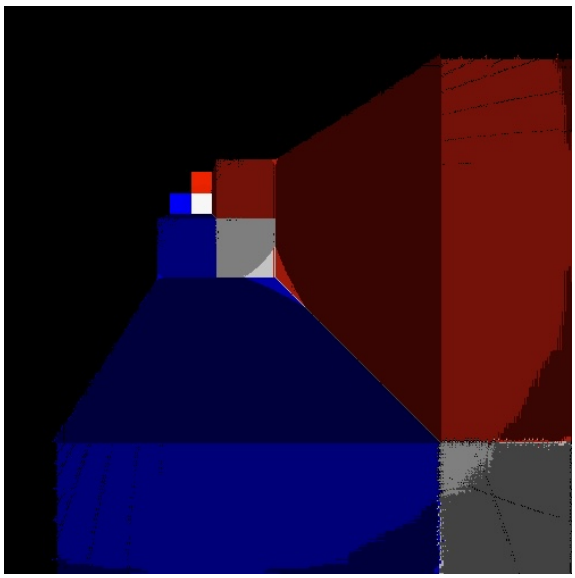
FPGA Voxel Integer Ray Tracer



ELEC50015: Electronics Design Project

Supervisor: Dr Edward Stott

Word Count: 9418

Project GitHub: https://github.com/yanniszioulis/ray-tracing

Sriyesh Bogadapati (EIE, 02207510)
Dinushan Edmond Camilus (EEE, 02256620)
Kiara Rao (EEE, 02030136)
Sne Samal (EIE, 02201807)
Meric Song (EIE, 01782878)
Yannis Zioulis (EIE, 02223077)

June 17, 2024

Abstract

This project details the implementation of a voxel-based integer ray tracer on a PYNQ-Z1 FPGA for accelerating mathematical visualisations. Ray tracing, chosen for its visual and computational complexity, is ideal for hardware acceleration due to its embarrassingly parallel nature. The design leverages integer arithmetic to enhance computational speed and accuracy, representing the environment using an octree structure for efficient memory usage and traversal. The implementation involved developing software prototypes in Python and C++, followed by hardware translation into System Verilog. Performance evaluations showed significant acceleration, with the FPGA implementation producing identical images 116 times faster than the C++ implementation. The system also includes a graphical user interface for real-time parameter adjustments and rendering feedback. Challenges such as integer normalisation and memory optimisation were addressed to ensure precise and efficient ray tracing. This project demonstrates the potential of FPGA-based acceleration for complex rendering tasks, providing insights into designing custom hardware for computation of mathematically intensive algorithms.

Contents

List of Figures

List of Tables

1 Project Introduction

The aim of this project was to implement mathematical acceleration on a PYNQ-Z1 FPGA. Examples of algorithms that could be accelerated with hardware whilst also providing a visual aspect include generating Mandelbrot or Julia set fractals, simulating Conway's game of life, and ray tracing.

Having explored multiple preliminary ideas, Voxel-based ray tracing was chosen to be implemented on hardware due to its strong visualisation aspect, algorithmic complexity and technical challenge. Since pixels in the final output image are independent of one another, ray tracing is *embarrassingly parallel* [7], making it a suitable choice for hardware acceleration for the end-of-year project.

1.1 Project Requirements [1].

The project brief provides the following system requirements that need to be satisfied in the design and implementation processes. Following the conclusion of the project, these requirements are evaluated against this project.

Mathematical function visualisation:

The system shall display a visualisation of a mathematical function that is computed in real-time.

Table 1: Mathematical visualisation requirements

#	**Requirement**
M1	The function shall be computationally intensive, such that it is not trivial to generate the visualisation at the required resolution and frame rate.
M2	The computation should be 'embarrassingly parallel', which means that multiple solutions of the problem, pixels in this case, can be computed independently with no data dependence between them.

PYNQ hardware:

The visualisation shall be generated with the supplied PYNQ-1000 SoC FPGA platform with an accelerator for the soft logic of the FPGA that computes the inner loops of the calculation

Table 2: PYNQ hardware requirements

#	**Requirement**
P1	The accelerator shall be described using Verilog or SystemVerilog.
P2	The accelerator shall provide an interface with the integrated CPU for the adjustment of parameters.
P3	The number formats and word lengths used in the accelerator should be selected to optimise the trade-off between visualisation accuracy and computational throughput.
P4	The computational throughput of the accelerated implementation shall exceed that of a CPU-only alternative programmed with C, C++ or Cython.

User Interface:

The system shall provide a user interface to enhance the function of the visualisation as an educational tool.

Table 3: User interface requirements

#	**Requirement**
U1	The user interface may be implemented using separate hardware to the visualisation computer.
U2	The user interface shall allow a user to adjust the parameters of the visualisation in an intuitive or interesting way.
U3	The user interface should provide information about the visualisation as an overlay on the image or via a separate medium.

1.2 Top-Level Project Specifications

Table 4: Top-Level Project Specifications

#	**Requirement**
1	Traversal: must be able to a voxel world space.
2	Integers: to improve computation speed, integers must be used as much as possible in calculations.
3	Inaccuracy minimisation: attempt to minimise inevitable inaccuracies due to the usage of integers.
4	Acceleration: the hardware must show significant acceleration to that of the software version.
5	Memory: make optimisations on the voxel world space to minimise storage requirements.
6	Parameters: the user must be able to specify camera parameters as well as colour parameters and receive the output in real time..
7	Colour generation: accurately recognises different colours of different objects within the world space.
8	Shading: models must implement some kind of shading to create depth effects.
9	Ray generation: the camera's viewpoint and orientation in the scene are correctly represented by the output image received.

1.3 Project Planning and Organisation

This is a group project and efficient collaboration is fundamental to the project's success. An agile methodology was adopted, including regular meetings among group members, frequent design iterations and effective time management. A Gantt chart was produced alongside a week-by-week plan, ensuring that the team met weekly goals. [A]

2 Ray Tracing Introduction

Rendering 2D images from 3D models is a task that is required in many industries such as video game development, animated film production and product design/manufacturing, and there are several different ways of rendering such images. The fastest and most commonly used way is a method called rasterisation, shown in Figure 1, which involves projecting vertices onto a 2D plane and filling in the resulting pixels with the corresponding object colours.

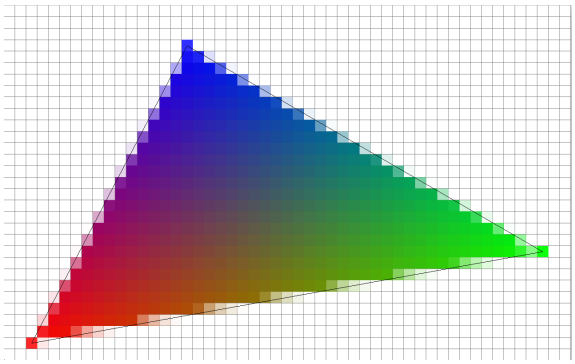


Figure 1: Rasterisation [2]

The project aims to accelerate a slower, more complex yet more accurate rendering method - ray tracing. Ray tracing is able to create images that better mimic the world than rasterisation because it is designed to represent the way light propagates through the world. It efficiently does this as it works to simulate the way light rays propagate through the world and end up on camera sensors in the event of taking a picture, only in reverse. That means that instead of simulating an immeasurable number of rays to find out how the entire 3D model is lit up, it works backwards, "shooting" a ray out of a "camera" for every pixel of the rendered image, as shown in Figure 2. The rays are propagated through the world space until an object is hit, and the corresponding pixel takes the colour of that object.

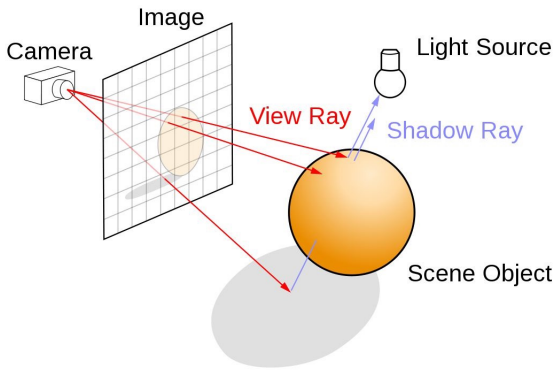


Figure 2: Ray tracing [3]

Advanced versions of ray tracing may also include light sources, meaning that rays can be simulated to traverse the world space until they hit some object, and then that ray can continue propagating towards the light source to calculate brightness - if there is an object between the light source and the point of impact of a ray, then the effect of a shadow can be simulated (something that cannot be done by rasterisation). Another feature of advanced ray tracing is the effect of shading which can be influenced by a number of factors such as object texture, and object reflectivity.

The ray tracer can take in an octree data structure as a representation of the environment to be

rendered. Rays are projected from a camera returning the colour of a material it hit, if it did hit something. Due to the time constraints of the project, shading was implemented, but reflections, shadows and textures were excluded, however, theoretical potential implementations are discussed in section 6.6.

2.1 Environment Representation

The world will be represented using an integer coordinate space, allowing all vector calculations to be simpler on the FPGA. Each coordinate will either be a 1x1x1 cube of empty space, with material ID 0 or said to contain material, with a corresponding material ID.

Users can build a 3D model using Unity to help visualise the model they are trying to render interactively. This model is then broken up into voxels which are then stored in memory, each voxel checks if it is overlapping with a `GameObject`'s collider, if so, it will store the material ID of that object, if overlapping nothing, it will store material ID 0.

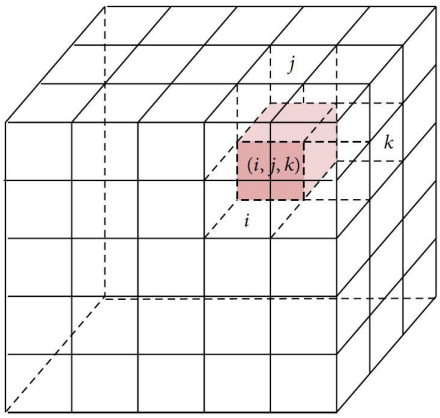


Figure 3: Voxel world space [4]

A naive approach would be to store this whole voxel world space in memory. This leads to a consistent memory requirement, however there could be large amounts of redundant information. This also makes world traversal slow as it would require memory access for every single integer step. If the memory requirement of the scene can be optimised, the time complexity will also be optimised.

In order to optimise the memory requirements of the object, two common methods considered were Bounding Volume Hierarchies [8] and Octrees. The latter was chosen as it is based on position access as opposed to geometric access. This is important as individual objects cannot be stored on the FPGA along with all their relevant information as unlike C++, SystemVerilog does not support object-oriented programming. If BVH were to be implemented, each node would have to store its own position data and children pointers, and every child node would have to iterate through and their position data would have to be checked. Due to this, octrees were selected. Any further optimised methods such as sparse voxel octrees also had to be rejected as the fixed size property of the bounding boxes in octrees to limit the unique handling of each bounding box; allowing a minimised number of states, and the information each node of the tree must store. By doing this, a more consistent time for the traversal of each bounding box is achieved.

2.1.1 Construction of the octree

In Unity, a C# script was written which takes in `GameObjects` and returns an octree data structure. This is done by:

- First, casting an axis-aligned bounding box (AABB), encompassing all the objects, rounding the side length up to the nearest power of 2 as due to being represented by bits, the coordinate space will span a power of 2.
- Then, if there are multiple materials; keeping in mind empty space is also considered a material; within this AABB, the cube is split into 8 smaller cubes as children of the root AABB. These 8 smaller bounding boxes will have a side length half of their parent.
- These AABBs are then inspected again, and further divisions are made if necessary until the leaf node is either fully encompassed by a single material or the AABB has reached unit side length; at which point, the first object in the array of `GameObjects` is taken to encompass the unit cube.

If a `GameObject` has not been assigned a material by the user, the script will also flag the node to indicate it has a non-null material, which is currently 0. By doing this, a material ID can then be assigned at a later parsing stage as a default material. This step is purely just for user-friendliness.

By using octrees, large areas of matching materials can be represented by a single larger bounding box. Where further precision is needed, traversal of the tree must be continued until a leaf node is found. This method is also very scalable, as by increasing the coordinate bit depth, more precise objects can be represented.

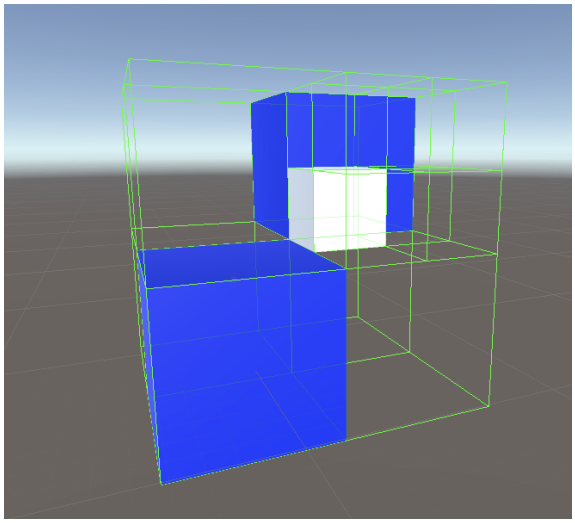


Figure 4: Visualisation of octants in Unity

For objects which are not aligned with the axis or have sizes which do not fully fill an octant, when the bounding boxes are cast since the smallest size is 1 unit, any parts of objects which are partially encapsulated in a box will be taken as fully encapsulating the unit voxel. As you increase the coordinate bit depth, the effect of this approximation will become more and more unnoticeable. To optimise the computing needed to construct the octree as well as the memory size of the octree, the colliders of each object will be rounded up to the nearest power of 2 to prevent it being broken up into singular voxels which result in the same final image.

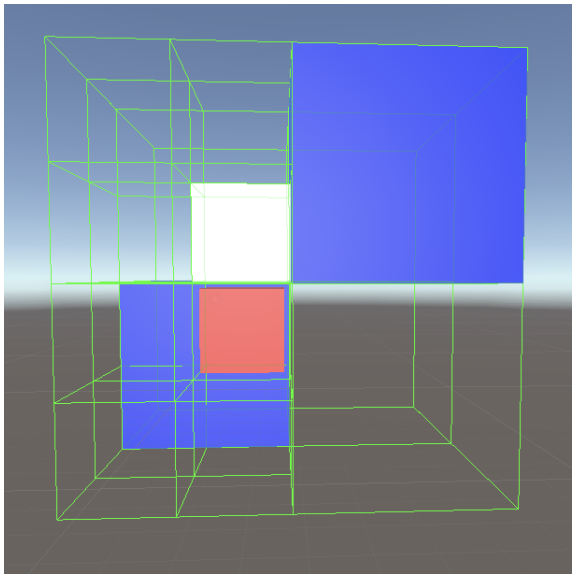


Figure 5: Visualisation of octant rounding in Unity

In Figure 5, the red cube has a side length of 3.5 units, and the white has 4 units. However, the final image will be the same as an integer coordinate space is being used; the box collider side length of the red cube is rounded up to 4 and thus contained in a single bounding box rather than across multiple unit cubes.

To store this information in memory, pointers will be used for nodes which have children. To get closer to reaching this step, an intermediary parser has been created in C# which traverses the octree and prints the information of each node into a text file, ready to be translated to a memory file.

2.1.2 Position access

In order to find the position within the octree using coordinate data, the octree is traversed top-down by checking the most significant bit of each coordinate to find the octant. Once this is done, the MIN and MAX positions of the current AABB and its size are updated. This process is repeated, with further bits if necessary, until a leaf node is found. Below is a 2D visualisation of this process, using quadrants for clarity instead of octants:

Coordinate bit depth

Using a 3 bit coordinate space in a 2D world gives an 8x8 world. This will affect how large the world is as well as how precise the voxels will be.

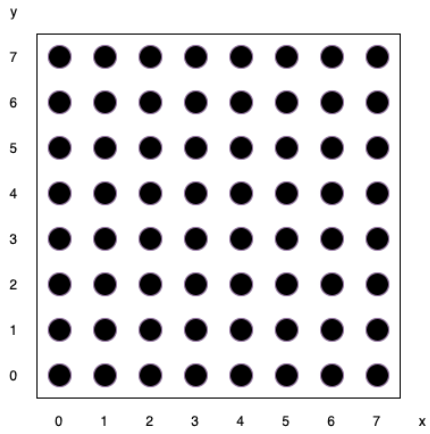


Figure 6: 3 Bit Coordinate Space

Objects in the world

2 squares of side length 2 are placed in this world.

