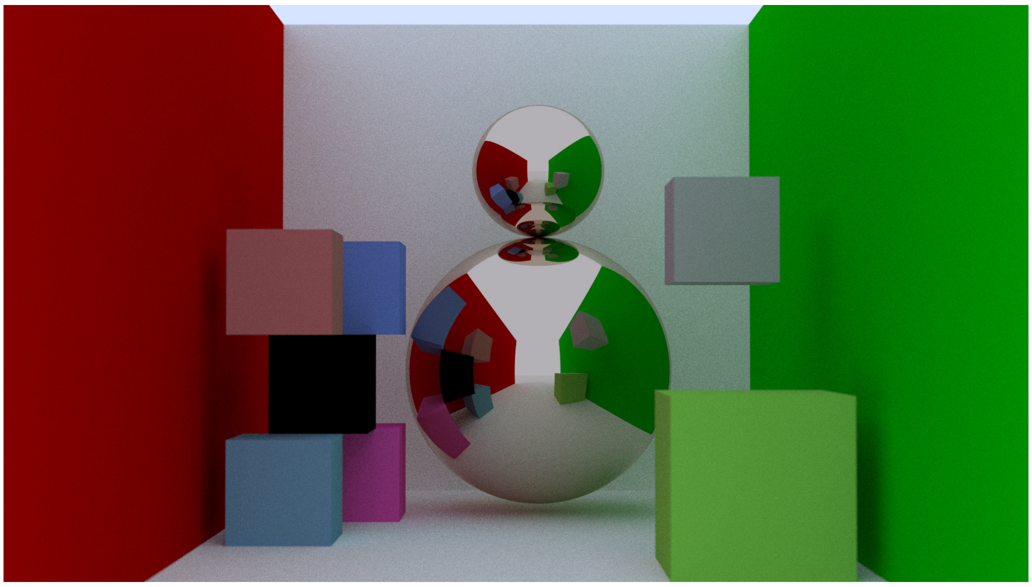


FIGURE 7.1: Cornell Box Scene

Spheres	Triangles
3	92

Double Teapot

In this scene we find 2 instances of another very famous object, the Utah teapot. A few spheres are added around and a mirror in the back.

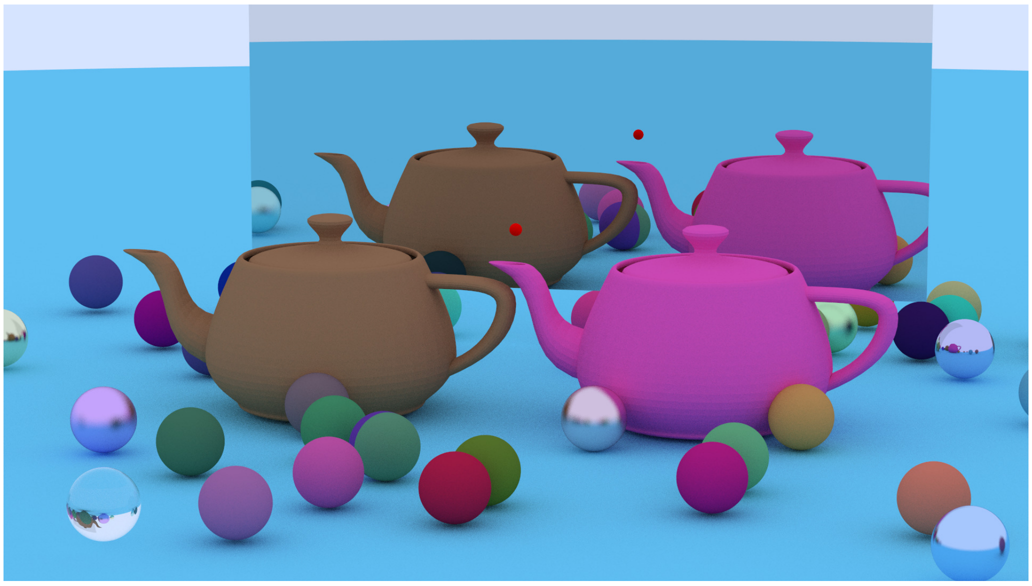


FIGURE 7.2: Double Teapot Scene

Spheres	Triangles
50	12642

Dragon Scene

The dragon scene consists of a dragon model.

