Imperial College London

Department of Electrical and Electronic Engineering

Mathematical Accelerator

Second Year Engineering Design Project
`ELEC50015`

Team Members:

Theodore Shah : 02380127
Lucas Venetz : 02387237
Showrob Das : 02220476
Leo Swataynon : 02397104
Jaber Ahmed : 02274584

Github. June 2025

Abstract

For the 2025 second-year Engineering Design Project, a real-time visualisation system was developed to render scenes using the ray marching technique on the PYNQ-Z1 FPGA. Ray marching is widely used for rendering implicit surfaces and volumetric effects but is typically reliant on high-performance GPUs. This project demonstrates how synthesised hardware acceleration on an FPGA can be used to offload the computational burden and enable interactive rendering.

The core hardware design, written in SystemVerilog, includes a pipelined fixed-point engine capable of evaluating signed distance functions (SDFs) in real-time. Look-Up Tables (LUTs) were used to optimise computational performance, and the module was thoroughly verified through extensive testbenching. The final design was synthesised and deployed on the PYNQ-Z1, making efficient use of its hardware resources.

To support user interactivity, a custom Python-based user interface was developed using Pygame. This interface allows users to adjust rendering parameters, which are transmitted to the FPGA over a UART serial connection. A serial receiver module was implemented on the FPGA to parse the incoming data and apply the updated parameters in hardware.

Throughout the project, careful attention was given to the integration between software and hardware, ensuring reliable communication and stable visual performance. Testing confirmed that the system met the functional requirements for real-time interaction and correct visual output. This project has enhanced our skills in digital design, hardware-software integration, and collaborative engineering, while also demonstrating the potential of FPGAs for educational and creative computational visualisation tasks.

Contents

1 Introduction

This report documents the development of a second-year engineering design project focused on building a near real-time mathematical visualisation tool using the PYNQ-Z1 FPGA board. The project integrates concepts from across the course, including digital systems, algorithm design, hardware-software integration, and user interface development.

The goal was to construct an interactive system that renders complex mathematical functions visually, using FPGA acceleration to offload computational tasks and enhance performance. The core rendering technique chosen was ray marching, due to its ability to produce visually compelling results while offering a structure well suited for parallelism and hardware implementation.

Throughout the project, the team followed a structured design process. This began with initial research and requirements capture, followed by hardware development, software construction, and full system integration. The result is a custom-built hardware pipeline implemented in SystemVerilog, paired with a responsive Python-based UI developed in Pygame, with communication between the two facilitated via a UART interface.

This report outlines each stage of the project, detailing design decisions, implementation strategies, encountered challenges, and the solutions employed. The final sections evaluate system performance and functionality, reflecting on how the project fulfilled its educational and technical objectives.

2 Planning and Design

2.1 Planning

The team placed strong emphasis on early planning to ensure smooth coordination and sustained progress. A collaborative environment was adopted from the beginning, with regular meetings held to share ideas, review developments, and allocate tasks efficiently.

The work was structured around weekly goals that supported the project's broader milestones, such as hardware development, user interface design, and more. Responsibilities were assigned based on individual strengths and interests, ensuring effective contributions across both hardware and software domains.

A shared daily project log was maintained to document progress, flag blockers, and support accountability. GitHub was used as the central version control platform to manage collaborative development and ensure code integrity. The team adopted an iterative approach, enabling regular testing, timely feedback, and ongoing refinement of the design.

Testing and documentation duties were shared across the group, and all members contributed to debugging and integration. This structure supported a steady pace of development and ensured that each part of the system evolved cohesively.

Date	Purpose
20/05	Initial discussion of project brief and ray marching concept
21/05	System architecture planning and task allocation
23/05	Early hardware design and UI testing
26/05	Midweek check-in and parameter selection discussion
28/05	Hardware module design review and debugging session
30/05	Python UI progress review and serial protocol planning
02/06	**Interim Presentation**
05/06	System-level testing and image output debugging
09/06	Final integration review and project log check
13/06	Final polish, formatting review, and submission planning
16/06	**Report deadline**
19/06	**Demo day**

Table 1: Record of Group Meetings

To provide structure to this process, the group established clear weekly goals spanning hardware, software, and integration tasks. These guided the project through its design and implementation phases. The agreed milestones for each development week are summarised below.

Week-by-Week Project Milestones

Week 1 – Planning / Setup

- Defined system architecture across hardware, UI, and communication layers
- Assigned responsibilities among team members
- Researched ray marching and fixed-point design trade-offs
- Set up GitHub repository and simulation/testbench environment
- Installed Vivado, PYNQ, and Python dependencies across devices

Week 2 – Architecture and Prototyping

- Finalised ray marching algorithm in Python and C++
- Began translating algorithm into Verilog for FPGA implementation
- Planned key UI functionality and parameter options
- Outlined UART serial communication protocol

- Allocated hardware modules and prepared interim presentation

Week 3 – Implementation

- Developed SystemVerilog modules based on the proposed architecture
- Built and tested an interactive UI using Pygame
- Implemented UART-based transmission between UI and FPGA
- Created testbenches for hardware modules and began system integration

Week 4 – Testing and Finalisation

- Completed full system integration and functionality checks
- Conducted testing for latency, frame rate, and visual correctness
- Refined UI layout and optimised hardware behaviour
- Finalised project report and prepared for demonstration

Task allocation was based on individual strengths, experience, and preferences, with responsibilities naturally divided across hardware development, software interfacing, and integration. While some members focused more on digital design and testing, others contributed to embedded programming, debugging, or user interface development. This flexible approach allowed each team member to take ownership of key components while still supporting one another during complex or cross-cutting tasks. All work was centralised on a shared GitHub repository to ensure collaboration and version control across the full system.

Team Member	Course	Core Responsibilities
Theo	EIE	Software implementation, Mid Level Module Development, VCD Scripts and Testbenches
Lucas	EIE	Lower Level and Top Level Module Development, Parallelisation, and Debugging
Showrob	EIE	Software implementation, Mid Level Module Development, and Debugging
Leo	EIE	Top Level Module Development, Pipelining for timing constraints, FPGA resource optimisation
Jaber	EEE	Top Level Module Development, UI Development and Integration

Table 2: Responsibility Delegation

The Gantt Chart detailing the planned project timeline can be seen in Figure 1.

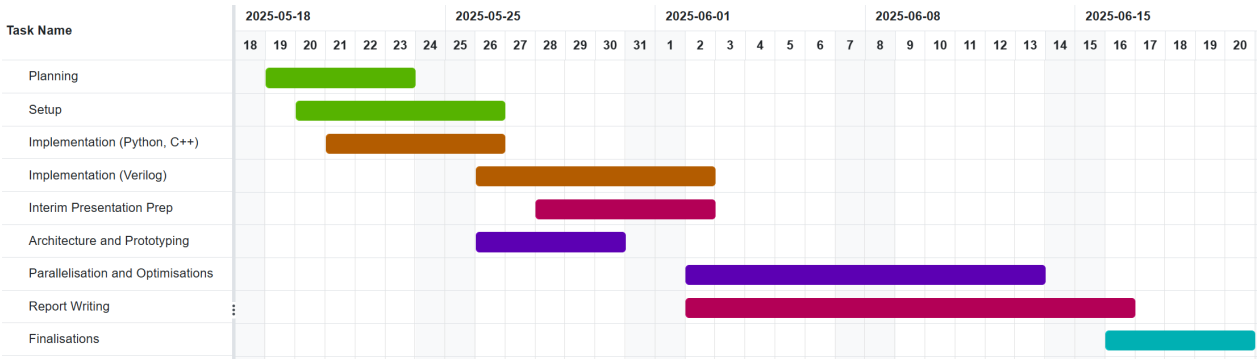

Figure 1: Gantt chart showing the planned project timeline

2.2 Design

2.2.1 Overview

To ensure that the project progressed toward a well-defined outcome, a robust set of development goals was established early on. These goals served as a continuous reference point throughout both the design and build stages.

In the opening week, team discussions played a vital role in forming a unified understanding of the project's purpose. These discussions were essential to align expectations and define concrete requirements. The agreed-upon requirements focused on ensuring that the system was accessible, engaging and interactive, reflecting both its intended use and the target audience. With these foundations in place, the team proceeded into an iterative cycle of design and development; ongoing reflection and evaluation were integral to ensuring that progress remained aligned with the original goals.

2.2.2 Ray Marching

Ray marching was selected as the target algorithm for this project due to its strong visual appeal and conceptual depth. It serves not only as a captivating rendering technique but also as an educational gateway into topics such as computational geometry, signed distance fields (SDF's) and real-time graphics on constrained hardware.

Ray marching operates by progressing rays through a 3D scene, using distance estimations to incrementally move through space until either an object is hit or the ray is determined to be a background ray (maximum number of steps is exceeded). To determine whether a ray "intersects" with an object, we use SDF functions to calculate the shortest distance between a point in space reached by marching our rays and the closest surface of an object. We continue marching that resulting distance until we've "hit" the surface and can draw the resulting pixel.

More simply, the ray marching process can be described as follows in Figure 2.

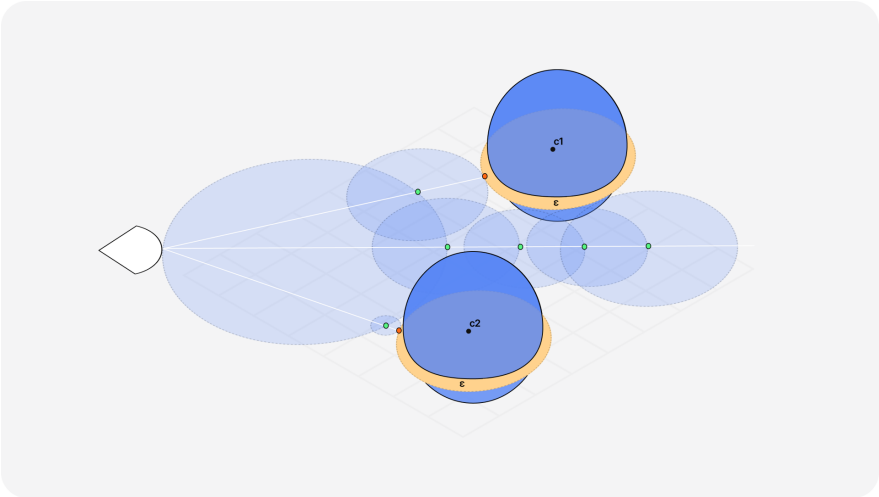

Figure 2: Ray marching diagram of two spheres [1].

- Each step of the raymarching goes as far as the shortest distance to a defined object (green points show where new raymarching process begins).
- If the distance between the point and the surface is below some constant ϵ this is considered a hit.
- If there is no hit, the process is continued from the new point (in green again) until a maximum number of steps is reached.

This technique allows for the rendering of complex implicit surfaces and volumetric effects with relatively simple formulas. Its reliance on mathematical functions rather than traditional meshes makes it highly suitable for FPGA acceleration, where fixed-function pipelines can exploit the algorithm's regular structure.

The use of signed distance functions means each pixel can be processed independently, making ray marching an ideal candidate for an FPGA accelerator. Ray marching also has strong potential educational value. It enables students to explore mathematical representations of geometry, understand the fundamentals of rendering pipelines, and perhaps even appreciate the performance benefits of hardware acceleration. By implementing ray marching on an FPGA, this project bridges the gap between high-level graphical concepts and low-level digital design, showcasing the power and flexibility of reconfigurable computing in a visually compelling way.

2.2.3 Requirements Capture

Prior to implementation, core functional and technical requirements were systematically defined to inform the design process. Table 3 summarises these specifications alongside their corresponding implementation solutions.