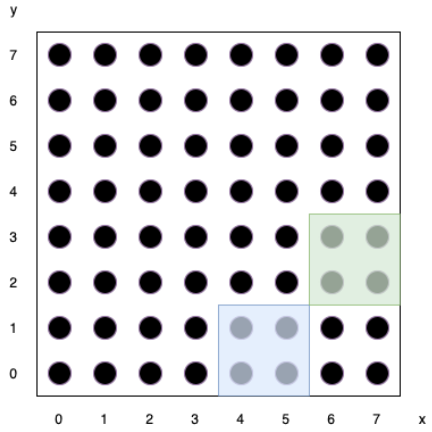


Figure 7: Placing Squares

Dividing the World

The world is then divided into quadrants, until each quadrant is encompassed by a single material. In this example, only an initial division and a further division in one quadrant are required.

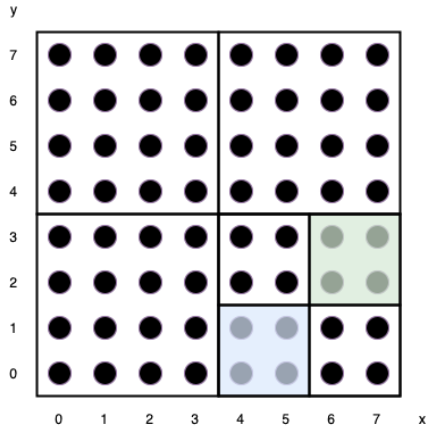


Figure 8: Quadtree construction

Ray Step

Consider a ray step in from an arbitrary position, it's new position shown by the red. This new position is stored.

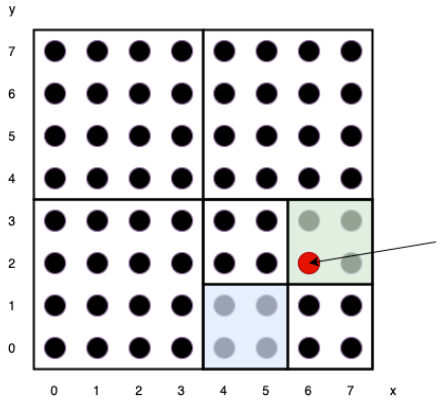


Figure 9: Ray enters position

Quadrant Selection

To find which quadrant to choose in 2D, the most significant bit of each position coordinate is considered. The MSBs can be read backwards to form a number in binary to choose the octant. The min and max of the current bounding box are now updated corresponding to:

$$\text{min\_x} + \text{MSB\_X} \times \text{half\_box\_size}$$

Similarly for y. The max coordinates are always calculated as an offset of half the parent box size from the min. After reading the memory location this corresponds to, once it is determined that this is not a leaf node,the depth must be increased, and the next most significant bit is checked.

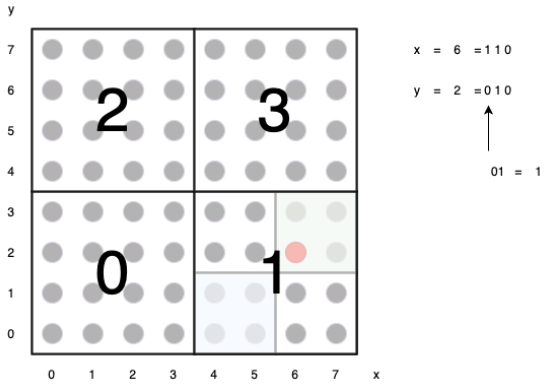


Figure 10: Quadrant selection from depth 0

Quadrant Selection

Now, by checking the next most significant bits of the position, it is inferred that the ray is within the third quadrant of this new space which, when checked, is a leaf node.

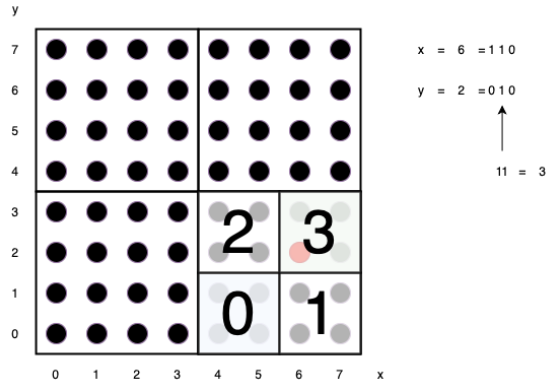


Figure 11: Quadrant selection from depth 1

2.2 Camera

To visualise the image, a camera is placed within the world and visualises the image through the collection of pixels on a plane to form a screen. The camera's normal vectors are defined; `up`, `right` and `lookAt` as parameters.

The `right` vector is defined to be the offset from the `lookAt` vector needed to move along the screen's x-axis to move by one pixel. Similarly, the `up` vector is defined as the offset to be subtracted to move down the y-axis of the screen by one pixel.

By doing this, the magnitude of the `lookAt` vector has the effect of controlling the field of view (FOV).

The camera is defined to point to the middle of the screen. The horizontal and vertical distances

from the centre of the image plane are then determined. The direction vector is determined through a weighted sum of the normalisation vectors with the pixel coordinates:

$$\text{Pixel Centre X} = C_x$$
$$\text{Pixel Centre Y} = C_y$$
$$C_x = \text{pixel no } \% \text{ image width} - \frac{\text{image width}}{2}$$
$$C_y = \frac{\text{pixel no}}{\text{image width}} - \frac{\text{image width}}{2}$$
$$\text{Direction Vector} = \text{Right } \cdot C_x \ + \text{Up } \cdot C_y \ + \text{LookAt}$$

Where pixel no is the current pixel, pixel no 0 is the pixel in the top left and pixel number (image width * image height) - 1 is the bottom right pixel.

2.3 World traversal

After generating a ray, its initial position is checked. Using this, information about its initial bounding box can be inferred as highlighted in section 2.1.2. In the case that the camera is in empty space, the ray must be stepped out of the bounding box.

This process starts by scaling the direction vector to be at least bigger than the side length of the bounding box. By doing this, a good initial guess for the first step is found. This initial guess is then added to the position vector, storing it temporarily, only updating the position if the ray is still within the same bounding box or if it has entered a position just outside the bounding box. This is done so that the ray does not overstep, skipping any boxes.

In this case, if the ray is not just outside the box, the direction vector is bisected, repeating this process of stepping, checking and bisecting until the ray has just emerged from the current bounds. Since the side length is set as the initial value, the ray is guaranteed to exit the bounding box over a finite number of steps as the total distance travelled will be greater than the longest straight line within the box, highlighted in Figure 12 below.

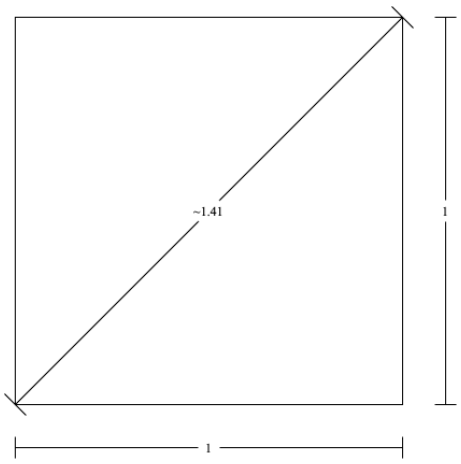


Figure 12: 2D representation

Here, a unit 2D square is considered. It is guaranteed that over a large number of steps, the ray will always step outside as the largest path is along the diagonal, which is shorter than just the first two steps. Extending this example to 3D, the longest path is now 1.732, which is exceeded by the first 3 steps. This also holds with boxes larger than the unit cube.

$$\sum_{n=0}^{\infty} \left(\frac{1}{2}\right)^n = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots$$

$$S = \frac{1}{1 - \frac{1}{2}} = \frac{1}{\frac{1}{2}} = 2$$

However, since the position vector is only conditionally updated, and integer vectors are used, it is no longer guaranteed to converge. To compensate for this, it is ensured that the direction vector can never fall to zero, it will always step further.

Once the ray exits the bounding box by one unit, it has entered an adjacent bounding box or exited the world space. At this point, the process is repeated until a node containing a non-zero material is found and a colour is returned, or the ray has exited the world space, and a default background colour is returned.

2.4 Colour Generation and Shading

Once a material ID in an octant is found, the colour of the cube is determined using the material ID look-up table. The material's colour could also simply be set as the output colour to implement a ray tracer with no shading.

However, if the camera position is also assumed to be the position of the light source as if it were the flash on a camera, the need for casting shadows can be disregarded, as objects under a shadow will not be seen by the camera at the light position anyway.

In the ray tracer, only matte objects are implemented, which are assumed to be perfect Lambertian reflectors [9].

Using this, shading can be implemented. To do so, the colour of the object must be scaled based on the reflected radiance. Since the camera position is also the light source position, a vector for the light direction is defined as the negative of the ray direction vector. The dot product of the normalised light direction vector and the surface normal can be taken, to approximate the radiance without the need to calculate the incident angle.

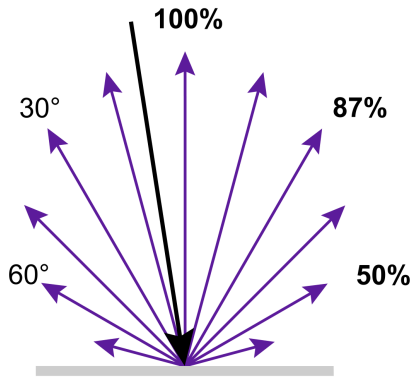


Figure 13: Lambertian Radiance [5]

Due to how world traversal was designed, the ray is known to be in the outermost integer position of the cube. Using this, at least one of the position components must match with either the AABB min or max position components as it must lie within one of these 6 planes. Further, a case statement to attain the surface normal can be defined.

After attaining the brightness factor, it is used to scale the RGB values and attain the new output colour.

$$\text{brightness\_factor} = \overline{\text{light\_dir}} \cdot \overline{\text{surface\_norm}}$$

2.4.1 Limitations

By attaining the surface normal through the position vector alone, with no regard for the direction vector, potential inaccuracies in the render along the edges and corners of each voxel may occur.

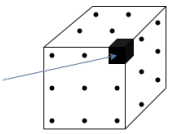


Figure 14: Ray intersects from the front face

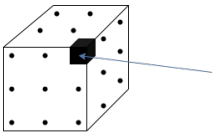


Figure 15: Ray intersects from the side face

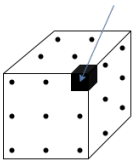


Figure 16: Ray intersects from the top face

As seen in figures 14, 15, and 16, to accurately determine the correct surface normal, the direction vector of the ray must be taken into account as well. However, this was neglected to prioritise ease of computation. The effects of doing this can be seen below in Figure 17 and 18.

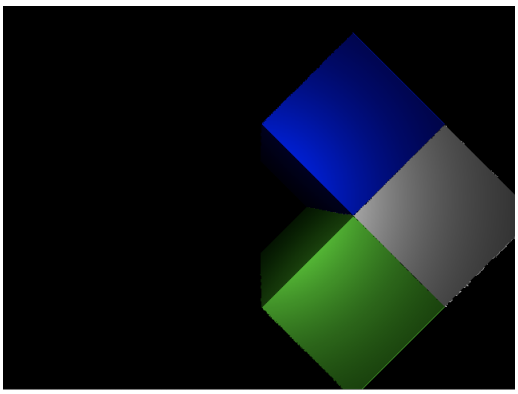


Figure 17: Outer edge along green and white cubes appears brighter than expected as the wrong surface normal is used.

Figure 18: Close up

3 Software Implementation (Python and C++)

3.1 Python Ray Tracer

To model the FPGA implementation and to compare the effects of hardware acceleration, a version in Python was made which mirrors the features and functions to be used.

The ray tracer will:

- Use an integer coordinate space to mitigate the requirement of floating point calculations in hardware
- Represent the world space using octrees, scaled by the coordinate bit depth
- Make all information dynamically accessible and calculations sequential so that they can be translated into memory hardware and SystemVerilog

The Python code is written following the structure highlighted in section 2, the code can be found in the appendix in section B.1.

3.1.1 Design analysis

This version takes in camera position vectors, camera direction vectors and image dimensions as parameters. Also modifiable at the top of the script by the developer for testing purposes is the octree, specified as a nested listed, the material ID mapping table and the coordinate bit depth. With these, image generation can be simulated to analyse the effects of different approximations to see if they are accurate enough. This would help influence decisions made in the design.

Using the Python script ray tracer, the effects of direction vector normalisation was immediately able to be seen after ray generation.

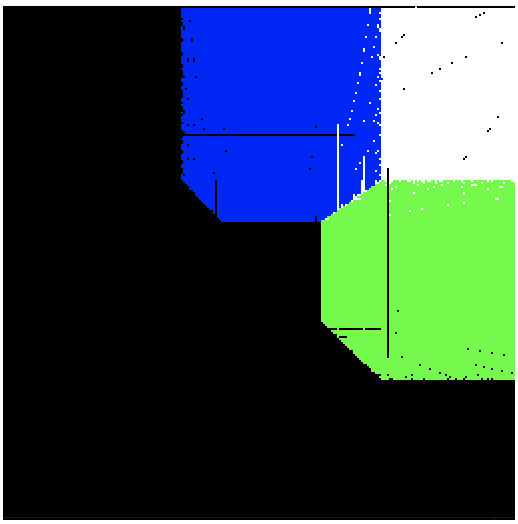


Figure 19: Image generated in Python without direction vector normalisation

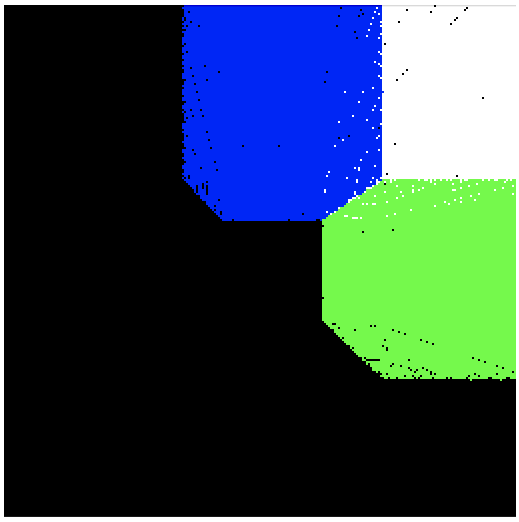


Figure 20: Image generated by Python Ray Tracer with direction vector normalisation

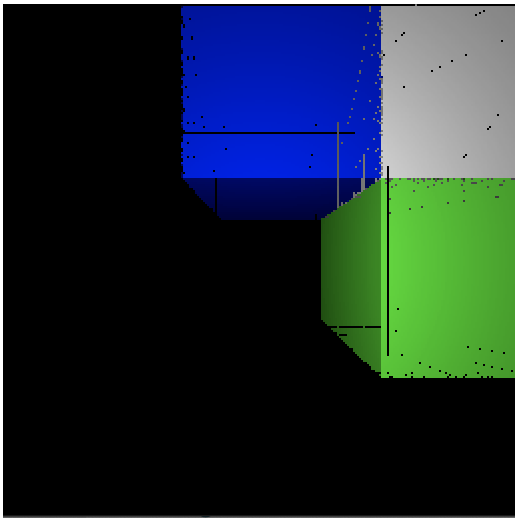


Figure 21: Image generated in Python without direction vector normalisation and shading

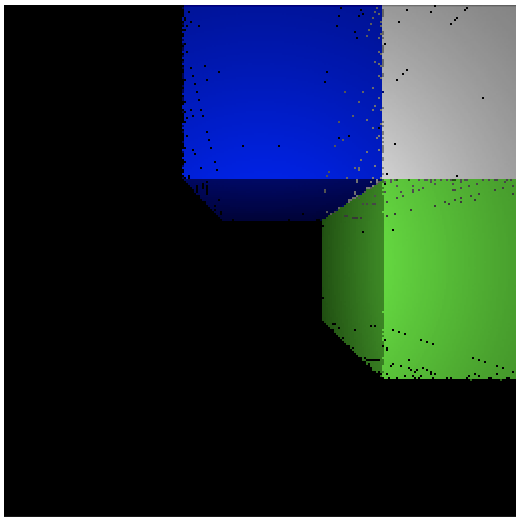


Figure 22: Image generated by Python Ray Tracer with direction vector normalisation and shading

By comparing the outputs, it is clear that the normalisation produces lines along steep edges of surfaces compared to the camera. This is due to the rounding of the position vector to integers leading to potential misses along certain planes.