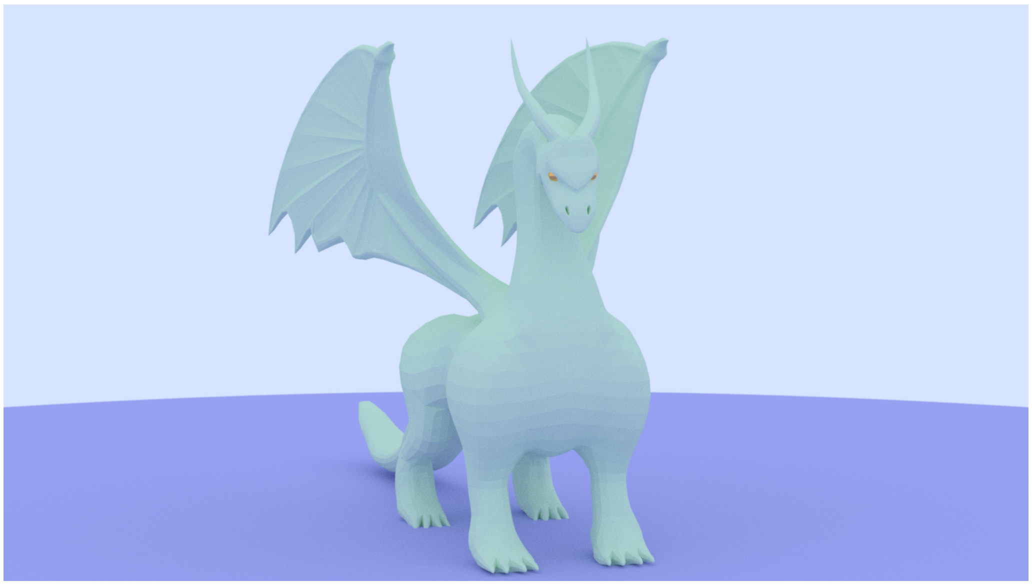


FIGURE 7.3: Dragon Scene

Spheres	Triangles
3	20384

Toy Plane Scene

This scene consists of a plane model. The original materials from the model are also present in this scene.