Requirement	Implementation Approach
Visualise a computationally intensive mathematical function	The application employs ray marching to evaluate signed distance functions (SDFs) for 3D scene rendering at a resolution of 640×480. Our aim is to have most of our objects displayed at a minimum of 20 frames per second. This allows us to demonstrate near real-time performance through FPGA acceleration.
Embarrassingly parallel computation for each pixel	Each pixel's ray is processed independently, leveraging the inherent parallelism of the FPGA to enable simultaneous execution across all pixels.
Exploits hardware acceleration effectively	Distance estimation and scene logic are offloaded to the programmable logic (PL) section of the FPGA, enabling parallel computation and significantly reducing CPU workload.
Modular and Scalable design	The architecture is partitioned into distinct modules ensuring scalability and ease of future enhancements. This modularity will be further elaborated in subsequent sections.
User Interface (UI) Requirements for Enhanced Educational Visualisation	The user-friendly interface combines intuitive sliders for camera and scene control with a clean layout, contextual tooltips, and practical presets.

Table 3: System Requirements and Implementation Approach

2.2.4 Hardware - Software Partitioning

The project was divided into hardware and software components to balance performance and flexibility. The ray marching algorithm, including distance estimation and loop control, was implemented in hardware to take advantage of parallel processing. The Python-based user interface handled all user interaction, including adjusting rendering parameters. Communication between the two domains was achieved using a UART connection, allowing the software to send updates to the hardware module with low-latency after user confirmation. This interactive approach ensured that parameter changes, including large shifts in camera or scene configuration, could be applied efficiently while keeping the system architecture modular and responsive.

2.2.5 Risk Assessment / FMEA

Throughout development, several measures were taken to minimise the risk of system failure. Key risks included unreliable UART communication, pipelining errors leading to incorrect timing, and mismatches between the software interface and the hardware protocol. To mitigate these, UART logging and debugging were used to verify serial transmission reliability,

and integration checks were performed throughout development. Simulation testbenches were used to validate pipelined hardware modules, while frequent synthesis and timing analysis ensured FPGA resource limits and timing constraints were respected. Version control was also maintained to manage stable builds and isolate regressions.

3 C++ Implementation

Prior to any System Verilog code design, we created a C++ version implementation. We did this for two reasons:

- To be able to compare the software implementation to the FPGA hardware design
- To better understand how to divide and allocate HDL modules and to verify logic

It has to be noted that the C++ code is a single cycle implementation, by using a nested for loop to iterate through every individual pixel. The C++ code graphical display is done by the SDL2 library.

```
for (int y = 0; y < height; ++y) {
            for (int x = 0; x < width; ++x) { //Raster pattern
                //Image plane height: scale * 2
                //Image plane width: aspect_ratio * scale * 2
                //(x + 0.5f) / width  => Maps the centre of the pixel to a range of [0,1] (Centre of a pixel is x+0.5)
                //2 * ... - 1   => Maps the pixels to a range of [-1, 1] (i.e leftmost pixel has value -1, and rightmost has value of 1)
                float px = (2.0f * ((x + 0.5f) / width) - 1.0f) * aspect_ratio * scale; //x direction vector of ray for this pixel
                float py = -(2.0f * ((y + 0.5f) / height) - 1.0f) * scale; //y direction vector of ray for this pixel
                //py is calculated in a similar way but the mapping of [-1, 1] needs to flipped as vertical pixel number increases as you go downwards the screen

                vec3 ray_dir(px, py, 1); //This ray goes 1 unit forward in the camera's local +z direction. px and py point the ray in the correct direction
                ray_dir = ray_dir.normalise(); //Normalise so that we can easily project it to certain distances

                float dist = raymarch(camera_pos, ray_dir); //Raymarch this certain ray
                vec3 p = camera_pos.addition(ray_dir.scalarMul(dist)); //p is the position of the ray after it is done ray marching. p = a + lambda * d

                float brightness = 0.0f;
                if (dist < 100.0f) { //If ray hit an object, calculate how lit up that object surface point is
                    vec3 normal = getSurfaceNormal(p); //Calculate normal vector of the surface at point p
                    vec3 lightDirection = lightPosition.addition(p.scalarMul(-1)).normalise(); //Calculate normalised light direction vector at point p
                    float diffuseCoeff = std::max(dot(normal, lightDirection), 0.0f); //Dot product both vectors to get the coeff that determines how "lit up" that point is. Read the blog to see how this works
                    brightness = diffuseCoeff; //DiffuseCoeff can take any value between 0 and 1
                }
                //If it didn't hit an object, then the brightness of that ray/pixel is 0
```

Figure 3: C++ Main pixel loop logic

```
float raymarch(vec3 ro, vec3 rd) {
    float rayDist = 0.0f;
    for(int i = 0; i < MAX_STEPS; i++) {
        vec3 p = ro.addition(rd.scalarMul(rayDist));
        float dS = scene(p);
        rayDist += dS;
        if(rayDist > MAX_DIST || dS < SURFACE_DIST) break;
    }
    return rayDist;
}
```

Figure 4: C++ Raymarcher Function

4 Hardware Design

4.1 Accelerator

4.1.1 Overview

The overall hardware design consists of multiple modules, each with their own characteristic function. An overall block design of the top level can be seen in Figure 16. Valid signals are used throughout every module to indicate when inputs and outputs are correct, while registers between modules are implemented to satisfy timing constraints at high clock frequencies.

4.1.2 Register File And Control Parameters

We wanted to accommodate as much user control as possible to our design. We have implemented multiple user interface features, such as:

- Rotation in all axis
- Variable zoom
- Scene selection
- Colour choice of scene

We have 8, 32 bit registers to control the parameters used in the modules. By considering the ranges and resolutions required by each parameter, we were able to fit all of the parameters in their respective fixed point representations within the 256 bits available. A list of parameters to be controlled is given:

Parameter	Component	Bit Width
Camera Forward Direction	x	18
	y	18
	z	18
Camera Right Direction	x	18
	y	18
	z	18
Camera Up Direction	x	18
	y	18
	z	18
Light Position	x	10
	y	10
	z	10
Colour Coefficient	R	10
	G	10
	B	10
Normal Factor Coefficient	–	32
Object/Scene Selection	–	2

Table 4: Bit allocation of key rendering parameters

The camera direction vectors are used to implement rotation in all axis, while a dynamic light position can be implemented using the light position coordinates. Changing the scene colour can be done by altering the RGB coefficients, while the normal factor coefficient is used to control the zoom. All of the previous parameters are represented using varying fixed point representation to best suit range and resolution constraints. The Object/Scene selection uses 2 bits to select between 4 different scene SDF's. The fixed point representations and the reasons for the representation choice are given below.

Use Case	Format	Total Bits	Resolution	Min Value	Max Value
Direction Vectors	Q1.17	18	0.000008	-1	0.999992
Light Position Vectors	Q7.3	10	0.125000	-64	63.875000
Colour Coefficients	Q0.10 (Unsigned)	10	0.000977	0	0.999023
Normal Factor Coefficient	Q8.24 (Unsigned)	32	0.000001	0	256.000000

Table 5: Fixed-point formats used for user control interface parameters

Time consideration was taken on the decision as to whether to pass in either 2 or 3 of the camera direction vectors. By passing in only 2 camera direction vectors, such as the forward and right components, you can determine the third component by computing the cross

product of the 2 input direction vector parameters. This comes with the advantage of being able to represent the direction vectors with a greater number of bits, 27 bits, as opposed to 18 bits. A greater number of bits to represent the vector fixed point components allows for either a greater range or better fractional resolution. The disadvantage of this approach is that 5 clock cycles are required to compute the cross product of the third vector and to normalise it.
We determined that the trade-off between better vector representation was not worth the increase in clock cycles, and decided on computing all direction vectors in software before writing all 9 parameters to the registers. We decided on representing both the light position coordinates and colour coefficients using 10 bits in their own fixed point representations to fit their respective ranges. The resolutions are sufficient to achieve accurate approximations. Both are strictly positive coefficients, so are unsigned fixed point representations, allocating an extra bit.
Table. 6 showcases how the parameters fit into the 8 registers available.

Register	**Bit Field Partition (MSB → LSB)**
Reg 0	Light.x[31:22] | Light.y[21:12] | Light.z[11:2] | objSel[1:0]
Reg 1	Normal Factor[31:0]
Reg 2	Forward.x[31:14] | Forward.y[13:0]
Reg 3	Forward.y[31:28] | Forward.z[27:10] | ColourCoeff.R[9:0]
Reg 4	Right.x[31:14] | Right.y[13:0]
Reg 5	Right.y[31:28] | Right.z[17:10] | ColourCoeff.G[9:0]
Reg 6	Up.x[31:14] | Up.y[13:0]
Reg 7	Up.y[31:28] | Up.z[27:10] | ColourCoeff.B[9:0]

Table 6: Visual breakdown of 32-bit register contents

4.1.3 Scene Query

The Scene Query module is a separate module used to implement signed distance functions, or SDF's, which are functions that store the geometric information for any scene. SDF's take in any 3D input coordinate and return the closest distance to any object surface as a signed fixed-point value. Common shapes have their own characteristic SDF's. Some examples are given below, where p is the input coordinate [2]:

$$\text{sdfSphere}(p, \text{halfwidth}) = \|p\| - \text{radius}$$

$$\text{sdfCube}(p, \text{halfWidth}) = \max\big(|p_x|, |p_y|, |p_z|\big) - \text{halfWidth}$$

SDF's can be used to render infinitely detailed geometry such as mandelbulbs and/or infinitely expansive scenes such as tunnels.

SDF's have varying complexity to them, and so therefore take a different number of clock cycles to evaluate. The SDF of a cube can be evaluated in a singular clock cycle, whereas for a sphere, calculating the magnitude of a vector requires the use of the inverse square module, which itself takes 3 clock cycles.

$$\|p\| = \sqrt{p_x^2 + p_y^2 + p_z^2} = \big(p_x^2 + p_y^2 + p_z^2\big) \times \text{inv\_sqrt}\big(p_x^2 + p_y^2 + p_z^2\big).$$

We took into account DSP and LUT utilisation, as well as timing constraints to selectively choose what different SDF's to implement in our design. We chose to include the following scenes and SDF's:

Scene	**Clock Cycles**
Sphere	5
Cube	1
Cube Lattice	2
Menger Sponge	8

Table 7: Clock cycle cost for each scene's SDF implementation

A object select signal controls which SDF to query at the present time, with valid in and valid out signals to indicate correct inputs and outputs respectively.
Lattice patterns can be generated by "folding" world space points into a single cell at the origin and evaluating this SDF instead. The Menger Sponge SDF has been pipelined to satisfy the timing constraints, due to it's more complicated combinational logic and complex operations.

```
module sdfInfiniteCube (
    input logic clk,
    input logic valid_in,
    input vec3 point,
    input fp radius,
    output fp outputDistance,
    output logic valid_out
);
    vec3 hhh;
    assign hhh = make_vec3(`FP_HALF, `FP_HALF, `FP_HALF);
    always_ff @(posedge clk) begin
        if(valid_in) begin
            outputDistance <= fast_cd(vec3_sub(vec3_fract(vec3_add(
               point, hhh)), hhh), `FP_QUARTER);
        end
        valid_out <= valid_in;
    end
endmodule
```

Figure 5: Cube Lattice SDF module

4.1.4 Ray Unit

The Ray Unit module is comprised of a Ray Generator and a Ray Unit. These two modules work sequentially with each other, with the Ray Generator acting first, and once finished, passes on information to the Ray Marcher to handle. A design of the Ray Unit can be seen in Figure 16. , within the top level module diagram.

Ray Generator

The Ray Generator module takes in a pixel coordinates and camera orientation vectors as inputs to determine a direction vector of the corresponding ray that intercepts the pixel specified. The ray direction is then passed from the Ray Generator module to the Ray Marcher module, with a valid signal to start the ray marching process. The module uses Q11.11 format to represent x and y coordinates, which aligns with screen resolution of 640x480.