Despite the significant difference in the generated images, it was decided not to normalise as the expensive cost of computing does not validate the benefits received.

3.1.2 Gamma correction

Another observation made after implementing shading was the generated image looked slightly off. This was due to the fact that the processing of the RGB values was done linearly, scaling by components of the light vector parallel to the surface normal, resulting in a linear colour distribution. However, the perception of colour is non-linear. What is perceived as a mid-grey in greyscale, is much closer to zero[10]. A common solution to this is to convert the colours into gamma space [11].

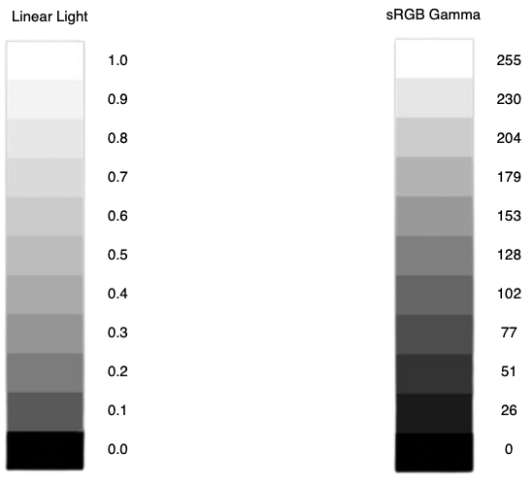


Figure 23: Linear colour scale and Gamma colour scale comparison [6]

A nominal value is to use 2.2 and raise normalised RGB values to this value. Instead, squaring the brightness factor (Figure 25) to create a non-linear distribution was sufficient.

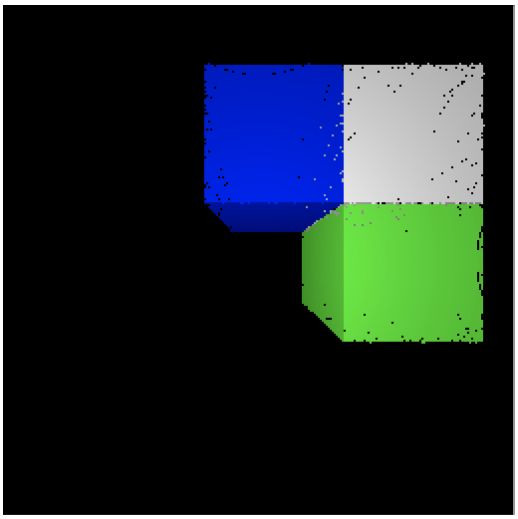


Figure 24: Initial render in linear colour space

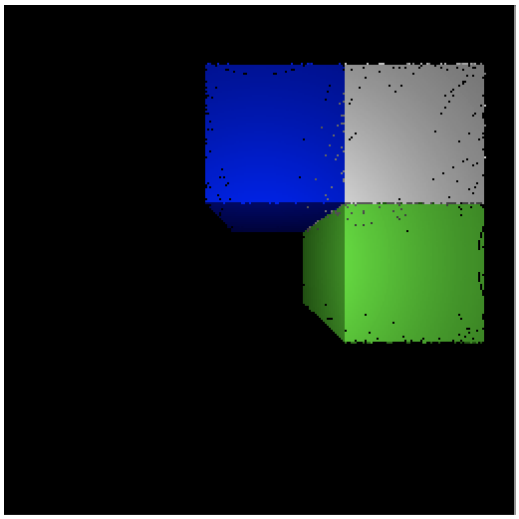


Figure 25: Gamma colour space approximation

3.2 C++ Ray Tracer

Due to ease of use, the first iteration was written in Python, however, to more accurately compare and model the final version, the ray tracer was written in C++. This version is used as a benchmark to measure the time taken to render, using the scene as a control and the number of cores/C++ as the independent variable.

The C++ version contains a more accurate memory mapping and traversal, which can be seen in the appendix. This version also does not normalise direction vectors. Normalisation is only required on the light vector for shading.

The code can be found in the appendix in section B.2.

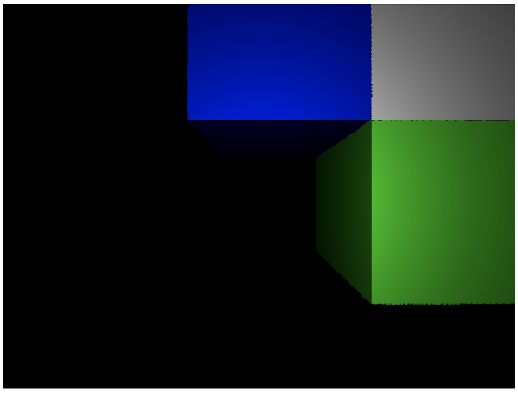


Figure 26: C++ render with shading

Figure 27: C++ render without shading

4 Hardware Implementation (System Verilog)

The single core is made up of two parts, surrounded by a top-level component (Ray Tracing Unit (RTU)):

- Ray generator: responsible for generating direction vectors for each ray corresponding to each pixel
- Ray processor: responsible for stepping each ray through the scene, traversing the octree to check for material hits and returning material colours or returning the background colour if there are no hittable objects in a given ray's path.

4.1 Ray Generator

The ray generator takes as input the image dimensions, coordinates for the camera position, the direction in which the camera is pointing and vectors for the right and up directions, these are parameters defined by the user and passed into the Programmable Logic (PL) from the Processing System (PS) via Memory-Mapped I/O (MMIO). As the image is scanned from pixel (0,0) to the bottom right, the up and right vectors specify how much to move the ray by, therefore, its magnitude represents the camera's focal length as well. The direction vectors are outputted to the ray processor. A new vector is generated only when the ray processor is ready to process a new one.

The Finite State Machine (FSM) used in the ray generator can be seen in Figure 28 and an explanation of each state is given in table 5.

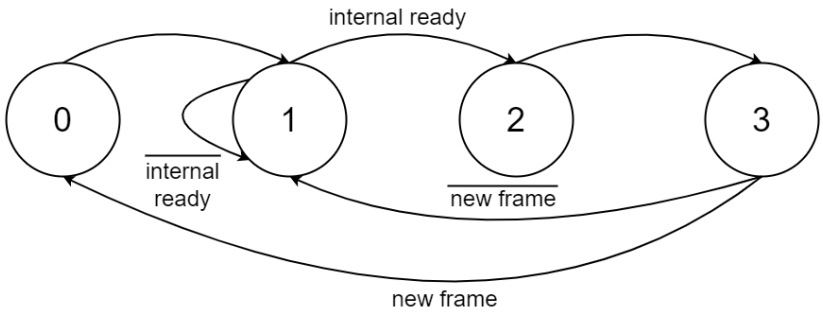


Figure 28: Ray generator state diagram

To ensure that potential failure modes are systematically identified and mitigated, a comprehensive Failure Modes and Effects Analysis (FMEA) on the subsystem design was conducted, this can be found in section D of the appendix.

Table 5: RayGenerator.sv State Descriptions

#	State	Description
0	IDLE	A reset state when a new frame is generated.
1	STALL	The ray generator is in this state until the ray processor is ready to receive a new direction vector for a new ray
2	GENERATE RAYS	Direction vectors are generated from the loop index, camera direction, up and right vectors.
3	UPDATE LOOP	Loop index is updated to be used for the next direction vector.

4.2 Ray Processor

The ray processor takes in the direction vector of a ray and based on the information it has on the scene, from the octree, it returns the appropriate RGB values. The processor follows a similar method of stepping the ray and traversing the scene to that of the software implementation. The ray starts from the camera and then is stepped through the current octant it is in, checking whether it hits something. If not, then the ray is stepped and the process is repeated until the ray hits something or the ray has left the environment space. If the ray hits a material, a brightness factor is applied to the RGB value to produce a more realistic shade to the surface the ray hits. A detailed state diagram and description are shown in Figure 29 and table 6 respectively show how this is achieved. To reduce complexity, the state diagram has been colour-coded to separate different stages of the FSM.

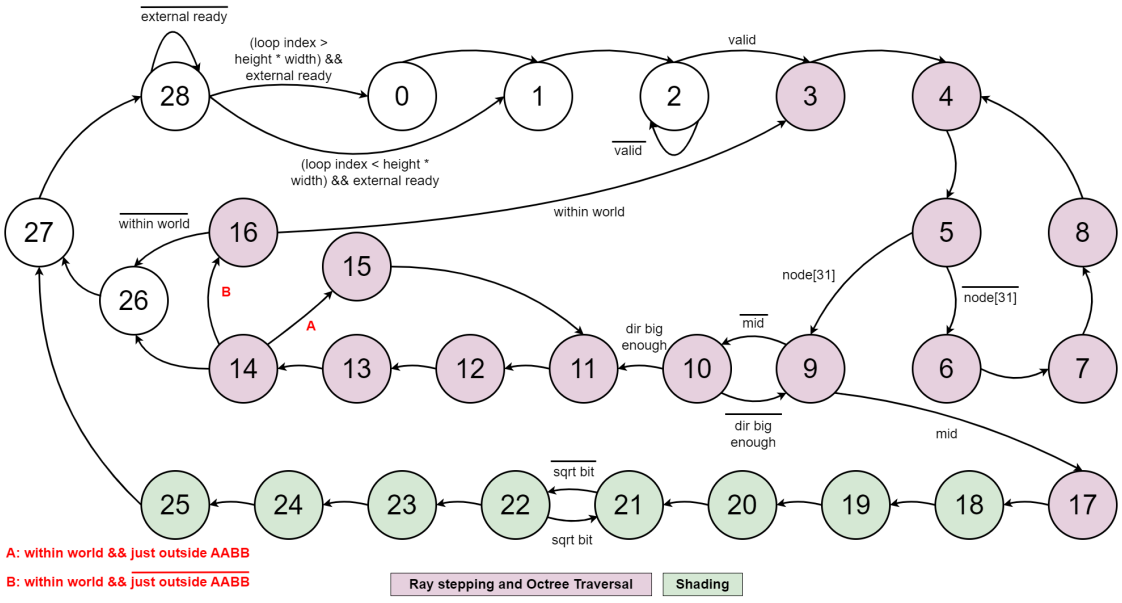


Figure 29: Ray processor state diagram

Table 6: RayProcessor.sv State Descriptions

#	State	Description
0	NEW FRAME	Reset world parameters for the processor
1	INITIALISE	Reset previous signals used in the processing of the preceding ray
2	IDLE	Validate the incoming direction vector from the ray generator, store it in registers for calculations and remember the original input vector.
3	RAY TRAVERSE INITIALISE	Reset the direction vector to the original input direction vector to step through a new octant in fewer steps.
4	RAY TRAVERSE OCTANT NO	Determine an index to work out which octant number the ray is in.
5	CHECK STATE	Stall state.
6	FIND OCTANT	Work out the octant number, increment the depth of the ray in the octree and adjust the octant size.
7	UPDATE MIN	Adjust minimum coordinates of the AABB the ray is currently in and determine the address of the sub-tree.
8	UPDATE MAX	Adjust maximum coordinates of the AABB the ray is currently in.
9	RAY PREP STEP	Determine a material ID for the ray's current position, and determine quantities needed to adjust the ray's direction vector if necessary.
10	RAY STEP ADJUST DIR VEC	If the magnitude of the ray's direction vector is smaller than the current octant's size, then double each component of the direction vector and use the new values in calculations. If it is big enough, then continue.
11	RAY STEP TEMP	Update the temporary position of the ray by adding the ray position to the current direction vector.
12	RAY STEP CHECK POSITION	Check whether the ray is within the world, within an AABB or just outside one (i.e. its coordinate is within one away from the min or max values)
13	RAY STEP CHECK AABB	Simplification state, use logic in state 12 and assign it to new variables for clarity and avoidance of repeated logic.
14	STALL	Stall state.
15	RAY STEP INSIDE	Adjust the actual position of the ray with the temporary position of the ray if the ray can travel to another octant. Reduce rounding errors to the ray's position.
16	RAY STEP OUTSIDE	Step outside the octant and reset any relevant signals to prepare for calculations for the ray's position in the new octant.

17	COLOUR FORMAT	With the returned material ID, assign RGB values to temporary registers for shading calculations.
18	SHADING 1	Determine light direction vectors and normal direction vectors based on the current position of the ray and its nearest AABB boundary.
19	SHADING 2	Calculate the magnitude of the light direction vector - the input of the square root sub-FSM
20	SQRT INIT	Set initialisation parameters for the square root sub-FSM (result = 0, iterative bit *(sqrt_bit) = 2^{30})
21	SQRT ITER	Iterative state where sqrt_bit shifts right and the result is adjusted accordingly on a bit-by-bit basis.
22	SQRT ITER BUFFER	Buffer state to allow for a clock tick for conditional state changes out of state 21.
23	SHADING 3 INIT	Normalised light direction vector initialised as 8096 x light direction vector.
24	SHADING 3	Normalisation approximation state.
25	SHADING 4	Compute the brightness factor - the dot product of the normalised light direction and collided plane normal vector.
26	RAY OUT OF BOUND	If the ray is out of the world, then return the background colour.
27	OUTPUT COLOUR INIT	Apply shading by multiplying the temporary RGB values by the determined brightness factor.
28	OUTPUT COLOUR	Output the RGB values to the pixel packer.

4.2.1 Implementing Shading on Hardware

The implementation of shading follows the same method as the software Python version of the ray tracer. This includes treating the camera as a "light source" - and thus defining the light direction as camera position minus current ray position. The brightness factor is then calculated by taking the dot product of the normalised light direction and the unit vector normal of the face of the bounding box that the ray has hit. Finally, the RGB value of the material of the object the ray has hit is multiplied by the brightness factor to compute the final colour of the pixel.

The hardware implementation of the procedure outlined above presented several challenges all centred around the fact that Vivado does not inherently support non-constant floating point variables, which is what the Python version makes use of. More specifically, attempting to normalise an integer vector by only using integer operations is technically impossible due to the necessity of a square root operation. This was avoided by implementing a square root approximation algorithm.

The square root approximation is computed using an iterative bit-wise approximation method, also known as the digit-by-digit algorithm. This involves an iterative refinement of the estimate by evaluating each bit of the input value (light vector magnitude) from the most significant to the least significant bit.

