UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Hardware Acceleration of Ray Tracing Algorithms using FPGAs

Diploma Thesis

Nikolaos Koxenoglou

Supervisor: Nikolaos Bellas

Volos 2021

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Hardware Acceleration of Ray Tracing Algorithms using FPGAs

Diploma Thesis

Nikolaos Koxenoglou

Supervisor: Nikolaos Bellas

Volos 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Επιτάχυνση Αλγορίθμων Ray Tracing με FPGAs

Διπλωματική Εργασία

Νικόλαος Κοξένογλου

Επιβλέπων: Νικόλαος Μπέλλας

Βόλος 2021

Approved by the Examination Committee:

Supervisor **Nikolaos Bellas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Dimitrios Katsaros**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Date of approval: 20-2-2021

Acknowledgements

I would like to take this opportunity to express my gratitude to my supervisor Prof. Nikolaos Bellas for his unwavering support and encouragement throughout my academic career. I would also like to thank PhD candidate Maria-Rafaela Gkeka for her help and support in the technical aspect of my diploma thesis.

Lastly this whole journey would not at all be possible if not for my family and friends support and I cannot thank them enough. Everyone around me showed extreme patience and understanding while I undertook this great journey of self-discovery as well as building and developing.

Especially I would like to thank my father Giorgos Koxenoglou and mother Antigoni Sofokleous for all the things in the world that they selflessly given to me while I was developing to who I am today. My sisters, Myrto and Eleni Koxenoglou, were always there supporting me and I can't thank them enough. Last but certainly not least I would like to thank my dear friend Stavros Simoglou for being there all the time helping me with all aspects of life in engineering.

Though I am ready to take on challenges previously unobtainable by my past self I am sure there is still more to be done in order to make me a better problem solver, engineer and most importantly a more complete person with a greater understanding of the world around me.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Nikolaos Koxenoglou
15-2-2021

Abstract

In computer graphics world render APIs, mainly OpenGL and DirectX for gaming, use rasterization to draw objects on screen. Rasterization is the process that takes a scene description usually in vector graphics and solves what we refer to as the visibility problem. When there is a scene to be rendered, we want to project that scene to a display or a print onto paper, we need to know what is visible from the point of view of the observer. A scene is usually comprised of 3D polygon objects which are themselves comprised of 3-pointed vertices known as triangles. Rasterization achieves this by projecting each 3D objects coordinates to 2D plane of the screen where real time rendering is needed, but it only accounts for the visibility problem [1]. Shading is the general process by which an object gets its colour and texture is not a part of it, it must be computed and or approximated afterwards reducing the realism of the scene.

This is where Ray Tracing comes in, it is focused around the image on contrast to rasterization which is focused on the objects in the scene [2]. To solve the visibility problem a ray is cast from each pixel and then each object is checked to see if that ray intersects said object or not. Afterwards if a successful intersection is made more rays spawn from that location to evaluate what colour that pixel will be. That way the shading process is integrated in the rendering process more neatly rather than being an afterthought.

There are a lot of Ray Tracing implementations but almost all of them involve high performance FPGAs and processors, the question is can we bring the same results to the table with a low power FPGA and processor combo and even achieve real time rendering? In short, the experiment yielded interesting results about what can and can't be done using such confided hardware limits as well as utilising only quick-to-market solutions like High Level Synthesis. Performance was not achieved with the ZedBoard ARM processor being quicker than our peripheral the conclusions we draw illuminate a path forward on how to approach ray tracing on hardware.

Περίληψη

Στον κόσμο των γραφικών στον υπολογιστή για τη σχεδίαση στην οθόνη χρησιμοποιούν στην πλειονότητά τους προγραμματιστικές διεπαφές που βασίζονται πάνω στο πλέγμα σάρωσης (rasterization). Αυτός ο αλγόριθμος είναι στην ουσία μια διαδικασία εύρεσης για το τι βλέπει ο παρατηρητής της εικόνας από μια τρισδιάστατη σκηνή που πρόβλημα ορατότητας. Η διαδικασία ξεκάνει με τη περιγραφή των αντικειμένων σε πολύγονα από τριάδες διανυσμάτων δηλαδή τρίγωνα. Έπειτα τα αντικείμενα προβάλλονται στην οθόνη από τον τρισδιάστατο χώρο στον δισδιάστατο. Αυτή είναι μια γρήγορή διαδικασία για αυτό και χρησιμοποιείτε σήμερα , κυρίος σε εφαρμογές ψυχαγωγίας όπου η ανάγκη για γραφικά πραγματικού χρόνου είναι μεγαλύτερη. Η διαδικασία αυτή έχει όμως ένα μεγάλο μειονέκτημα, η απόφαση για το τι χρώμα και ύφη πρέπει να έχει το αντικείμενο στο κάθε εικονοστοιχείο γίνετε προσεγκίστικά. Εδώ είναι που έρχεται ο αλγόριθμος ακολουθίας ακτίνας που είναι πιο συγκεντρωμένος γύρο από την εικόνα πάρα τα αντικείμενα μέσα στη σκηνή. Για να λύσει το πρόβλημα της ορατότητας εκπέμπει και ακολουθεί τη διαδρομή μιας ακτίνας από το σημείο όρασης του παρατηρητή μέσα από το κάθε εικονοστοιχείο μέχρι να συναντήσει ένα αντικείμενο. Όταν η ακτίνα έρθει σε επαφή με το αντικείμενο γεννιούνται περεταίρω ακτίνες για εντοπισμό όλων των πηγών που μπορεί να επηρεάσουν το χρώμα και υφή του αντικειμένου. Έτσι η διαδικασία εύρεσης χρώματος και υφής ενός εικονοστοιχείου είναι ακριβώς μέσα στην διαδικασία σχεδιασμού. Υπάρχουν πολλές υλοποιήσεις του αλγορίθμου αλλά σχεδόν όλες χρησιμοποιούν υψηλής επίδοσης και ενεργειακής κατανάλωσης υλικό. Η ερώτηση είναι αν μπορούμε να φέρουμε τα ίδια αποτελέσματα στο τραπέζι με πολύ χαμηλότερης επίδοσης και ενεγρειακής κλάσης υλικό. Τα πειράματα και ο σχεδιασμός έδειξαν ότι αυτό εντέλει δεν είναι εφικτό με ένα καθόλα ενδιαφέρον τρόπο. Τα όρια του υλικού που είχαμε στη διάθεση μάς δεν μας σε συνδυασμό με τη χρήση πιο άμεσων από σχεδίαση σε προϊόν μέθοδών δεν μας έδωσαν την επίδοση που ψάχναμε αλλά μας έδειξαν τι πλαίσιο σκέψης και υλικό χεριάζετε.

Table of contents

List of figures

List of tables

Abbreviations

e.g.	exempli gratia - for example
etc.	et cetera - and other similar things
FPS	Frames Per Second
SW	Software
HW	Hardware
FPGA	Field Programmable Gate Array
ASIC	Application-Specific Integrated Circuit
GPU	Graphics Processing Unit
SLI	Scalable Link Interface
CPU	Central Processing Unit
SoC	System-on-Chip
HLS	High Level Synthesis
I/O	Inpu/Output
RT	Ray Tracing
RTI	Ray Triangle Intersection
MT	Moeller-Trumbore
BVH	Bounding Volume Hierarchies
PCIe	Peripheral Component Interconnect express
PCI	Peripheral Component Interconnect
2D	Two Dimensional
3D	Three Dimensional
HP	Half Precision
FP	Floating Point

Chapter 1

Introduction

Our everyday life is impacted in one way or another by computer graphics. From small things like TV commercials to feature length film and from video games to any application that requires the rendering of 3D graphics to a 2D screen. The main methods over the years to achieve this with is rasterization (common in PC gaming) and ray tracing (in movie animation the first movie being Pixar's Monsters Inc.). In other words, rasterization is mostly used for real-time applications where frame rates must be at least above 25 FPS, the more the better but 25 is the minimum number to achieve fluid animations, rasterization is preferred. This is also the prevailing technology used in all graphics processors today. This technology though can only get you so far, the last major graphics leap happened with the release of Crysis in November 13, 2007 developed by Crytek, Saber Interactive, Crytek GmbH, after that rasterization algorithm started to show its limitations when it comes to incorporating ever increasing need to evolve visual effects such as advanced reflections, lighting etc.

Ray tracing is quite old as an algorithm, the first ray tracing scene was taking two weeks to render at Bell labs in 1978. Nowadays it is still a heavy computationally procedure that is being lifted from its mostly animation off-line rendering duties into the mainstream with the introduction of Nvidia RTX 20 series GPUs in September 20, 2018. And with the release of RTX 30 series in September 16, 2020 the future of real-time ray tracing is ever nearer. More actually Nvidia's implementation still renders a scene using rasterization but with added hardware it can produce stunning lighting visuals.

1.1 Problem Statement

Ray tracing is an algorithm that exhibits high parallelism and is very computationally heavy a combination favouring FPGAs and GPUs. Alas high-power GPUs [4] [5] have all the potential in the world to develop such demanding rendering techniques due to having little resections in area or power. Cited papers utilize powerful FPGAs also made an appearance in research papers using systems like:

1. Multi-FPGA Xilinx Virtex-E prototyping system [6]
2. SGI RASC R100 Blade [7]
3. Xilinx Virtex-II 6000-4 FPGA [8]

With the last one achieving 20 - 60 FPS in some scenes. Or used low power systems, like Xilinx Spartan-3E 1600 [9] with limited rendering capabilities.

The problem is, can a more modern but power efficient and more importantly smaller FPGA achieve similar results while being general enough to be able to render a variety of scenes. A general-purpose ray tracing renderer accelerated with the power and re-configurable FPGA will be very useful in several scenarios. In addition to this goal modern design flows tend to move further away from low level descriptions like Verilog HDL and VHDL to higher level solutions like HLS.

1.2 Contribution and Content Mini-map

In the grand scheme of things, the research contacted focused on getting better than the on-board ARM processor for starters and the use of quick to market solutions like HLS while utilising a low power fabric in ZedBoard development board.

1. Researching various Ray Tracing implementations
2. Developing a base accelerator
3. Optimising accelerator
4. Deriving behaviour conclusions based on performance and other aspects of the design

This thesis was developed and documented in the following chapters:

Chapter 1 Introductory chapter setting the stage for the project and analysing the problem this thesis is trying to address.

Chapter 2 Background information regarding ray tracing, developments that have been made, selected platform for development

Chapter 3 Presenting why this particular software base and implementation was selected

Chapter 4 Utilizing HLS to build and optimise a basic accelerator

Chapter 5 Using the conclusions to further understand how to best approach ray tracing algorithms

Chapter 2

Background

2.1 What is Ray Tracing

Ray tracing produces stunning visuals in the movie industry as well as gaming using Nvidia RTX graphics cards that can make someone believe that it is very complex. On the contrary it is an intuitive and simple algorithm. On the most basic level it mimics the way our eyes perceive the world by collecting light rays bouncing from various objects. Thus, what we see is the objects colour and texture as it is affected by environmental contributions like direct lighting, reflections from other objects indirect lighting. We also observe the objects texture directly. Stating these obvious facts is important to help us realise that a renderer doesn't have this intuitive knowledge like we have and lacks the ability to observe its 3D environment. So, in order to build a frame, the renderer needs to create the image pixel by pixel while trying to figure out what the camera sees like Figure 2.1.

Getting more technical we start with a list of objects that comprise the scene, said objects are part of the world. Through a viewpoint, in computer graphics world this is referred to as camera or eye, we draw the scene. As mentioned before ray tracing is trying to solve the visibility problem, as opposed to light transport algorithms that simulate the way light propagates a scene. The solution to that problem is simply what the camera sees.

This way when a ray is generated for each pixel the algorithm tests for a possible intersection with an object. Each object is a polygon mesh of triangles. If an intersection occurs then that objects material and other options such as texture, colour and material are used to decide what colour it is. These are the primary rays and they serve the purpose to establish the first contact or intersection with an object. Secondary rays are generated from that point,

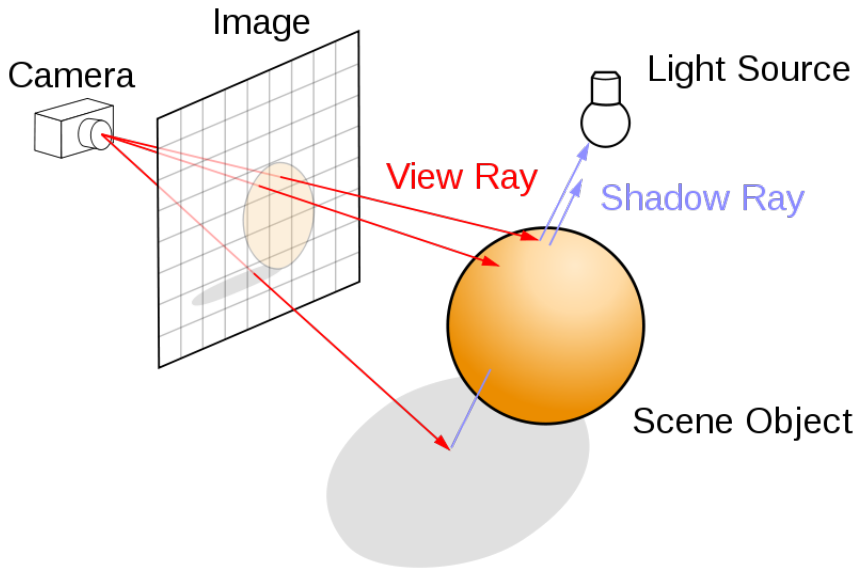

Figure 2.1: Simple RT Example

depending on the object's material properties, to determine the objects lighting effects Figure 2.2.

1. Shadow form other objects
2. Reflections in the case of mirrors
3. Refraction in the case of water or glass

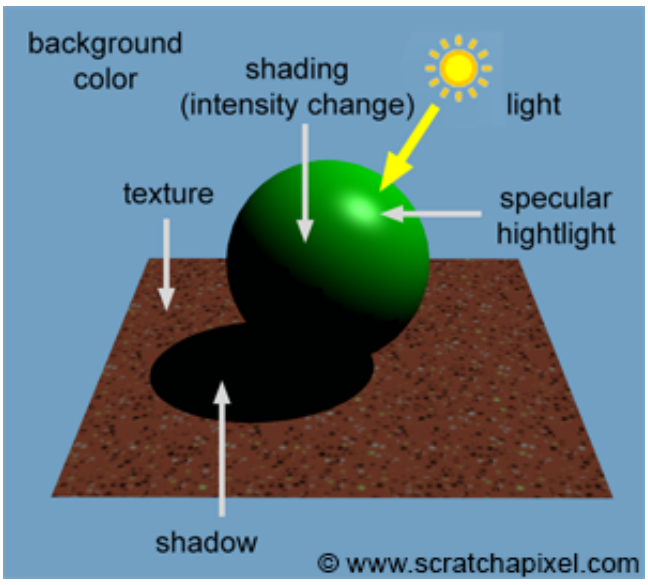

Figure 2.2: Some 3D Rendering Effects in Action

From that alone we can see how the process to decide what colour value each pixel ends up having is more neatly integrated with the render algorithm. Rays that end up outside of the view window or camera are discarded. This however is very computationally taxing notwithstanding that it has very high degrees of parallelism.