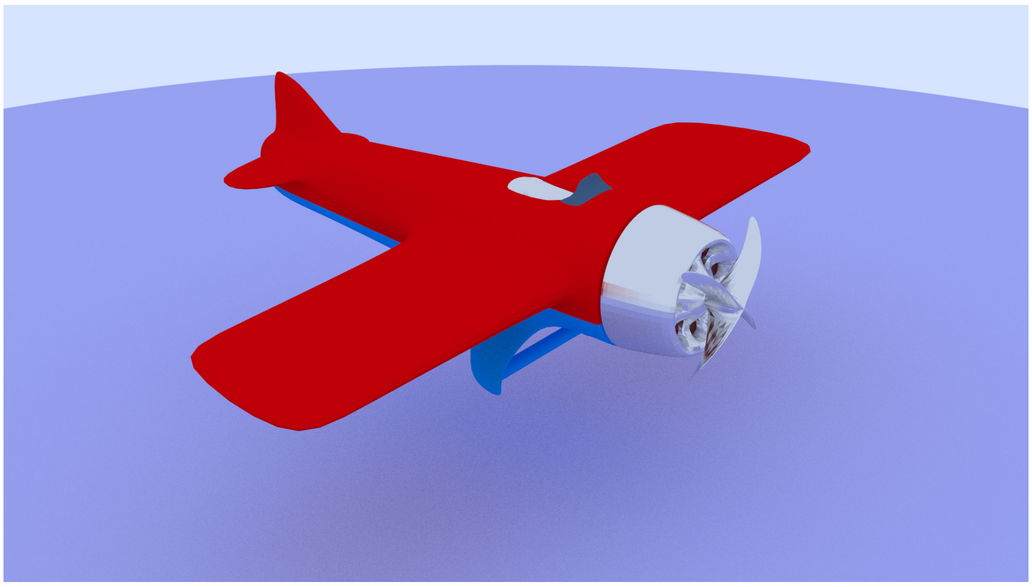


FIGURE 7.4: Toy Plane Scene

Spheres	Triangles
1	21936

Cars Scene

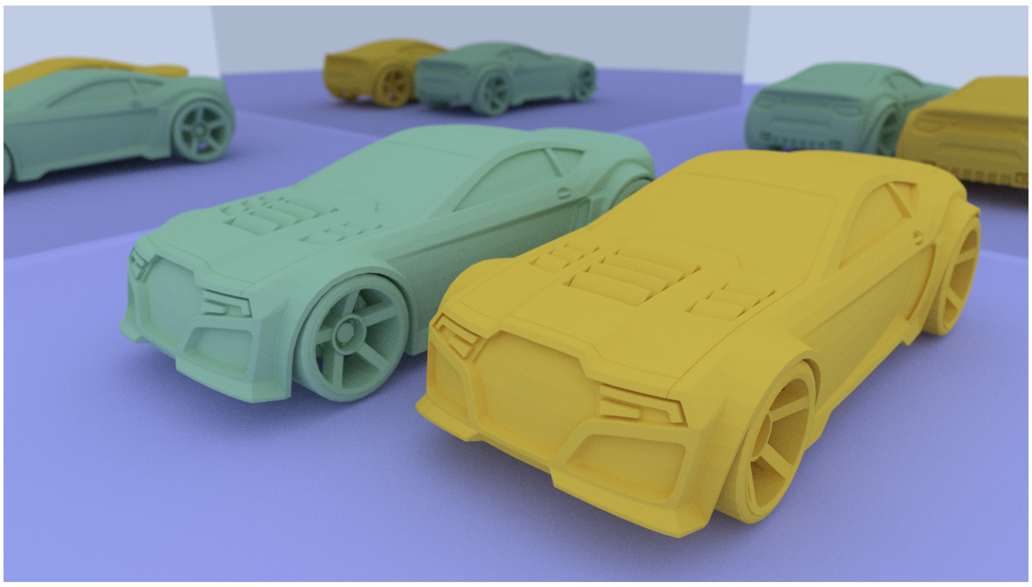


FIGURE 7.5: Cars Scene

This scene consists of 2 car models. There are mirrors placed in the back at a 90°angle so that more rays will bounce around the scene. They also create the effect of quadrupling the objects placed in front of the mirrors.

Spheres	Triangles
1	45120

Friends Scene

This scene consists of two famous characters from computer games, a fall guy from the game "Fall Guys" and Cuphead from the game "Cuphead". Once again, there are mirrors placed on the ceiling, on the floor and the front and back wall. As it is expected, this scene is darker since the ceiling is blocked off and objects collect light energy only from the rays that escape from the sides. We also can see the "infinity" effect of putting 2 mirrors against each other.

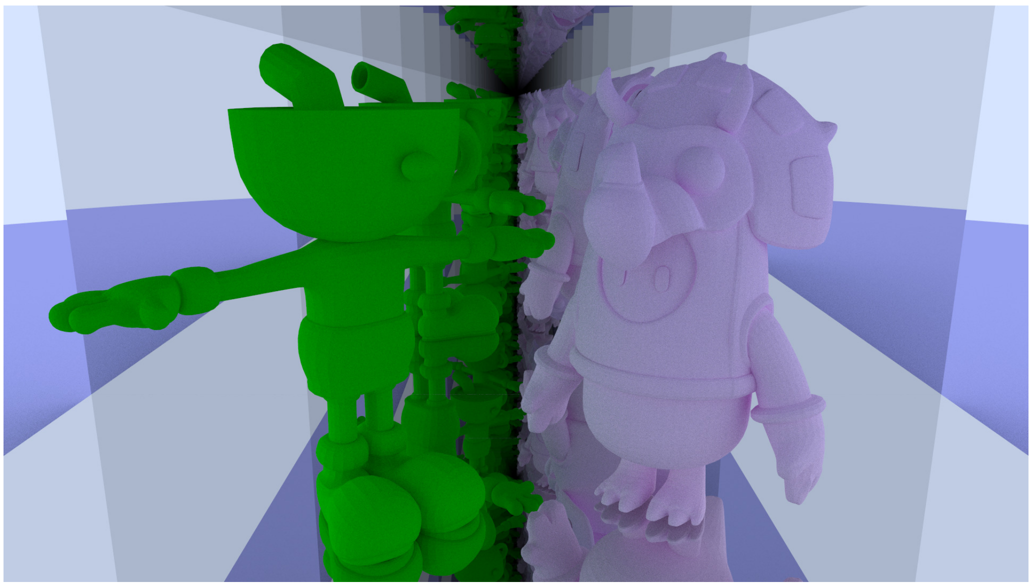


FIGURE 7.6: Friends Scene

Spheres	Triangles
1	56654

7.4 Performance Results

We tested the above scenes under 2 different sampling settings, 10 sample per pixel and 100 sample per pixel. For the CPU implementation 3 different settings were used single thread, 6 thread and 12 thread. We present, below, in tabular form and in histograms the overall timing and total rays cast across different platforms.