Initially, the result `sqrt_res` is set to zero, and the test bit `sqrt_bit` is initialised to 2^{30}, the highest power of four less than the theoretical maximum input for a 32-bit number. In each iteration, a

temporary value `sqrt_temp` is calculated by adding the current test bit to the estimated result. If this temporary value does not exceed the input, it is subtracted from the input, and the estimate is updated to include this bit. If not, the estimate is simply right-shifted to discard the current bit. The test bit is then shifted to positions to the right (divided by 4) and the process repeats until all positions have been evaluated. To better demonstrate the way the approximation algorithm works, an example of finding the square root of 144 is shown below, and the snippet of this part of the code can be found in C.1

Square Root of 144 Initial Setup

- **Input:** `sqrt_input = 144`
- **Initial sqrt_res:** 0
- **Initial sqrt_bit:** $1 \ll 6 = 64$ (Highest power of 4 smaller than input)

Iteration 1

Current sqrt_bit:	64
Calculate sqrt_temp:	$\text{sqrt\_temp} = \text{sqrt\_res} + \text{sqrt\_bit} = 0 + 64 = 64$
Comparison:	$\text{sqrt\_input} \geq \text{sqrt\_temp} \quad \Rightarrow \quad 144 \geq 64$ True
Action:	$\text{sqrt\_input} = \text{sqrt\_input} - \text{sqrt\_temp} = 144 - 64 = 80$
	$\text{sqrt\_res} = (\text{sqrt\_res} \gg 1) + \text{sqrt\_bit} = (0 \gg 1) + 64 = 64$
Update sqrt_bit:	$\text{sqrt\_bit} = \text{sqrt\_bit} \gg 2 = 64 \gg 2 = 16$

	sqrt_res:	1
sqrt_input:	10010000	(144)
−	01000000	(64)
	01010000	(80)

Iteration 2

Current sqrt_bit:	16
Calculate sqrt_temp:	$\text{sqrt\_temp} = \text{sqrt\_res} + \text{sqrt\_bit} = 64 + 16 = 80$
Comparison:	$\text{sqrt\_input} \geq \text{sqrt\_temp} \quad \Rightarrow \quad 80 \geq 80$ True
Action:	$\text{sqrt\_input} = \text{sqrt\_input} - \text{sqrt\_temp} = 80 - 80 = 0$
	$\text{sqrt\_res} = (\text{sqrt\_res} \gg 1) + \text{sqrt\_bit} = (64 \gg 1) + 16 = 32 + 16 = 48$
Update sqrt_bit:	$\text{sqrt\_bit} = \text{sqrt\_bit} \gg 2 = 16 \gg 2 = 4$

	sqrt_res:	11
sqrt_input:	01010000	(80)
−	01010000	(80)
	00000000	(0)

Iteration 3

Current sqrt_bit:	4
Calculate sqrt_temp:	$\text{sqrt\_temp} = \text{sqrt\_res} + \text{sqrt\_bit} = 48 + 4 = 52$
Comparison:	$\text{sqrt\_input} \geq \text{sqrt\_temp} \quad \Rightarrow \quad 0 \geq 52$ False
Action:	$\text{sqrt\_res} = \text{sqrt\_res} \gg 1 = 48 \gg 1 = 24$
Update sqrt_bit:	$\text{sqrt\_bit} = \text{sqrt\_bit} \gg 2 = 4 \gg 2 = 1$

		sqrt_res:	110
	sqrt_input:	00000000	(0)
−	00110100	(52)	(TOO LARGE)
		00000000	(0)

Iteration 4

Current sqrt_bit:	1
Calculate sqrt_temp:	$\text{sqrt\_temp} = \text{sqrt\_res} + \text{sqrt\_bit} = 24 + 1 = 25$
Comparison:	$\text{sqrt\_input} \geq \text{sqrt\_temp} \quad \Rightarrow \quad 0 \geq 25$ False
Action:	$\text{sqrt\_res} = \text{sqrt\_res} \gg 1 = 24 \gg 1 = 12$
Update sqrt_bit:	$\text{sqrt\_bit} = \text{sqrt\_bit} \gg 2 = 1 \gg 2 = 0$

		sqrt_res:	1100
	sqrt_input:	00000000	(0)
−	00011001	(25)	(TOO LARGE)
		00000000	(0)

Final Result

- **Final sqrt_res:** 12

Initially, the components of the light direction vector are scaled by 8096. This scaling is performed to bring the values into a fixed-point range that facilitates efficient integer arithmetic. Having computed `sqrt_res`, the next step is to divide the `light_dir` vector by it to calculate `normalised_light_dir`, however, such a computationally expensive division gives a worst negative slack (WNS) of -99 ns and caused distortions in the colours of the image.

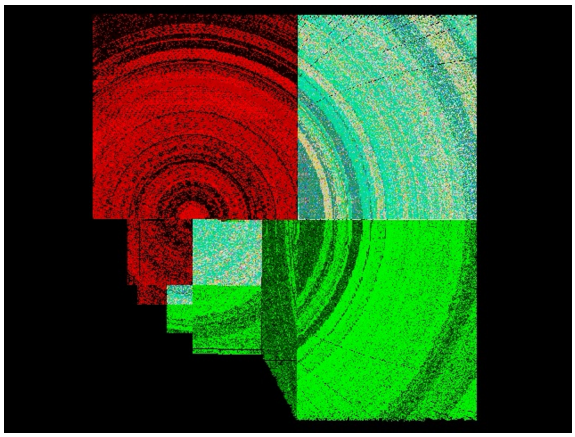


Figure 30: Colour Distortion due to -99ns WNS

As such, the division was approximated by `sqrt_res` by using right shifts. The number of right shifts is determined based on the range of `sqrt_res`. For instance, if `sqrt_res` is greater than or equal to 1024, the light direction components are right-shifted by 10 bits, effectively dividing them by 1024. Similarly, other ranges are handled with corresponding shifts, ensuring the light direction vector components are appropriately scaled down to a normalised range.

This approximated division is repeated - while also performing the same shifts to `sqrt_res` until `sqrt_res` is less than 2. This process normalises the light direction vector components and the brightness factor within a range of 0-8095. The final RGB values are then divided by 8096 (logical right shift by 10 bits) to bring them to a range between 0 and 255.

Although this method only approximates the normalisation, the resulting precision is sufficient for shading calculations in the context.

4.3 Memory

The ROM module was implemented in SystemVerilog to hold the octree structure in memory. This ROM is hard-wired to hold the contents of a memory file and is generated by a Python script that parses the text file generated from the Unity 3D model. (see 2.1) The consistency and symmetry of the octree environment representation structure allowed the representation of each level of the octree using 8 consecutive memory words in the ROM. The offset of each word describing an octant is consistent within its section. 2.1.2.

Every word in the ROM can either be a pointer or a material tag.

Whenever an octant of space cannot be fully described by a single material - and the space has to be divided further to fully describe it - the memory word describing it is a pointer to a new level in the octree - a new block of 8 words. (The pointer points to octant 0 of the new level.)

Oppositely, whenever an octant can be fully described by a single material it takes a material tag. To differentiate between pointers and material tags, a mask of 0xFFFFFFxx was chosen. This enables the use of 8 bits to specify the material - meaning up to $\{2\hat{8}\}$ materials can be supported. Material ID to RGB value mappings are hard-coded into the Ray Processor unit.

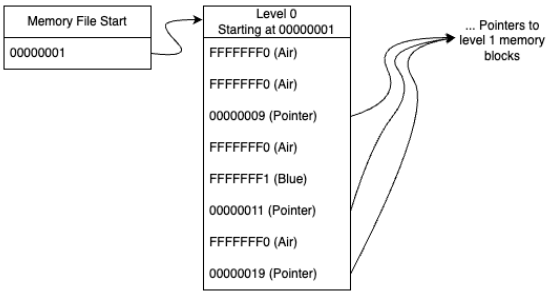


Figure 31: Memory structure diagram

The SystemVerilog Ray Processor unit can traverse the memory structure described in Figure 31 based on knowledge of its current location coordinates, and the minimum/maximum coordinates of the current octant in which it is in. If the current word is not a material tag, it traverses the memory file further. If it is a material tag, based on whether it is free space (material ID 0) or not, it decides to either step further along the world or return the colour of the object and enter the shading stage.

4.4 Ray Tracing Unit (RTU)

The ray generator and ray processor are connected via the RTU top-level module. The RTU is connected to the provided pixel packer, sending pixel data to the VDMA.

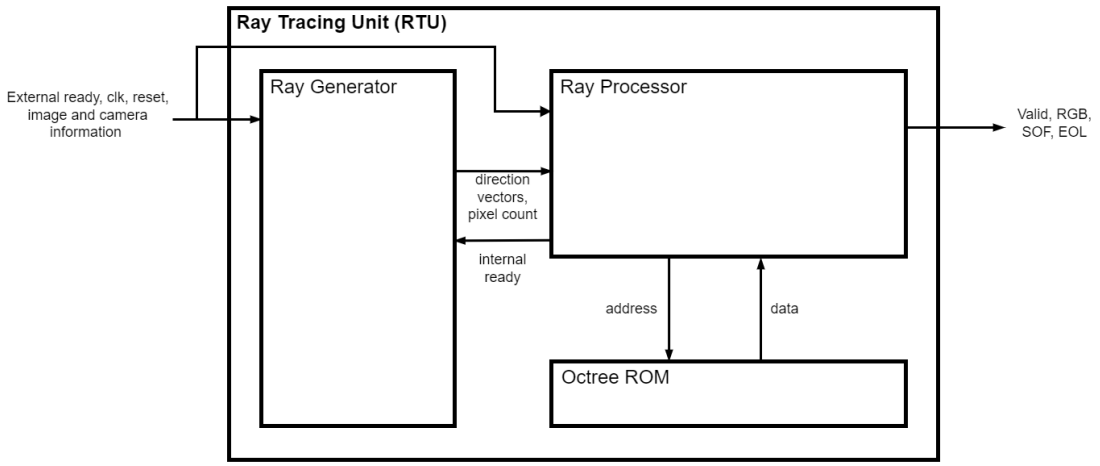


Figure 32: Top-level diagram of RTU

Through Verilator simulation, Figure 33 was generated. The same image was later generated in hardware.

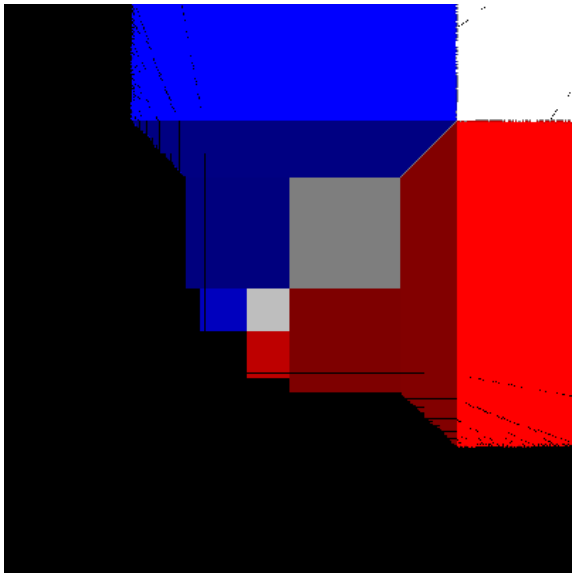


Figure 33: Image from Simulation

Both the software and hardware simulation have minimal inaccuracies. A perfect image would show completely filled cubes. The inaccuracies, shown in Figure 33, relative to a perfect image, are black pixels in areas where block colours are expected. These inaccuracies are more prevalent on the edges of cubes, shown in Figure 34. These are a result of rounding errors.

Figure 34: Increased frequency of rounding errors on shape edges

4.4.1 Minimising Rounding Errors

By using integer components for the direction vectors, there will be inaccuracies when bisecting the direction vector due to rounding. This cannot be avoided, however, the effect of the final step can be minimised.

An effect of rounding is that the direction vectors are no longer guaranteed to converge just outside the bounding box, meaning unit stepping must be considered. Below in Figure 35 is a 2D demonstration of two potential methods.

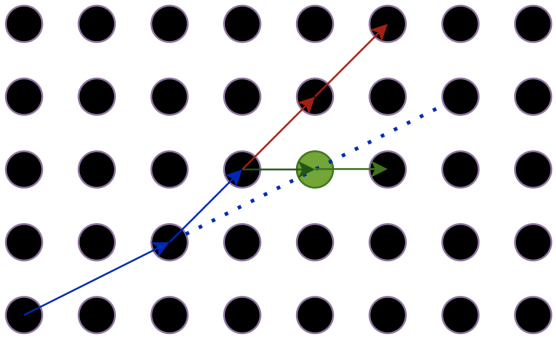


Figure 35: Rounding errors from bisection

The blue ray signifies a direction vector and the dotted line signifies its 'ideal' path. After a step, the components are bisected. The x component is 2 and can be halved to 1, however, the y component is already 1, and ideally, should go to 0.5. This will be rounded up to 1. The next step is already out of the desired trajectory. From now on, the difference in the two cases can be seen.

The first is to keep going, rounding 0.5 up to 1 every single time, resulting in a direction vector of (1,1) every time an integer stepping within a bounding box is used.

The second is to store an underflow bit for all the components. When this is high for the first time, the component is rounded up to one. Upon subsequent bisections, this will be set to 0 as theoretically, it should be near to 0.25, which rounds down to 0.

In order to make sure that the distance vector never becomes 0, as convergence outside the bounding box is no longer guaranteed using integers, a component only goes to 0 if there is at least 1 other non-zero component in the next step. By doing this, in 2D, it means the unit steps can be in 3 directions, (1,0), (1,1) and (0,1). In 3D, this is extended to going from being able to unit step in 8 directions (the vertices of a cube) to 26 directions (the faces, edges and vertices of the cube), allowing for a more accurate direction vector. From 35 you can see that it will intersect the ideal path.

The SystemVerilog code for this procedure can be seen in section C.2 of the appendix.

4.5 Parallelisation

Before, pixel generation was done sequentially. There are two main tasks that are executed one after another: ray generation then ray processing. If the maximum time taken to complete one task is $t_{process}$, then the total time required to process N rays to generate an image sequentially is given by:

$$t_{sequential} = N \cdot 2 \cdot t_{process}$$

Where:

$$t_{process} = \max(t_{generate},\ t_{processing})$$

Concurrent processing of pixels can be done with multiple RTUs operating in parallel. Define the number of RTUs as R, then the processing time can be reduced to:

$$t_{parallel} = \left\lceil \frac{N}{R} \right\rceil \cdot M \cdot t_{process}$$

Parallelisation of image generation can be done in 2 ways. The first is splitting the target image into equal neighbouring sections such that each RTU generates pixels for one section. For example,

with four RTUs, the image can be split into quarters and each RTU generates pixels for one quarter. The second option is to have each RTU generate neighbouring pixels, i.e. for four RTUs, pixel 1 is generated by RTU 1, pixel 2 by RTU 2, pixel 3 by RTU 3, pixel 4 by RTU 4, pixel 5 by RTU 1, pixel 6 by RTU 2 and so on. Due to the nature of the provided pixel packer [1] module, parallelisation via the second option was chosen. This can be implemented through the addition of a new module: Pixel buffer.

A pixel buffer is a module that stores incoming pixel data from each ray processor and releases them sequentially to the pixel packer when it is ready. With two ray processors, their respective RGB values are stored in the pixel buffer. The pixel buffer starts with the first set of RGB values writes it to the pixel packer and then moves the next. After the pixel buffer is done processing a signal, it sends a "ready" signal to its respective ray processor so that it can start processing a new ray should the following ray processors take more time processing a ray. This method of sending individual ready signals to ray processors as opposed to having a single ready that is set high when all rays are sent to the buffer was chosen to reduce the time in between subsequent ray processing. The state diagram is shown in Figure 36. and descriptions in table 7. These changes have affected the top-level RTU and have been reflected in Figure 37.

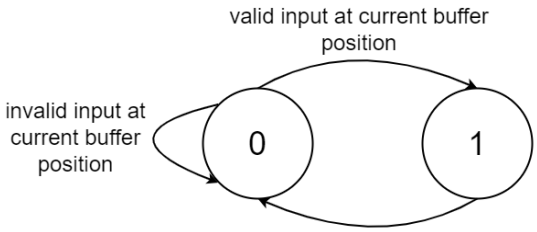


Figure 36: Pixel buffer state diagram

Table 7: PixelBuffer.sv State Descriptions

#	**State**	**Description**
0	IDLE	A waiting state to check whether the current pixel buffer slot has valid data. If it does, then proceed to write to the pixel packer in the next state. Before leaving this state, change which slot to look at so that the correct slot is returned to for polling.
1	WRITE PIXEL	The RGB, SOF and EOL values are written to the pixel packer. Return to the IDLE state afterwards.

[1] More information on the pixel packer is given in the next section

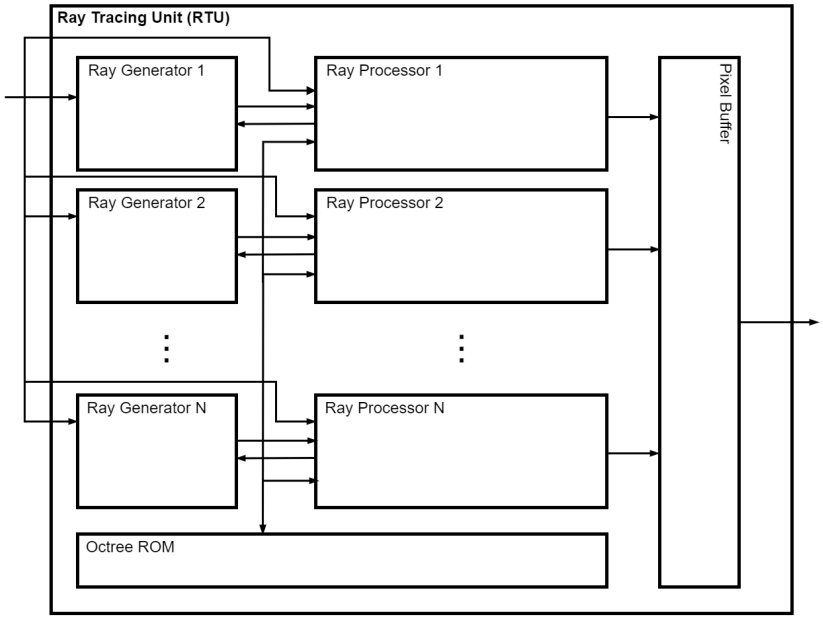


Figure 37: RTU with parallel ray processors and pixel buffer

Whilst more cores improve performance, there is an element of diminishing returns. As the number of cores increases, more resources get used and a smaller increase in performance is noticed. Figure 38 shows the effect of increasing the number of ray processors against the number of clock cycles. The number of clock cycles is expected to decrease at a rate inversely proportional to the number of ray processors. In Figure 38 it can be seen that the actual results follow this trend, however, there is a clear offset due to the pixel buffer moving through the buffer and outputting each pixel.[2]

Due to resource limitations outlined in section 6.4, parallelisation with 2 RTUs was chosen.