2.2 Used Platforms

Initial experiments run on my personal laptop equipped with an Intel Core i5 6200U dual core CPU with Hyper Threading and able to boost to 2.8 GHz on all cores and 16 GiB of DDR3L 1600MHz RAM. Having an ultra-low voltage CPU as a starting point serves as an example of a low power general purpose processor. Intel Vtune profiling program was utilized to profile the algorithm and establish hot spots that can be accelerated with the use of a hardware accelerator or many.

2.2.1 FPGA & ZedBoard

Field Programmable Gate Arrays FPGAs are somewhere between general purpose processors running an operating system and an ASIC designed for that specific application. Being equipped with an array of re-configurable logic blocks, block ram elements as well as both integer and floating-point arithmetic units they can be used to build hardware that comes very close to printed silicon performance. FPGA referrers to the programmable logic chip which doesn't come on its own, the board is a big part of the broad appeal and varied usage scenarios. It holds the fabrics I/O and interfaces with the outside world and that can also dictate the FPGAs effectiveness and performance e.g. PCIe connectivity, SD card slots, USB, UART serial connection etc. Notable usage scenarios include but not limited to:

1. Video & Image processing
2. High Performance Computing and Data storage
3. ASIC Prototyping

According to the application and price range there are a lot of offerings from Xilinx and other manufacturers like Altera (Intel), and Lattice Semiconductor amongst others that vary according to the size and capability of the FPGA programmable fabric to the number and bandwidth of on and off board interfaces.

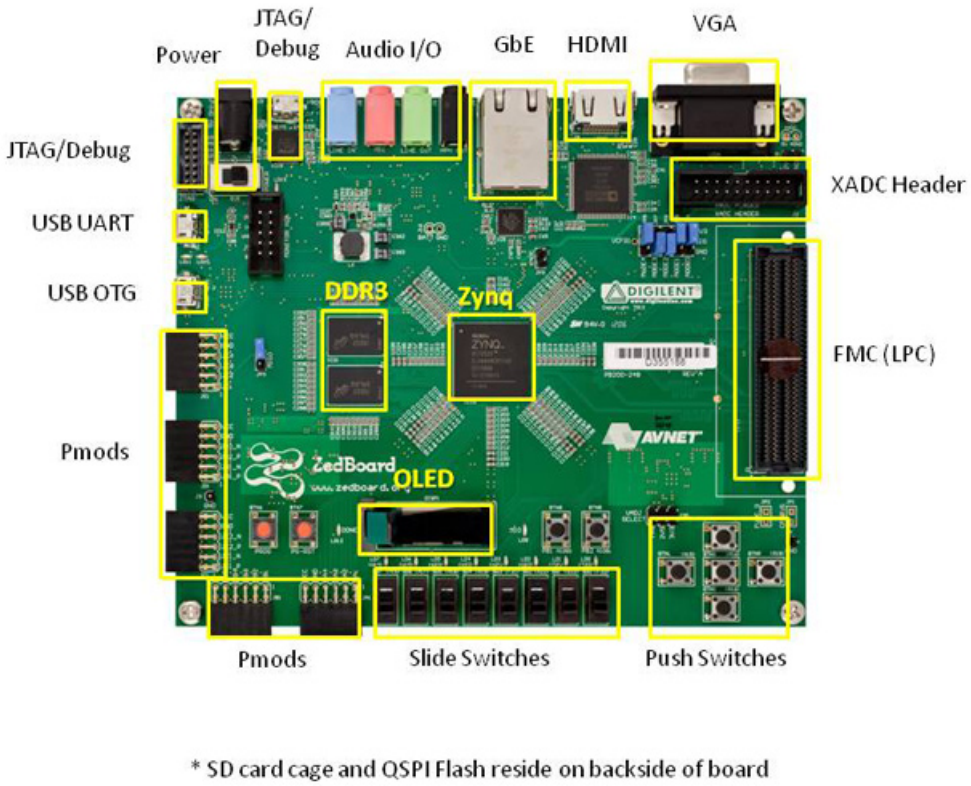

Figure 2.3: ZedBoard Board Diagram

ZedBoard:

The board chosen for this project is ZedBoard figure 2.3, block diagram pictured here 2.4, using Xilinx HLS bare metal accelerator design flow. It is equipped with Xilinx Zynq-7000 (XC7Z020 [10]) all programmable SoC which contains two ARM A9 processors clocked at 667 MHz and an Artix-7 FPGA fabric with a maximum clock of 250 MHz. That value varies depending on the design and overall fabric utilisation. It offers a good balance of performance and power efficiency for its size requiring only passive cooling with a small aluminium heat-sink as well as a plethora of connection interfaces. In this particular development board we can take advantage of the ARM processors and create a low overhead application that offloads computation tasks on the hardware accelerator on the FPGA fabric and not have the need to include an operating system like peta-Linux, used by Xilinx on their boards. Running bare metal application means that there is a small piece of software containing addresses and functions for hardware and board accelerators without the need for a fully-fledged operating system. Such action also has its limitations.

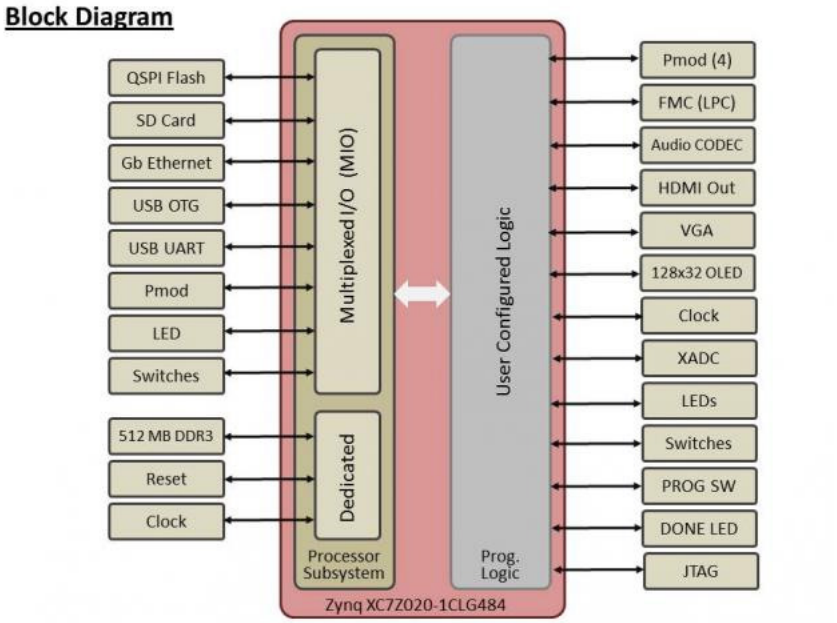

Figure 2.4: ZedBoard Block Diagram

HLS:

High level synthesis is a program, in this case Xilinx Vitis HLS, that produces hardware accelerators based on C/C++ source codes. Said codes need modifications to facilitate a smooth transformation from software code to hardware description. In laments terms OS functions and calls, variable size buffers etc. need to be removed or specified statically [11]. Ports need to set-up as to establish I/O communication with the rest of the system. Digital design good practices also apply here as we can model the software we use as a base aimed at creating as simple and strait forward software description for our hardware. As an automated solution it doesn't lend it self to much tinkering with the produced hardware making fine control over the hardware difficult to impossible.

2.3 Notable Research

2.3.1 FPGA

Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip [8]:

Research into this topic using FPGA platforms has been attempted before with very good results in while using dynamic scene rendering. One particularly successful is the real-time Ray Tracing of Dynamic Scenes on an FPGA Chip by Computer Science, Saarland Uni-

versity, Germany. Utilizing high performance at the time Xilinx Virtex-II 6000-4 FPGA contained on an Alpha Data ADM-XRC-II PCI-board they managed to have 20 - 60 FPS on various dynamic scenes on a full ray tracing pipeline on a single FPGA with only 90 MHz. Expanding upon an established architecture, written with JHDL (Just-Another Hardware Description Language), and using the power of the FPGA fabric to exploit the inherent parallelism of the algorithm they achieved these impressive results.

This work is based on a powerful and large fabric with many FPU available to the user and being connected to the PCI interface. As well as low-level structural hardware description language. Both powerful tools in the hands of a capable digital designer.

Caching architecture for flexible FPGA ray tracing platform [12]:

Computation performance improvements were not the only area of research into FPGAs and ray tracing, data structures and traversal were also under the microscope. It is not only important to increase processing power of a system by increasing parallel processing of said data, you also need high enough throughput to be able to feed that system with data. Using a naive implementation can lead to ineffective use of available bandwidth and space.

An improved data structure, like BVH and KD trees, and access method is also needed for an improved throughput performance. In that publication memory access patterns, different cache types and data replacement methods. A more flexible memory controller was the answer that took advantage of information existing only at run-time. Evaluating said experimental observations that team created a novel cache structure and replacement policy that alleviated the memory bottlenecks that accompanied performance scaling.

2.3.2 Nvidia RTX

Turing GPUs [4] made their debut with hybrid rendering combining traditional rasterization and ray tracing. While rasterization is typically faster to acquire an objects visibility, determining that instead of the primary ray. RT is being utilized for shooting secondary rays that it would be possible to create smooth correct reflection, refraction, shadow, and other effects rasterization is forced to make simplifications and approximation errors. Both Turing [4] and Ampere [5] architectures feature dedicated RT cores and BVH traversal units to accelerate RTI with the latter providing significant performance improvement when it came to RTI. In all the above research the conclusion is the same. RT is achievable with good performance on either powerful FPGAs with proper data structure management or powerful

GPUs with hybrid rendering. Nvidia also utilizes tensor flow cores as artificial intelligence de-noising filter in an effort to reduce the number of rays needed for each scene as well as preform resolution up-scaling using a smaller resolution image as a start. This improves the performance further.

All those implementations have something in common, the usage of powerful expensive hardware. Can we achieve similar results with the smaller and less expensive ZedBoard?

Chapter 3

Base SW Rendering Engine

3.1 Chosen Implementation

According to the research studied the main differentiating factor between RT implementations is to decide how one handles ray/object intersection. What granularity level do we choose for our system, having a different intersection procedure according to each object type and shape, be it sphere, cube or parallelepiped even. A complex natural scene consists of many different objects and if performance is our goal this will have to be streamlined.

Luckily for us 3D graphics have a solution for this problem already. By creating all objects with a simple geometry that is booth simple to intersect and versatile enough to shape other objects with this problem is solved. Such object is the simple triangle, defined by 3 vertices and occupying the same plane making it easier to detect intersections as we will see afterwards is a very solid choice. All convectional 3D renderers render objects as polygon meshes of triangles for example in figure 3.1.

Ray triangle intersection is the most difficult part of the Ray Tracing algorithm. Triangles are not really a geometry type of their own. Rather, they are a subset of the polygon primitive type. Intersection between triangle and a ray is simple and lends itself to various optimizations.

Several papers [8] [6] [13] came to the conclusion that Möller-Trumbore [14] RTI version is the most efficient and has the best performance and it offers best compatibility with existing software platforms.

The general roadmap involves building a Ray Tracing renderer that will have its more computationally intense parts accelerated with hardware accelerators. Internal data structures

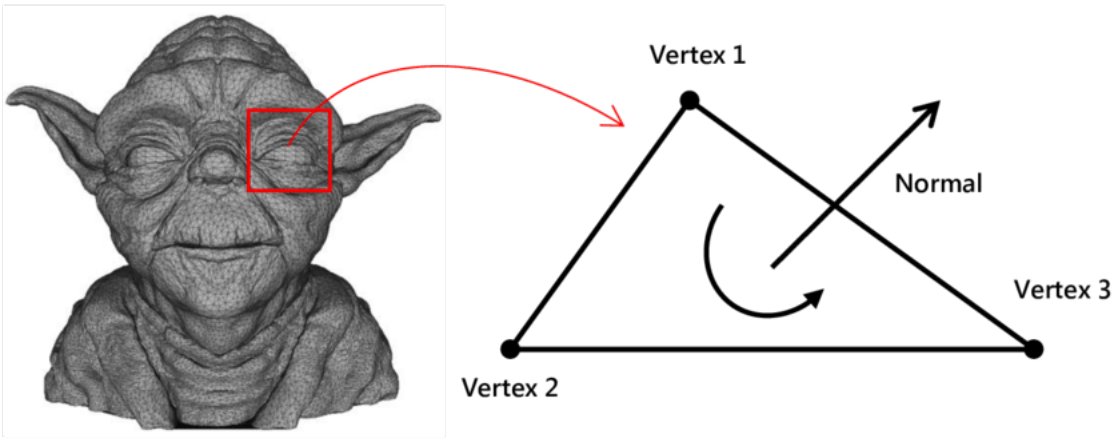

Figure 3.1: Example polygon mesh from triangles [3]

as well as their traversal will have to be optimized to achieve near real time performance. Such tasks will be initially carried out by the ARM processor of our board.

3.2 Ray Triangle Intersection Algorithm

Based on the source code and explanation from this site [15] that provides a whole rendering tutorial on the general fundamentals of rendering in both rasterization and ray-tracing algorithms.

I took the code from the base Ray triangle intersection and future HW accelerator and run it forming first impressions and figure out the maths behind the basic ray triangle intersection before moving to the next more complete Ray Tracing renderer.

The initial phase the program has hardwired geometries for three triangles, it calculates only the ray triangle intersection and returns a static colour. This is the principle essence of a Ray Tracing algorithm among others. Thus, it was decided this was a good starting point.

The code was cleaned and documented as it was filled with code from previous steps of this tutorial, such explanations were filled accordingly.

We start with basic equations for the ray triangle intersection. The first equation 3.1 where t is the distance from the ray origin, O is the ray origin and R is the direction.

$$P = O + tR \tag{3.1}$$

The MT algorithm utilizes the above equation along with the property of the parameterization of P (the intersection point in terms of barycentric coordinates with the plane) as seen in barycentric coordinates. In equation 3.2 A, B, C are the vertices of a triangle and u, v, w

the barycentric coordinates, which are usually normalized $u + v + w = 1$. This property can help us reduce operations needed to calculate w from u and v $w = 1 - u - v$. The u and v coordinates

- Can't be greater than 1 nor lower than 0
- The sum can't be greater than 1 (u+v <= 1)
- They express coordinates of points defined inside a unit triangle

Thus 3.2 becomes 3.3

$$P = uA + vB + wC \tag{3.2}$$

$$P = (1 - u - v)A + uB + vC \tag{3.3}$$

Developing we get

$$P = A - uA - vA + uB + vC = A + u(B - A) + v(C - A) \tag{3.4}$$

Based on that equation and combining with 3.1 with 3.4 we get 3.5. The left side of the equation can be viewed as the transformation that moves the triangle from its original world space position to the origin. The other side has the effect of transforming the intersection point from x,y,z space to t,u,v space.

$$O - A = -tD + u(B - A) + v(C - A) \tag{3.5}$$

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

The solution of a system of linear equations can be found in terms of determinant using Cramer's rule. Scalar triple product is defined to be the dot product of the vectors with the cross product of the other two, it can also be understood as the determinant of a 3x3 matrix.

Thus re-writing equation 3.5 using Cramer's rule we get

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times D) \cdot D \end{bmatrix} \tag{3.6}$$

Where $T = O - A$, $E_1 = B - A$, $E_2 = C - A$, $P = (D \times E_2)$,$Q = (T \times E_1)$.