To generate rays from each pixel, the pixel coordinates are converted into normalised range of [-1, 1], known as Normalised Device Coordinates(NDC). Applying NDC ensures that our ray generation module is consistent with computer graphics convention also easier to adapt for changes such as different screen resolution. Afterwards, the ray direction is calculated from camera space to world space using the rotation vector composed of camera's basis vector as shown in the equations below.

$$\begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} = \begin{bmatrix} \text{right}_x & \text{up}_x & -\text{forward}_x \\ \text{right}_y & \text{up}_y & -\text{forward}_y \\ \text{right}_z & \text{up}_z & -\text{forward}_z \end{bmatrix} \cdot \begin{bmatrix} ndc_x \\ ndc_y \\ 1 \end{bmatrix}$$

$$\begin{aligned} r_x &= ndc_x \cdot \text{right}_x + ndc_y \cdot \text{up}_x - \text{forward}_x \\ r_y &= ndc_x \cdot \text{right}_y + ndc_y \cdot \text{up}_y - \text{forward}_y \\ r_z &= ndc_x \cdot \text{right}_z + ndc_y \cdot \text{up}_z - \text{forward}_z \end{aligned}$$

The camera basis vectors are derived from its position and target. The forward vector is the normalised direction from the camera to the target. The right vector is the normalized cross product of the forward vector and world up [0,1,0], and the up vector is the cross product of right and forward.

Since normalisation and cross product operations are area consuming, the camera forward and up vectors were precomputed in software and passed to the hardware as registers. The camera right vector was then calculated in hardware using the cross product of forward and up. The decision to compute camera right vector in hardware was made because only 8 registers available, camera up and forward takes 6 registers.

Ray Marcher

The Ray Marcher module carries out the iteration of ray marching for a particular ray. Each ray is projected from the ray origin, through a certain pixel on the screen, and into the scene. The Ray Marcher takes a dynamic number of clock cycles to complete, as different rays take a different number of steps before reaching a termination condition. Rays that are projected into the background take more steps, and therefore more clock cycles, than rays that hit an object earlier. A finite state machine is used to model the process. A flowchart of the ray marcher finite state machine is shown in Figure 6.

The state machine consists of 3 states; IDLE, STEP, DONE. The state machine starts in the IDLE state upon a reset signal and waits for an incoming valid signal from the Ray Generator. Once the incoming valid signal is asserted, the state changes to an STEP state, where upon every step iteration, the Ray Marcher module queries the scene. By querying the scene with the current end position of the ray, the submodule returns the closest distance to any object in the scene. There are 3 termination conditions to exit the STEP state of the ray marching process:

1. The length of the ray, or the distance travelled of the ray is greater than the `MAX_DIST` predefined parameter.
2. The number of step iterations the ray has gone through is greater than the `MAX_STEPS` predefined parameter.
3. The closest distance to an object in the scene is smaller than the `SURFACE_DIST` predefined parameter. In this case, we can determine that the ray has hit an object, and we can set the "hit" flag high.

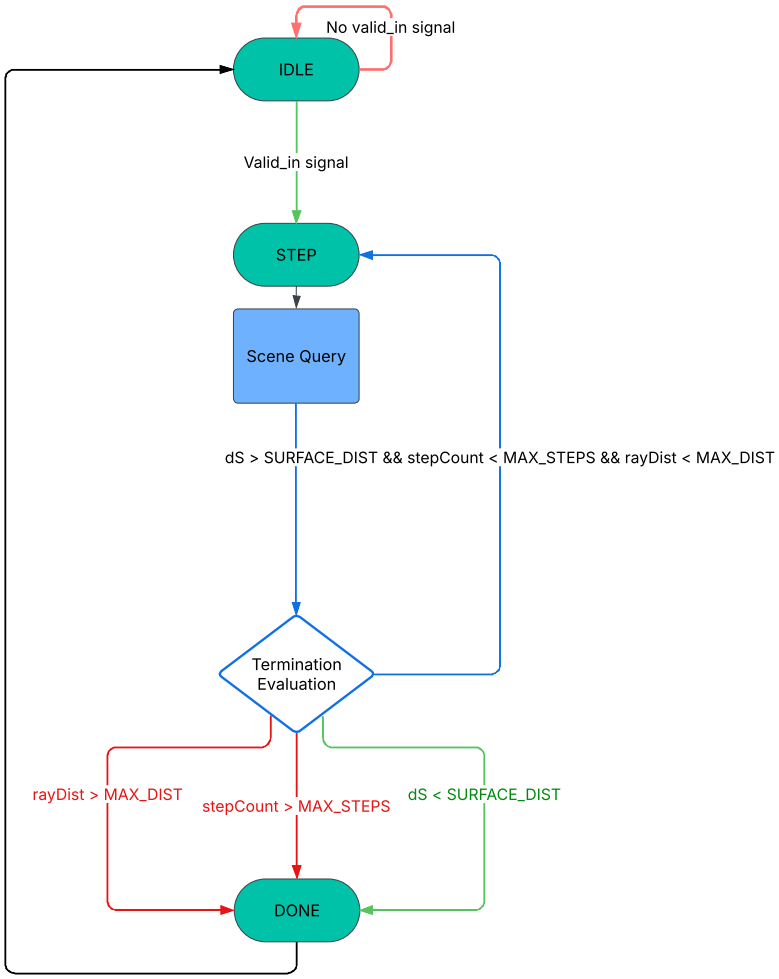

Figure 6: Flowchart diagram of the ray-marching state machine.

The surface distance parameter is predefined as 0.01, and describes how close a ray needs to be to an object to be determined to have intercepted an object and counted as a "hit". This parameter is necessary to terminate the ray marching process on rays that are not normal to an object surface. By setting the parameter to a very small value, we can achieve a truer shape and scene, at the cost of taking a greater number of clock cycles before a "hit" is determined. For objects of radius 1, we determined that the value of 0.01 was sufficient in obtaining an accurate image while not taking an excessive number of clock cycles.

```
STEP: begin
        stepVec = vec3_scale(rayDir, rayDist);
        position = vec3_add(rayOrigin, stepVec);
        if(submodule_finished) begin
            if (rayDist > MAX_DIST || stepCount >= MAX_STEPS) begin
                hit_internal = 1'b0;
                nextState = DONE;
            end
            else if (dS < SURFACE_DIST) begin
                hit_internal = 1'b1;
                nextState = DONE;
            end
            else begin
                nextState = STEP;
                submodule_valid_in_next = 1'b1;
            end
        end
        else begin
            nextState = STEP;
            submodule_valid_in_next = 1'b0;
        end
        point_next = position;
    end
DONE: begin
    nextState = IDLE;
    point_next = '0;
end
```

Figure 7: Ray Marcher FSM code snippet

If all of the termination conditions are not met, the ray marcher returns to the STEP state and undergoes another step iteration. However, if any of the termination conditions are met, the ray marcher proceeds to the DONE state. In this state, the end position of the ray, a flag to indicate if the ray "hit" an object, and a valid signal, are asserted as outputs of the ray marcher module, and therefore outputs of the ray unit. Following the DONE state, the finite state machine returns to the IDLE state to await another incoming valid signal of another different ray.

4.1.5 Buffer Manager

The pixel packer module is designed to take one pixel at a time in raster order. This poses an issue to our design as our ray marchering process has variable computation time for each ray direction, meaning we would have to deal with out-of-order completion when rendering pixels in parallel.

We decided the best way to accomplish this was to have a small circular buffer for each unit, and implement an interleaved pixel assignment pattern (i.e Ray Unit 0 processes pixels 0,4,8..., Ray Unit 1 processes pixels 1,5,9...). This approach also offers a more balanced load distribution as adjacent pixels are more likely to have similar computational times. A simple pixel counter determines which Ray Unit should provide the next pixel in raster order.

Once a Ray Unit has completed its pixel, the result is stored at the current write location, and the write pointer is advanced. The Ray Unit is then immediately assigned to its next pixel according to the interleaved assignment pattern. Read pointers for each buffer are only advanced once pixels are consumed for raster output. Pointer wrapping was implemented using the modulo operator with the buffer depth.

```
buffer_full = ((write_ptrs[j] + 1) % BUFFER_DEPTH) == read_ptrs[j];
if(valid_out[j] && !buffer_full) begin
    buffers[j][write_ptrs[j]].surface_point <= surface_points[j];
    buffers[j][write_ptrs[j]].hit <= hits[j];
    buffers[j][write_ptrs[j]].valid <= 1'b1;
    // Wrap around
    write_ptrs[j] <= (write_ptrs[j] + 1) % BUFFER_DEPTH;
    pixel_assignments[j] <= pixel_assignments[j] + RAY_UNITS;
end
```

Figure 8: Write Pointer Logic

A 3-state FSM is also used to check buffer space for each Ray Unit so that work on a pixel can start independently as soon as a unit is free and its buffer has capacity. If a Ray Unit is free and the buffer has space (based on the `buffer_can_accept[i]` flag), it is put in a `BUSY` state. When computation is complete, the unit goes to the `IDLE` state if the buffer has space. Otherwise it is held at the `WAITING` state to hold the result data until buffer space is available.

Our initial approach for a round-robin buffer selection was to use an FSM to cycle between Ray Units in raster order. However, we quickly realised that FSM state transitions would require sequential state updates and unnecessary synchronization complexity, and would require a new state for each additional Ray Unit. The modulo approach would achieve the same result with a single operation.

```
next_unit = pixel_counter % RAY_UNITS
pixel_ready = buffers[next_unit][read_ptrs[next_unit]].valid
```

```
next_unit = pixel_counter % RAY_UNITS;
        pixel_ready = buffers[next_unit][read_ptrs[next_unit]].valid
           ;
        if(pixel_ready) begin
            surface_point_out <= buffers[next_unit][read_ptrs[
               next_unit]].surface_point;
            hit_out <= buffers[next_unit][read_ptrs[next_unit]].hit;
            pixel_valid_out <= 1'b1;

            buffers[next_unit][read_ptrs[next_unit]].valid <= 1'b0;
            read_ptrs[next_unit] <= (read_ptrs[next_unit] + 1) %
               BUFFER_DEPTH;
            pixel_counter <= pixel_counter + 1;
        end
```

Figure 9: Read Pointer Logic

```
generate
    for (genvar i = 0; i < RAY_UNITS; i++) begin : space_check
        always_comb begin
            buffer_can_accept[i] = ((write_ptrs[i] + 1) %
               BUFFER_DEPTH) != read_ptrs[i];
        end
    end
endgenerate
```

Figure 10: Full Logic

To ensure we were maintaining raster order output, we created mock Ray Units that would simulate a random processing delay. We would manually monitor the read/write pointers for each buffer and ensure no buffer overflow would occur. The testbench for this module can be found in `tb_buffer_manager.sv`.

We also considered using a single shared buffer with coordinate-indexing where units would be constrained to work within certain scanline windows. (Without a scanline the buffer would have to accommodate for every pixel which would be far too large). While this approach would improve memory utilization, there would be no change in the speed-up, and we determined the added complexity and potential for access conflicts outweighed the benefits.

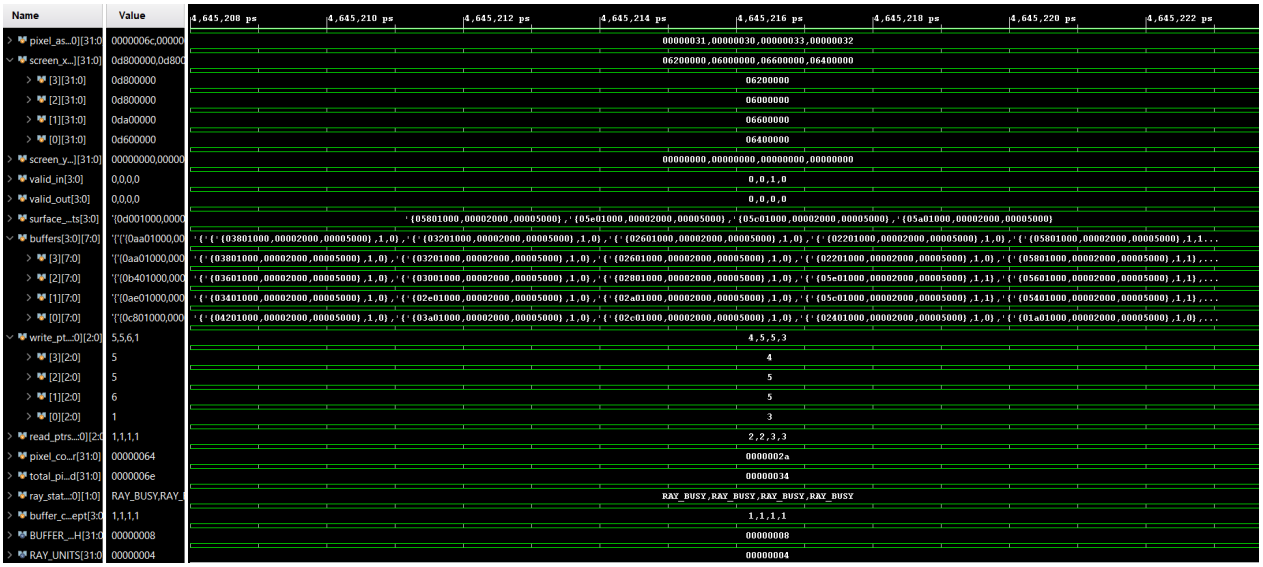

Figure 11: Waveform of Ray Unit Buffering.