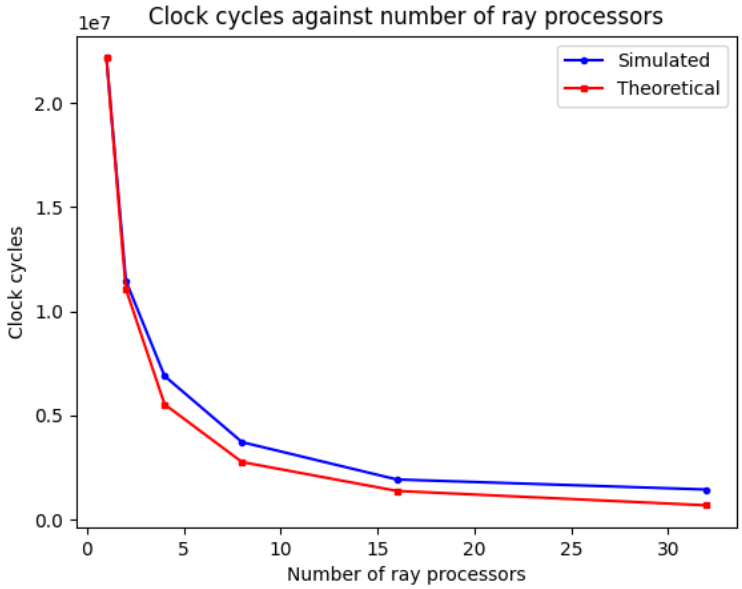


Figure 38: Effect of parallel ray processors on computation speed

[2]To reduce time between runs, these results were obtained via Verilator simulation as opposed to on the board.

4.6 Top Level

The RTU is responsible for producing RGB values from the given scene. However, RGB data needs to be sent to the VDMA module to be kept in a buffer until a full frame is successfully read, after which a full frame can be displayed on the PS.

Formatting RGB (24-bit) into 32-bit data is done by a provided pixel packer module. As the name suggests, this module packs the free 8 bits remaining of the word with 8 bits of the next pixel. This combination of RTU and pixel packer is wrapped in a new top-level module, pixel generator, shown in Figure 39. As mentioned in the previous section, there is only one-pixel packer in the parallelised implementation, so whilst the generation of pixel RGB data is done in parallel, output to the VDMA is done serially.

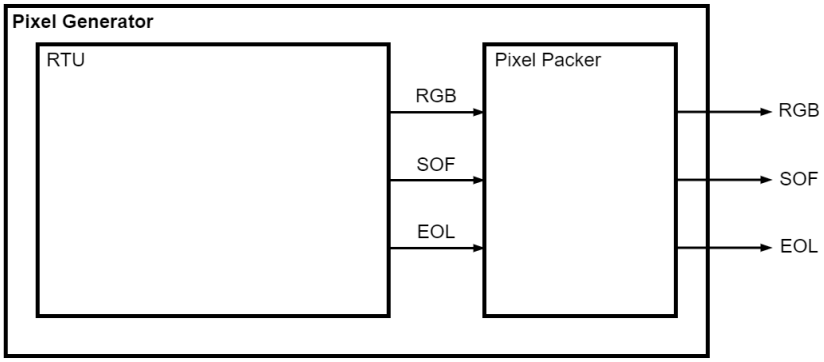


Figure 39: Pixel generator top level

Figure 39 shows 2 new signals: `SOF` and `EOL`. These are signals needed by the VDMA. `SOF` is the "start of frame signal" more commonly known as `tuser`. This signal tells the VDMA when the RGB data is that of a new frame. `EOL` or `tlast` is a signal that tells the VDMA that a pixel in the buffer is the last of a line, the number of EOL signals should be the same as the image's height [12].

This concludes the hardware implementation of the Ray Tracer, the results are discussed in section 6.1.

5 Graphical User Interface

As per the project requirements, a suitable GUI is necessary for users to interact with the ray tracer. The GUI is responsible for:

- Sending parameters such as camera position, camera direction, scene choice and background colour, to the FPGA.
- Receiving rendered images and displaying them to the user.

Pygame Library and Numpy was used to develop GUI:

5.1 Parameters

The following parameters were passed in as user input:

1. Camera position
2. Camera direction
3. Right vectors
4. Up vector
5. Background colour

The GUI collects user-readable data and formats it into the MMIO registers.

Each of these quantities fits into a single 32-bit register that can be written to via the PS and accessed via the PL. Figure 40 describes the use of registers to transfer parameters from PS to PL. **Figures showing side by side of changed camera position and background colour**

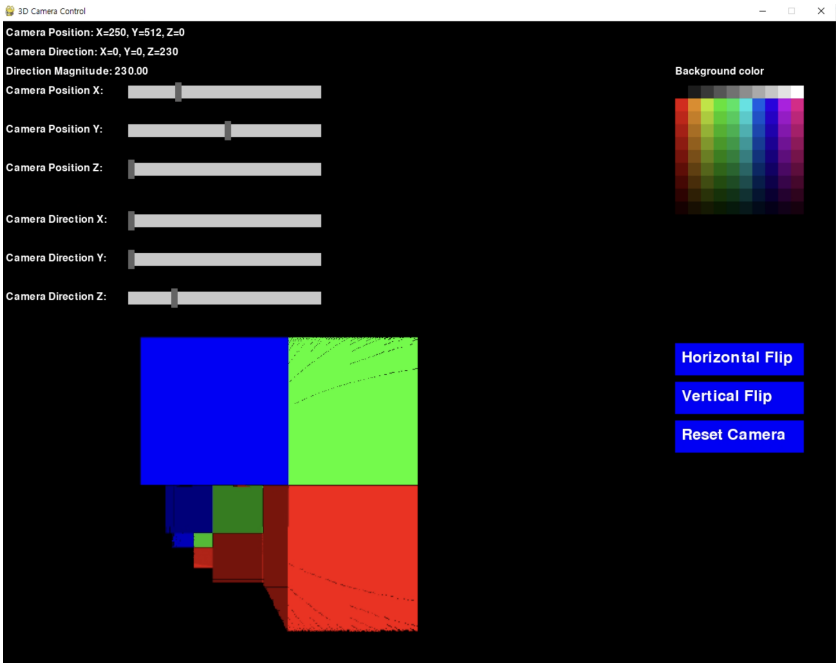
Function	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Camera position	gp0			Z Component										Y Component										X Component									
Camera direction	gp1			Z Component										Y Component										X Component									
Camera right	gp2			Z Component										Y Component										X Component									
Camera up	gp3			Z Component										Y Component										X Component									
Background colour	gp4									Blue								Green												Red			

Figure 40: MMIO parameter register mapping

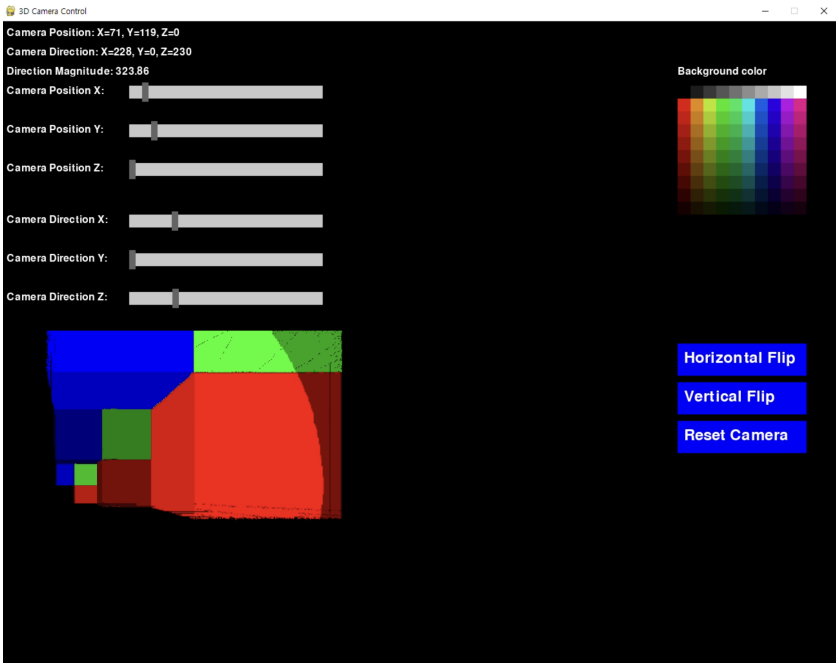
5.2 Client Interface

The client graphical user interface offers a user-friendly experience for interacting with the ray-traced environment. It provides intuitive controls for camera manipulation, allowing users to explore the scene from various perspectives.

5.2.1 Camera Controls



(a) Parameter 1



(b) Parameter 2

Figure 41: Camera parameter updates

Depicted in Figure 41, adjustments can be made to the camera's position and direction using X, Y, and Z component sliders. For a more immersive experience, the GUI also incorporates a familiar W/A/S/D and mouse control scheme, mimicking the feel of a rendering engine like Blender or Unity3D.

The GUI offers a user-friendly single-click RGB selector and 3 quick camera adjustment buttons(horizontal/vertical flip, reset) for effortlessly adjusting the rendered frame.

5.2.2 Nonstop rendering

To achieve near real-time, non-stop rendering, the client application transmits camera parameters to the Frame API server using TCP requests. This continuous communication loop facilitates video-like frame updates on the image section of the GUI.

5.2.3 Performance Considerations

While hardware limitations on the PYNQ board and the inherent overhead of TCP communication prevent true real-time frame rates, the core functionality of hardware-accelerated rendering aligns with the project's main goal. Currently, frame updates occur at approximately 200 milliseconds per frame. This latency can be reduced through further parallelisation and the implementation of a direct HDMI connection, aiming for smoother rendering experiences in the future.

5.3 Frame API Interface

On the server side, a Python-based TCP server acts as the API endpoint. The client application interacts with this server by requesting newly rendered frames at 200-millisecond intervals (this interval can be customised). In response, the server transmits the requested frame data as packed byte arrays containing RGB information.

5.3.1 FPGA Integration Frame Data Validation

The PYNQ library is used to initialise overlay using a Vivado-generated bitstream file. Additionally, the built-in register file interface of the pixel generator supports dynamic changes to camera parameters based on each client request.

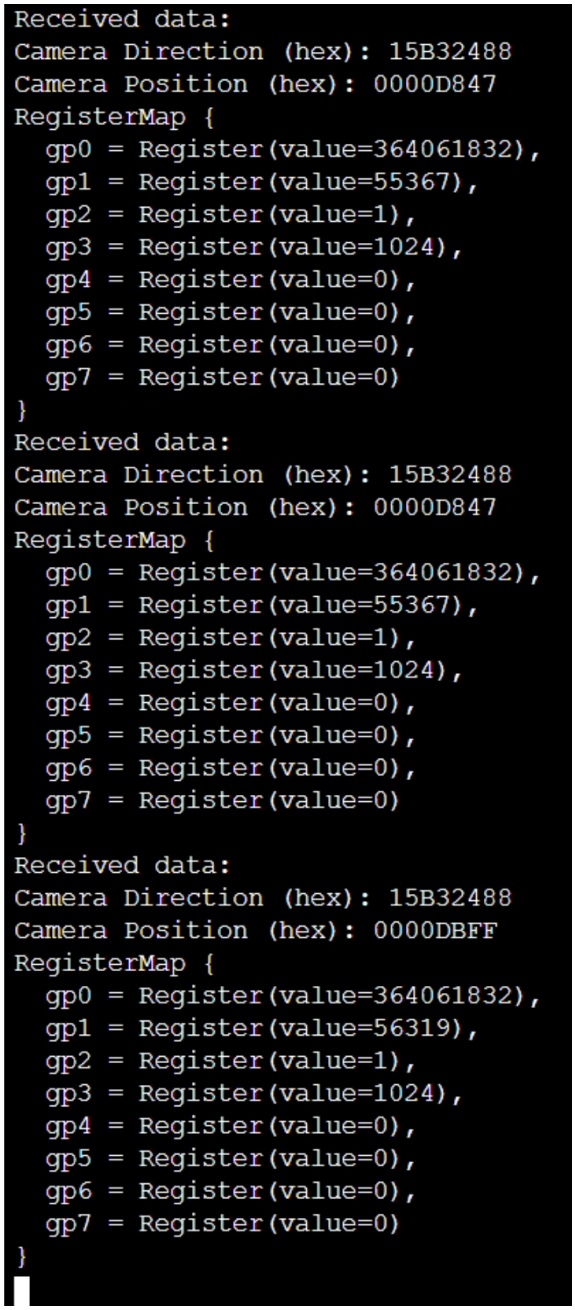


Figure 42: Server console output

To ensure the transmission of fully rendered frames, the server reads the frame data four consecutive times before accepting it as final. Also for debugging purposes, the server console is set to print out the current regfile values to ensure parameter processing as shown in Figure 42

This approach is an attempt to ensure the completeness of the rendering process before the frame is sent to the client. While effective, this method can be further optimised by implementing a dedicated "ready-to-send" signal within the FPGA's ray processing module.

6 Evaluation

6.1 Performance Evaluation

Measuring the performance of the hardware consisted of three parts: comparing against the C++ and Python implementations, evaluating accuracy and evaluating the algorithmic complexity.

6.1.1 Comparing software and hardware

The hardware ray tracer was tested against both C++ and Python equivalent implementations. For a fair evaluation of performance, figure 43 was used to measure computation speeds of varying image dimensions. For additional context, the software was run on a 1.4GHz Quad-Core Intel Core i5 processor. [13].

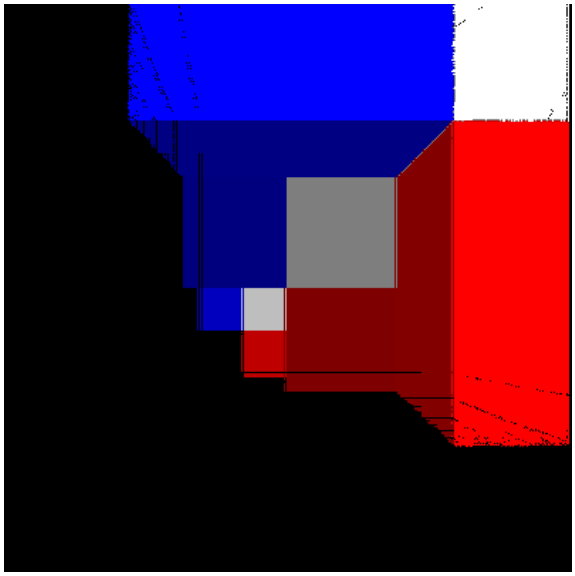


Figure 43: Control Image

Figure 44 shows run times for both the Python and C++ ray trace implementations. They both increase linearly with the number of pixels. In comparison, figure 45 shows a much quicker run time.