In order to connect the maths with the code the above becomes

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(-dir \times v_0v_2) \cdot v_0v_1} \begin{bmatrix} ((orig - v_0) \times v_0v_1) \cdot v_0v_2 \\ (-dir \times v_0v_2) \cdot (orig - v_0) \\ ((orig - v_0) \times v_0v_1) \cdot -dir \end{bmatrix} \tag{3.7}$$

Where in the code $P = pvec = (D \times E_2) = (-dir \times v_0v_2)$, $Q = gvec = (T \times E_2) = ((orig - v_0) \times v_0v_1)$, $tvec = orig - v_0$ and $det = pvec \times v_0v_1$

The intersection checks based on this equation are then simple, firstly we check if the intersection point lies on the plane. With culling enabled

- If the dot product of the ray direction and the triangle normal (det) is, or is close to 0 the triangle and the ray are parallel
- If the dot product of the ray direction and the triangle normal (det) is negative the triangle is back-facing

With culling disabled we only check the first condition. It is also important to note that with culling disabled we need to normalize u by multiplying it with the inverse of det to properly test the intersection.
Then we move to calculate u, v and t.

$$u = (tvec \cdot pvec) * 1/det \tag{3.8}$$

$$v = (-dir \cdot gvec) * 1/det$$

$$t = (v_0v_2 \cdot gvec) * 1/det$$

There is no intersection

- If u is either lower or greater than 1
- If v is lower than 0 and the sum of u and v are greater 1

During this initial run we observed the Ray Triangle intersection algorithm in various scenarios. These involved single triangle intersection as well multiple triangle with and without *culling*, culling is the idea of not rendering back-facing triangles in order to reduce intersection calculations. Back-facing triangles have their normal pointing in a different direction than the ray direction.

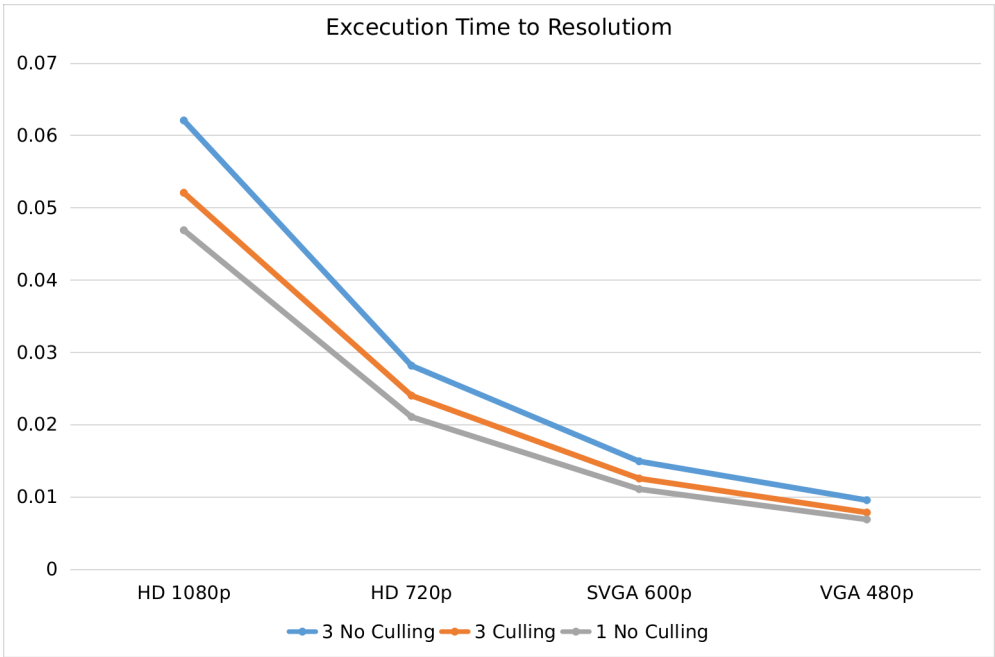

Figure 3.2: RTI performance observations on 3 triangle scene

We wanted to benchmark the basic ray triangle intersection to get an idea how it would perform in a completed system with mesh of triangles and Phong effects. What can be further optimized on this routine in order to speed up and achieve real time rendering. How much the resolution and culling affect execution time.

We tested 3 cases with 4 different resolutions figure 3.2

1. 3 triangles with no back facing ones (No culling)

2. 3 triangles with 2 back facing ones (Culling)

3. 1 triangle not back facing (No culling)

The first is used to establish a baseline for a simple shape. The second serves the purpose to evaluate how much performance can we gain by disregarding the back facing when calculating primary rays. The third test is to evaluate the raw performance of the algorithm with having only one triangle in the scene.

We found that resolution affects the performance greatly. Lower resolutions have orders of magnitude quicker execution time.Going from 1080p to 720p gained huge time advantage and still maintaining HD spec. Culling offered less performance improvement. Going from 3 to 3 with no culling reduced the time. But even so less triangles was faster still.

3.3 Render system

In this step I studied each of the tutorial stages:

1. Introduction to Polygon Meshes [16]
2. Ray-Tracing a Polygon Mesh [17]
3. Transforming Objects using Matrices [18]
4. Introduction to Shading [19]
5. The Phong Model, Introduction to the Concepts of Shader, Reflection Models and BRDF [20]
6. Global Illumination and Path Tracing [21]

To build a minimal yet functional ray tracing renderer that would serve the purpose of our gold software base. The final system includes reading, traversing, and rendering polygon mesh as well as basic shading that includes diffuse, reflect, reflect-refract and Phong model effects.

Global illumination and path tracing introduced additional load other that RTI which was the focus for an initial implementation. As it stands direct lighting is the only way a scene is illuminated. Indirect lighting is not accounted for reducing the realism but keeping the process more straight forward. The source suggested using the Monte-Carlo method to reduce the need for a lot of rays for the contribution calculation.

It was deemed enough to achieve a baseline performance reading on the whole idea of Ray Tracing (ray triangle intersection, triangle traversal and shading) while having minimal complexity as to not cause many development headaches. The base C++ source code was expanded upon with dynamic object and scene option file input parser and argument reader. Every bit of code was documented as to what they do. And a PSNR comparing program was built to check each implementation result against golden samples produced by golden software implementation.

The rendering process has two main stages, visibility (can the user see an object from his perspective) and the shading (what colour will those pixels have). The visibility problem has to do with the shape an object has and whenever that object is visible from the perspective of

the user. This is the part of the rendering equation where the ray triangle intersection comes in.

Its ultimately concluded by calling the *trace* function and following the list *Intersect, Ray Triangle Intersection* below to determine if an intersection occurs. When such an intersection occurs, we need to figure out the colour of that pixel. Shading is exactly that. It is the process in which we calculate the colour of the intersected pixel.

A more general view of the flow of the renderer:

1. Parse Data (Scene Data, Object Geometry and Option Data)
2. Render Loop (Frame Loop)
3. Cast Ray (Shading Pixel)
4. Trace (Object Loop)
5. Intersect (Triangle Loop)
6. Ray Triangle Intersection (Ray Triangle Calculations)

Data structures are set-up to use a C++ class for vectors and matrices with build in functions for maths operations from addition and multiplication to dot and cross products. This simplified the initial implementation by having this abstraction over maths operation and it facilitated the easier implementation of visual effects.

Polygon Meshes:

The renderer relies on a polygon mesh description (format created by [16] explained in Appendix A) in order to render objects on the screen as well as my own scene and object data description file formats (also explained in Appendix A). This was done to facilitate easier reading of more complex objects. All that data is parsed at the beginning of the render process and are kept in memory. As now the renderer stands it only supports the rendering of 1 frame only. That can tell us if a potential use of the system in a renderer with user controls and video game like setting would be real-time if 1 frame takes less than 0.033 seconds to render.

Render:

Rendering an image is done with a double for *j for height, i for width* loop. As we traverse each pixel *height x width* calculations need to be made to determine several parameters about

the final image, static ones for the whole image and dynamic ones that change with each pixel.

Static calculations, same for the whole image:

- scale is found by $scale = tan(degreestoradians(FOV * 0.5))$
- aspect ratio of image $imageaspectratio = width/height$
- ray origin $origin = cameratoworld * zerovector$

Dynamic calculations that are pixel bound and need to be calculated for every pixel

- x normalized coordinate for the pixel $x = (2*(i+0.5)/width-1)*imageaspectratio* scale$
- y normalized coordinate for the pixel $y = (1 - 2 * (j + 0.5)/height) * scale$
- ray direction $direction = cameratoworld * (x, y, -1)$
- ray direction normalized

After all parameters are calculated the we cast the ray by calling the *castRay* function which considers these parameters: ray origin, ray direction, object list, light list, and scene options in order to determine if there is valid intersection between a ray and and object and if there is what colour must be shaded based on the objects, and lights properties.

Cast:

At this point we are preparing for the ray to be traced to a potential triangle intersection checking to see if the maximum depth has been reached or not. In the case that it has been reached we simply return the background colour.

This is where the *trace* function is called. It is given the ray origin, direction as well as an object list and a struct to place intersection point data if such point exists. If a successful intersection is found, then the function returns true based on the intersection details and object properties we colour the pixel accordingly. The hit pint is evaluated from the ray origin, direction and the distance to the hit point as such $hitPoint = orig + dir * isect.tNear$. Data like the objects normal, texture coordinates are retrieved using *getSurfaceProperties* from the object list.

After that it is a simple matter of finding the object type and colouring/ illuminating the pixel as per the effects in the **Shading** section.

Trace:

Tracing a ray in this implementation is naive yet effective. It allows us to focus our attention to first solve bugs in the integration with the hardware accelerator and then choose to upgrade them in order to achieve more performance.

It is a simple procedure to check each object from the list of objects and then Calling the *Intersect* function to check each triangle within that object if that falls within the rays path and how close it is from the origin.

Intersect:

The intersection is solved by calling the *rayTriangleIntersect* function, that will be later replaced by a hardware accelerator, for each triangle in that object. That function takes care of RTI as explained before.

Shading:

The shading effects include

- Diffuse shading using a generated pattern or solid colour
- Reflection effects
- Reflection and refraction effects
- Phong shading using a more fine tuned colour using weights for diffuse, specular effects

Diffuse:

The simple defuse shading is done through the *illuminate* function. It takes in to account the hit point, light direction, light intensity and how far is it from the ray origin. This function takes uses the type of light point or distant. Both belong in the class light which contains information like light to world matrix, colour, intensity and the illuminate function prototype. Then each light source has its own class with the implemented illuminate function. Different light sources require different illuminate functions and are implemented as such.

All effects that use light sources in any way have their contribution added up in a loop.

Illuminate/ Light class:

Point light acts like a localized light source. A pint light has a position which is found by multiplying the 0 vector with the light to world matrix. This is done due to the assumption that point lights are created in the world origin space and then are placed in the correct position

by the light to world matrix. It is unaffected from rotations. It lights up other diffuse objects using the following formula:

- $lighttoworldmatrix * zerovector$ get light position
- $lightdirection = Hitpoint - Lightposition$
- compute the square distance $r2 = lightdirectionnormalized$
- normalize incident light ray direction $lightdirection/ = distance$
- final light intensity with square falloff $lightintensity = colour{*}intensity/(4{*}pi{*}r2)$

Distant light sources are simpler they only take into account the direction of the light source, colour and intensity. It computes direction instead of position in order to replicate the distant nature of the light type. It acts like a sun like source as it light ups all objects from the same direction. In oder to calculate the direction we multiply the light to world matrix with the (0, 0, -1) and then normalized.

- $lighttoworldmatrix * (0, 0, -1)$ light direction becomes the calculated direction
- $lightintensity = colour * intensity$ light intensity is found by taking into account the colour
- distance is equal to float max

After the illumination part is calculated we call the *visibility = !trace(...)* function once again in order to determine if the intersection point is in the shadow or not. If the trace function is evaluated as true we want to colour that pixel dark for that means it is some shadow.

A simple pattern is calculated on various geometric shapes such as (Blurry chequerboard, Diagonal chequerboard, Stripped, Grey chequerboard, Solid Grey Colour). For simplicity we chose to go with a solid grey colour.

$$hitcolour+ = visibility * pattern * lightIntensity$$
$$*max(0, (hitNormal) \cdot (-lightDirection))$$

Reflection:

Reflection is a simple effect and it is calculated using the *reflect* function with incident view direction and the normal of the surface. This function is also used in the reflection and refraction effect.

After that the hit colour is added as a contribution with a recursive call to the *castRay* function with the data from that point with depth + 1. This is akin to generating rays to find what the reflection adds to that pixel.

$$reflection = incidentviewdirection - 2 * incidentviewdirection \cdot objectnormal$$

$$reflection = normalizereflection$$

$$hitColor+ = 0.8 * castRay(hitPoint + hitNormal * options.bias,$$

$$R, objects, lights, options, depth + 1);$$

Reflection and Refraction:

This effect is a bit of a more complex affair. We need to figure out how much light reflected and how much continues or transmitted, start by computing Fresnel parameters based on ray direction hit point normal, intersected object index of refraction.

This implementation disregards the effect of light attenuation and absorption that happens in transparent materials.

Fresnel (refracting vs reflecting light) is the effect involving transparent objects like water, glass or anything light can pass through and gives us the above amount. This will give us the relation of reflected light and help us describe the transmission of light when incident on a surface normal. When the angle of incidence decreases the transmitted light is increasing and vice-versa.

To calculate the Fresnel parameters, we begin by clamping (limiting the position in an area, commonly used in computer graphics) the $incidentlightdirection \cdot hitpointnormal$ between -1 and 1 values. The result of clamping helps to determine if the incident ray hits the object from outside or inside by checking the sign.

We also set several parameters with 1 and the ior and then swap them if the incident ray is positive that will help us calculate the kr factor using trigonometric functions.

If total internal reflection is occurring, the phenomenon where the angle of incident light is greater the critical angle such that 100% of the light incident is reflected, kr factor is 1.

Such an occurrence is caused by the light interacting with materials with each successive interaction having a smaller index of reflection.

Light is simulated to travel as two waves (parallel and perpendicular polarized) perpendicular to each other and to find kr will need to calculate the ratio of reflected light for these two waves, the average gives us the ratio of reflected light.

$$cost = sqrtf(std :: max(0.f, 1 - sint * sint));$$

$$cosi = fabsf(cosi);$$

$$Rs = ((etat * cosi) - (etai * cost))/((etat * cosi) + (etai * cost));$$

$$Rp = ((etai * cosi) - (etat * cost))/((etai * cosi) + (etat * cost));$$

$$kr = (Rs * Rs + Rp * Rp)/2;$$

After the fresnel coefficients we must figure out if the ray is inside or outside. A shadow bias is added to account for the shadow displacement needed for various scenes and object properties. It is usually very small.

$$outside = direction \cdot hitnormal < 0$$

$$bias = objectbias * hitnormal$$

Finally if the value of kr coefficient is less than 1 we treat the effect as a refraction otherwise as a reflection. Both make use of recursive call of the *castray* function , with a max depth limit as mentioned above, in order to find the colour contribution in of the pixel.

In the case of refraction the *refract* function is called to find the refraction direction and then decide if the ray is outside or not by adding or subtracting the shadow bias.

Refract:

This function serves to calculate the refraction direction. As with the *Fresnel* calculations begin with *clamp* function and the setting of *etai=1,etat=ior* parameters. If the result of the *clamp* function is negative that means the ray is outside the surface of the object adn the cos(theta) must be positive. In the case of the ray being inside the surface cos(theta) is already positive so, we just swap the refraction indices and reverse the normal direction. After that the function simply returns the k factor.