Total Rays (in millions)						
	Cornell box	Dragon	Plane	Double Teapot	Cars	Friends
10 samples	107	34	45	58.3	84.5	110
100 samples	1,100	341	450.5	582.8	846	1,100

TABLE 7.1: Total rays cast for each scene

1920x1080 | 10 samples (in sec)					
	CPU 1T	CPU 6T	CPU 12T	GPU	U50
Cornell box	152.166	123.553	132.528	79.215	43.182
Dragon	237.897	48.031	52.155	30.012	47.765
Plane	203.548	54.923	52.261	33.292	41.799
Double Teapot	207.123	72.993	67.043	45.176	35.926
Cars	1285.14	301.354	197.316	124.96	177.874
Friends	549.013	134.206	121.185	74.998	71.703

TABLE 7.2: Timing results for |1920x1080p-10 sample| scenes

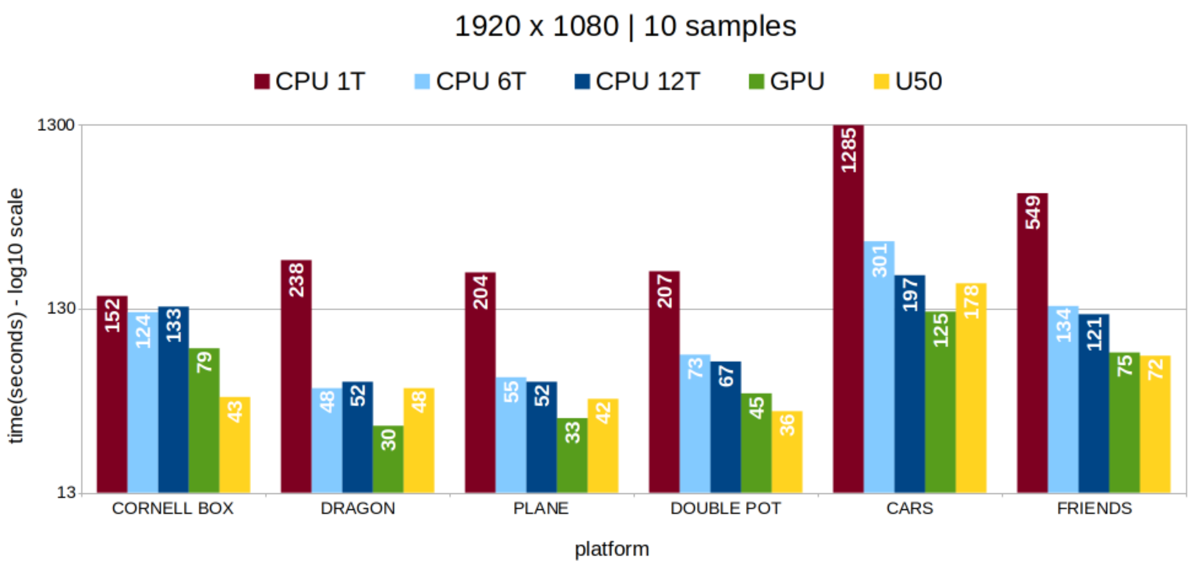


FIGURE 7.7: Execution times across platforms for 10 samples

1920x1080 | 100 samples (in sec)					
	CPU 1T	CPU 6T	CPU 12T	GPU	U50
Cornell box	1537.764	1265.86	1333.28	774.152	423.909
Dragon	2413.95	496.311	475.433	293.326	466.781
Plane	2010.38	550.62	511.492	325.712	404.335
Double Teapot	2046.18	734.68	671.015	450.096	355.254
Cars	12748.58	3163.08	1943.89	1245.12	1795.89
Friends	5566.99	1304.87	1210.36	745.123	712.532