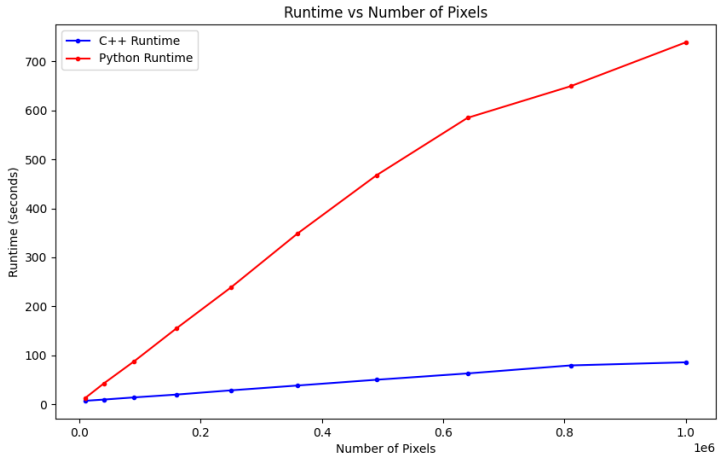


Figure 44: Graph of time taken against pixels generated

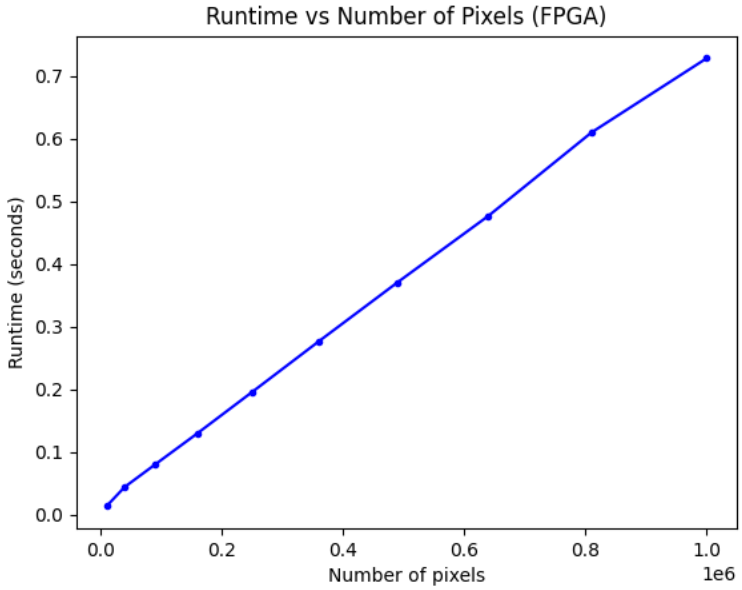


Figure 45: Graph of time taken against pixels generated on the FPGA

Comparing the linear line of best fits, the results are as follows:

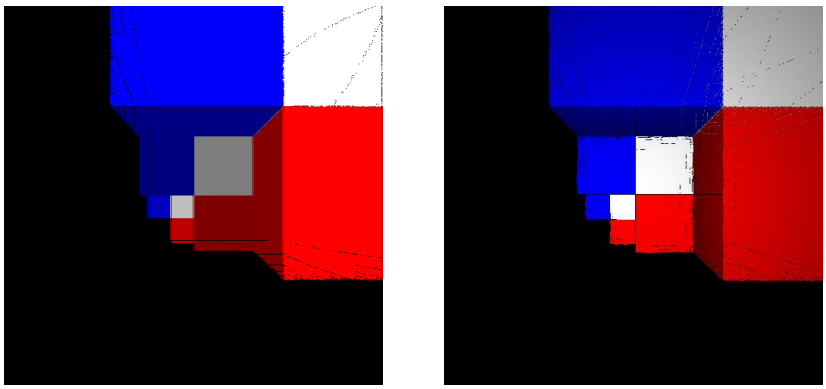
$$\text{FPGA: } y = 7.246 \cdot 10^{-7}x + 0.0134$$
$$\text{C++: } y = 8.407 \cdot 10^{-5}x + 7.038$$
$$\text{Python: } y = 7.684 \cdot 10^{-4}x + 36.77$$

The gradient of the equation is the time taken to compute one pixel. It can be sees that the FPGA, is $\frac{8.407 \cdot 10^{-5}}{7.246 \cdot 10^{-7}} = 116.02$ times faster than the C++ implementation and $\frac{7.684 \cdot 10^{-4}}{7.246 \cdot 10^{-7}} = 1060.45$ times

faster than the Python implementation. This is a significant acceleration relative to the software versions and is mainly down to the use of integers over floating pointer numbers throughout the algorithm.

6.1.2 Accuracy

Figure 46 compares a hardware-generated image to a software one. Upon first glance, it is clear that octree traversal is functional. There are inconsistencies with shading that are yet to be fixed. There are also multiple rounding errors and inconsistencies, whose effect has been minimised as outlined in section 4.4.1.



(a) FPGA (b) Software

Figure 46: Hardware-software comparison

To determine how accurate the hardware ray tracer is, images from identical scenes generated from hardware and software were put through a diff checker [14]. Figure 47 shows clear inaccuracies in the shading aspect. The inaccuracies are due to the decision to implement ray tracing purely using integers as such, normalisation and shading brightness factors are slightly less accurate in hardware, however, the hardware shading still creates a sense of depth within the scene.

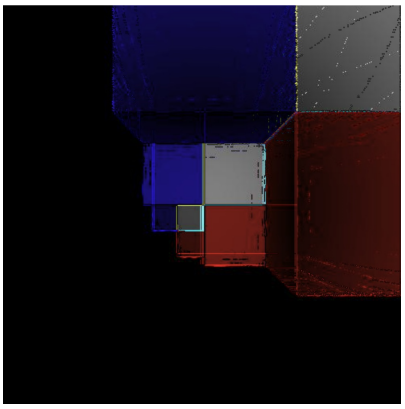


Figure 47: Results of difference checking

6.1.3 Algorithmic complexity

To measure the algorithmic complexity of the ray tracer, the time taken to generate images of increasing sizes was measured. Figure 48, shows that the dual-core has linear time complexity.

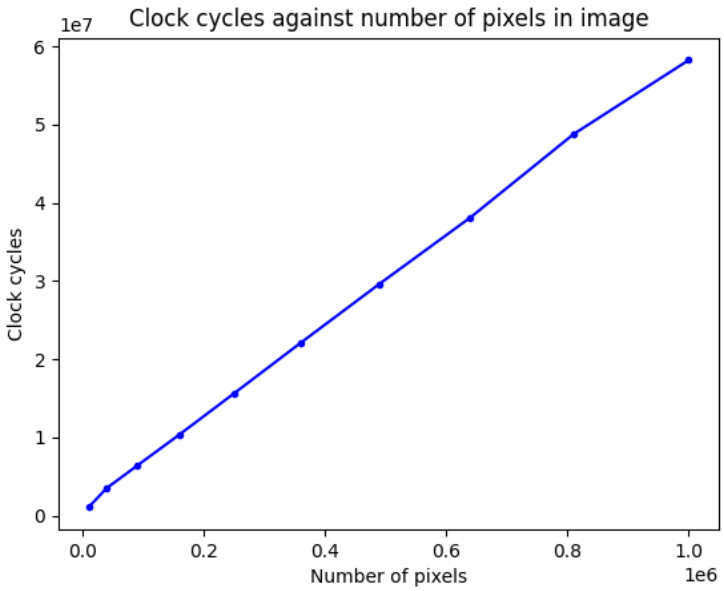


Figure 48: Clock cycles required frame against number of pixels in frame

This is because ray generation has complexity $O(1)$, octree traversal and ray stepping has complexity $O(\log(m)+p)$ where m is the depth of the octree and p is the number of steps for ray stepping. Shading has k steps therefore a complexity of $O(k)$. Therefore for n rays, the time complexity is:

$$\text{Time Complexity} = O(n + n\log(m) + np + nk)$$

The above implies non-linear time complexity: $O(n\log(m)$. However, since m is the depth of the octree, which was set to a maximum of 10, due to the 1024 coordinate space, the time complexity of the ray stepping and octree traversal can be approximated to $O(n+pn)$ where n is the number of rays and p is the number of steps following traversal. Replacing this with the above time complexity, linear time is indeed achieved:

$$\text{Final Time Complexity} = O(2n + np + nk)$$

6.2 Ray Processor Deconstruction

From the section **??**, it is clear that there are many states for a single-state machine. An optimisation for the sake of module structure involves splitting the ray processor into multiple stages, each with its own FSM. The ray processor does two main jobs:

- Step the ray through the world, returning valid RGB values for each pixel.
- Apply a brightness factor to the RGB values to achieve more realistic shading.]

Therefore, the ray processor was reduced into a ray stepper and ray shader as shown in Figure 49. This not only cleans up code but opens up the RTU to potential pipelining.

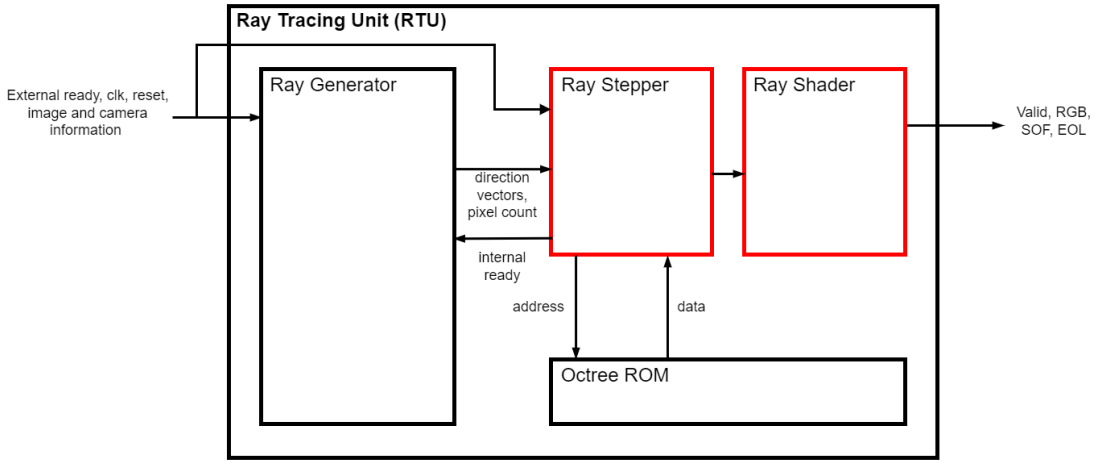


Figure 49: RTU (not parallelised for clarity) with broken down ray processor

6.3 Pipelining

Pipelining can be implemented by inserting pipelining registers between the ray generator and ray stepper, and between the ray stepper and ray shader, as shown in Figure 50.

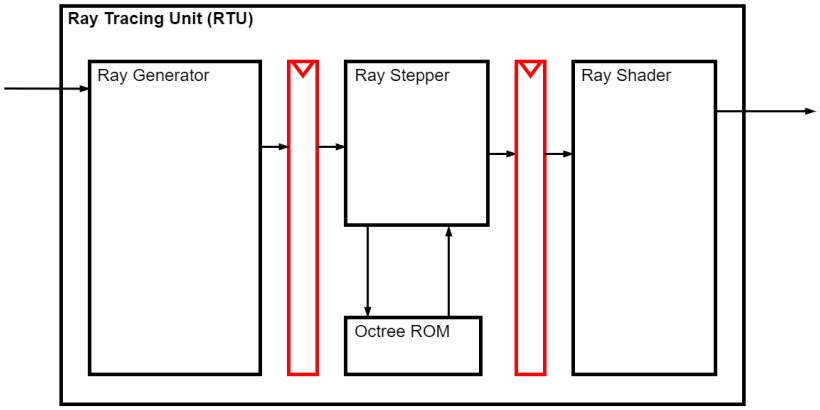


Figure 50: Pipeline RTU (not parallelised for clarity)

As shown in Figure 51, pipelining shows that different tasks can be executed concurrently. With three tasks, ray generation, stepping and shading, 3 units can be set up for executing a single task in the procedure.

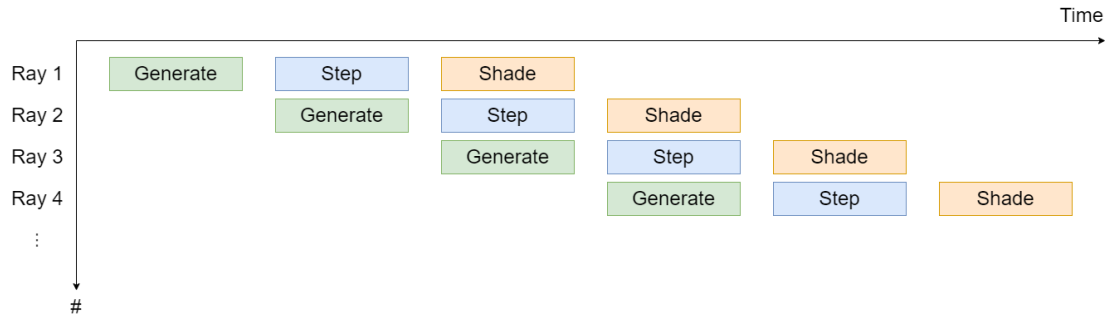


Figure 51: Pipelined execution

The generator will produce a ray and pass it to the stepper. Whilst the stepper processes this ray, a new ray can be generated. For a total of N rays, the total processing time can be reduced to a maximum of:

$$t_{pipeline} = (N + 2) \cdot t_{process}$$

Where

$$t_{process} = \max\left(t_{generate},\ t_{step},\ t_{shade}\right)$$

From 4.5, this is an improvement to a sequential time of: [3]

$$t_{sequential} = N \cdot 3 \cdot t_{process}$$

6.4 Resource Utilisation

When implementing a single core design, a resource utilisation report was used, as shown in Figure 52. The components that are of concern are the pixel generator and the video top-level module that includes modules for HDMI and VDMA.The resource breakdown for each can be found in the table 8 and table 9 respectively.

Resource	Utilization	Available	Utilization %
LUT	37906	53200	71.25
LUTRAM	2304	17400	13.24
FF	51795	106400	48.68
BRAM	68	140	48.57
DSP	60	220	27.27
IO	118	125	94.40
MMCM	2	4	50.00

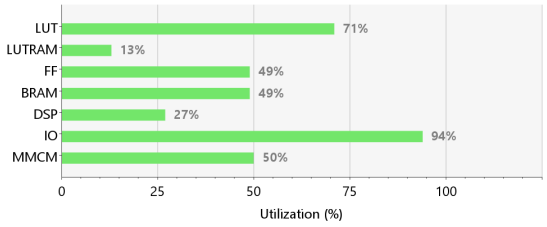


Figure 52: Single core utilisation report

[3] Now adjusted with 3 stages as opposed to previous 2 stage RTU

Name	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	Block RAM Tile	DSPs
pixel generator	3473	1242	35	0	0	42
pixel packer	30	27	0	0	0	0
ray tracing unit	3361	883	3	0	0	42
octant rom	2	5	0	0	0	0
ray generator	852	131	0	0	0	13
ray processor	2504	747	3	0	0	29
Percentage	6.53	1.17	0.13	0	0	19.09

Table 8: Resource utilisation for pixel generator and sub-components

Name	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	Block RAM Tile	DSPs
video	12446	22097	384	0	15	18
Percentage	23.39	20.77	1.44	0	10.71	8.18

Table 9: Resource utilisation for the video module

From the breakdown of resources, it is clear that extensions of the single-core design are limited by the DSPs. The PYNQ board has a total of 220 DSP slices [15], of which 42 are taken by a single RTU and 18 by the video module. That leaves a total of 160 that can be used for a parallelised version. The pixel buffer mentioned in section 4.5 for 2 cores, takes up resources outlined in table 10. The changes required for parallelisation also modified the ray generator resource utilisation as shown in table 11.

Name	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	Block RAM Tile	DSPs
Pixel buffer	102	84	0	0	0	2
Percentage	0.19	0.08	0	0	0	0.45

Table 10: Resource utilisation for the pixel buffer

Name	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	Block RAM Tile	DSPs
ray generator (parallel)	894	131	0	0	0	15
Percentage	0.5	0.12	0	0	0	6.36

Table 11: Resource utilisation for the ray generator in parallel

From this, the number of cores that can feasibly be implemented on the board can be determined:

- There are 220 DSPs in total.
- 18 are taken by the video block.
- 2 are taken by the pixel buffer
- 44 are taken by the ray processor and ray generator combination.

- It is indicated that there are 200 DSPs allocated solely for the ray generator and ray processor combination
- This implies there are 4 ray processors and generators operating in parallel.

However, during implementation on a setup with 4 cores, the implementation failed due to insufficient LUTs. The presence of other modules in the base design consumed significant LUT resources, making them a limiting factor. Consequently, it was decided to implement only 2 cores in parallel, thereby reducing the runtime to half that of a single core. Due to time constraints, the decision was made to maintain the base design as it stood. Given more time, additional cores would have been preferred for implementation.

6.5 Memory

Currently, a custom-built ROM module is used to store the rendering environment using an octree data structure. However, ROMs are read-only, preventing users from quickly changing the rendering environment from the processing system via the AXI protocol. To address this limitation, Vivado offers Block Memory IP and AXI BRAM Controller IP. These tools can be used to generate and utilize BRAM, enabling efficient communication and data storage between the processing system and programmable logic. Figure 53 is an example implementation of BRAM in Vivado block diagram using the Vivado IPs and rtl bus interface set up in IP packager of pixel_generator IP.