4.1.6 Object Surface Vectors

The outputs of the Ray Unit are passed into the getSurfaceVectors module, which is an intermediate module between the Ray Unit and the Shading module. In the case of a ray "hit" with an object, the module calculates both the normal vector and light vector at that certain collision surface point. These vectors are required by the following shading module to calculate the shading coefficients to colour in the pixel. Both vectors are normalised using the inverse square module.

$$\hat{p} \;=\; \frac{p}{\sqrt{p_x^2 + p_y^2 + p_z^2}} \;=\; p \;\times\; \text{inv\_sqrt}\big(p_x^2 + p_y^2 + p_z^2\big).$$

Computing the normal of a surface can be done using the gradient of the SDF at the required surface point. Surface normals must be perpendicular to the surface, so therefore they coincide with the field's gradient. We will exploit differentiation from first principles to obtain an accurate approximation, where h is the small difference used. We settled on a value of

0.01 for h. We chose to evaluate using techniques that use central differences, as this ensures the normals follow the surface without any directional bias due to uneven surface changes. We used a technique that follows the shape of a tetrahedron:

$$\begin{aligned}\nabla \,\text{sceneQuery}(p) \;\approx\; & (1,-1,-1)\,\text{sceneQuery}\big(p+(h,-h,-h)\big)\\ & +(-1,-1,\,1)\,\text{sceneQuery}\big(p+(-h,-h,\,h)\big)\\ & +(-1,\,1,-1)\,\text{sceneQuery}\big(p+(-h,\,h,-h)\big)\\ & +(\,1,\,1,\,1)\,\text{sceneQuery}\big(p+(h,\,h,\,h)\big)\end{aligned}$$

This approach requires the evaluation of 4 scene queries. We parallelised this by instantiating 4 sceneQuery modules, to quarter the number of required clock cycles. Normalising produces the normal vector.

$$\mathbf{n}(p) \;=\; \frac{\nabla \,\text{sceneQuery}(p)}{\big\|\nabla \,\text{sceneQuery}(p)\big\|}$$

```
always_comb begin
      if (stage1_valid) begin
          if (hit_in_2) begin
              pos_xyy = vec3_add(p_2, vec3_scale(h_xyy, eps));
              pos_yxy = vec3_add(p_2, vec3_scale(h_yxy, eps));
              pos_yyx = vec3_add(p_2, vec3_scale(h_yyx, eps));
              pos_xxx = vec3_add(p_2, vec3_scale(h_xxx, eps));
          end
          reg_hit_in_1 = hit_in_2;
          reg_p_1 = p_2;
      end
  end

   sceneQuery getClosestDist_xyy (
      .clk(clk),
      .rst(rst),
      .valid_in(stage1_valid),
      .pos(pos_xyy),
      .obj_sel(obj_sel),
      .closestDistance(dS_xyy),
      .valid_out(module_finished_xyy)
  );
```

Figure 12: Normal vector calculation with 1 of 4 the *sceneQuery* modules

The second required vector, the light direction vector, can be calculated from the difference between the light position and the surface point.

$$\mathbf{L}(p) \;=\; \frac{\mathbf{p}_{\text{light}} - \mathbf{p}}{\big\|\mathbf{p}_{\text{light}} - \mathbf{p}\big\|}$$

To satisfy timing constraints, the module has been pipelined into 4 stages. This required adding registers to store the valid and hit flags, and to pass on vector information throughout the different stages. Registers were also needed to store and pause the pipeline when calling the sceneQuery submodule, and to resume when the valid out signal of the sceneQuery is asserted.

4.1.7 Shading

This module implements a Lambertian reflectance with an added ambient lighting component.

$$L_d = k_d \cdot I \cdot \max(\mathbf{n} \cdot \mathbf{l}, 0)$$

L_d is the diffuse light intensity coefficient, k_d is the diffuse reflectivity coefficient, I is the incident light intensity, and n and l are the normal and light direction vectors respectively at a certain surface point.

$$\mathbf{n} \cdot \mathbf{l} = \|\mathbf{n}\| \, \|\mathbf{l}\| \cos\theta$$

Both vectors have been calculated and normalised by the previous module, getSurfaceVectors. Normals parallel with the light direction vector will produce a lighting coefficient of 1, whereas normals perpendicular to the light direction vector will produce a coefficient of 0.

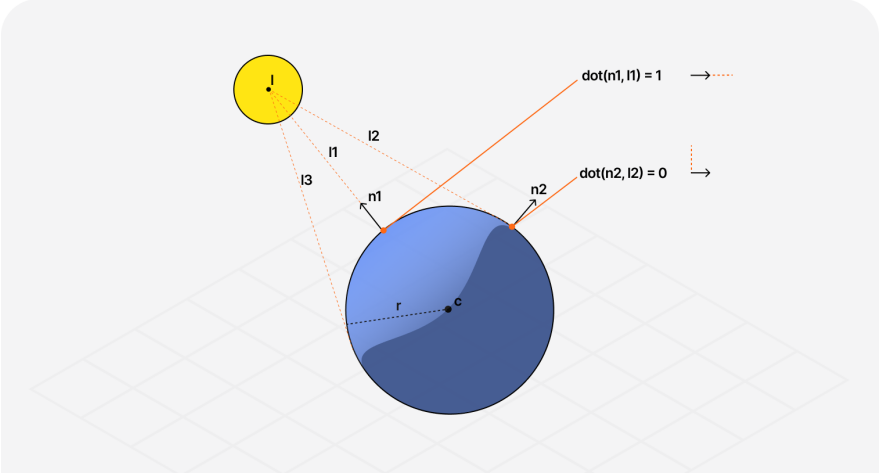

Figure 13: Diagram of light coefficient evaluation. [1].

The diffuse term quantifies how much light is incident on a particular surface point, by calculating the dot product between the surface normal vector and light direction vector. A ternary operator is then used to clamp the result to zero if its negative to consider only incoming external light onto the surface. To prevent areas not directly lit from appearing completely black, we introduced an ambient component that approximates scattered light hitting the surface, making it is easier for users to see the 3D object. Only a simple ternary operator on the Y component of the normal vector was used to approximate the ambient term, making it efficient while enhancing visual clarity.

Initially, the shading logic was implemented using `always_comb` for quick functional verification. However, the module was later pipelined to fix timing issues. It is currently divided into 4 stages, making sure every stage has minimal multiplication.

4.1.8 Counter Buffer (FIFO)

The counter buffer lies the end of the pixel generation stages as a final module stage that directly passes data to the pixel packer. The purpose is to properly sync the rgb_out, valid_out, sof and eol signals. There are multiple methods that can achieve the same result, such as packing the sof and eol through all the processes so that each rgb_out will always have the correct sof and eol value. However, this means an extensive utilization of buses and registers; as a result, a counter that increments for every valid_in from the shading module is generated that automatically keeps track of the pixel count, thus, the sof and eol. Most importantly, this allows a handshake sequence with the packer. The handshake sequence is controlled by the input ready signal from the pixel packer - valid_in along with other signals should only go high when the pixel packer is ready.

Initially, to perform this safe handshake in the single-cycle and pipelined design, a coordinate counter was chosen. The coordinate counter works by relying on a FSM that cycles between just 2 stages IDLE, PUSH. When IDLE, the counter waits for a valid_in pulse from shading which increments the coordinate count and shifts the state to PUSH and when PUSH, it waits for a ready indicator from the packer to produce a valid_out high, pass data to the packer, and then shifts the state back to IDLE.

```
Time resolution is 1 ps
SOF Ok on frame 0
EOL Ok on line 0
EOL Ok on line 1
EOL Ok on line 2
EOL Ok on line 3
EOL Ok on line 4
EOL Ok on line 5
EOL Ok on line 6
EOL Ok on line 7
EOL Ok on line 8
EOL Ok on line 9
EOL Ok on line 10
EOL Ok on line 11
EOL Ok on line 12
```

Figure 14: SOF and EOL handshake testing

This immediately poses an issue when integrated with the parallel units because of the continuous burst of valid_in from the shading module. A FSM without any memory storage will always skip the subsequent data. As a result, a small 8 by 32-bit FIFO replaces the FSM to enable a burst of input. The synchronous FIFO allows simultaneous reading and writing of data while keeping track through write and read pointers and a full indicator. The counter that determines the sof and eol value remains relatively the same and increments whenever a read action occurs.

```
    // Internal FIFO buffer
    logic [OUT_WIDTH-1:0] buffer [0:BUFFER_SIZE-1];
    logic [$clog2(BUFFER_SIZE)-1:0] wr_ptr, rd_ptr;
    logic [$clog2(BUFFER_SIZE+1)-1:0] count;

    // Pixel coordinate counters
    logic [$clog2(SCREEN_WIDTH)-1:0] x, x_temp;
    logic [$clog2(SCREEN_HEIGHT)-1:0] y, y_temp;

    logic full;

    assign full = (count == BUFFER_SIZE);
```

Figure 15: FIFO & Counter signals for counter_buffer module

4.2 Optimisations

4.2.1 Pipelining

The first step of optimisation from the software implementation is the inclusion of pipeline. The purpose of pipelining is not only to reduce the timing issues faced during synthesis and implementation, but also to increase the throughput of the single-cycle design. The pixel generation process can be broken down into 4 main stages: Hit point generation, normal & light vector calculation, object colouring, and handshake counter Figure 16. In addition to each module having its own pipelining stages, the top module (*Full Module) also implements pipeline registers to separate the aforementioned stages.

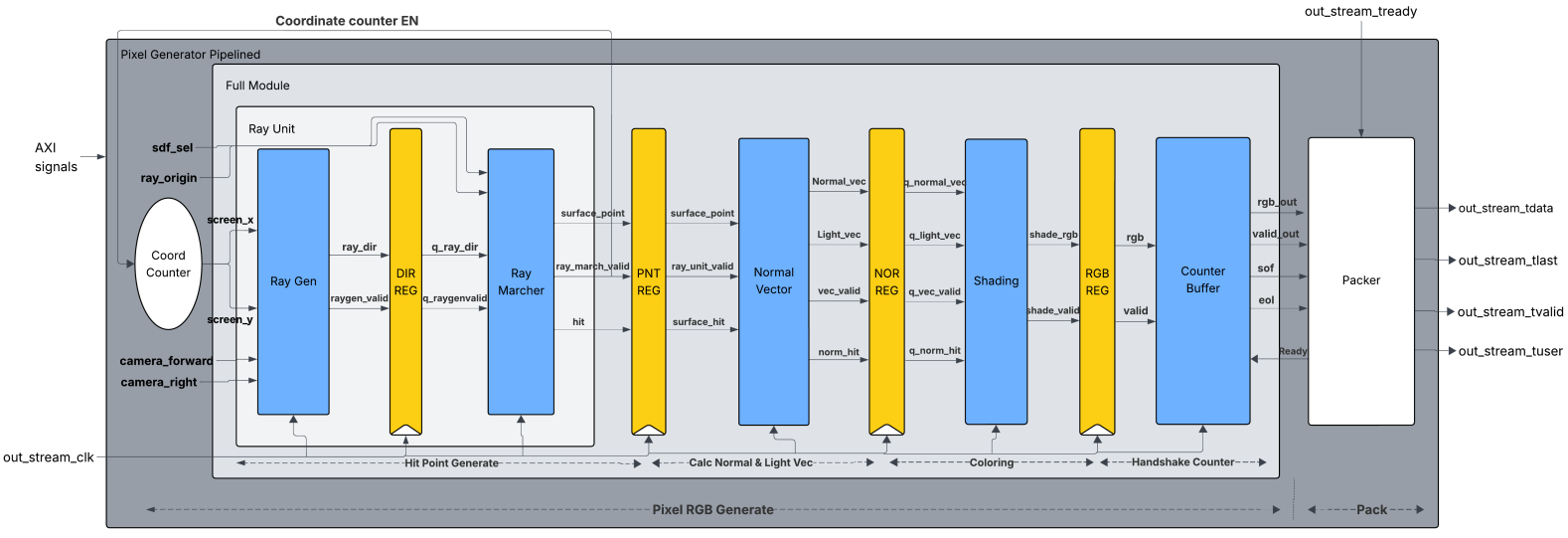

Figure 16: Top-level diagram of the pipelined design

The biggest difference to a single-cycle design is the coordinate counter loop enable signal which initiates the start of a new pixel coordinate (*screen_x, screen_y) and a valid_in signal. As opposed to waiting for the entire pixel RGB generation to complete, the EN signal will be routed to the Ray Unit valid_out (*ray_march_valid) and initiate the coord counter loop after the surface point for the previous coordinate has been calculated. Pipelining will allow multiple pixels to be processed in the subsequent stages after Ray Unit in the same instance, in turn, substantially increasing the throughput.