In the case of reflection, the *reflect* function is called to find the reflection direction and then decide of the ray is outside or not by adding or subtracting the shadow bias.

The final hit colour is determined by combining reflection colour and refraction colour accordingly.

$$hitColour+ = reflectionColour * kr + refractionColour * (1 - kr);$$

Phong:

Phong effects are in essence a more realistic diffuse effect in combination with specular reflections, as all objects have a diffuse and specular nature to their appearance. It adds weights and extra calculations to determine how big the specular spot will be and how glossy or mate a surface will look.

Determining the colour of a pixel is the same as the simple defuse effect up to the point where we find out if the point lies in a shadow. After that we compute the diffuse component based on if the point is under shadow or not, the objects albedo factor (the ratio of reflected light over the amount of incident light), light intensity and the dot product of hit normal and light direction.

$$diffuse+ = visibility * objectalbedo * lightintensity$$
$$*max(0, hitnormal \cdot -lightdirection)$$

The specular component is found by using the *reflect* function and calculating using the following equation, n is the specular exponent factor, it controls the size of the specular spot

$$specular+ = visibility * lightintensity * pow(max(0, reflection \cdot -raydirection), n)$$

As mentioned above the contribution of all the light sources are added up to the above components. At the end the hit colour is determined by adding the Phong components together along with their respected weights from each object's properties.

$$hitcolour = diffuse * intersectedobjectKdfactor$$
$$+specular * intersectedobjectKsfactor$$

3.3.1 Testing Methodology

Correction checking and scene description:

In this step the main goal was to expand the testing suit of scenes and establish a more reliable way to measure what the actual algorithm is doing the next step.

Scenes were added using the already constructed objects geometric descriptions from the code source but now they cover a wider range of the renderer's capabilities. In essence we "activate" a more complete part of our renderer to better identify performance hot-spots and transfer said functionality to a hardware accelerator. This is important because due to the range of visual effects supported these scenes represent each effect from different object materials to different type and light colour.

For that reason, after making sure by optical evaluation that the baseline golden software is functioning correctly gold image samples were created. These samples, seen in Appendix B are used to check the correct render outputs after hardware design and optimisation or software optimisation runs.

3.4 Software Baseline & Hot-spot Analysis

Performance baseline was set by running seven different scenes, see Appendix B, on my personal laptop as described in section 2.2 using the described gold software baseline.

Resolution	Glass Pen B.1	Simple Plane B.2	4 Glasses B.3	4 Glasses Point B.4	Simple Plane 2 B.5	Utah Teapot B.6	All Materials B.7
720p	150.64	0.07	26.24	33.33	0.08	82.15	1658.88
1080p	302.81	0.15	58.32	74.1	0.16	182.23	3745.63

Table 3.1: Baseline Software Ray Tracing Performance

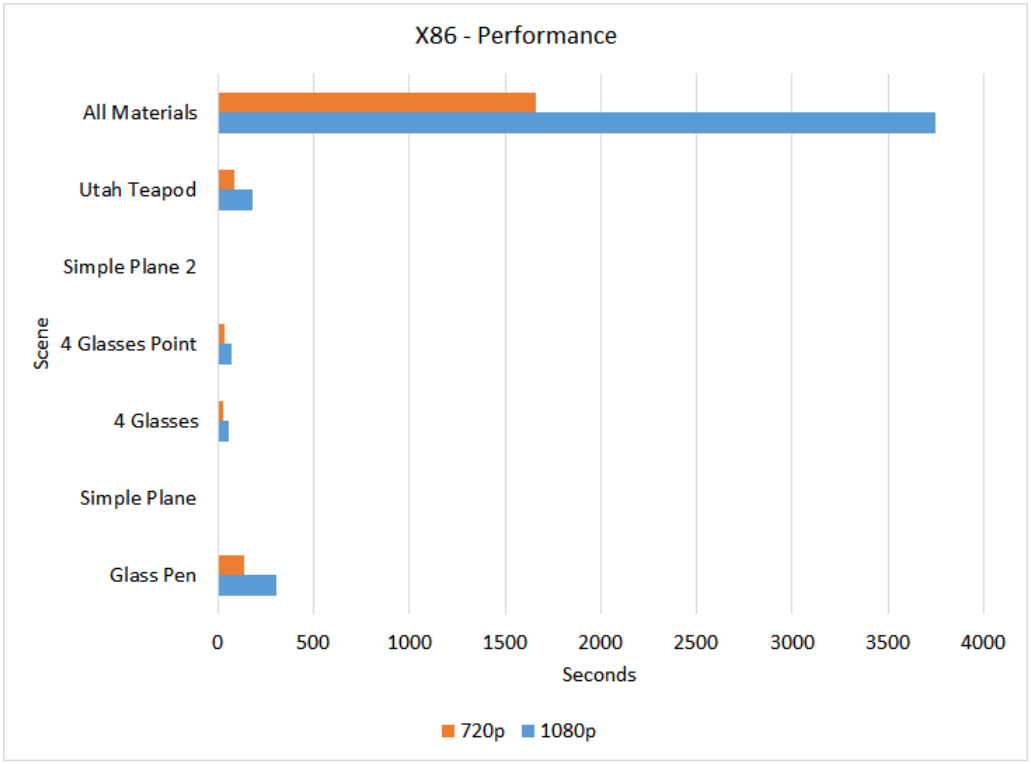

Figure 3.3: X86 Performance

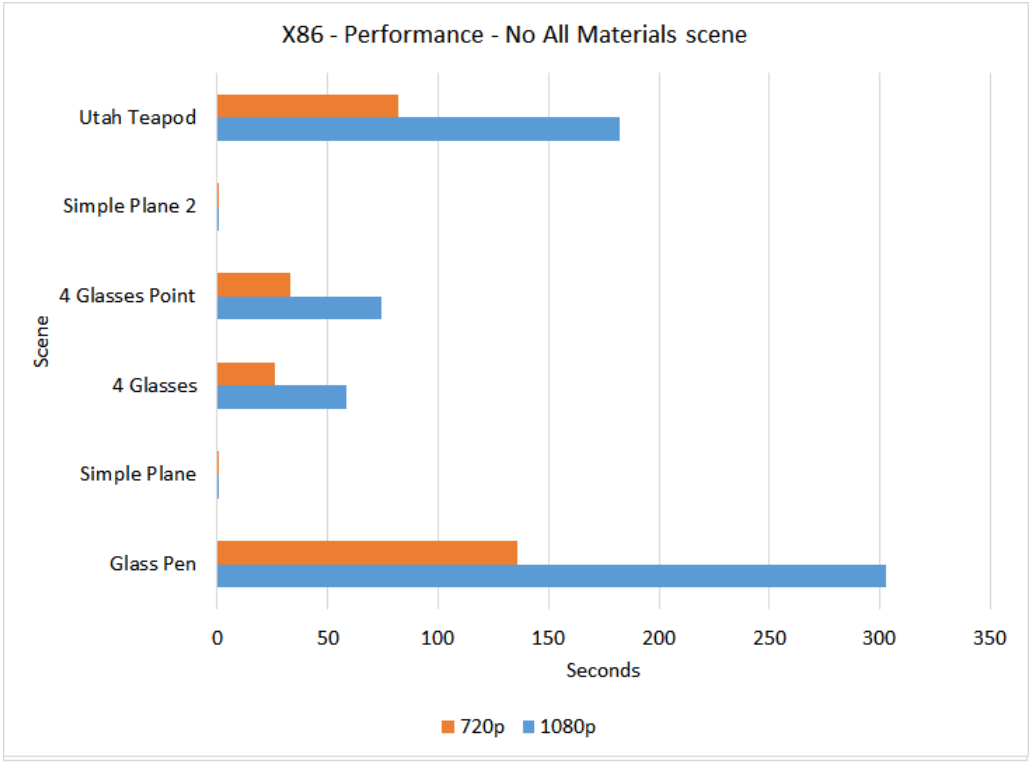

Figure 3.4: X86 Performance - No All Material scene

Hot-spot analysis:

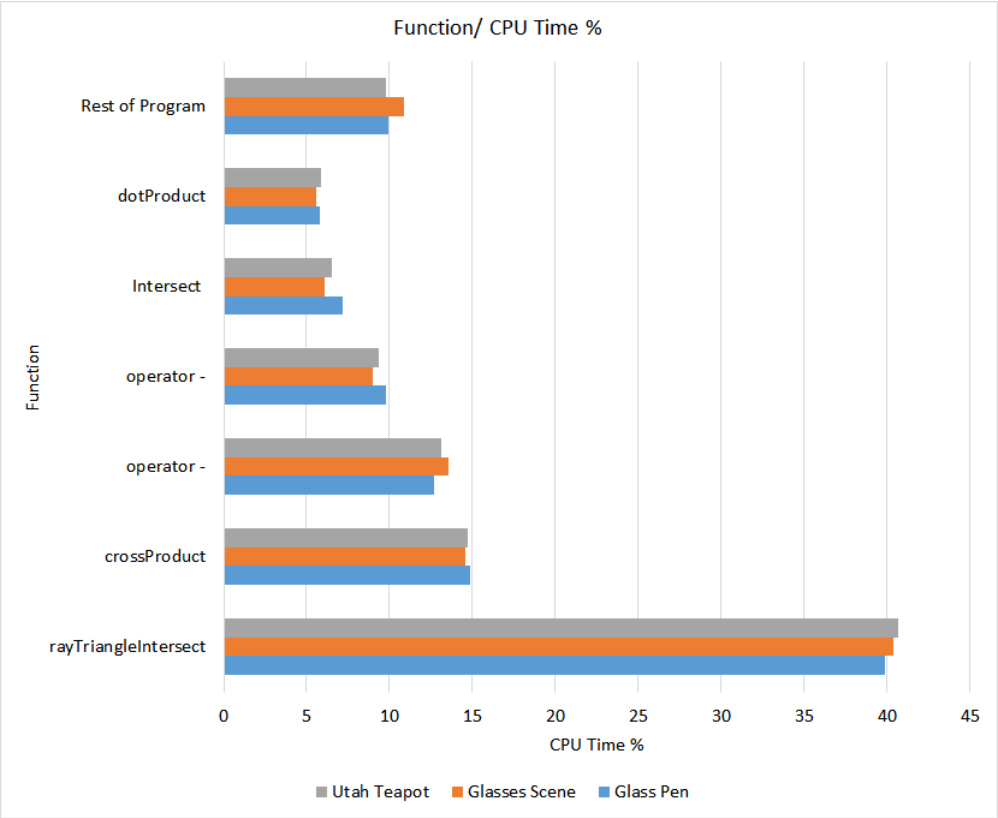

Figure 3.5: Function CPU time per overall CPU time in % - Using Class Functions

The performance hotspots of the base code were evaluated in Intel's Vtune Analysis software. Gold program was selected for this test. Benchmark scenes were defined as those that required a lot of time to render giving us ample samples to work with as well as testing various effects glass pen B.1, 4 glasses B.3 and Utah teapot B.6 initially.

The following graph shows the overall execution time for the scenes split to the functions that were identified as performance hotspots 3.5, using bottom up analysis. Results paint an interesting picture about the destination of spend time.

As mentioned, vectors are stored and handled by a class, Vec3f, which has maths operations build in. In this case Vtune reports these functions as different hotspots in the calling function ray triangle intersect. Although from initial tests it is pretty apparent what function is a performance hotspot.

To make sure no other call to the Vec3f internal functions like crosProduct and others interfered the test was rerun with all material B.7, glass pen B.1 and Utah teapot B.6. The function was modified by removing the calls to Vec3f functions and in-lining all math functions using floats. That way we have a single function that Vtune can track for hotspots by itself.

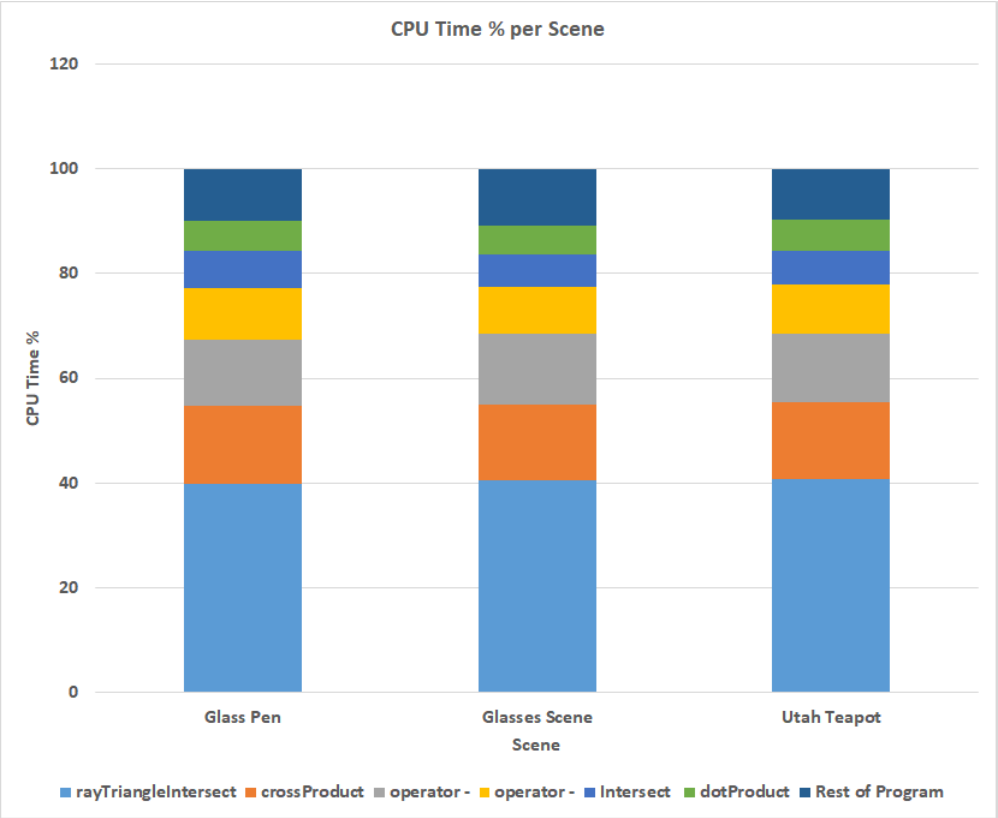

Figure 3.6: Function CPU time per overall CPU time in % - Using Class Functions

The results in figure 3.7 confirm the earlier results with the majority of CPU time being spend in *rayTriangleIntersect* function, the next in the function that traverses all the triangles of each object and the rest of the program.