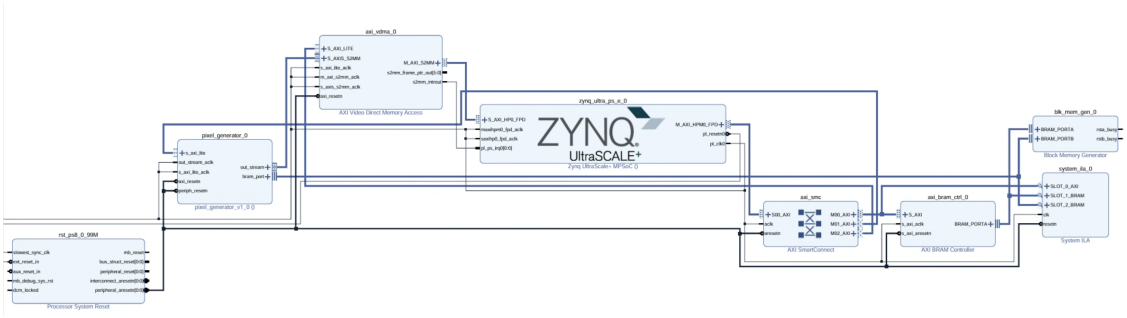


Figure 53: BRAM implementation block diagram

To communicate with the BRAM from a custom IP without building an AXI Master interface, a relatively simple BRAM bus interface (physical ports) is built into the top module of the custom IP. Then, the logical ports of the bram_rtl interface are directly wired to the physical ports defined in the custom IP. This approach allows full utilization of BRAM from both the processing system (AXI4) and programmable logic (BRAM_RTL) by using true dual-port BRAM. After this implementation, rendering environments can be changed in near real-time, similar to how camera parameters are adjusted and new frames are generated in GUI.

6.6 Additional Features

The built ray tracer can be further extended to produce more realistic renders. The current model has octree traversal, camera movement, basic material colours and shading. To produce more realistic images, reflections, shadows, light sources and more advanced material properties could be implemented. More details on the theory around these features can be found below.

6.6.1 Reflective Materials

In order to implement reflective materials, materials are first identified as reflective in nature. With the current memory space, a separate flag would be used. A choice to split either the address space or material space would be made. If using the address space, this would limit the number of nodes available, meaning the maximum complexity of the scene would decrease. Taking a bit away from the colour space would limit the amount of colours that can defined from 256 to 128. This, however, does not directly correspond to the colour depth [16], as the shading feature will still allow for a deep colour image. The choice is down to the user, reliant on the purpose of the FPGA.

After this, when a reflective material is hit, the component of the direction vector matching the surface normal will be flipped and a counter will need to be incremented to keep track of the number of reflections. This is so that no infinite loops are created from having multiple reflective materials. Afterwards, the ray propagates as usual.

If the ray is rather forced to stop propagation due to hitting some limit on the number of reflections, the returned colour must be the average of all the colours of the reflective materials it has hit. In order to do this, on each reflection, a temporary average RGB is stored, which will be returned in the case of reflection timeout.

6.6.2 Shadows and Light Sources

Shadows and light sources come hand in hand. Currently, the effect of shadows is mimicked while working around actually having to define a separate light source. In order to implement light sources, a point source of light can be defined and passed in as a parameter. Upon calculation of the light direction vector, instead of being the opposite of the ray direction, it will now be from the hit position to the light position and the following logic is the same.

6.7 Review with Project Requirements

Mathematical function visualisation:

Table 12: Mathematical visualisation requirements evaluation

#	Requirement	Evaluation
M1	The function shall be computationally intensive, such that it is not trivial to generate the visualisation at the required resolution and frame rate	Voxel-based ray tracing satisfies this requirement due to the computation required to step a ray in the world, check for hits or not, and determining appropriate colour shades based on hit angles. This process needs to be repeated for every pixel in an image and has been optimised through the use of octrees, and integer-based square root algorithms.

M2	The computation should be 'embarrassingly parallel', which means that multiple solutions of the problem, pixels in this case, can be computed independently with no data dependence between them.	This requirement is satisfied. In ray tracing, there is no case where one pixel is dependent on another. WParallel ray tracing has been simulated with multiple cores, proving that this algorithm can be parallelised effectively.

PYNQ hardware:

Table 13: PYNQ hardware requirements evaluation

#	Requirement	Evaluation
P1	The accelerator shall be described using Verilog or System Verilog.	Requirement satisfied, all code for ray tracing hardware acceleration has been written in System Verilog.
P2	The accelerator shall provide an interface with the integrated CPU for the adjustment of parameters.	Requirement satisfied. MMIO was used to write to registers using the PS and then read from them via PL. The registers are used to store camera location, direction, focal length details and background colour,
P3	The number formats and word lengths used in the accelerator should be selected to optimise the trade-off between visualisation accuracy and computational throughput.	Requirement satisfied. When implementing shading, square root approximations were obtained using just integers, trading off some accuracy for computation. Vector data has been chosen to be 10-bit each so that all 3 components can fit inside a 32-bit word.
P4	The computational throughput of the accelerated implementation shall exceed that of a CPU-only alternative programmed with C, C++ or Cython.	Requirement satisfied. Based on results outlined in section 6.1,image generation was accelerated by a factor of 116 compared to the C++ implementation.

User Interface:

Table 14: User interface requirements evaluation

#	Requirement	Evaluation
U1	The user interface may be implemented using separate hardware to the visualisation computer.	Requirement satisfied, a Pygame GUI was developed to receive images generated by the FPGA and send input parameters to the board.
U2	The user interface shall allow a user to adjust parameters of the visualisation in an intuitive or interesting way.	Requirement satisfied, the user is able to send camera position, direction, up and right normal vectors as well as an image background colour.
U3	The user interface should provide information about the visualisation as an overlay on the image or via a separate medium.	Requirement satisfied, the image, updated with the input parameters if needed, is sent and displayed to the user via a TCP connection.

6.8 Review with Top Level Project Specification

Table 15: Top-level project specifications evaluation

#	Requirement	Evaluation
1	Traversal: must be able to a voxel world space.	Requirement satisfied, rays are able to successfully traverse a voxel world space and generate an accurate scene for a given input.
2	Integers: to improve computation speed, integers must be used as much as possible in calculations.	Requirement satisfied, integers were used throughout all calculations, including in a square root algorithm.
3	Inaccuracy minimisation: attempt to minimise inevitable inaccuracies due to the usage of integers.	Requirement satisfied, rounding errors were minimised through the use of an underflow bit.
4	Acceleration: the hardware must show significant acceleration to that of the software version.	Requirement satisfied, the hardware accelerated ray tracer has a 116 gain over the equivalent in C++ and 1060 compared to Python.
5	Memory: make optimisations on the voxel world space to minimise storage requirements.	Requirement satisfied, octrees were used as the data structure for environment representation.
6	Parameters: the user must be able to specify camera parameters as well as colour parameters and receive the output in real-time.	Requirement satisfied, camera direction, position, up and right vectors as well as background colour were passed in as user input via MMIO.
7	Colour generation: accurately recognises different colours of different objects within the world space.	Requirement satisfied, the ray tracer accurately hits objects and returns the correct material ID corresponding to a valid RGB value.
8	Shading: models must implement some kind of shading to create depth effects.	Requirement satisfied, shading was implemented purely using integers.
9	Ray generation: the camera's viewpoint and orientation in the scene are correctly represented by the output image received.	Requirement satisfied, the ray generator produces direction vectors accurately corresponding to each pixel in the image.

References

[1] E. Stott, "Mathematics accelerator guide," https://github.com/edstott/EE2Project/blob/main/doc/accelerator-guide.md, 2023, accessed: 2024-06-15.

[2] E. . PA1, "Rasterisation image from eecs 487 pa1," 2015, accessed: 2024-06-15. [Online]. Available: https://web.eecs.umich.edu/~sugih/courses/eecs487/f15/pa1/index.html

[3] N. Developer, "Ray tracing," 2018, accessed: 2024-06-15. [Online]. Available: https://developer.nvidia.com/discover/ray-tracing

[4] B. Krüger, "Vtk marching cubes discussion," 2020, accessed: 2024-06-15. [Online]. Available: https://discourse.vtk.org/t/vtkmarchingcubes-vtkdiscretemarchingcubes-does-not-produce-a-closed-mesh-surface/4312

[5] A. D. Ryer, "The light measurement handbook," 1997, accessed: 2024-06-08. [Online]. Available: https://cgvr.cs.uni-bremen.de/teaching/cg_literatur/ILT-Light-Measurement-Handbook.pdf

[6] M. Moulin, "Linear, gamma and srgb color spaces," 2018, accessed: 2024-06-15. [Online]. Available: https://matt77hias.github.io/blog/2018/07/01/linear-gamma-and-sRGB-color-spaces.html

[7] A. Chalmers, E. Reinhard, and T. Davis, *Practical Parallel Rendering*. CRC Press, Mar. 2011.

[8] Wikipedia, "Bounding Volume Hierachy," 2024, Accessed: 2024-05-18. [Online]. Available: https://en.wikipedia.org/wiki/Bounding_volume_hierarchy

[9] C. Mobley, "Lambertian brdfs," 2021, accessed: 2024-06-08. [Online]. Available: https://www.oceanopticsbook.info/view/surfaces/lambertian-brdfs

[10] Unity, "Unity documentation - colour space," 2022, accessed: 2024-06-01. [Online]. Available: https://docs.unity3d.com/Manual/LinearLighting.html

[11] Wikipedia, "srgb," 2024, accessed: 2024-06-01. [Online]. Available: https://en.wikipedia.org/wiki/SRGB

[12] A.-S. V. Interface, *Scene Change Detection LogiCORE IP Product Guide*, 2024, accessed: 2024-06-15. [Online]. Available: https://docs.amd.com/r/en-US/pg322-v-scenechange-detect/AXI4-Stream-Video-Interface

[13] Intel Corporation, "Intel Core i5-8257U Processor 6M Cache up to 3.90 GHz," 2024, Accessed: 2024-06-16. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/191067/intel-core-i5-8257u-processor-6m-cache-up-to-3-90-ghz.html?wapkw=8257u

[14] Checker Software Inc., "Compare images to find their differences - Diffchecker," 2023, Accessed: 2024-06-16. [Online]. Available: https://www.diffchecker.com/image-compare/

[15] Digilent, *PYNQ-Z1 Reference Manual*, Digilent Inc., 2018, https://digilent.com/reference/_media/reference/programmable-logic/pynq-z1/pynq-rm.pdf.

[16] Wikipedia, "Color depth," 2024, Accessed: 2024-06-17. [Online]. Available: https://en.wikipedia.org/wiki/Color_depth

A Project Management

A.1 Gantt Chart

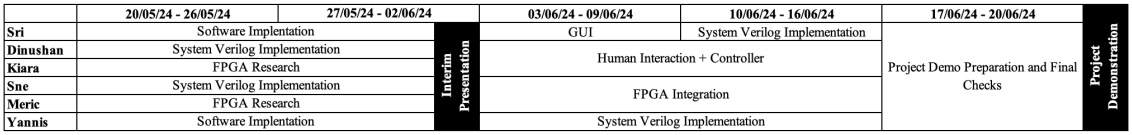


Figure 54: Gantt Chart

A.2 Weekly Plan

Week	**Aims**
Week 1 (20/05/24 - 26/05/24)	Research potential ideas, get familiar with the FPGA Investigate specifications and potential limitations with the hardware
Week 2 (27/05/24 - 02/06/24)	Prototype a software ray tracer and think about System Verilog Implementation. Analyse memory of models to be built and compare to storage requirements, analyse outcome of bit depth and clock cycles for each stage in simulations.
Week 3 (03/06/24 - 09/06/24)	Interim Presentation. Make progress on System Verilog implementation. Work on getting existing modules integrated with Vivado IPs and getting it on the FPGA. Start work on GUI and control. Start writing project report.
Week 4 (10/06/24 - 16/06/24)	Complete basic FPGA image generation, implement octree traversal, shading and parallelisation on System Verilog. Complete GUI and human interaction aspects. Work and finish report.
Week 5 (17/06/24 - 20/06/24)	Demo Week. Complete any outstanding tasks. Ensure the complete system is fully functional.

Table 16: Weekly Plan

B Software Implementation

B.1 Python Ray Tracer

```python
import numpy as np
from PIL import Image
from tqdm import tqdm

# Parameters
coord_bit_length = 10
cam_pos = np.array([200, 300, 0])
cam_norm = np.array([0, 0, 70])
cam_up = np.array([0, 1, 0])
cam_right = np.array([1, 0, 0])
im_height = 256
im_width = 256
#octree = [0, 0, 0, 0, [0,0,0,0,3,2,4,[0,3,0,0,1,[0,0,0,0,0,1,2,3],2,1]], 2, 3, 1]
octree = [0, 0, 0, 0, 0, 2, 3, 1]
material_table = [[0, 0, 0], [255, 255, 255], [0, 255, 0], [0, 0, 255], [255, 0, 0]]

# Image placeholder
image = np.zeros((im_height, im_width, 3), dtype=np.uint8)

def roundPosition(position):
    return np.round(position).astype(int)

def toBinaryStr(value, bit_length):
    return format(value, f'0{bit_length}b')

def withinAABB(position, aabb_min, aabb_max):
    return np.all(position >= aabb_min) and np.all(position <= aabb_max)

def justOutsideAABB(position, aabb_min, aabb_max):
    return np.any(position == aabb_min - 1) or np.any(position == aabb_max + 1)

def traverseTree(ray_pos, node, oct_size, aabb_min, aabb_max):
    depth = 0
    x_bin = toBinaryStr(ray_pos[0], coord_bit_length)
    y_bin = toBinaryStr(ray_pos[1], coord_bit_length)
    z_bin = toBinaryStr(ray_pos[2], coord_bit_length)
    while isinstance(node, list) and depth < coord_bit_length:
        octant = int(z_bin[depth] + y_bin[depth] + x_bin[depth], 2)
        depth += 1
        oct_size /= 2
        aabb_min = aabb_min + (oct_size * np.array([int(x_bin[depth - 1]), int(y_bin[depth - 1]), int(z_bin[depth - 1])])).astype(int)
        aabb_max = aabb_min + np.array([oct_size - 1, oct_size - 1, oct_size - 1]).astype(int)
        node = node[octant]
    return node, oct_size, aabb_min, aabb_max

def stepRay(ray_pos, ray_dir, oct_size, aabb_min, aabb_max):
    while np.linalg.norm(ray_dir) < oct_size:
```

```
48          ray_dir *= 2
49      while not justOutsideAABB(ray_pos, aabb_min, aabb_max):
50          temp_position = ray_pos + ray_dir
51          temp_position = roundPosition(temp_position)
52          if withinAABB(temp_position, aabb_min, aabb_max) or
            ↪  justOutsideAABB(temp_position, aabb_min, aabb_max):
53              ray_pos = temp_position
54          ray_dir /= 2
55      return ray_pos
56
57  def apply_gamma_correction(color, gamma=2.2):
58      return np.clip(255 * (color / 255) ** (1 / gamma), 0, 255).astype(np.uint8)
59
60  total_pixels = im_height * im_width
61
62  with tqdm(total=total_pixels, desc="Rendering", unit="pixel") as pbar:
63      for y in range(im_height):
64          for x in range(im_width):
65              centered_x = x - (im_width / 2)
66              centered_y = (im_height / 2) - y
67              ray_dir = (cam_right * centered_x + cam_up * centered_y + cam_norm)
68              ray_dir = ray_dir / np.linalg.norm(ray_dir)
69              ray_pos = np.copy(cam_pos)
70              ray_pos = roundPosition(ray_pos)
71
72              world_size = 2**coord_bit_length
73              world_min = np.array([0, 0, 0], dtype=int)
74              world_max = np.array([world_size - 1, world_size - 1, world_size -
                ↪  1], dtype=int)
75              aabb_min = world_min
76              aabb_max = world_max
77              root = octree
78
79              while withinAABB(ray_pos, world_min, world_max):
80                  mid, oct_size, aabb_min, aabb_max = traverseTree(ray_pos, root,
                    ↪  world_size, world_min, world_max)
81                  if mid == 0:
82                      ray_pos = stepRay(ray_pos, ray_dir, oct_size, aabb_min,
                        ↪  aabb_max)
83                  if mid > 0:
84                      hit_normal = np.array([0, 0, 0])
85                      if ray_pos[0] == aabb_min[0]:
86                          hit_normal = np.array([-1, 0, 0])
87                      elif ray_pos[0] == aabb_max[0]:
88                          hit_normal = np.array([1, 0, 0])
89                      elif ray_pos[1] == aabb_min[1]:
90                          hit_normal = np.array([0, -1, 0])
91                      elif ray_pos[1] == aabb_max[1]:
92                          hit_normal = np.array([0, 1, 0])
93                      elif ray_pos[2] == aabb_min[2]:
94                          hit_normal = np.array([0, 0, -1])
95                      elif ray_pos[2] == aabb_max[2]:
96                          hit_normal = np.array([0, 0, 1])
97
98                      light_dir = cam_pos - ray_pos
99                      light_dir = light_dir / np.linalg.norm(light_dir)
```

```
                    brightness_factor = (np.dot(light_dir, hit_normal))**2

                    colour = np.array(material_table[mid]) * brightness_factor
                    colour = np.clip(colour, 0, 255).astype(np.uint8)
                    colour = apply_gamma_correction(colour)

                    break
            else:
                colour = [0, 0, 0]

            image[y, x] = colour

            pbar.update(1)

print("done")
img = Image.fromarray(image, 'RGB')
img.save('ray_traced_image.png')
img.show()
```