Based on the simulations, the frame generation speed of the single-cycle design is roughly $\approx$ 0.195 second/frame for a 10 ns period clock cycle. In comparison, the frame generation speed of the pipelined design is roughly $\approx$ 0.138 second/frame for the same clock period (Fig 17). The difference is an approximated 30% reduction and increases the frame rate to roughly 7.2 frames per second.

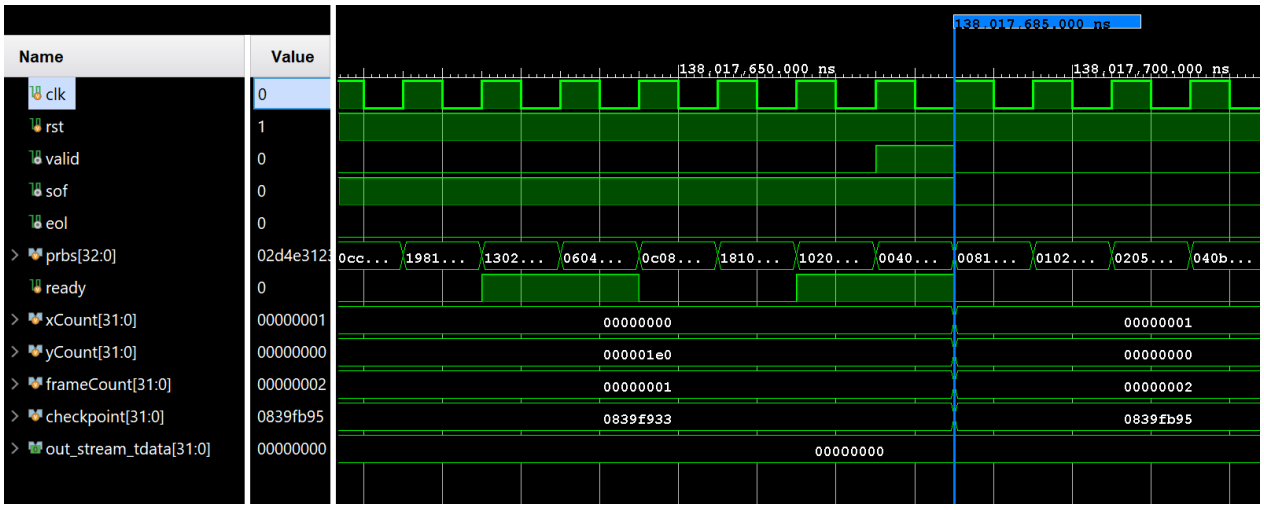

Figure 17: 1 line generation speed for pipelined design (simulation)

4.2.2 Parallelisation

The ray marchering process for each pixel is independent, enabling us to parallelise rendering across multiple internal Ray Units. As demonstrated in Figure 15, rendering time decreases with additional Ray Units, but the speedup is not proportional to the number of units deployed. While theoretically we should expect linear scaling, the overall frame time remains constrained by the other pipeline stages, particularly the surface normal calculation and shading stages which create a performance ceiling.

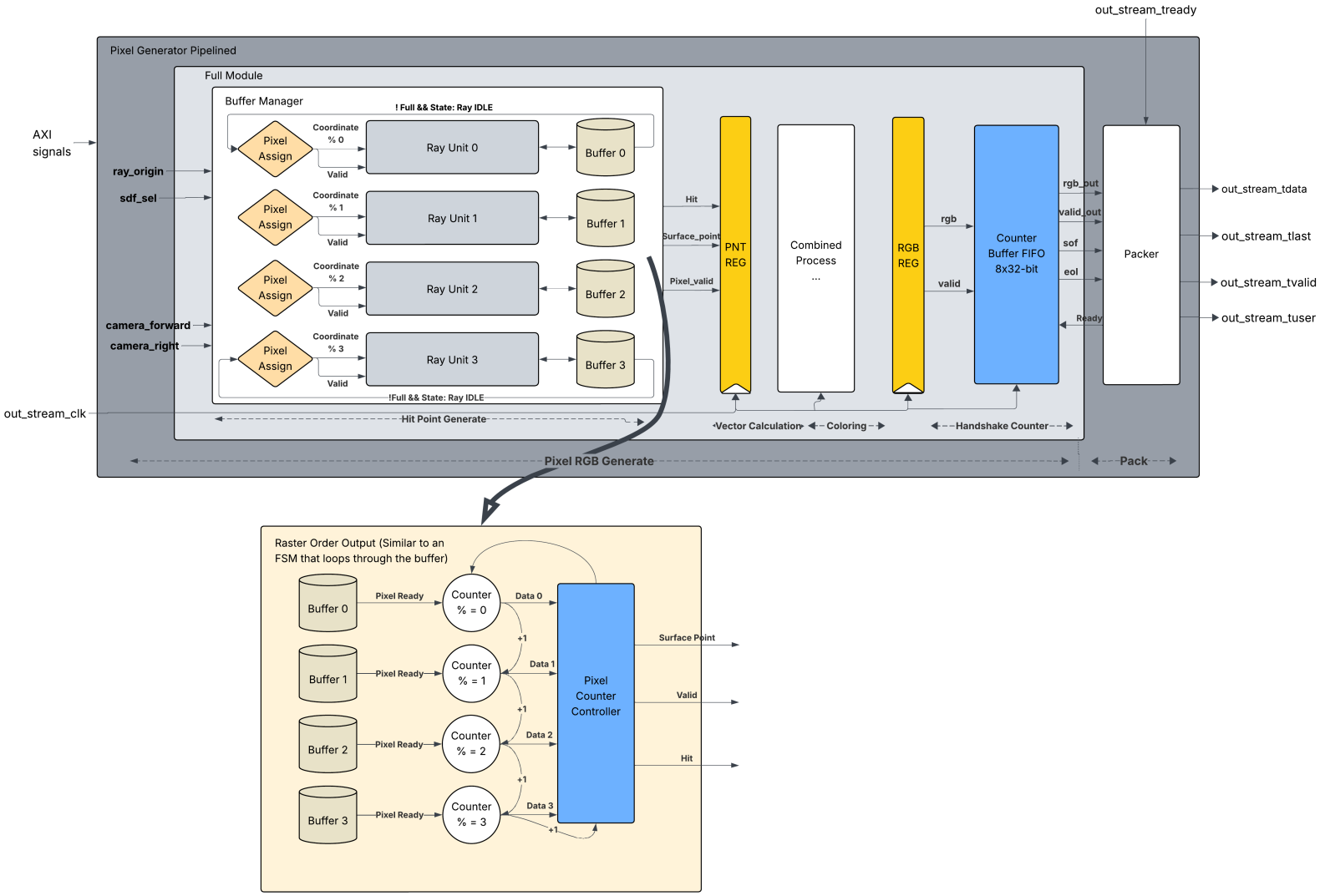

Figure 18: Top-level diagram of the pipelined design

Beyond these algorithmic bottlenecks, the more immediate limitation is resource availability within the FPGA. Each Ray Unit requires approximately 5000 LUTs for implementation. Our baseline pipelined version with no parallelisation (single Ray Unit) consumes around 24,000 LUTs, while only about 40,000 LUTs are available to us in the PYNQ board. This leaves us only about 15,000 LUTs available for additional Ray Units, theoretically limiting us to 3 additional units. Future performance improvements would require us to optimize our LUT slice utilization in the Ray Unit, or possibly using multiple FPGA boards for further parallelisation.

The performance data was obtained by systematically incrementing the number of Ray Units in the parallel architecture and measuring how long it took to complete one frame. We used SystemVerilog generate blocks to parametrise the number of Ray Units, ultimately determining that 4 Ray Units provided optimal resource utilisation. A buffer depth of 16 was

more than sufficient to handle latency variations in the ray marcher process while minimizing memory utilization. Shallower buffers would risk stalling when faced with higher scene complexity.

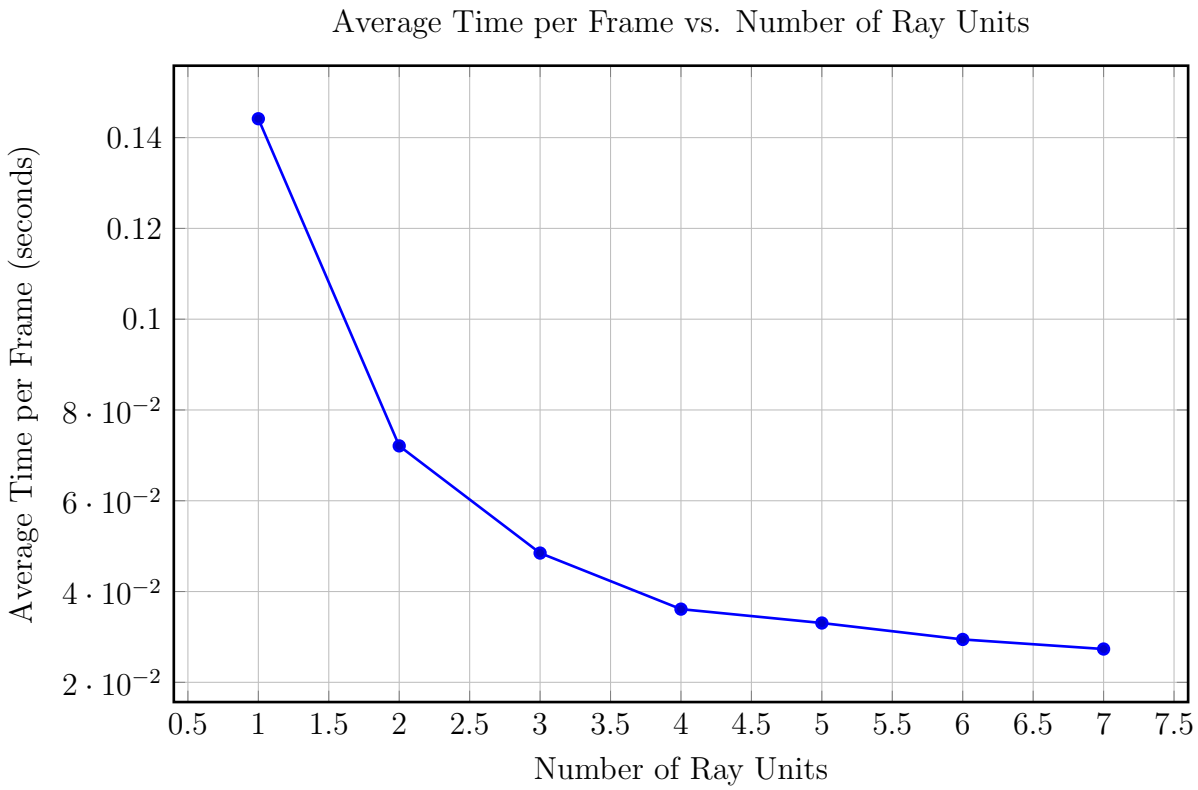

Figure 19: Frame Generation Performance for Parallel Ray Units

4.2.3 Number Representation

Initially, we adopted a word length of 32 bits to provide additional precision tolerance for potentially complex SDF queries such as the rendering of a Menger sponge tunnel. Although 24 bits would have been sufficient for our current world sizes and accuracy requirements for basic SDF operations, we decided on keeping the extra bits for greater tolerance.