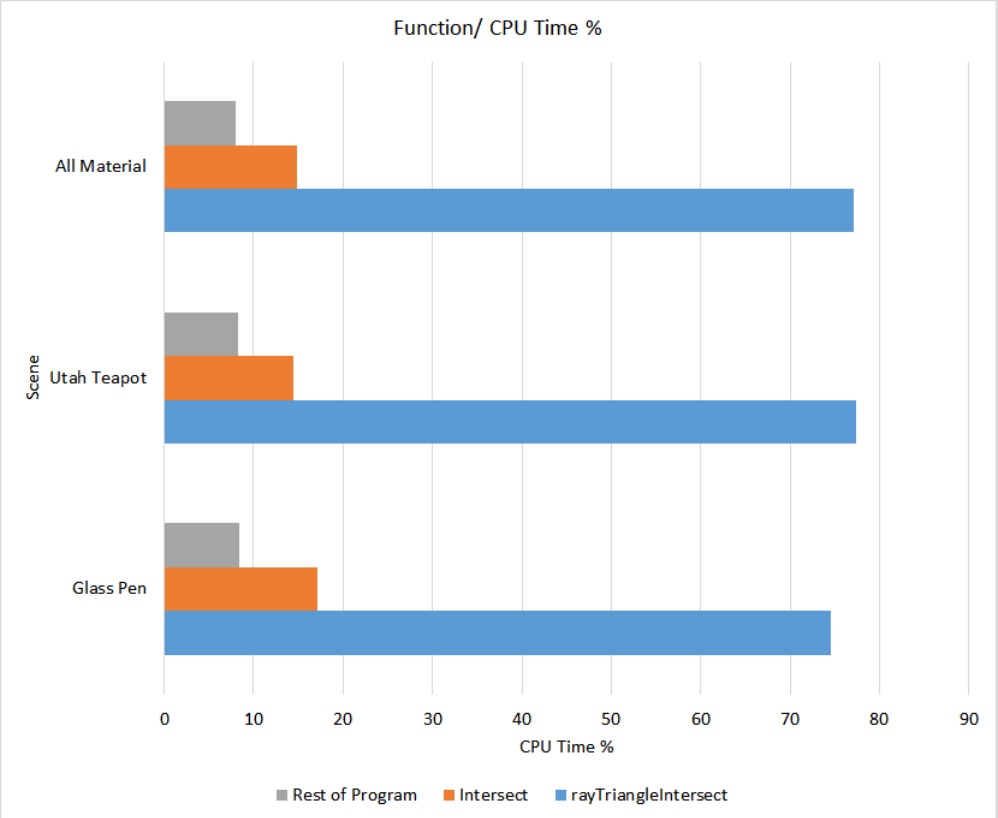

Figure 3.7: Function CPU time per overall CPU time in % - In-lined Function

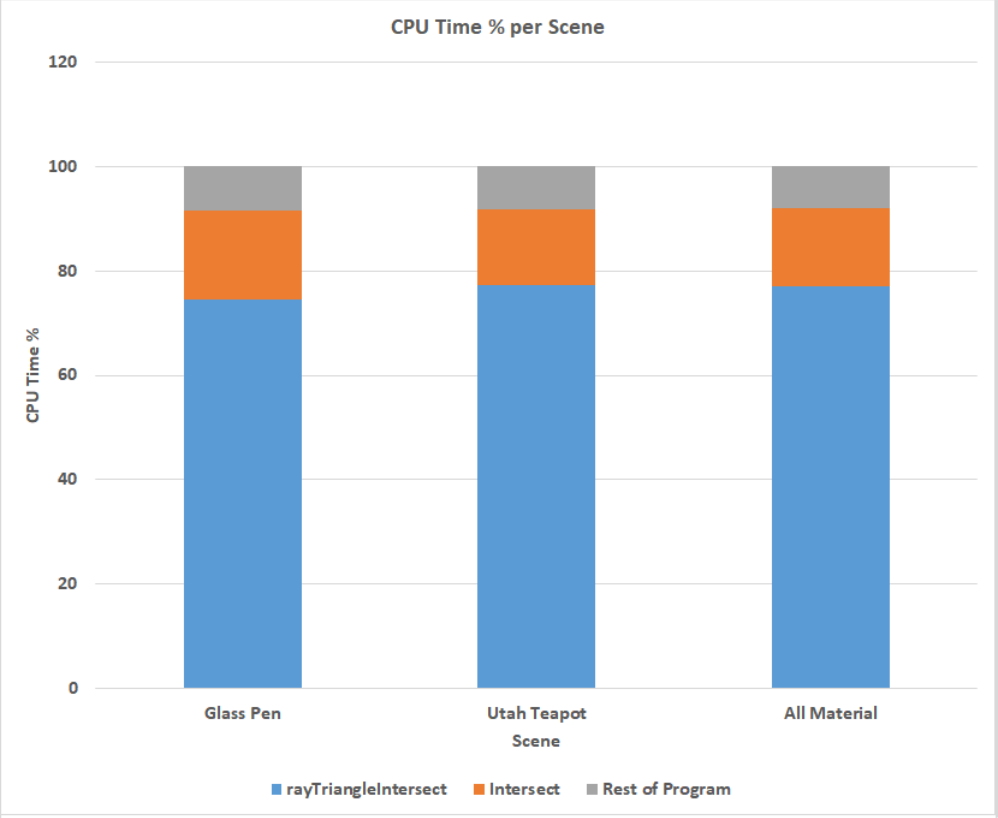

Figure 3.8: Function CPU time per overall CPU time in % - In-lined Functions

Chapter 4

HLS stage

4.1 HLS base accelerator

Defining the I/O and more importantly which part of the render process to build in to HW was decided basted on the performance and hot-spot analysis performed in the previous phase.

Ray triangle intersection took the majority of execution time so it was a good place to start.

That raises the need to transfer 5 inbound vectors as input

- Rays origin point
- Rays direction
- Triangles vectors (v0, v1, v2)

and 3 as output on a successful intersection

- Distance to intersection point
- Coordinates on the intersected triangle (u, v)

Software platform:

At first to have a more level playing field and give out first accelerator a fight chance performance will be compared not with the x86 implementation but with the on-board ARM processor of out starter development board *ZedBoard*. That will be the software baseline we will try to beat.

Initial plan:

The first part focused on using the code as it stands and producing the hardware accelerator with no modifications simply be using HLS. Using bare metal to use our accelerator as a starting point is a more strait forward way to gauge the performance of a design. With petalinux, Xilinx board compatible linux distribution, and use Ubuntu ARM existing as a future step that will unlock OpenMP functionality.

HLS specific pragmas will be set up for the I/O ports using the Xilinx recommended options. As the program is using class and addresses (pointers) to transfer the data back and forth *m axi* was the way to go. Setting the port offset to slave and a depth multiple of 2 (the Vec3f class contains only 3 floats as data so padding is required but is performed by Vitis HLS). An *axi lite* return port must be used regardless of the return value of our accelerator.

Thus I/O will be done by using the Vitis HLS produced functions to set pointers and use the Vec3f class as the software does.

Backup plan:

Due the complexity of today's HLS methods and the abstraction level of C++ class data type situations my arise a that will be beyond our control (long critical paths, hls compiler having insufficient data) backup plan will be in-place. We can skip all the C++ complexity and add the maths operations directly in the accelerator code to be compiled in to hardware. Having that in mind we can also utilize a simpler port standard *axi lite* to transfer individual float numbers for each coordinate we need to transfer.

Inlining the functions eliminates the need to use the Vec3f class build-in functions. After that we can just transfer a float for each of the coordinates needed part (x, y, z).

Software role:

Initial integration on the board will be done through the ARM processor as a standalone hardware accelerator. The on-board processor will take care of the rendering flow as described in the flow diagram. Further optimisations can pe implemented at later stages by instantiating more accelerators to take advantage of parallel nature of the Ray Tracing algorithm, moving more time-consuming functions on the hardware or by optimizing object/triangle data structure during traversal.

Table 4.1: HLS Latency Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)	
min	max	min	max	min	max
43	123	0.430us	1.23 us	44	124

Table 4.2: HLS Utilization

BRAM	DSP	FF	LUT	URAM
16	11	11	35	0

4.2 Code preparation and building basic HW functionality

Main Plan:

The main plan for this step is just make it work. We need to establish, as close to the software golden code as possible, a performance baseline and work from there. The results of this analysis will provide valuable information about what optimisation strategy to implement, more importantly what aspects of the algorithm need modifications and at which granularity. This stage will also revile to us which plan is more suitable as a baseline accelerator.

HLS stage:

In the initial stage the function *RayTriangleIntersect* was used as it is used in the golden system. Utilizing Vec3f class to transfer data vectors as well as have the maths operations build in. This is as close to the software golden sample as you can get but it's not what can be described as hls friendly code due to the structure of the classes in C++.

Baseline synthesis with a target 10ns period of the accelerator yielded about 130 clock cycles needed for the result to be produced.

To ensure the correct functionality of the integrated code C simulation was run to verify the result and the correct operation of the transition from a function in the program to a accelerator.

Vivado stage:

Vidado is a pretty strait forward, you initialize the platform you choose with a board preset in our initial experiment *ZedBoard* and after that create the block diagram containing

all the necessary hardware IPs including out accelerator. IN future steps more than 1 hardware accelerator can be used. This is also where we use the *ZedBoard* configuration to increase the fabrics clock speed in later stages.

Vitis IDE stage:

Code was prepared for the first run as mentioned in the previous step by using the function as is. We used the same golden code as a base and appended the necessary functions to set accelerator ports as well as functions to start the accelerator. It's important to note that all object and scene data parsing had to be rewritten due to Xilinxs different way of handling file-system and file operations in general. It uses a fat based file-system and utilized specific read, write etc functions.

The run did not run successfully due to a need to transfer float values from the software *ARM* to the accelerator.

This was remedied by using typecasting and value redirection, returning the output form the accelerator to an integer and then typecasting it to a float.

We had limited success using this method because despite the data being transferred correctly this time they suffered form possible setup/hold violations as the data were transferred in inconsistent positions. Such a behaviour would have forced us to reduce the fabric frequency reducing the potential performance gain. A timing violation implies the existence of a long critical path which is something that is a good practice to avoid.

Back-up Plan:

The backup plan as it was described in the previous step was to inline the functionality of the *Vec3f class* in the code and use simple float input and outputs.

HLS stage:

The code was first developed and tried in an equivalent C++ software implementation with all the necessary changes to the function as well as calling said function to evaluate correct functionality. After that it was a simple plan to build the hardware accelerator according to the new software.

The thing that becomes immediately appended is the reduction in clock cycles.

Table 4.3: HLS Latency Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)	
min	max	min	max	min	max
32	110	0.320us	1.10 us	33	111

Table 4.4: HLS Utilization

BRAM	DSP	FF	LUT	URAM
0	12	3	12	0

This reduction in latency based on the initial synthesis that targets the same clock period shows us that this base accelerator utilized less logic to perform the same operation thus it required less pipeline stages that can be added later to increase the clock speed.

Vivado stage:

As the main plan described the accelerator was added to the block diagram in a different *Vivado* project in order to test them both separately.

Vitis IDE stage:

The modified code from the main plan was adjusted to run the new accelerator, with the increased number of ports and the float data transfer fix already applied and executed on the *ZedBoard*. The result was a correct I/O transfer and the execution of the program terminated successfully.

Final chosen plan and why:

A the end we chose to proceeded with the back-up plan to the next stages and profile it's initial performance. The use of *axi lite* is not a big barrier and the reduced utilization is a bigger advantage than the ability to transfer data from the *master axi* bus.

4.3 Base Performance and Optimisation Paths

Base accelerator performance is as shown below along with *ARM*.

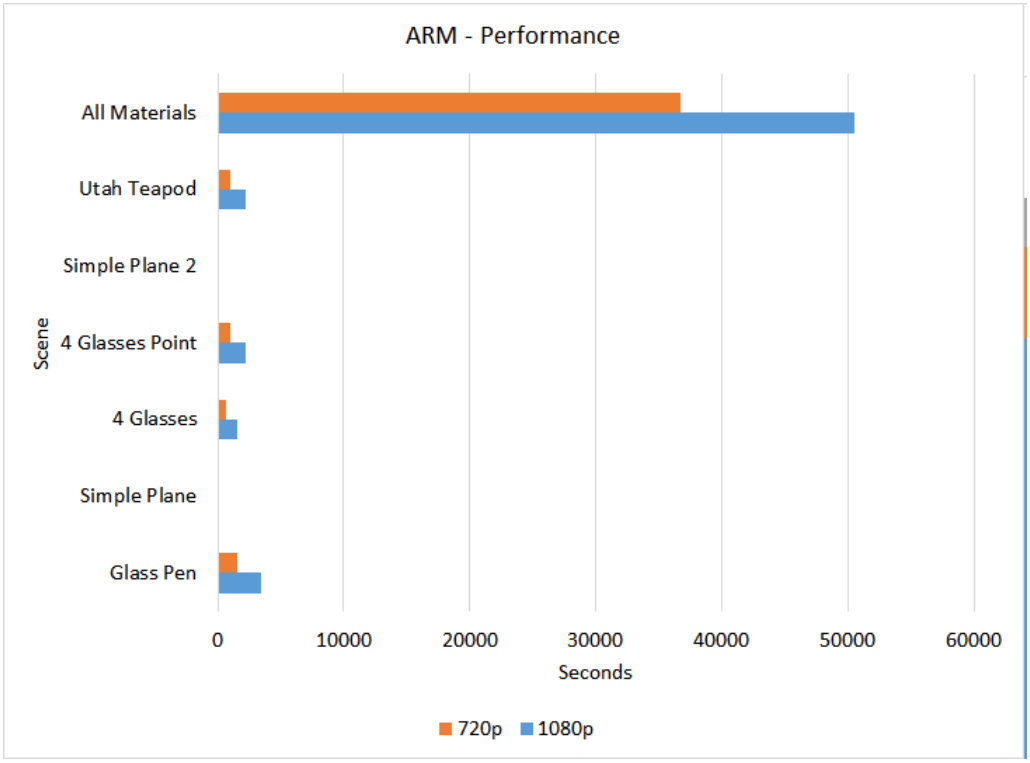

Figure 4.1: ARM Performance

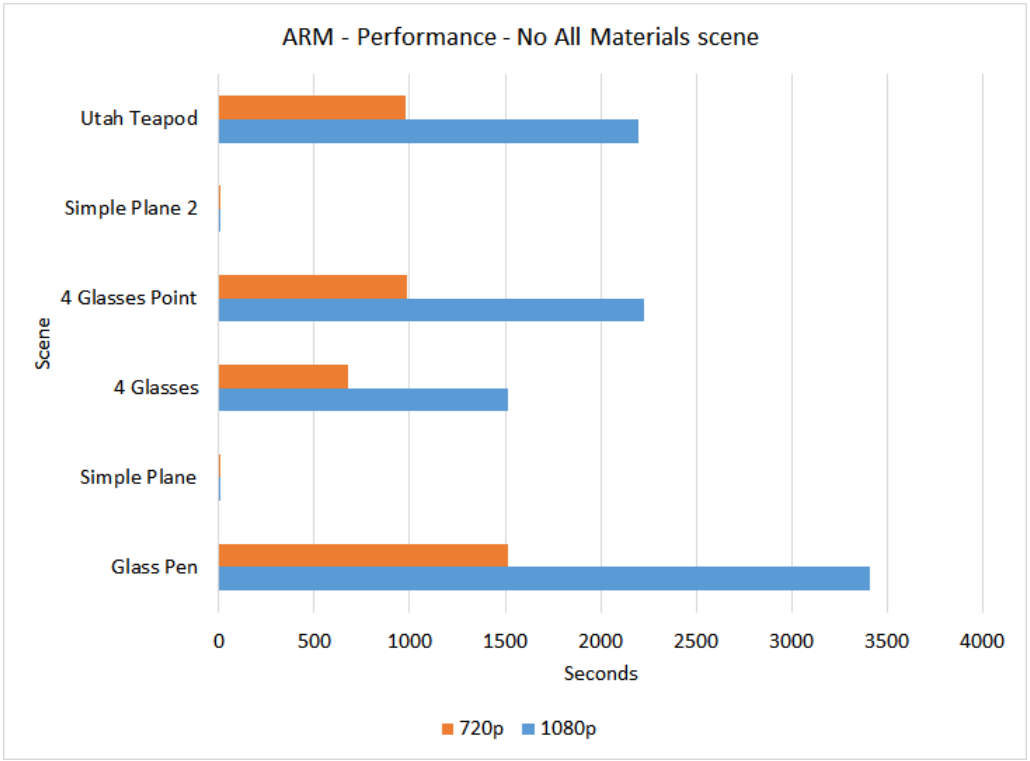

Figure 4.2: ARM Performance - No All Materials scene

Resolution	Glass Pen B.1	Simple Plane B.2	4 Glasses B.3	4 Glasses Point B.4	Simple Plane 2 B.5	Utah Teapot B.6	All Materials B.7
720p	1514.191	0.6871	675.041	989.516	0.686	977.49	36723
1080p	3406.54	1.54	1518.185	2224.497	1.5432	2200.64	50479

Table 4.5: Baseline ARM Ray Tracing Performance

Arm performance figure 4.5 shows that even with a long run-time the strategy, rendering finished within reasonable time.