B.2 C++ Ray Tracer

```
#include <vector>
#include <cmath>
#include <bitset>
#include <string>
#include <tuple>
#include <iostream>
#include <fstream>

using namespace std;


int coord_bit_length = 10;

vector<int> cam_pos = {200, 300, 0};
vector<int> cam_norm = {0,0,230};
vector<int> cam_up = {0,1,0};
vector<int> cam_right = {1,0,0};

int im_width = 1000;
int im_height = 1000;

const vector<bitset<32>> octree = {
    bitset<32>(1),
    bitset<32>(0x80000000),
    bitset<32>(0x80000000),
    bitset<32>(0x80000000),
    bitset<32>(0x80000000),
    bitset<32>(0x80000000),
    bitset<32>(0x80000002),
    bitset<32>(0x80000003),
    bitset<32>(0x80000001)
};
```

```
// bit 31 is set for material
vector<vector<int>> material_table = {{0,0,0}, {255,255,255}, {0,255,0}, {0,0,255}, {255,0,0}, {255,255,0}};

vector<vector<int>> image;

bool withinAABB(vector<int> position, vector<int> aabb_min, vector<int> aabb_max){
    return position[0] >= aabb_min[0] && position[1] >= aabb_min[1] && position[2] >= aabb_min[2] && position[0] <= aabb_max[0] && position[1] <= aabb_max[1] && position[2] <= aabb_max[2];
}

bool justOutsideAABB(vector<int> position, vector<int> aabb_min, vector<int> aabb_max) {
    bool outside_x = (position[0] == aabb_min[0] - 1 || position[0] == aabb_max[0] + 1);
    bool outside_y = (position[1] == aabb_min[1] - 1 || position[1] == aabb_max[1] + 1);
    bool outside_z = (position[2] == aabb_min[2] - 1 || position[2] == aabb_max[2] + 1);

    bool within_x = (position[0] >= aabb_min[0] && position[0] <= aabb_max[0]);
    bool within_y = (position[1] >= aabb_min[1] && position[1] <= aabb_max[1]);
    bool within_z = (position[2] >= aabb_min[2] && position[2] <= aabb_max[2]);

    return (outside_x && within_y && within_z) || (within_x && outside_y && within_z) || (within_x && within_y && outside_z) || (outside_x && outside_y && within_z) || (outside_x && within_y && outside_z) || (within_x && outside_y && outside_z) || (outside_x && outside_y && outside_z);
}


tuple<bitset<32>, int, vector<int>, vector<int>> traverseTree(vector<int> ray_pos, int oct_size, vector<int> aabb_min, vector<int> aabb_max){
    int depth = 0;
    bitset<32> node(octree[0]);
    bitset<10> x_bin(ray_pos[0]);
    bitset<10> y_bin(ray_pos[1]);
    bitset<10> z_bin(ray_pos[2]);

    while(!node.test(31) && depth < coord_bit_length){

        string temp_str;
        temp_str += z_bin.to_string()[depth];
        temp_str += y_bin.to_string()[depth];
        temp_str += x_bin.to_string()[depth];
        int octant = stoi(temp_str, nullptr, 2);
        depth += 1;
        oct_size /= 2;

        aabb_min[0] = aabb_min[0] + (x_bin[coord_bit_length-depth]*oct_size);
        aabb_min[1] = aabb_min[1] + (y_bin[coord_bit_length-depth]*oct_size);
        aabb_min[2] = aabb_min[2] + (z_bin[coord_bit_length-depth]*oct_size);

```

```
76          aabb_max[0] = aabb_min[0] + oct_size-1;
77          aabb_max[1] = aabb_min[1] + oct_size-1;
78          aabb_max[2] = aabb_min[2] + oct_size-1;
79
80          int nodeIndex = int(node.to_ulong()) + octant;
81          node = octree[nodeIndex];
82      }
83      return make_tuple(node, oct_size, aabb_min, aabb_max);
84  }
85
86  vector<int> stepRay(vector<int> ray_pos, vector<int> ray_dir, int oct_size,
    ↪  vector<int> aabb_min, vector<int> aabb_max){
87      while(ray_dir[0]*ray_dir[0] + ray_dir[1]*ray_dir[1] + ray_dir[2]*ray_dir[2] <
        ↪  oct_size*oct_size){
88          ray_dir[0] *= 2;
89          ray_dir[1] *= 2;
90          ray_dir[2] *= 2;
91      }
92      while(!justOutsideAABB(ray_pos, aabb_min, aabb_max)){
93          vector<int> temp_pos = ray_pos;
94          temp_pos[0] = temp_pos[0] + ray_dir[0];
95          temp_pos[1] = temp_pos[1] + ray_dir[1];
96          temp_pos[2] = temp_pos[2] + ray_dir[2];
97          if(withinAABB(temp_pos, aabb_min, aabb_max) || justOutsideAABB(temp_pos,
            ↪  aabb_min, aabb_max)){
98              ray_pos = temp_pos;
99          }
100
101         ray_dir[0] = (abs(ray_dir[0])<=1) ? (ray_dir[0]>=0 ? 1 : -1) :
            ↪  ray_dir[0]/2;
102         ray_dir[1] = (abs(ray_dir[1])<=1) ? (ray_dir[1]>=0 ? 1 : -1)  :
            ↪  ray_dir[1]/2;
103         ray_dir[2] = (abs(ray_dir[2])<=1) ? (ray_dir[2]>=0 ? 1 : -1)  :
            ↪  ray_dir[2]/2;
104     }
105     return ray_pos;
106 }
107
108 vector<double> normaliseVector(const vector<double>& vec) {
109     vector<double> normalisedVec;
110     double magnitude = 0.0;
111
112     for(int_fast32_t i = 0; i < vec.size(); ++i){
113         magnitude += vec[i] * vec[i];
114     }
115
116     magnitude = sqrt(magnitude);
117
118     if(magnitude > 0.0){
119         for(int i = 0; i < vec.size(); ++i){
120             normalisedVec.push_back(vec[i] / magnitude);
121         }
122     }
123     return normalisedVec;
124 }
125
```

```
126 void writePPM(const string& filename) {
127     ofstream file(filename);
128     if (file.is_open()) {
129         file << "P3\n" << im_width << " " << im_height << "\n255\n";
130         for (const auto& row : image) {
131             for (int i = 0; i < row.size(); i += 3) {
132                 file << row[i] << " " << row[i + 1] << " " << row[i + 2] << " ";
133             }
134             file << "\n";
135         }
136         file.close();
137     } else {
138         cerr << "Unable to open file";
139     }
140 }
141 
142 int main(){
143     cout << "started" << endl;
144     for(int y = 0; y < im_height; y++){
145         for(int x = 0; x < im_width; x++){
146             int centered_x = x - (im_width / 2);
147             int centered_y = (im_height / 2) - y;
148             vector<int> ray_dir = cam_norm;
149             ray_dir[0] += (centered_x*cam_right[0] + centered_y*cam_up[0]);
150             ray_dir[1] += (centered_x*cam_right[1] + centered_y*cam_up[1]);
151             ray_dir[2] += (centered_x*cam_right[2] + centered_y*cam_up[2]);
152 
153             vector<int> ray_pos = cam_pos;
154 
155             int world_size = pow(2, coord_bit_length);
156             vector<int> world_min = {0, 0, 0};
157             vector<int> world_max = {world_size-1, world_size-1, world_size-1};
158 
159             int oct_size = world_size;
160             vector<int> aabb_min = world_min;
161             vector<int> aabb_max = world_max;
162 
163             bitset<32> node;
164             vector<int> colour;
165             while(withinAABB(ray_pos, world_min, world_max)){
166 
167                 tie(node, oct_size, aabb_min, aabb_max) = traverseTree(ray_pos,
                    ↪  world_size, world_min, world_max);
168                 unsigned int mid = node.to_ulong() & 0x7FFFFFFF;
169 
170                 if(mid==0){
171                     //cout << "air" << endl;
172                     ray_pos = stepRay(ray_pos, ray_dir, oct_size, aabb_min,
                        ↪  aabb_max);
173                 }
174                 if(mid>0){
175                     vector<int> hit_normal = {0,0,0};
176                     if (ray_pos[0] == aabb_min[0]){
177                     hit_normal = {-1, 0, 0};
178                     }
179                     else if(ray_pos[0] == aabb_max[0]){
```

```
180                         hit_normal = {1, 0, 0};
181                     }
182                     else if(ray_pos[1] == aabb_min[1]){
183                         hit_normal = {0, -1, 0};
184                     }
185                     else if(ray_pos[1] == aabb_max[1]){
186                         hit_normal = {0, 1, 0};
187                     }
188                     else if(ray_pos[2] == aabb_min[2]){
189                         hit_normal = {0, 0, -1};
190                     }
191                     else if(ray_pos[2] == aabb_max[2]){
192                         hit_normal = {0, 0, 1};
193                     }
194 
195                     vector<double> light_dir = {static_cast<double>(-ray_dir[0]),
                        ↪  static_cast<double>(-ray_dir[1]),
                        ↪  static_cast<double>(-ray_dir[2])};
196                     light_dir = normaliseVector(light_dir);
197 
198                     double brightness_factor = pow((hit_normal[0]*light_dir[0] +
                        ↪  hit_normal[1]*light_dir[1] + hit_normal[2]*light_dir[2])
                        ↪  , 1.4);
199 
200                     // cout << "Material: " << mid << " found at x: " << x << "
                        ↪  y: " << y << endl;
201                     colour = material_table[mid];
202                     double adjusted_colour_r = colour[0] * brightness_factor;
203                     double adjusted_colour_g = colour[1] * brightness_factor;
204                     double adjusted_colour_b = colour[2] * brightness_factor;
205                     colour = {static_cast<int>(round(adjusted_colour_r)),
                        ↪  static_cast<int>(round(adjusted_colour_g)),
                        ↪  static_cast<int>(round(adjusted_colour_b))};
206                     break;
207                 }
208             }
209             if(colour.empty()){
210                 colour = {0, 0, 0};
211             }
212 
213             image.push_back(colour);
214 
215         }
216     }
217 
218     writePPM("output.ppm");
219     cout << "done" << endl;
220 }
221 
```

C System Verilog Code

C.1 Square Root Approximation in SV

```
SHADING_2: begin
    magnitude_shading <= (light_dir_x**2 + light_dir_y**2 + light_dir_z**2)
        ↪ ;
    sqrt_input <= (light_dir_x**2 + light_dir_y**2 + light_dir_z**2);
end

SQRT_INIT: begin
    sqrt_res <= 0;
    sqrt_bit <= 1 << 30;
end

SQRT_ITER: begin
    if (sqrt_bit != 0) begin
        sqrt_temp = sqrt_res + sqrt_bit;
        if (sqrt_input >= sqrt_temp) begin
            sqrt_input <= sqrt_input - sqrt_temp;
            sqrt_res <= (sqrt_res >> 1) + sqrt_bit;
        end else begin
            sqrt_res <= sqrt_res >> 1;
        end
        sqrt_bit <= sqrt_bit >> 2;
    end
end

SQRT_ITER_BUFFER: begin
end

(State transitions below)

SHADING_2: begin // 19
    next_state = SQRT_INIT;
end
SQRT_INIT: begin // 20
    next_state = SQRT_ITER;
end
SQRT_INIT_BUFFER: begin // 21
    if (sqrt_bit > sqrt_input) begin
        next_state = SQRT_ITER;
    end else begin
        next_state = SHADING_3_INIT;
    end
end
SQRT_ITER: begin // 22
    next_state = SQRT_ITER_BUFFER;
end
SQRT_ITER_BUFFER: begin // 23
    if(sqrt_bit != 0) begin
        next_state = SQRT_ITER;
    end else begin
        next_state = SHADING_3_INIT;
    end
end
```

C.2 Minimisng Rounding Errors in SV

```
1  if(reg_ray_dir_x == 1) begin
2  underflow_x <= 1;
3      if(underflow_x == 0 || (reg_ray_dir_y == 0 || reg_ray_dir_z == 0) || (
           ↪ underflow_x == 1 && underflow_y == 1 && underflow_z == 1) )
           ↪ begin
4          reg_ray_dir_x <= 1;
5      end
6  end
7  else begin
8      reg_ray_dir_x <= reg_ray_dir_x >>> 1;
9  end
10
11 if(reg_ray_dir_y == 1) begin
12     underflow_y <= 1;
13     if(underflow_y == 0 || (reg_ray_dir_x == 0 || reg_ray_dir_z == 0) || (
           ↪ underflow_x == 1 && underflow_y == 1 && underflow_z == 1)) begin
14         reg_ray_dir_y <= 1;
15     end
16 end
17 else begin
18     reg_ray_dir_y <= reg_ray_dir_y >>> 1;
19 end
20
21 if(reg_ray_dir_z == 1) begin
22     underflow_z <= 1;
23     if(underflow_z == 0 || (reg_ray_dir_x == 0 || reg_ray_dir_y == 0) || (
           ↪ underflow_x == 1 && underflow_y == 1 && underflow_z == 1)) begin
24         reg_ray_dir_z <= 1;
25     end
26 end
27 else begin
28     reg_ray_dir_z <= reg_ray_dir_z >>> 1;
29 end
```

D FMEA of Subsystem Design

Project	FPGA Voxel Ray Tracer								Version	1					
Sub assembly	Ray Generation								Date	6/17/2024					
Prepared by	Kiara Rao								Page ____ of _____	1 of 1					
Approved by	Sne Samal								Notes						
Process Purpose	Potential Failure Mode	Severity	Potential Causes of Failure	Occurence	Process Control	Detection	RPN	Recommended Actions	Area / person responsible	Delivery date	Action taken	Severity	Occurence	Detection	New RPN
Generate rays from camera to scene	Incorrect ray direction	9	Incorrect direction vector calculation	5	Direction vector validation	4	180	Implement validation checks for direction vectors	Sne, Sri	06/06/2024	Added validation checks for direction vectors	9	3	2	54
Generate rays from camera to scene	Incorrect loop index update	8	Logic errors in FSM	4	FSM State Validation	3	96	Enhance FSM logic and test coverage	Sne, Sri	06/06/2024	Improved FSM logic and added test cases	8	2	2	32
Generate rays from camera to scene	Mismatched image dimensions	7	User input error, miscalculation	4	User input validation	3	84	Validate input parameters	Sne, Sri	06/06/2024	Added input parameter validation	7	2	2	28
Generate rays from camera to scene	Incorrect focal length	7	Incorrect magnitude of right/up vectors	4	Vector magnitude validation	3	84	Validate vector magnitudes	Sne, Sri	06/06/2024	Enhanced vector magnitude validation	7	2	2	28
Generate rays from camera to scene	Delayed ray generation	6	Inefficient FSM state transitions	3	Performance profiling	3	54	Optimise FSM transitions	Sne, Sri	06/06/2024	Optimised FSM state transitions	6	2	2	24
Generate rays from camera to scene	Incorrect initial state	8	Incorrect FSM initialisation	3	Initialisation validation	2	48	Validate FSM initial state	Sne, Sri	06/06/2024	Added FSM initial state validation	6	2	2	24