TABLE 7.3: Timing results for | 1920x1080p-100 sample | scenes

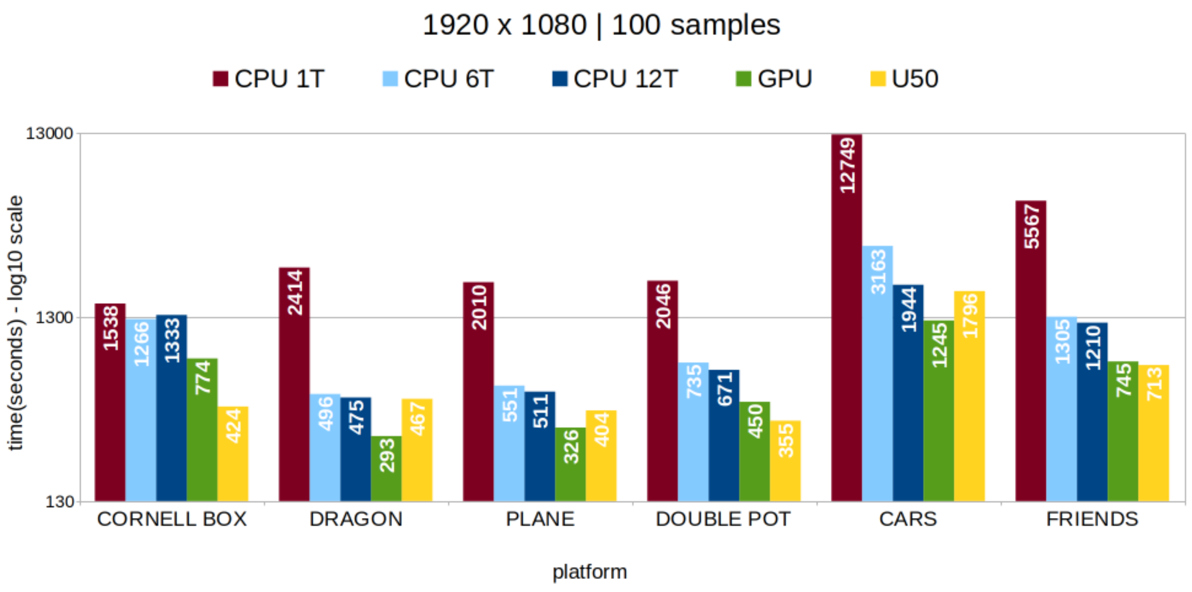


FIGURE 7.8: Execution times across platforms for 100 samples

Performance discussion

As we can see, our FPGA implementation performs fairly well. As we can see our FPGA implementation is able to beat the CPU equivalent on all of our datasets. On the bigger scenes, it doesn't manage to beat the GPU implementation across the board, but they perform tightly close. We also need to consider the power consumption of these 2 platforms. Even though we couldn't measure it, based on our knowledge, we expect the FPGA to be operating at around half the power of the GPU. We believe the major reason that the FPGA loses performance is because of the way that the feedback loop is performed. Instead of data being routed back internally, the data is collected at the end of the pipeline and sent back to the host as a big chunk. The host then copies this chunk of data from the output buffer to the input buffer and feeds back into the kernel. This creates some sort of synchronization that isn't needed and can be avoided. Also BRAM and URAM data are copied over and over when 1 time should be enough. This process is repeated many times until the final image is created. As future work, rerouting this feedback loop to be done internally should notably increase performance.

An odd result in our testing was around the Cornell Box scene. This scene is the smallest of the bunch and we can see that it doesn't scale as expected on the CPU. We measured a high amount of cache misses when deploying more threads on this scene. This led to a higher amount of instructions being issued and eventually caused a hazard for the threads. We believe this has to do with the small size of the BVH. Since the BVH is small and there aren't many nodes in it, we believe the threads were attempting to read similar memory elements resulting in the problem. Further testing can confirm our assumption, but the process in this scene is identical to the others. Nonetheless, in this scene we see the greatest performance difference of the FPGA compared to the other platforms.

In the Cars scene we also observe a mediocre speedup from the FPGA compared to the CPU. We believe this has to do with the amount of extra rays which are fed back into the feedback loop. Resolving that issue should relieve the situation.

At last we can see that across all platforms performance scales with the amount of samples linearly, which means that 10x samples equal 10x time. This is also the case for the total rays created.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Ray Tracing is becoming a very "hot" topic when it comes to computer graphics. The ray tracing algorithm is a very computationally exhaustive algorithm that was mainly used by 3D film making companies. In the recent years the hardware caught up to the demand and more people are able to have access to this method of rendering. Major companies have already started pursuing the creation of ray tracing specific platforms that aim to accelerate the process.