We chose fixed-point representation over floating-point for a number of reasons. Floating-point units consume significantly more LUT resources compared to fixed-point arithmetic, and tend to have variable latency depending on operand values. The dimensions of the ray marcher world space is set at 10, so we chose Q8.24 for our fixed point representation, which is sufficiently large to handle our distance vector calculations with a precision of 2^{-24}. High fractional precision is particularly crucial for accurate ray stepping and surface detection. The 24-bit fractional component ensures that rounding errors accumulated during the ray marching process remain negligible compared to typical surface intersection tolerances or other quantization errors.

```
generate
    for (genvar i = 0; i< RAY_UNITS; i++) begin : ray_unit
        ray_unit ray_unit_ins (
            .clk(clk),
            .rst_gen(rst),
            .screen_x(screen_x[i]),
            .screen_y(screen_y[i]),
            .valid_in(valid_in[i]),
            .camera_forward(camera_forward),
            .camera_right(camera_right),
            .ray_origin(ray_origin),
            .sdf_sel(sdf_sel),
            .surface_point(surface_points[i]),
            .valid_out(valid_out[i]),
            .hit(hits[i])
        );
    end
endgenerate
```

Figure 20: Instantiating Ray Units in Parallel

Vector and other normal fixed-point arithmetic operations could be efficiently implemented through the use of automatic functions in SystemVerilog, leveraging the `vector_pkg.svh` header at the start of every module. For example, multiplication operates in a similar way to integers and just requires shifting the result by 24 bits.

We also implemented a hardware-optimized, pipelined inverse square root module for vector normalization in fixed-point arithmetic. This is achieved by first normalizing the input to the range $[0.5, 1]$ by shifting the input depending on the position of the most significant bit using a priority encoder. A BRAM lookup table is then used to find the inverse square root of the normalized input using 2^{12} pre-computed values in the range $1/\sqrt{0.5}$ and 1, before scaling back to its previous magnitude. We could also reuse the module to perform regular square roots to find the magnitude of a vector by multiplying the output of the function by its input squared i.e $\sqrt{x} = x^2 \cdot \frac{1}{\sqrt{x}}$.

This method results in nearly zero quantization error for most inputs between 0.5 and 100 using only 16 KB of BRAM.

We also experimented with using one or two iterations of the Newton Rhapson method instead of a lookup table. We determined that using two iterations of the algorithm modified to avoid using dividers would take 6 cycles (10 cycles if folded) for sufficient precision, compared to 3 cycles with our current implementation. Another option we looked at was to use an optimal linear interpolation algorithm which would only take 2 cycles, but had a precision of 2% which would introduce unacceptable error accumulation over the many sdf evaluations required per frame. Based on these experiments, we determined that the BRAM lookup table would provide the best balance between speed, precision and hardware complexity.

```
typedef logic signed [`WORD_WIDTH-1:0] fp;
typedef struct packed {
    fp x;
    fp y;
    fp z;
} vec3;

function automatic fp fp_mul(input fp a, input fp b);
  logic signed [63:0] result;
  result = $signed(a) * $signed(b);
  result = result >>> `FRAC_BITS;
  return result[31:0];
endfunction

function automatic fp vec3_dot(input vec3 a, input vec3 b);
  return fp_mul(a.x, b.x) + fp_mul(a.y, b.y) + fp_mul(a.z, b.z);
endfunction

function automatic vec3 vec3_cross(input vec3 a, input vec3 b);
  vec3_cross.x = fp_mul(a.y, b.z) - fp_mul(a.z,b.y);
  vec3_cross.y = fp_mul(a.z, b.x) - fp_mul(a.x, b.z);
  vec3_cross.z = fp_mul(a.x, b.y) - fp_mul(a.y, b.x);
endfunction
```

Figure 21: Some examples of automatic functions for fixed point vector arithmetic

```
casez (x)
    // ...
    32'b00000000000000001????????????: begin norm_x_next = x <<
       9;  exp_adj_next = 5'h11; end
    32'b000000000000000001???????????: begin norm_x_next = x <<
       10; exp_adj_next = 5'h12; end
    32'b0000000000000000001??????????: begin norm_x_next = x <<
       11; exp_adj_next = 5'h13; end
    32'b00000000000000000001?????????: begin norm_x_next = x <<
       12; exp_adj_next = 5'h14; end
    32'b000000000000000000001????????: begin norm_x_next = x <<
       13; exp_adj_next = 5'h15; end
    // etc.
endcase
```

Figure 22: Priority Encoder to determine MSB

Table 8: Error Analysis Summary for Inverse Square Root (Worst case input at $x = 32.5$)

Metric	**Value**
Maximum Absolute Error	1.314×10^{-4}
Maximum Relative Error	0.011847%
Average Absolute Error	2.410×10^{-5}
Average Relative Error	0.004433%

5 Software Design

The software component was developed to enable interactive control of the hardware-based ray marcher through a lightweight Python interface. It consists of two main parts. The first of these parts is a Pygame based graphical user interface (GUI) running on a local PC, and the second is an embedded Python script running on the PYNQ board. These components work together to transmit user-specified parameters to the hardware, allowing for flexible and responsive scene updates.

5.1 UI Overview

The user interface was designed to serve as a bridge between the user and the ray marching hardware engine, enabling interactive control over scene parameters without requiring any hardware level modifications. Built using the Pygame library, the interface provides a simple and accessible way to experiment with the rendering process.

Through adjustable sliders and control buttons, users can modify inputs such as camera position, object configuration, and other visual parameters. These adjustments allow for meaningful exploration of the system's capabilities, and help demonstrate how user input can influence hardware-rendered output. While minimal in design, the interface prioritises responsiveness and ease of use, making it well suited for live demonstrations and iterative testing.

The development of the user interface was guided by a set of educational criteria established at the start of the project. These included requirements for interactivity, intuitive parameter adjustment, and clear visual feedback. The UI was also expected to operate independently from the rendering hardware to enable flexible deployment. These criteria shaped the design choices discussed in the following sections.

5.1.1 Pygame UI Construction

The graphical user interface was constructed using the Pygame library, selected for its simplicity and effective support for interactive 2D applications. The layout was designed to prioritise clarity, ease of use, and visual accessibility. A small informal survey was conducted with peers to gather feedback on the initial design. Based on the responses, improvements

were made to improve visual structure, simplify navigation, and apply a palette suitable for colour-blind users. Online simulation tools were also used to verify the visual contrast under common conditions such as deuteranopia.

The interface was built using manually positioned widgets such as sliders and buttons, arranged in logical groups for each control category. The design emphasised simplicity and responsiveness during interactive demonstrations and testing. Event handling was implemented using Pygame's input system, enabling fluid user interaction.

A full breakdown of the adjustable parameters, button functions, and communication logic is provided in the following sections.

Figure 23: Screenshot of the Pygame-based user interface.

5.1.2 Adjustable Parameters

Camera Controls

The interface includes three sliders for controlling the camera's rotation along the X and Y axes, as well as a zoom level. These sliders allow the user to modify the camera's viewing angle and distance, thereby changing the perspective from which the ray-marched scene is rendered. Since the camera orbits around fixed radius, its position in 3D space can be calculated using rotation angle. The angle allows us to calculate the camera position, which will be subsequently used to calculate the values of camera basis vector, since cross product and normalisation is resource intensive it can be offloaded to a python script running in CPU. Camera basis vectors will be sent through registers.

The camera position vector $\vec{C}$ in Cartesian coordinates is given by:

$$\vec{C} = \begin{bmatrix} r \cdot \cos(\phi) \cdot \sin(\theta) \\ r \cdot \sin(\phi) \\ r \cdot \cos(\phi) \cdot \cos(\theta) \end{bmatrix}$$

where:

- r is the zoom level (distance from the origin),
- θ is the horizontal angle (azimuth),
- ϕ is the vertical angle (elevation).

The camera basis vectors are calculated as follows:

$$\vec{f} = \frac{\vec{T} - \vec{C}}{\|\vec{T} - \vec{C}\|} \quad \text{(forward vector)}$$
$$\vec{r} = \frac{\vec{f} \times \vec{u}_{\text{world}}}{\|\vec{f} \times \vec{u}_{\text{world}}\|} \quad \text{(right vector)}$$
$$\vec{u} = \vec{r} \times \vec{f} \quad \text{(up vector)}$$

Each slider operates over a fixed range that mirrors the fixed-point format used in the hardware registers. Specifically, rotation values range from –128 to +127, while zoom ranges from 0 to 10. These ranges were chosen to ensure a direct mapping from UI inputs to register values, avoiding the need for additional scaling.

To support more accessible and precise adjustments, the sliders are operable via both mouse and keyboard. The user can use the Tab key to cycle between sliders, the arrow keys to adjust values, and Enter to confirm changes.

The following Python snippet shows how the camera sliders are defined within the UI logic:

Listing 1: Camera slider configuration with matching hardware range

```
SLIDERS = {
    "Rotation X axis": (-127.0, 128.0, None, "{:.3f}"),
    "Rotation Y axis": (-127.0, 128.0, None, "{:.3f}"),
    "Zoom":            (0.0,    10.0,  None, "{:.2f}"),
}
```

Object Properties

Users can cycle between four predefined geometric shapes using the shape selection button in the lower-right corner of the interface. The available options are Sphere, Cube, Infinite

Cube, and Tunnel, each offering a distinct rendering profile to demonstrate the visual impact of different scene geometries.

Internally, each shape is associated with a numeric index ranging from 0 to 3, matching the 2-bit encoding used in the hardware. This direct mapping avoids the need for string parsing on the FPGA and allows for efficient object selection through a single register write.

The current shape is displayed on the button itself, and pressing the button cycles through the available options. Like the other parameters, the selected shape is only confirmed when the user presses the "Confirm" button, helping to prevent accidental changes.

Listing 2: Shape selection logic in the UI

```
shape_labels = ["Sphere", "Cube", "Infinite Cube", "Tunnel"]
shape_keys   = ["sphere", "cube", "infinite_cube", "tunnel"]
shape_i      = 0  # index to track current shape

# toggle shape on button click
elif shape_btn.collidepoint(mx, my):
    shape_i = (shape_i + 1) % len(shape_labels)
```

Lighting Controls

The interface includes three sliders for adjusting the position of the light source along the X, Y, and Z axes. These parameters control the origin point from which light is emitted within the scene and have a direct influence on how surfaces are shaded. This affects the perceived depth and highlights of the rendered object, helping illustrate how light direction impacts the visual outcome.

Each slider operates over a fixed range from –64 to +63.875, with a step size of 0.125. This resolution was chosen to align with the fixed-point representation used in hardware, ensuring that the values can be transferred directly to registers without additional scaling or rounding logic.

As with the other controls, users can interact with the sliders using both mouse and keyboard inputs. The current values are held in temporary memory until confirmed, at which point the lighting parameters are transmitted to the FPGA.

This adjustability allows users to experiment with different lighting conditions and observe their impact on the scene, reinforcing the educational goals of the system.

Colour

The interface includes a Hue slider that allows users to control the colour of the rendered object. This parameter is useful for adjusting the visual style of the scene and enables users to experiment with different aesthetics without requiring any changes to the hardware.

Internally, the Hue value is stored as a normalised float between 0.0 and 1.0. When the Confirm button is pressed, this value is converted into an RGB triplet using Python's `colorsys.hsv_to_rgb()` function. This conversion is necessary because the shading module in the hardware expects input in RGB format. The RGB values are then formatted into the UART command string sent to the FPGA:

Listing 3: Converting hue to RGB for UART transmission

```
hue_n = sliders["Hue"]["value"]
r, g, b = colorsys.hsv_to_rgb(hue_n, 1, 1)
cmd = f"R={r:.3f},G={g:.3f},B={b:.3f},..."  # snippet simplified for
    clarity
```

The interface also includes a colour swatch beside the slider, which updates automatically based on the selected hue. This provides immediate visual confirmation of the selected hue, making the process more intuitive and engaging. The colour selection feature also supports experimentation with styling and contributes to the project's educational aim of making the visualisation accessible and interesting.