Resolution	Glass Pen B.1	Simple Plane B.2	4 Glasses B.3	4 Glasses Point B.4	Simple Plane 2 B.5	Utah Teapot B.6	All Materials B.7
720p	N/A	10.71	N/A	N/A	10.69	N/A	N/A
1080p	N/A	24.0236	N/A	N/A	23.9776	N/A	N/A

Table 4.6: Baseline HW accelerator Ray Tracing Performance

This was not the case for the baseline HW performance that required more than six hours to even reach 50% on the less demanding scenes. Figure 4.6 N/A entry referees to that.

The General optimisation goal is to surpass the performance of the software implementation running on the *ARM* core of the *ZedBoard*. This can be achieved several ways and there are a lot of potential areas to explore and try make better. Starting with the most basic of design tactics in an effort to get more performance from our hardware to more in-depth techniques.

4.3.1 Fabric Frequency

The simplest and more apparent optimisation is the clock speed the accelerator is increasing either or both the target period during *HLS* synthesis or/and the frequency of the *Programmable Logic Fabric* of the FPGA. Doing the latter without doing the former is overclocking. It has beed proved to be an effective way to increase performance if one does not meet *SDC Silent Data Corruption*.

Table 4.7: Performance runs for: simple plane 1 scene in seconds and PSNR

Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
23.94	26.87	21.95	23.17	15.33	14.66	17.36
inf	inf	inf	inf	33.75	inf	inf

Table 4.8: Run HLS Utilization

Run	BRAM	DSP	FF	LUT	URAM
Run 1:	0	12	3	12	0
Run 2:	0	11	4	13	0
Run 3:	0	12	3	12	0
Run 4:	0	11	4	13	0
Run 5:	0	13	4	12	0
Run 6:	0	12	3	13	0
Run 7:	0	11	5	15	0

Targeting anything below 5ns period resulted in timing violation warning from the *HLS* so it was decided to be left at that.

The following tests experiments run in order to determine

- Run 1: Synthesis targeting 10ns(100MHz) - Fabric @ 10ns(100MHz)
- Run 2: Synthesis targeting 5ns(200MHz) - Fabric @ 10ns(100MHz)
- Run 3: Synthesis targeting 10ns(100MHz) - Fabric @ 4ns(250MHz)
- Run 4: Synthesis targeting 5ns(200MHz) - Fabric @ 4ns(250MHz)
- Run 5: Synthesis targeting 15ns(66.6MHz) - Fabric @ 4ns(250MHz)
- Run 6: Synthesis targeting 12ns(83.3MHz) - Fabric @ 4ns(250MHz)
- Run 7: Synthesis targeting 4.5ns(222.2Mhz) - Fabric @ 4.34ns(230MHz) actual (214 MHz)

Utilization figures for the above runs.

Table 4.9: Run Vivado Utilization

Run	LUT	LUTRAM	FF	DSP	BUFG	Power
Run 1:	8	1	5	13	3	1.722 W
Run 2:	8	1	6	12	3	1.753 W
Run 3:	9	1	5	13	3	1.808 W
Run 4:	8	1	6	12	3	1.879 W
Run 5:	9	1	5	14	3	1.841 W
Run 6:	9	1	4	13	3	1.813 W
Run 7:	8	1	6	12	3	1.884 W

Table 4.10: HLS Latency Summary

Run	Latency (cycles)		Latency (absolute)		Interval (cycles)	
	min	max	min	max	min	max
Run 1:	32	110	0.320 us	1.10 us	33	111
Run 2:	62	220	0.310 us	1.10 us	63	221
Run 3:	32	110	0.320 us	1.10 us	33	111
Run 4:	62	220	0.310 us	1.10 us	63	221
Run 5:	25	85	0.375 us	1.275 us	26	86
Run 6:	30	106	0.360 us	1.272 us	31	107
Run 7:	74	259	0.333 us	1.165 us	75	260

Testing the accelerator reviled that the simple plane scene took the least amount of run time and the image quality was correct. The test was run for a 720p version of the scene with no problems whatsoever.

However although there is an important performance increase that comes with overclocking. Specifically when it is combined with timing violations. This is wrong from a digital design perspective despite the fact that out accelerator managed to get the best performance in Run 6 with no image artifacts (Silent Data Corruption) whatsoever. This effectively showed us that the run is frequency based, the faster the accelerator can run the faster we can set the fabric frequency in order to achieve faster results.

With this knowledge at hand we chose a Run 7 that combines the performance of over-

clocking with no slack violations. Reducing the clock uncertainty makes the clock margin smaller that would normally be used to account for increase in delays derived from RTL logic synthesis, place, and route.

Enabling unsafe math optimisations from the solution settings helped to achieve the last run removing the slack violation in HLS with no result on the final image.

Something that also provided a performance edge was enabling *-O3* compiler optimisations for the rest of the render in *Vitis IDE*.

Conducting the experiments with synthesis and fabric periods gives us an idea for how the accelerator reacts to different target synthesis periods and passing through *vivado* flow. That way if this yields any additional performance we can incorporate the above best options and have an even greater leap in performance.

AXi Interface change:

The interface we chose is *AXI4 Lite*. This application does not depend on high data throughput but quick and immediate transfers. Thus there is no merit moving to a faster interface line *Master AXI* as the I/O required will always be the same regardless. This is also based on the analysis perspective in *Vitis HLS*, data input takes 1 cycle.

4.3.2 HLS Directives

In this stage of the project the accelerator consists of mathematical operations only with buffer/registers used for storing intermediate results. For the most part HLS optimisation directives revolve around data throughput and loops. Thus in this stage they would contribute no performance benefit as the accelerator is called frequently to calculate data and return a result. In future optimisation with an onboard buffer and multiple accelerators we can apply *HLS Directives* effectively.

4.3.3 Arbitrary/Half Precision

Floats are a general and known problem in computation problems often leading to greater delays due to the high amount of hardware cycles it requires to produce a result as well as the fabric area they occupy. Thus the apparent next step was to try and use fixed point arithmetic for the calculations. As *Xilinx* made *FP 16-Half Precision* data type deprecated we opted to try *C++ Arbitrary Precision* library as a starting point.

As calculated the minimum bits required to represent the numbers used are 12 decimal and 9 fractional bits. Tests were conducted with values bellow and above said limit.

This yielded positive results as the cycles and latency needed to produce a result were significantly reduced.The first number represents the total number of bits, the second number the decimal bits, the next two are the rounding and overflow models. Utilizing a starter type with 10 decimal bits and 6 fractional and going through the *Vivado* flow the results on the image were not promising. No intersection was detected and the render came out blank. That also removed a critical path data dependency opening up the next optimisation which is the replacement of 1/x division.

Experimenting with various sizes and *Ap Fixed* rounding and overflow options we got the same result.

- AP RND ZERO that rounds the number to the nearest representable value specifically zero, for positive values deleting the redundant bits and adding least significant bits to get he nearest representable value for negative values.
- AP SAT SYM that saturates the value in case of overflow to the maximum or in case of underflow to the negative maximum.
- 16 10 Failed.
- 16 10 AP RND ZERO AP SAT SYM Failed.
- 16 13 Failed.
- 16 13 AP RND ZERO AP SAT SYM Failed.
- 21 12 Failed.
- 25 10 Failed.
- 25 10 AP RND ZERO AP SAT SYM Failed.
- 32 20 Failed.
- 32 20 AP RND ZERO AP SAT SYM Failed.
- 32 16 Failed.
- 32 16 AP RND ZERO AP SAT SYM Failed.

- 64 32 Board over-utilisation.

There is loss of precision when converting a float to fixed point but no loss when converting from fixed point to float. The area used for the accelerator was increased generally increased with no increase in performance despite of the promising results in *HLS Analysis* perspective and general reductions in delay and cycles.

The Ray triangle intersection was not performed as it was expected. Experiments on the *X86* gold code renderer showed that input coordinates can be truncated down to only 3-4 fractional bits with minimal loss of precision of the final render. This showed us that the incorrect detection of the ray triangle intersection is caused by the fact that the main algorithm is designed to perform calculations on the unit triangle. Subsequently allocating more bits for the fractional part and having a bigger maximum number that can be represented was not the solution as the problem lies with the precision possible.

After all of that failed, despite being deprecated in the *HLS* documentation, we attempted to use half precision data type. This data type is composted by 1 sign bit, 5 exponent and 10 mantissa as opposed to 32 bit float with 1 sign, 8 exponent and 23 mantissa.

Such action did not provide the design with a massive reduction in cycles to produce a result nor did we did not achieve lower synthesis period. Although in theory we expected some gains going to *fp16* we did not observe such behaviour. Running at 17.9130 seconds this is slower than the float version

Moreover the accelerator did not achieve the correct results with the final render having visual artifacts B despite there being no timing violations whatsoever. The pattern of said artifacts are produced at the point where we cast a ray to determine if the intersected point lies in a shadow or not. The circular shape of the disturbances indicate that the distances cant be represented with 16 bits and thus a point that would normally lie in the path if the distant light source are detected to be in a shadow. Disabling *unsafe math optimisation* did nothing.

4.3.4 1/x Elimination

The algorithm requires the calculation of *1/determinant* as such this division is something that takes a lot of time. Another potential optimisation is to replace the *1/x* with a lookup table of pre-calculated values [22] using a method to scale the number between 0 and 1 or 1 and 2 reducing the execution time problem to an area one. Alas such strategies can be available to us if a data dependency on the critical path wasn't halting the computation progress.

As mentioned above the fixed point arbitrary precision and half precision data types did not yield correct results thus leaving the division dependency unaltered.

In case of a better performance gain using other methods this can be performed to provide a boost and free computational resources in the FPGA slice.

4.4 Multiple accelerators

Multiple accelerators can be used as the algorithm has very high parallelization potential. Each pixel is independent from neighbouring pixels. This combined with the fact that we have all the available data at hand we can choose at which granularity level we parallelize and what procedure of the rendering process.

As the accelerator stands we can approach this from several angles stating with top down view and render even and odd lines on two parallel running render loops thus effectively splitting the render process in two. This is a more high level strategy that lends itself to more performance gains than any other, Nvidia was using a very similar strategy in *SLI* connected GPUs with Split Frame Rendering [23]. Subsequently an even-odd split of the rendering workload will provide a performance boost to every scene it is preferable to be added at the end and exploit potential parallelization in other more fine grained areas first, such as eliminating the need to call the accelerator again and again for each interaction by splitting the load to two accelerators and equipping them with buffers. This will have the advantage of fewer data transfers needed and will give us a potential performance boost to scenes with only one object.

Object level parallelism can be added in a further stage of the project. A potential outcome would be to have two accelerators for Ray triangle intersection calculations in an object and then two or four sets in order to cover the object list. This will require storing the results and then deciding which will be the final intersection for that pixel and decide its colour afterwords. This gives us a total of eight accelerators as a potential target.

4.4.1 Buffer on accelerator

Multiple accelerators, Triangle level granularity:

At first we tested in software a buffered implementation with a single accelerator in order to test increased procedures going from ARM to accelerator. After a successful test we can

Table 4.11: Buffer implementation Vivado Utilization

LUT	LUTRAM	FF	BRAM	DSP	BUFG	Power
38	42	16	99	62	3	2.65 W

spilt the load of testing the tingles of a single object to multiple accelerators. The buffer will contain three arrays with data of two categories, static and dynamic. Static data involves data that are calculated on a frame or pixel basis they include the ray direction and origin. Dynamic data are comprised of an array with the vertex position in space and a list of the index to create each triangle. Using this tactic as mentioned in previous chapters helps save space for bigger triangle meshes.

Master axi interface will be utilised as from this point forward we will be dealing with arrays instead of scalar variables, thus using burst mode with *memcpy*. In order to apply this optimisation we used a buffer to hold all the information that would be used in the computation. As the computation takes place at the triangle level we need to store vertex position in space for each object we need to process. Keeping the storage cost down each object has only the absolute necessary vertex position needed to describe an object and using an index array. Those are stored in a format different from the format that what our hardware accelerator can realistically be able to be expected to handle. The procedure falls on the software side of the implementation to package the input data in such a way that the accelerator can access easley. This does introduce additional processing time but will greatly help us in the multiple accelerator run. For floating point vertex positions it was determined that the most complex object in our library needed 9723 vertex position floats amounting to 38 MiB of data and 12719 50 MiB going to index storage. Having that procedure take that much time it needs to be offset by utilizing multiple accelerators splitting the processing load of determining if an intersection occurred for that particular object.

These modifications had an undesired effect, the immediate and substantial increase in area consumed bu the accelerator with worst performance. Utilizing almost 99 present of the available Block RAM as seen below.

Even though the average runtime of each function intersection function was 0.0702 this added up to the already huge hardware cycles the accelerator needed to produce a result.

Before appling the multiple accelerator run the introduction of a loop in the *HLS* accelera-

Table 4.12: Loop Pipeline implementation Vivado Utilization

LUT	LUTRAM	FF	BRAM	DSP	BUFG	Power
37.42	42	16	99	62	3	2.308 W

tor unlocked several optimisation pragmas to help us increase the performance per accelerator further. These include

- Dataflow
- Pipeline
- Loop Unroll
- Loop flatten
- Array partitioning

Dataflow:

This optimisation was not able to be applied due to dependency on previous loop iteration. Various changes to improve that were made but none were able to mitigate this problem without altering the functionality. Thus this optimisation was not applied.

Loop Pipeline - rewind:

The pipeline directive will allow *Vitis HLS* to reduce the interval it takes for the next iteration of a loop by enabling concurrent execution of read, process and write tasks within the loop. Using the rewind option further attempts to further eliminate the pause occurring after one loop ends and the next begins thus creating a system with continues loop pipelining.

Array Partitioning:

Array partitioning splits the array into smaller arrays or even individual registers. Doing this we can increase the throughput of the accelerator as well as enabling more memory ports to be used. In conjuction with loop pipelining we should be able to see performance improvements albeit with a lot of area overhead. The partitioning value was selected to be 3 due to the access pattern of the main loop which has the pipeline pragma enabled.