In its core the ray tracing algorithm is heavily parallelizable so the FPGA can be a great candidate platform to gain advantage from this. The algorithm's heaviest job revolves around identifying which object does a ray intersect with. When it comes to very complex scenes, checking every polygon on the scene to find the correct one becomes obsolete. Using acceleration structures is essential to ensure the smooth operation of the algorithm. These structures create a hierarchy tree of recursively smaller and smaller groups of polygons in the scene based on their relative position which can be refereed to, to find the correct objects with much less effort. All rays

To make this algorithm run on other platforms drastic changes need to be made. Our algorithm was mainly recursive and written in an inheritance manner. These needed to be removed or changed on platforms that don't support them. We created a setup that uses repetitive logic and removed inheritance to make the algorithm work on GPUs using the OpenCL platform. The main issue was the traversal of the BVH acceleration structures which we overcame by using repetitive logic and a small cache array.

More changes needed to be issued to make the algorithm work on FPGAs. Once again, recursion is not an option. We opted to use a pipeline streaming setup. Using the same logic as before was not good enough because it was not able to be pipelined. We broke the BVH traversal down into a cascaded setup where every level of the tree has a unique hardware unit and BRAM associated with it and is connected to the previous and next unit with FIFO streams. Data moves from the root to the leaves of the tree while being processed on the way. Data for the tree nodes are placed into BRAM and URAM for faster memory accesses but that limits the possible maximum size of the BVH. This way we were able to achieve a setup were a ray can be processed every 2 cycles.
Our architecture has a big drawback regarding the feedback loop which creates a type of unnecessary synchronization and more I/O between the host and kernel platforms. Rerouting the feedback loop is the next step to achieve greater performance with this architecture.
Another issue we needed to solve was random random generation which was done by using LFSR registers.

Even though close, FPGA results weren't great compared to the GPU. We did observe some speedup, especially on smaller scenes, but on bigger scenes the result was underwhelming. That said, if we factor in power consumption the results are a lot more promising. Rerouting the feedback loop is expected to solve a lot of issues and provide considerable performance.

8.2 Future Work

In order to further improve the quality and performance of our introduced ray tracer work can be done in many different areas.
First of all work can be done to add more features to the ray tracer:

- Shadow Rays or Importance Sampling can be applied for better shadow quality.
- Light sources and directional lighting support.
- More materials and texture support can be added (wood etc.)
- Switch the BVH tree for a K-D tree to avoid double intersections and reduce computations.
- Better bounding-box creation method for triangle polygons.

- When creating the BVH or K-D tree, factor in camera position for better tree quality(as proposed in [8]).
- Denoise filters could also be another way to reduce the amount of samples Monte Carlo integration requires to wield visually acceptable results.

Regarding the FPGA architecture work can also be done to improve the performance:

- Re-route the feedback loop so that it doesn't go back to the host, but is done internally.
- Better memory management. "Enum" types can be used instead of integers where possible to save memory and maybe fit in a few extra stages.
- Multi-kernel process. Since the kernel itself is leaving some FPGA area unused a multi kernel setup can be explored, but with consideration to BRAM and URAM(maybe with multi-port BRAM/URAM).
- Using a K-D tree would make some logic redundant, which can be redesigned for better space and resource management.

External Links

[1] “Rasterization video”. In: (). URL: https://www.youtube.com/watch?v=brDJVEPOeY8.

[2] “Ray Tracing slide show”. In: (). URL: https://slideplayer.com/slide/10086872/.

[3] “Scrachapixel book”. In: (). URL: https://www.scratchapixel.com/.

[18] “Barycentric coordinate system”. In: (). URL: https://en.wikipedia.org/wiki/Barycentric_coordinate_system.

[19] “Cramer’s Rule”. In: (). URL: https://en.wikipedia.org/wiki/Cramer%27s_rule.

[20] “Scalar Triple Product”. In: (). URL: https://en.wikipedia.org/wiki/Triple_product.

[22] “Github Project that reads triangles from file”. In: (). URL: https://github.com/bicknyers/raytracer-cpp.

[23] “Khronos 2011 openCL Overview”. In: (). URL: https://www.khronos.org/assets/uploads/developers/library/2011_GDC_OpenCL/AMD-OpenCL-Overview_GDC-Mar11.pdf.