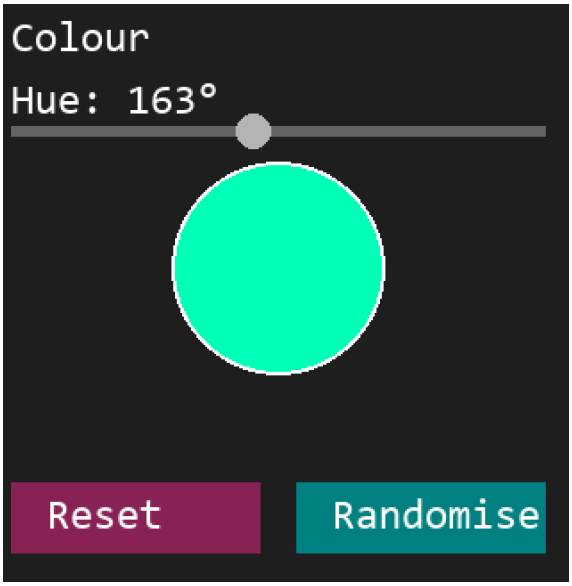

Figure 24: Colour swatch showing the selected hue value and visual feedback.

Reset and Randomise Buttons

Each parameter group in the interface is supported by Reset and Randomise buttons. The Reset button restores the sliders to their default values, while the Randomise button generates new values within the specified range. This functionality is provided for camera, lighting, and colour parameters, allowing users to easily explore a wide range of configurations.

The inclusion of the Randomise button aligns with the system's educational goal by encouraging curiosity and engagement. It offers a quick way to demonstrate how different parameter

combinations affect the rendered output, especially during live demonstrations or classroom settings.

Figure 25: Reset and Randomise buttons grouped under each parameter section.

All adjustable features described in this section were implemented to support the educational design criteria, including interactivity, experimentation, and clear user feedback, while ensuring full compatibility with the hardware logic.

5.1.3 Serial Transmission Protocol

The user interface transmits user-defined parameters to the FPGA over a UART serial connection. When the Confirm button is pressed, the system constructs a formatted ASCII string that includes only the final, confirmed values. These include zoom level, light origin coordinates, object shape, and RGB colour values, as well as the direction vectors derived from the camera rotation inputs. The corresponding hardware register allocations and bit widths are summarised in Table 4.

Listing 4: Formatted UART string containing only the values transmitted to the FPGA

```
# Assume direction vectors (forward, right, up) are precomputed from RX/RY
cmd = (f"FX={fx:.3f},FY={fy:.3f},FZ={fz:.3f},"
       f"RX={rx:.3f},RY={ry:.3f},RZ={rz:.3f},"
       f"UX={ux:.3f},UY={uy:.3f},UZ={uz:.3f},"
       f"ZM={zm:.2f},"
       f"LX={lx:.3f},LY={ly:.3f},LZ={lz:.3f},"
       f"R={r_:.3f},G={g_:.3f},B={b_:.3f},"
       f"OBJ={sk}\r\n")
ser.write(cmd.encode('ascii'))
```

The complete message is encoded in plain ASCII, structured as a comma-separated list of key–value pairs (e.g., FX=...,OBJ=...). This format is easy to parse on the receiving end and was designed to keep integration with the hardware simple.

5.2 Embedded Python Script

The embedded Python script runs on PYNQ board's ARM processor, the Zynq. It listens for serial data from the UI and writes the received parameters to hardware registers via the AXI interface.

5.2.1 UART Interface

The embedded Python script uses a UART interface to receive serial messages from the host PC. Upon receiving a message, it performs basic validation to check formatting and completeness. Once validated, the parameters are parsed and passed directly to the logic responsible for AXI register updates.

UART was chosen for its simplicity, and dependable point-to-point communication. Its deterministic nature helps reduce the chance of packet corruption or loss in controlled environments, making it suitable for reliably transmitting discrete control updates between the UI and FPGA.

5.2.2 AXI Write Logic

The AXI write logic is responsible for transferring parsed UI parameters into memory-mapped registers on the FPGA. Because the slider ranges were carefully chosen to align with the hardware's fixed-point representation, values such as zoom level, light origin coordinates, and colour coefficients can be sent directly without additional scaling.

Each parameter is assigned to a specific register in the PYNQ overlay. Floating-point values from the UI are cast to integers before transmission to match hardware expectations. The example below illustrates how values might be written to registers:

Listing 5: Writing confirmed values to hardware registers

```
regmap.gp0 = int(zoom)              # Zoom level
regmap.gp1 = int(light_x)           # Light position X
regmap.gp2 = int(light_y)           # Light position Y
regmap.gp3 = int(light_z)           # Light position Z
regmap.gp4 = shape_index            # Shape identifier (0-3)
# Additional registers used for colour and camera vectors...
```

This direct memory-mapped register access reduces latency and simplifies debugging, while ensuring that all UI-driven changes are accurately reflected in hardware behaviour.

6 Integration and Testing

6.0.1 UART Interface

6.1 Testbench Development

Alongside creating the modules in System Verilog, we designed testbenches for each module. By ensuring verification of lower level modules, this would help us save time debugging higher level modules. We used both Icarus and Vivado to run simulations, while GTKWave was

utilised to view waveforms for debugging purposes. When implementing new features such as new SDF's and camera orientation, having testbenches for every module proved essential to saving time.

6.2 Functional Verification

6.2.1 Unit Testing

Ray Marcher Ray Marcher was validated by applying rays from various positions and directions to evaluate its direction .The purpose of the module is to implement ray marching algorithm, as described in Section 3.1.4. Each test case is chosen to stimulate edge cases such as diagonal intersection, vertical hit , and also clear misses. Since the value of sphere and position are known, it is possible to verify if it should be a hit or a miss. To verify the position of objects,the expected intersection was stimulated in python script and compared against the hardware verification to check the correctness.

```
task automatic apply_ray(input real ox, oy, oz, dx, dy, dz);
    ro = make_vec3(to_fixed(ox), to_fixed(oy), to_fixed(oz));
    rd = make_vec3(to_fixed(dx), to_fixed(dy), to_fixed(dz));

    @(negedge clk);
    valid_in <= 1;
    @(negedge clk);
    valid_in <= 0;

    wait(valid_out);
    @(posedge clk);

    $display("Ray: ro=(%.2f,%.2f,%.2f), rd=(%.2f,%.2f,%.2f) --> Hit
       =%.4f, Point=(%.2f,%.2f,%.2f), Dist=%.4f",
      ox, oy, oz, dx, dy, dz, hit, point.x, point.y, point.z,
         from_fixed(distance));
  endtask

  // Simulation
  initial begin
    $display("=== rayMarcher simulation ===");

    $dumpfile("rayMarcher_test.vcd");
    $dumpvars(0,tb_rayMarcher);

    rst_n <= 0;
    #20;
    rst_n <= 1;

    apply_ray(0.0, 0.0, 1.0, 0.0, 0.0, -1.0);   // Straight ray
       forward
    apply_ray(1.0, 1.0, 0.0, -0.70710678, -0.70710678, 0.0); //
       Diagonal toward center
    apply_ray(0.0, 5.0, 0.0, 0.0, -1.0, 0.0);  // From above
       downward
    apply_ray(0.0, 0.0, 1.0, 0.0, 1.0, 0.0);   // Miss upward

    #100;
    $display("=== Test Complete ===");
    $finish;
  end
```

Figure 26: Testbench code snippet for the `rayMarcher` module

Ray Generator The main function of Ray generator module is to generate ray direction which has been mentioned previously in Section 3.1.4. The testbench snippet provided be-

low demonstrates part of the extensive testing of the module by stimulating with various pixel coordinates, including edge cases which was verified by calculation through manual calculation.

The structure of testbenches has been similar, with varied inputs and helper functions used to convert values to fixed-point representations.

```
function automatic real from_fixed_Q11_21(fp val);
    return $itor($signed(val)) / 2097152.0; // 2^21
endfunction

function automatic fp to_fixed_Q11_21(input real val);
  return $rtoi(val * (2.0 ** 21));
endfunction
```

Listing 6: Testbench code snippet

6.3 Top Level Simulation

Once unit tests had verified the functionality of all submodules, we moved on to simulate the top module. By converting a VCD signal into an PNG image, we were able to quickly preview a frame for the current design, which avoided the time consuming process of synthesising on an FPGA to view a display. The following python script uses two signals to generate an image frame. An asserted *validout* signal indicates a pixel that has undergone the full process. The 24 bit RGB *shade_out* signal was used to colour in that specific pixel. This script can only generate a valid frame in a single cycle implementation, as it uses a counter to track the pixel position in raster order.

```
from vcdvcd import VCDVCD
from PIL import Image

# --- CONFIGURATION ---
vcd_file = "Path To VCD File"
shade_signal  = "tb_fullModule.shade_out[23:0]"
valid_signal  = "tb_fullModule.valid_out"
width, height = 640, 480
output_image = "Path To Output Image"
print("Parsing  V C D  ")
vcd = VCDVCD(vcd_file, signals=[shade_signal, valid_signal],
   store_tvs=True)

shade_tv = vcd[shade_signal].tv
valid_tv = vcd[valid_signal].tv

def binstr_to_rgb(bitstr):
    val = int(bitstr.zfill(24), 2)
    return ((val >> 16) & 0xFF,
            (val >>  8) & 0xFF,
             val        & 0xFF)

pixels = []
i_s = i_v = 0
cur_shade = "0"*24
cur_valid = "0"
total = width * height

while len(pixels) < total and (i_s < len(shade_tv) or i_v < len(
   valid_tv)):
    t_s = shade_tv[i_s][0] if i_s < len(shade_tv) else float('inf')
    t_v = valid_tv[i_v][0] if i_v < len(valid_tv) else float('inf')

    if t_s <= t_v:
        # just update the current shade
        cur_shade = shade_tv[i_s][1]
        i_s += 1
    else:
        # valid changed only here do we emit
        cur_valid = valid_tv[i_v][1]
        i_v += 1
        if cur_valid == '1':
            pixels.append(binstr_to_rgb(cur_shade))

# build & save the image
img = Image.new("RGB", (width, height))
img.putdata(pixels)
img.save(output_image)
print("Done ->", output_image)
```


Figure 27: Python script to translate a VCD signal to a PNG image

7 Performance Analysis

7.1 Frame Rate & Latency

To determine the effectiveness of each hardware version - single-cycle, pipelined, parallel -, the FPS and latency are calculated and averaged from 5 trials. Due to the nature of the process, the trials exhibit almost identical timing so only the average is presented.

Table 9: Performance Comparison Across Implementations

Implementation	**Time (s)**	**FPS**	**Clock (MHz)**	**Latency (ns)**	**Speedup**
C++	0.075	13.3	—	—	1.0x
Python	4.320	0.23	—	—	0.017x
HW (Single-Cycle)	0.195	5.13	100	500	0.39×
HW (Pipelined)	0.138	7.25	100	550	0.55×
HW (Parallel x4)	0.037	27.7	100	900	2.08×

Note: Table 9 presents the performance analysis of different implementations to render a **sphere** using our ray marching algorithm. To evaluate the performance of the final design (4x Parallel ray units), the C++ code, executed using Intel Core i9-13900H 13th Gen, is considered as the base implementation to satisfy the project requirement. The clock frequency used for the pixel generator is kept at 100 MHz across all implementations.

Foremost, the latency between hardware differs with a sudden increase from the single-cycle & pipelined to the parallel implementation. This increase is owed to the rise in complexity of the parallel version which requires additional modules, particularly the buffer manager. Although a 450 ns gain from the pipelined implementation can be seen as substantial, a 900 ns latency is sufficient for a graphics rendering and throughput is significantly more relevant. Subsequently, to compare the throughput, the generation time taken per frame and FPS data are utilized in the speed-up ratio calculation, as shown in Table 9.