Loop Unroll - factor N:

Loop unrolling effectively copies the loop body enabling some portions of the loop to execute in parallel. Due to the variable loop ending and the large area requirements partial

Table 4.13: Array Partitioning implementation Vivado Utilization

LUT	LUTRAM	FF	BRAM	DSP	BUFG	Power
50	56	21	99	84	3	2.774 W

loop unrolling was selected to be tried. In *Vitis HLS* if array partitioning is used pars of the loop accessing the partitioned data are unrolled. The unroll factor selection was done by estimating the appropriate area to performance balance.

Table 4.14: Loop Unroll implementation Vivado Utilization

LUT	LUTRAM	FF	BRAM	DSP	BUFG	Power
73	24	42	54	100	3	3.055 W

All the above avenues of experimentation led to very high utilisation of the FPGA fabric and even if the buffer size was reduced, DSP utilization exceeded 50% of available modules. Effectively killing our multi-accelerator strategy.

4.4.2 Multiple accelerators No Buffer:

In this last trial we attempted to use the single accelerator with the best performance and minimal area utilisation. This included using the *ARM* processor to divide the load between the available accelerators as a load balancer would.

The plan consists of fitting as many accelerators as the *ZedBoard* can handle and then using the triangle loop to activate different accelerator depending on how many triangles we need to calculate. The correct intersection result will be chosen based on the return value of the accelerator, true on a valid intersection and false on no intersected detected, and the final intersected triangle would be selected based on the distance form the origin which is one of the accelerators outputs.

This was found to be the only way to both keep the design small and simple to fit a relatively small *FPGA* fabric like the one on the *ZedBoard* and keep it power efficient.

Alas this route proved too much for the fabric of our *ZedBoard* to handle. With simple plane scene B.2 taking as much time as the single accelerator the extra overhead of setting the accelerator ports proved to be disastrous. Due to the extended usage of the ZedBoards fabric

the we couldn't run the accelerator at its rated frequency either getting us below 200MHz and into 150MHz. This is one other major factor why the multiple accelerator was slower.

accelerators	Time	LUT	LUTRAM	FF	DSP	BUFG	Power
x8	26.81 s @ 100 MHz	57	5	48	95	3	2.398 W
x4	21.46 s @ 100 MHz	29	3	25	47	3	2.031 W
x2	17.03 s @ 187 MHz	15	2	12	24	3	2.031 W
x2	24.70 s @ 100 MHz	15	2	12	24	3	1.856 W
x1	17.36 s @ 214 MHz	8	1	6	12	3	1.884 W
x1	23.94 s @ 100 MHz	8	1	5	13	3	1.722 W

Table 4.15: Multiple accelerators performance and Vivado Utilization

Only Running the x2 accelerators at the highest frequency we got some tangible results with a small improvement in performance. That being said other scenes required more than three hours to run.

Chapter 5

Conclusions

5.1 Closing Arguments and Conclusions

Although promising, the size and available hardware on the *ZedBoard's* programmable fabric didn't allow very much room to take advantage of the inherent parallelism offered for such applications. The hard number crunching problem the accelerator was called to solve didn't lend it self to very much room for optimisations.

As stated above reduced precision data types (arbitrary and half precision) were promising they did not prove to be working with the way the algorithm handles ray - triangle interaction. Using the unit triangle to reduce the calculations needed to perform the intersection increased the precision needed dramatically. As stated using buffered accelerators to avoid continued and frequent data transfers to the accelerator didn't yield the result expected and the area utilisation skyrocketed. With very poor performance per accelerator and no room to explore multiple accelerator options this strategy was abandoned.

The final results are produced running the basic benchmark scenes simple plane 1 B.2 and simple plane 2 B.5 as they are the only ones that run within reasonable time in both 1080p and 720p experiments.

Pitting the best accelerator performance achieved against ARM and X86 further proves our point that more complex and powerful fabric is needed.

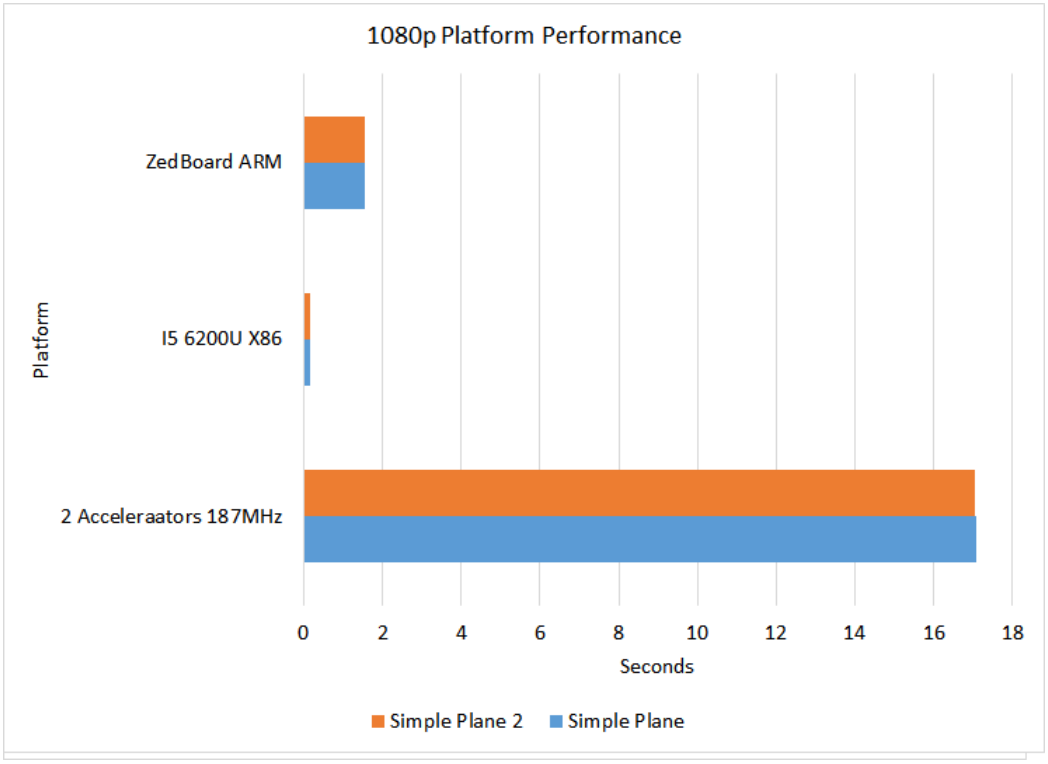

Figure 5.1: Performance Between Platforms 1080p

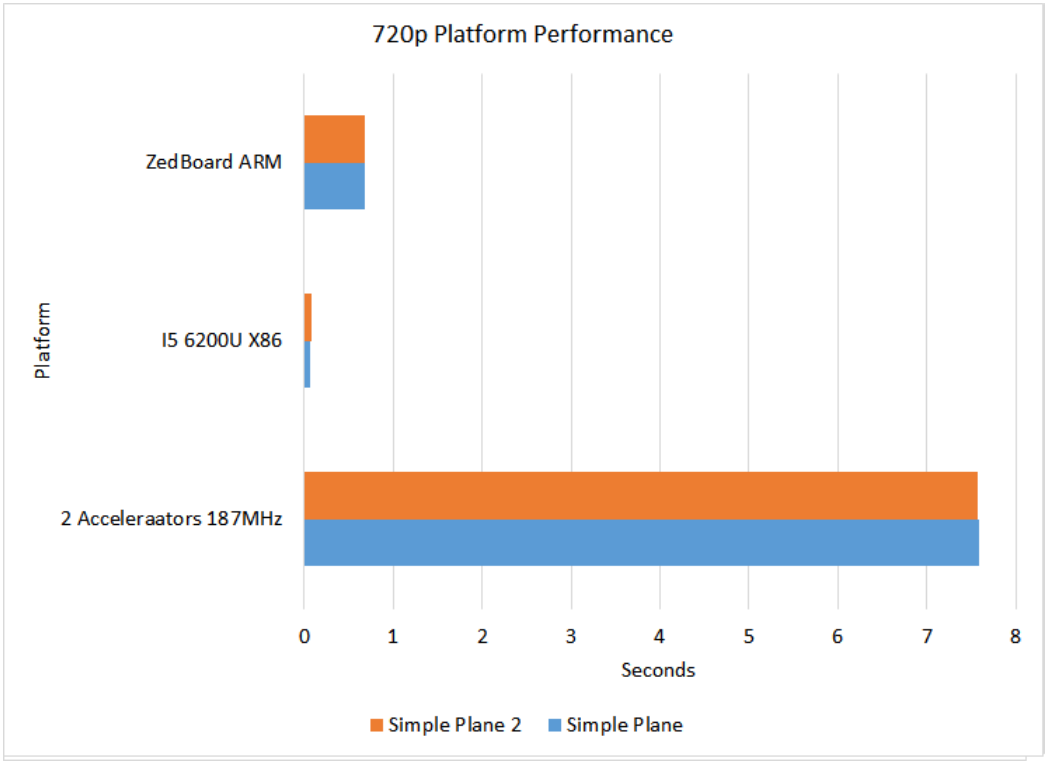

Figure 5.2: Performance Between Platforms 720p

Below are all the accelerator optimisation paths against each-other, runs that produces

artifacts were omitted.

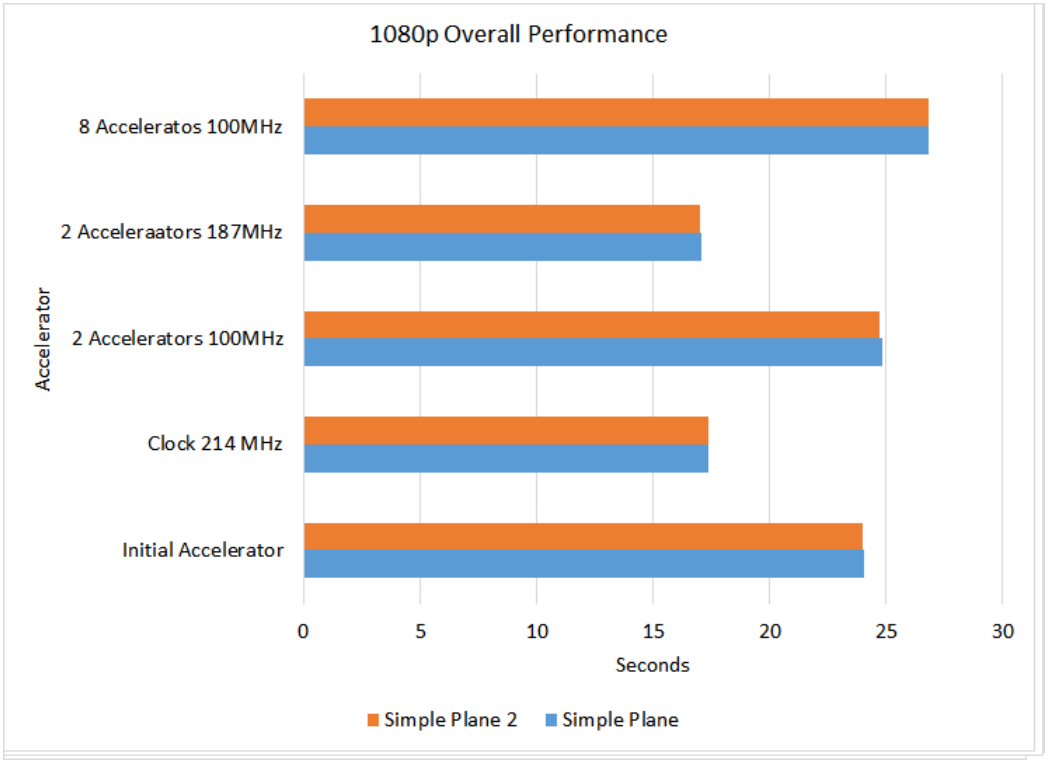

Figure 5.3: Performance Between accelerator Optimisations 1080p

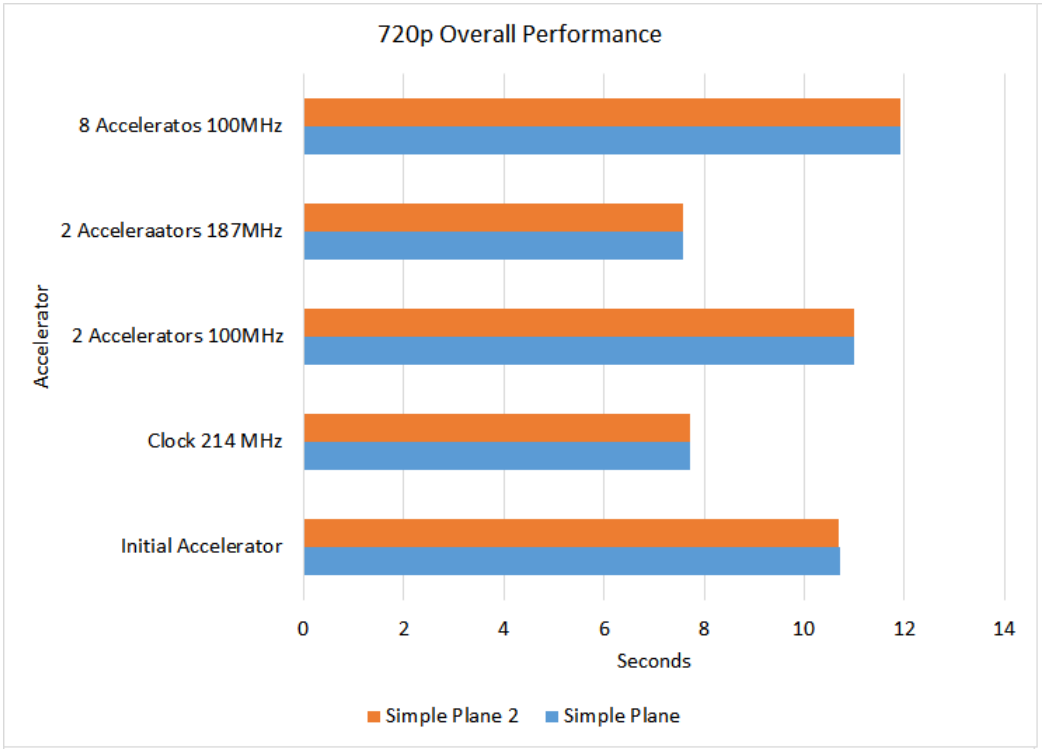

Figure 5.4: Performance Between accelerator Optimisations 720p

Although the results were not what we expected this shows us that Ray tracing algorithms need excessive modifications in order to tun on low performance and power hardware effectively and needs a lot more optimisations in order to level the playing field with x86. This shows us that FPGA needs more CPU GPU type relationship and much higher performance fabric as an extra processing part connected with a PCI Express interface.

5.2 Future Work

Future work on the project includes the usage of a smarter more efficient data structure and traversal algorithm for each pixel to minimize the number of intersections to be checked.

Designing the accelerator with Verilog HDL will be also on the list as HLS programs generally don't offer the same level of performance as a good digital designer can produce. On that note a more "hands on" approach would make sure more parts of the algorithm would receive individual attention. Therefore designed to be also accelerated using the FPGAs fabric instead of containing everything on on a single accelerator. Critical path control and a more decentralized approach would help greatly in that regard. A hybrid rendering approach like the one Nvidia uses [4] [5] would also be an option. Although an optimized pure RT implementation is generally preferable in order to utilize multiple parts of our hardware design increasing the overall system efficiency.

Certainly more powerful FPGAs can be utilized in the future with more processing power and area than *ZedBoard* and with faster connectivity to the outside world e.g. PCIe. This would unlock more powerful CPUs and access more direct access to data.