Bibliography

[1] "Rasterization video". In: (). URL: https://www.youtube.com/watch?v=brDJVEPOeY8.

[2] "Ray Tracing slide show". In: (). URL: https://slideplayer.com/slide/10086872/.

[3] "Scrachapixel book". In: (). URL: https://www.scratchapixel.com/.

[4] Jefferson Amstutz et al. "An Evaluation of Multi-Hit Ray Traversal in a BVH using Existing First-Hit/Any-Hit Kernels". In: *Journal of Computer Graphics Techniques (JCGT)* 6 (2015).

[5] Peter Shirley. *Ray Tracing in One Weekend*. Dec. 2020. URL: https://raytracing.github.io/books/RayTracingInOneWeekend.html.

[6] Jean-Colas Prunier. *Scratchpixel book*. URL: https://www.scratchapixel.com/.

[7] Greg Humphreys Matt Pharr Wenzel Jakob. *PBR book*. URL: https://www.pbr-book.org/.

[8] Warren Hunt and William R. Mark. "Ray-specialized acceleration structures for ray tracing". In: *2008 IEEE Symposium on Interactive Ray Tracing*. 2008, pp. 3–10. DOI: 10.1109/RT.2008.4634613.

[9] Per H. Christensen et al. "Ray Tracing for the Movie 'Cars'". In: *2006 IEEE Symposium on Interactive Ray Tracing*. 2006, pp. 1–6. DOI: 10.1109/RT.2006.280208.

[10] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. "Saarcor: a hardware architecture for ray tracing". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 2002, pp. 27–36.

[11] Jörg Schmittler et al. "Realtime ray tracing of dynamic scenes on an FPGA chip". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 2004, pp. 95–106.

[12] Jae-Ho Nah et al. "RayCore: A ray-tracing hardware architecture for mobile devices". In: *ACM Transactions on Graphics (TOG)* 33.5 (2014), pp. 1–15.

[13] Jae-Ho Nah et al. "T&I engine: Traversal and intersection engine for hardware accelerated ray tracing". In: *Proceedings of the 2011 SIGGRAPH Asia Conference*. 2011, pp. 1–10.

[14] Felix Winterstein, Samuel Bayliss, and George A. Constantinides. "FPGA-based K-means clustering using tree-based data structures". In: *2013 23rd International Conference on Field programmable Logic and Applications*. 2013, pp. 1–6. DOI: `10.1109/FPL.2013.6645501`.

[15] Yulin Chen et al. "Ray Tracing on Single FPGA". In: *2021 IEEE Asia-Pacific Conference on Image Processing, Electronics and Computers (IPEC)*. 2021, pp. 1290–1294. DOI: `10.1109/IPEC51340.2021.9421209`.

[16] Tomas Möller and Ben Trumbore. "Fast, minimum storage ray-triangle intersection". In: *Journal of graphics tools* 2.1 (1997), pp. 21–28.

[17] Wikipedia. *Möller–Trumbore intersection algorithm — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=M%C3%B6ller%E2%80%93Trumbore%20intersection%20algorithm&oldid=1007449813`. [Online; accessed 19-February-2022]. 2022.

[18] "Barycentric coordinate system". In: (). URL: `https://en.wikipedia.org/wiki/Barycentric_coordinate_system`.

[19] "Cramer's Rule". In: (). URL: `https://en.wikipedia.org/wiki/Cramer%27s_rule`.

[20] "Scalar Triple Product". In: (). URL: `https://en.wikipedia.org/wiki/Triple_product`.

[21] Timothy L Kay and James T Kajiya. "Ray tracing complex scenes". In: *ACM SIGGRAPH computer graphics* 20.4 (1986), pp. 269–278.

[22] "Github Project that reads triangles from file". In: (). URL: `https://github.com/bicknyers/raytracer-cpp`.

[23] "Khronos 2011 openCL Overview". In: (). URL: `https://www.khronos.org/assets/uploads/developers/library/2011_GDC_OpenCL/AMD-OpenCL-Overview_GDC-Mar11.pdf`.

[24] Jonathan Tompson and Kristofer Schlachter. "An introduction to the opencl programming model". In: *Person Education* 49 (2012), p. 31.

[25] Maria George and Peter Alfke. "Linear feedback shift registers in virtex devices". In: *Xilinx apprication note XAPP210* (2007).