The C++ implementation was already able to execute the algorithm relatively quickly at up to 13.3 frames/second. The efficiency can be proven when compared to the Python performance with a speedup ratio of 0.003x. While both the single-cycle and pipelined performance are both lower than C++ by 0.39x and 0.55x respectively, the hardware implementation with 4x parallel ray unit **doubles** the performance of the C++. This successfully showcases the effectiveness of the final hardware design.

Figure 28 shows the performance variation across different SDF primitives, showing how the performance scales with scene complexity. Rendering speed is proportional to the number of iterations per pixel, so more complex scenes like the Menger Sponge would require 8 cycles to compute, while the Cube Lattice would only require 2 cycles to compute, a nearly 4-fold difference.

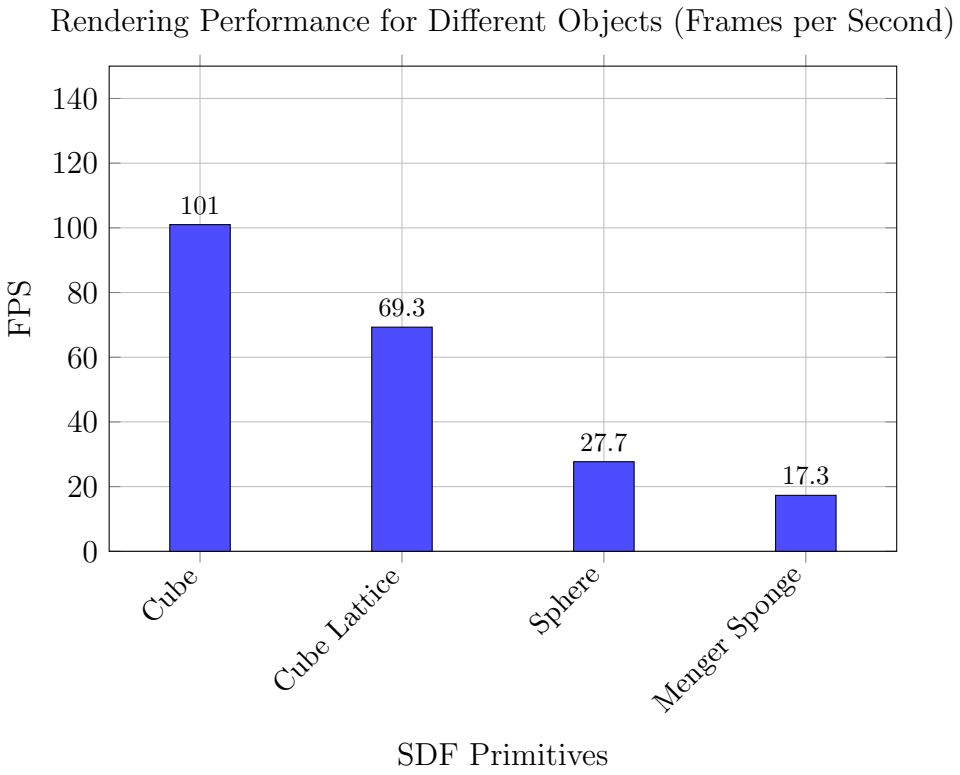

Figure 28: FPS performance for different SDF primitive types with 4 Ray Units in parallel.

7.2 Visual Output Correctness

The available objects/scenes were rendered at a pixel resolution of 640 x 480. The results were excellent with all the scenes generated as intended. The images (Fig. 29) are displayed below and can qualitatively evaluated to be successful.

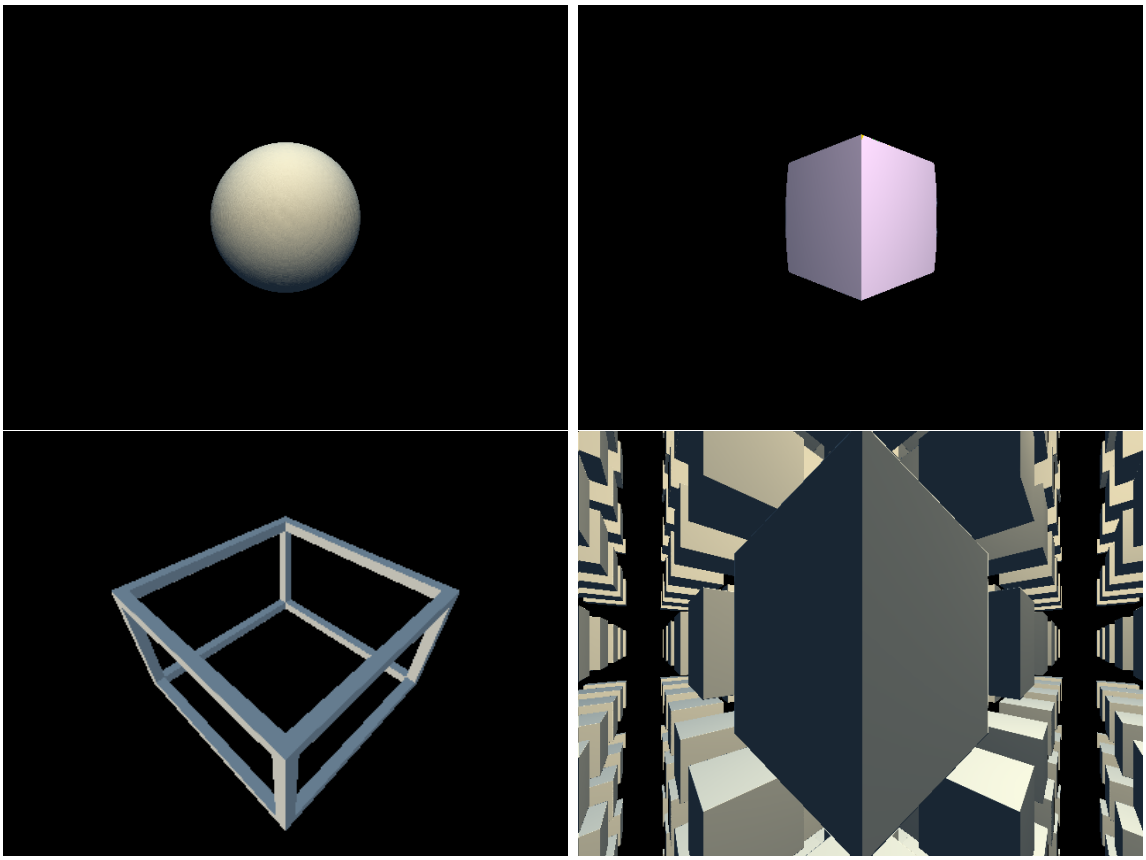

Figure 29: Scenes Rendered Images

7.3 Resource Utilisation

Table 10: Module and Resource Utilization Summary

Module	**LUTs**	**BRAM**	**DSPs**
Top (Pixel Generator)	39,500	34	220
- Buffer Manager	21,000	16	200
– Ray Unit	5,000	4	50
- Surface Calculation	11,000	18	14
- Shading	2,500	0	6
- Counter Buffer	60	0	0
- Packer	29	0	0

Shown in the Table 10, the parallel design utilizes all the DSPs and had to allocate some arithmetic calculation to the look-up tables. The number of LUTs is still within the limit even when combined with the video IP.

8 Conclusion

Our ray marching system successfully achieved our primary objectives, demonstrating real-time rendering of multiple object scenes at 640×480 resolution with frame rates exceeding 20 frames per second. Despite encountering synthesis complications during development, particularly LUT resource constraints, the final hardware implementation delivered excellent performance. We also met our secondary objective of creating an intuitive, visually engaging user interface with multiple features designed with accessibility and education use in mind, and are particularly happy with the object rotation.

9 Evaluation

9.1 Achievement of Requirements

Requirement	Implementation Approach	Met?
The system shall display a visualisation of a mathematical function that is computed in real time	Ray marching of SDFs is used to generate a live 3D scene.	✓
The function shall be computationally intensive, such that it is not trivial to generate the visualisation at the required resolution and frame rate	SDF evaluation of complex objects and iterative ray stepping per pixel is computationally demanding.	✓
The computation should be 'embarrassingly parallel'	Each pixel's ray is processed independently across 4 parallel Ray Units, exploiting per pixel independence.	✓
The visualisation shall be generated with the supplied PYNQ-Z1 SoC FPGA board	Ray marching and shading are computed on the PYNQ-Z1's programmable logic, with HDMI output used for display.	✓
The accelerator shall be described using Verilog or SystemVerilog	All hardware implementation and optimisations are written in System Verilog.	✓
The accelerator shall provide an interface with the integrated CPU for the adjustment of parameters	An AXI-lite interface is used to access register which is used to change parameter of our SDF query to select object and different Camera Position .	✓
Number formats and word lengths should optimise accuracy vs throughput	Custom fixed-point formats are used to balance precision and performance.	✓
Accelerated implementation shall exceed CPU-only performance	Frame rate of the FPGA-accelerated version exceeds C++ in CPU implementation.	✓
The system shall provide a user interface to enhance educational use	A UI supports interaction and visual learning.	✓
UI may be implemented using separate hardware to the visualisation computer	UI is hosted separately and communicates with PYNQ over UART.	✓
UI shall allow adjustment of parameters in an intuitive way	Sliders that directly translates to camera moving is provided,and object colour changes are visible in UI in real time.	✓
UI should provide information via overlay or separate display	UI includes labelled sliders with overlays for easy understanding.	✓

Table 11: Requirements Compliance Table

9.2 Limitations

A major constraint in performance optimization that we faced was optimising our DSP usage. In retrospective, our word width of 32 bits was largely unnecessary. Our SDF calculations require a substantial number of multiplications to perform, and our higher word width led to the consumption of more DSP resources than necessary, imposing a considerable resource overhead. Reducing the word length to 24 bits would have yielded significant performance improvements by freeing up LUT slices for optimisations elsewhere such as additional Ray Units. It would have been better to analyse precision requirements carefully from the start so there would have been no need to revert to a wider implementation for "tolerance".

Regarding the user input, a significant constraint is our limitation to 8 registers for user input parameters. This limits the variety of scenes that users would be able to create and manipulate. Our current register allocation supports basic parameters such as camera position, rotation, zoom and colour, but provides no capacity for advanced SDF techniques such as union and intersection of multiple objects or procedural noise. To support additional features we would need to transition to a BRAM-based input parameter system. This would require additional logic for BRAM interfacing, and careful resource allocation given our existing resource constraints.

9.3 Future Improvements

There were several distinct improvements we could have made with regards to performance optimization and the addition of more features.

- **Checkerboard Rendering:** A common rendering technique we could have employed was checkerboard rendering, where we render only every other pixel and use bilinear interpolation to reconstruct the missing pixels using its neighbours. This would have given a clear 2× FPS improvement.
- **Colour Rendering Techniques:** Applying a dithering filter would have significantly improved perceived colour depth and improve accessibility. Our current system outputs to a limited colour palette, but temporal or spatial dithering could have improved the perceived contrast between different surfaces and prevented harsh colour transitions.
- **Colouring Circuit Pipeline:** The circuit architecture could also have been better pipelined, particularly in the surface normal calculation and shading modules, which represent a potential bottleneck in the rendering pipeline. Our decision to limit the system to single-pass rendering was appropriate given our time and resource constraints, and techniques such as deferred shading would have required significant frame buffer memory that would not have been worth the improvement in image.
- **SDF Manipulation Features:** SDF's can be manipulated to make unusual shapes. The union and intersection of different objects can be obtained by taking the maximum and minimum of the object SDF's respectively. Having further user controls such as

incorporating the union and intersection of objects could further the educational value of the project.

References

[1] Maxime Heckel. *Painting with Math: A Gentle Study of Raymarching*. [Online; accessed May 10, 2025]. 2023. URL: https://blog.maximeheckel.com/posts/painting-with-math-a-gentle-study-of-raymarching/.

[2] Inigo Quilez. *Distance functions*. [Online; accessed June 16, 2025]. URL: https://iquilezles.org/articles/distfunctions/.