Research in this topic already has shown that this is achievable using older FPGA fabrics more modern ones will be able to interface with modern renderers and graphical APIs to provide RTI acceleration.

As a last future plan, expanding the rendering engines capabilities and supported effects such as Anti Aliasing and indirect lighting is always a welcome improvement.

Bibliography

[1] Rasterization: a practical implementation. https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation. Date Accessed: 26-08-2020.

[2] An overview of the ray-tracing rendering technique. https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview. Date Accessed: 26-08-2020.

[3] Giao Pham, Suk-Hwan Lee, and Ki-Ryong Kwon. Interpolating spline curve-based perceptual encryption for 3d printing models. *Applied Sciences*, 8:242, 02 2018.

[4] Nvidia turing gpu architecture. Nvidia Turing GPU Architecture whitepaper. Date Accessed: 20-09-2020.

[5] Nvidia ampere ga102 gpu architecture. NVIDIA Ampere GA102 GPU Architecture whitepaper. Date Accessed: 20-09-2020.

[6] J. Fender and J. Rose. A high-speed ray tracing engine built on a field-programmable system. In *Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT) (IEEE Cat. No.03EX798)*, pages 188–195, 2003.

[7] MARKUS BILLETER OLA BÅNGDAHL. Fpga assisted ray tracing, rendering large offline scenes. Master's thesis, Department of Computer Engineering Computer Graphics Research Group CHALMERS UNIVERSITY OF TECHNOLOGY, 2007.

[8] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang Paul, and Philipp Slusallek. Realtime ray tracing of dynamic scenes on an fpga chip. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware 2004, Grenoble, France, August 29-30, 2004*, pages 95–106, 01 2004.

[9] Racing the beam ray tracer. https://tomverbeure.github.io/rtl/2018/11/26/Racing-the-Beam-Ray-Tracer.html#introduction. Date Accessed: 01-04-2020.

[10] Zynq-7000 soc data sheet: Overview (ds190). https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. Date Accessed: 9-011-2020.

[11] Vitis hls coding styles. https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitis_hls_coding_styles.html#iyg1582649282811. Date Accessed: 9-011-2020.

[12] J. Liaperdos, A. Arapoyanni, and Y. Tsiatouhas. Caching architecture for flexible fpga ray tracing platform. *Journal of Parallel and Distributed Computing*, 104:61–72, Jun. 2017.

[13] J. Schmittler, I. Wald, and P. Slusallek. Saarcor – a hardware architecture for ray tracing. *Proc. of the Conference on Graphics Hardware 2002*, pages 27–36, 2002.

[14] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2, 08 2005.

[15] Ray tracing: Rendering a triangle. https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection. Date Accessed: 26-08-2020.

[16] Introduction to polygon meshes. https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-polygon-mesh. Date Accessed: 26-08-2020.

[17] Ray-tracing a polygon mesh. https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-polygon-mesh. Date Accessed: 26-08-2020.

[18] Transforming objects using matrices. https://www.scratchapixel.com/lessons/3d-basic-rendering/transforming-objects-using-matrices. Date Accessed: 30-08-2020.

[19] Introduction to shading. https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading. Date Accessed: 15-09-2020.

[20] The phong model, introduction to the concepts of shader, reflection models and brdf. https://www.scratchapixel.com/lessons/3d-basic-rendering/phong-shader-BRDF. Date Accessed: 18-09-2020.

[21] Global illumination and path tracing. https://www.scratchapixel.com/lessons/3d-basic-rendering/global-illumination-path-tracing. Date Accessed: 20-09-2020.

[22] M. J. Schulte and J. E. Stine. Symmetric bipartite tables for accurate function approximation. In *Proceedings 13th IEEE Sympsoium on Computer Arithmetic*, pages 175–183, 1997.

[23] Sli: Split frame rendering. https://docs.nvidia.com/gameworks/content/technologies/desktop/sli.htm. Date Accessed: 01-04-2020.

APPENDICES

Appendix A

Software Documentation

A.1 File Formats

A.1.1 Object Options Data file

Object Option data (.ood)

- Object to world coordinates
- Set the material type(0-Diffuse, 1-Reflection, 2-ReflectionAndRefraction, 3-Phong)
- Index of refraction (also sometimes referred to as ior)
- albedo = reflect light / incident light
- phong model diffuse weight
- phong model specular weight, control the size of matte lighting spot
- phong specular exponent, control the size of specular spot

A.1.2 Scene Options Data file

Scene Option data (.sod)

- Set resolution width
- Set resolution height
- Field of view changes how much of the scene is visible

- Set background colour when no intersection occurs
- Camera to world Set the camera to a position in the scene
- Sets shadow bias
- Sets a limit to how many rays we ”chase” to find an objects contribution
- Number of lights in the scene
- Type of light
- Select the light colour
- Select the light intensity
- Light to world coordinates

A.1.3 Geometry file

Geometry file data (.geo)

- The first number defines the number of faces making up the mesh.
- The second and third line is just a series of integers representing the face index and the vertex index arrays.
- The next line contains the vertex position data.
- The next lines contains the normal data.
- The last line contains the texture coordinates data.

Appendix B

Images

B.1 Sample Test Scenes

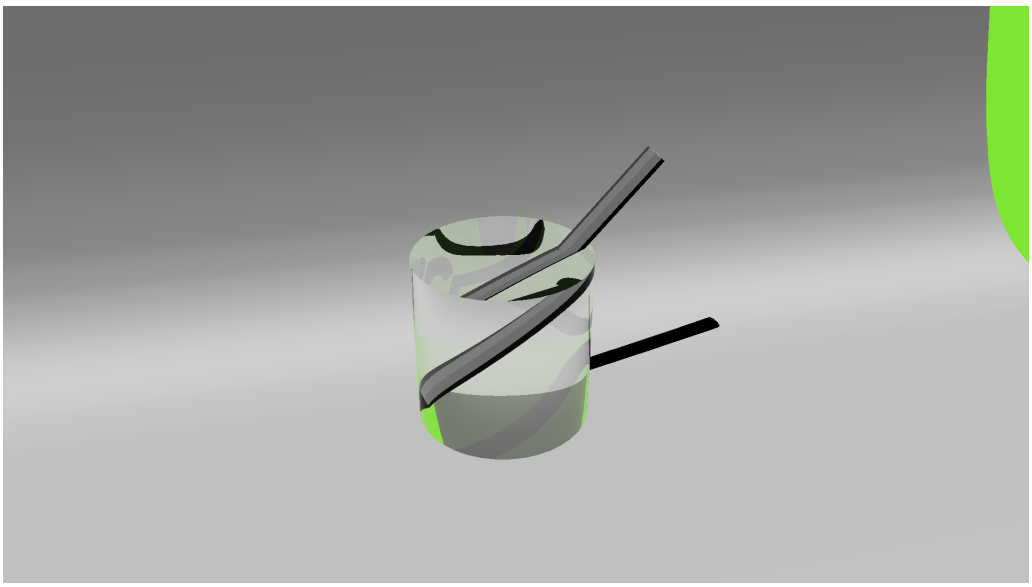

Figure B.1: Scene features a plane and a glass with reflection and refraction effect

Figure B.2: Simple plane scene consisting of only one object and two triangles

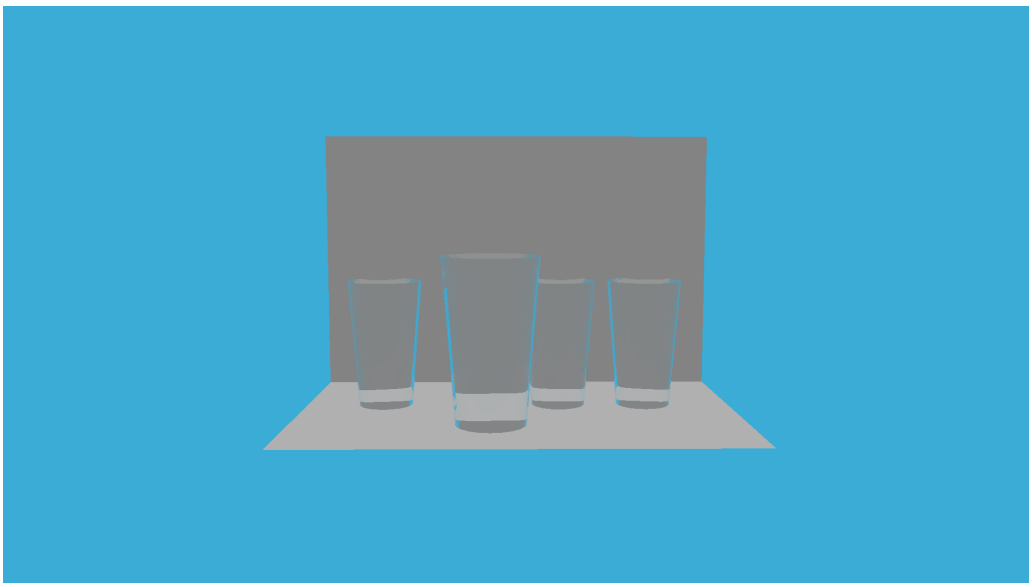

Figure B.3: Scene features a plane and 4 glasses illuminated by distant lighting

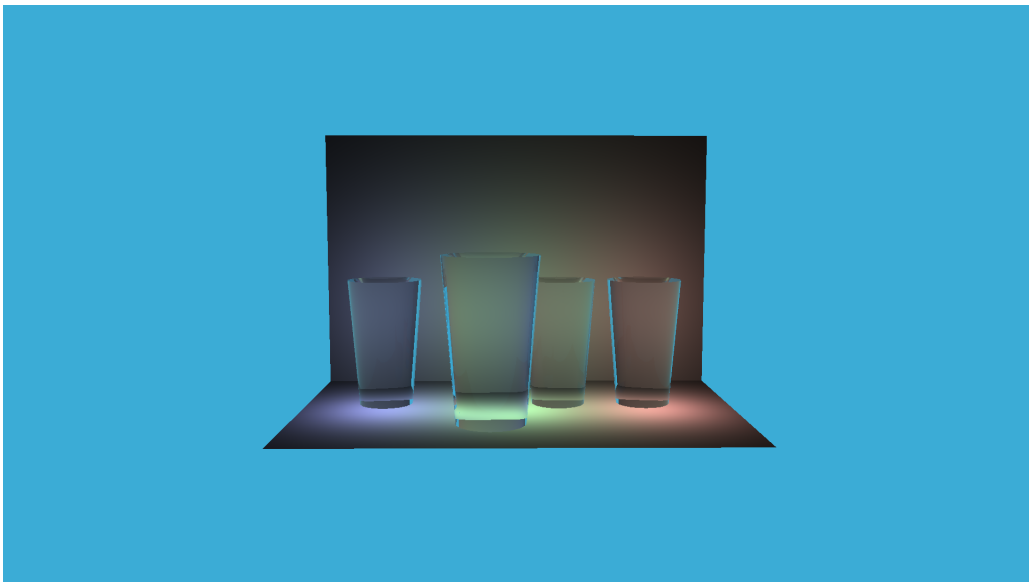

Figure B.4: Scene features a plane and 4 glasses illuminated by point lighting

Figure B.5: Simple plane scene consisting of only one object and two triangles

Figure B.6: Utah teapot famous render object in the world of computer graphics

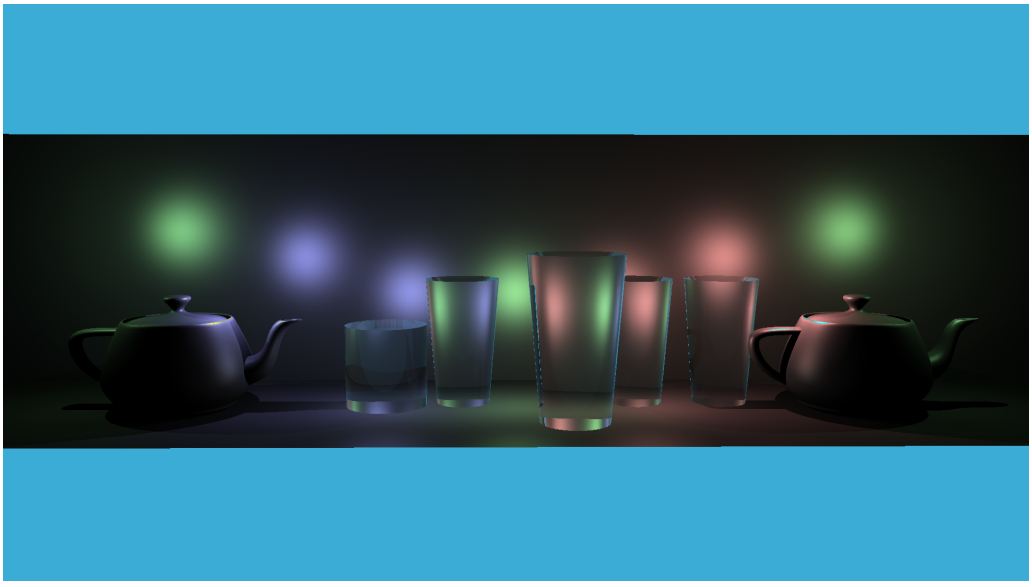

Figure B.7: A scene featuring illumination from point lights Phong and reflect refract object types

B.2 FP 16 Artifatcs

Figure B.8: FP 16 Artifacts Sample Scene Simple Plane B.2

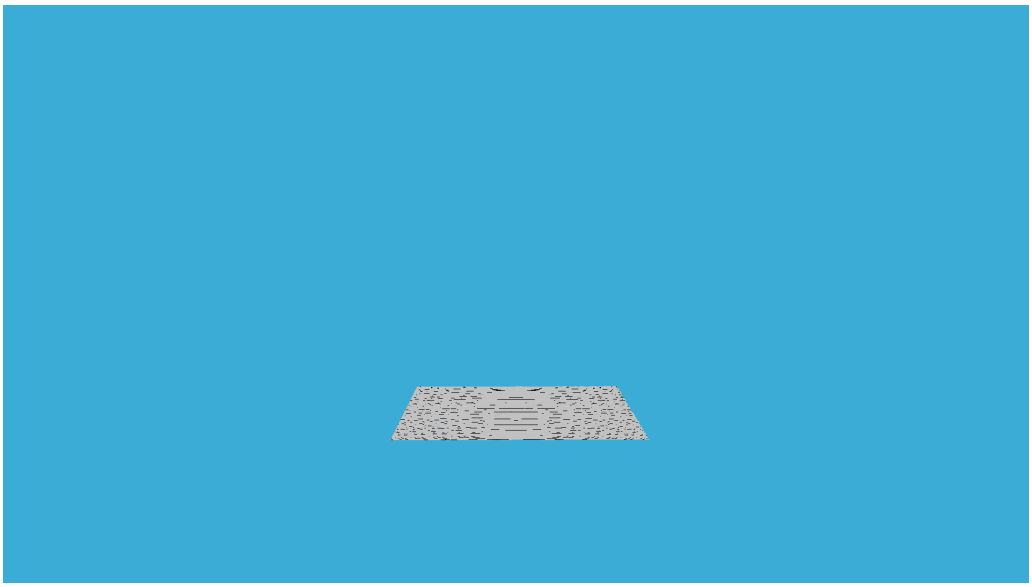

Figure B.9: FP 16 Artifacts Sample Scene Simple Plane 2 B.5