REND3R: Raytracing Engine and Networked Device for 3D Rendering

Dev Chheda
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
dchheda@mit.edu

Moruph Osuolale
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
moruph@mit.edu

***Abstract*—We propose REND3R, a hardware-based 3D graphics pipeline capable of rendering three-dimensional primitives to a VGA-connected screen given commands over an Ethernet interface. The system can parse graphics commands, complete the geometry calculations necessary for mapping 3D to 2D space, and paint vertices, edges, and planes to the screen. We implement two versions of REND3R, one which uses rasterization rendering and another which uses raytracing rendering. Each rendering mode requires distinct computations, but the overall structure of the system is similar in both.**

***Index Terms*—Digital systems, Field programmable gate arrays, Domain-specific architecture, Instruction set architecture, Graphics processing, Computer networks**

I. Introduction

A. Overview of 3D rendering techniques

In general, 3D rendering is accomplished by keeping track of three main aspects that constitute a "scene":

- the camera
- light sources
- geometric objects

By calculating the relationship between these three things, we can determine the value for every pixel on the screen.

For a rasterization implementation of rendering, geometric objects are represented as triangles in three dimensional space, which can be projected to the two-dimensional viewport of the camera. The color of a particular triangle depends on how the triangle faces any light sources. Then for every pixel on the screen, we check to see if it is within any of the calculated 2D triangles, and if so, we assign to the pixel the color of the enclosing triangle.

The raytracing implementation of rendering involves casting rays from the focus of the camera, through each pixel in the viewport, onto the 3D scene. In order to calculate the value of each pixel, we first determine the nearest geometric object that each ray collides with by intersecting the ray with every object in the scene. Then, we compute the lighting (and potentially reflections/refractions) using recursive raycasts, and mix with the color of the object to compute the pixel value. So, in raytracing rendering, it is most convenient to represent geometric objects using analytically described surfaces which for which we can easily compute ray intersections. We specify these shape types further in Section II-D.

B. Physical System

The entirety of the hardware needed to run the system exists onboard the Digilent Nexys 4 DDR FPGA. To interface with the system, an ethernet cable is used to send commands, and a VGA cable is used to output display to a screen.

C. System Design Overview

The REND3R system consists of three primary stages:

- intepreter: The system receives commands over ethernet which are written into an instruction bank[1]. the instructions are read out and processed by the instruction processor. Instructions that involve updating properties of the 3D objects require writing to the memory bank. Other control instructions are handled by passing certain signals to the rendering controller.
- geometry processor: The rendering controller manages the scheduling of computations based on control events passed from the instruction processor and based on the rendering mode. The rendering controller passes data as scheduled from the memory bank to the renderer, which actually completes the rendering computations. As the renderer produces results, the controller tabulates and writes pixel values into the frame buffer.
- display: The VGA state machine implements the VGA specification and passes pixels from the frame buffer to the FPGA's VGA lines.

A diagram of these stages is shown in Figure 1.

II. Instruction Set Architecture

We define a custom 32-bit instruction set architecture for specifying graphics commands to our processor. The instructions set allows the programmer to fully specify a frame-by-frame 3D graphics sequence.

The processor deals with four fundamental types of instructions: frame instructions (F-type), camera instructions (C-type), lighting instructions (L-type), and shape instructions (S-type). Each instruction type is designated a unique opcode, and each has its own division of bits into various fields, as shown

[1]Due to time constraints we were not able to configure ethernet tools on our computers to actually send ethernet packets to the FPGA board. Instead, we load the instruction bank from a preset `main.mem` file.

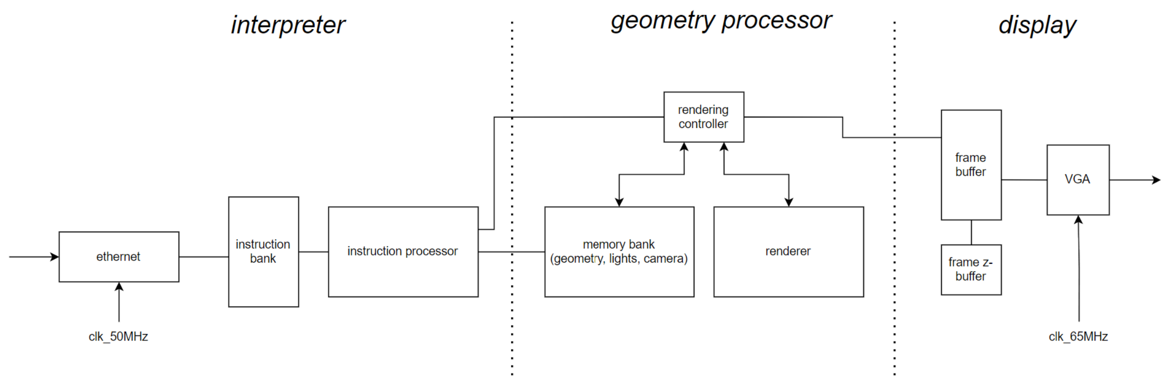

Fig. 1: High-level system overview

in Figure 3. S-type instructions are further divided into SE-type and SD-type, as discussed further in Section II-D. We currently implement 8 total instructions, as shown in Figure 2.

A. F-type instructions

Frame instructions are used to signal frame-level and control events. The type of frame event is determined by the func bits (instruction[10:9]). We currently support the following functions:

- 2'b00 = end render. This signifies the end of a render.
- 2'b01 = new render. This signifies the beginning of a new render and clears all existing shape and lighting information
- 2'b10 = new frame. This signifies the beginning of a new frame.
- 2'b11 = loop render. This signifies to loop the rendering by jumping to the top of buffered instructions.

However, as seen in Figure 3, there are many unused bits in the F-type instructions, allowing for many possible extensions. For example, one extension might involve allowing the programmer to specify a desired frame-rate, which would signal the processor to drop computations to match the frame-rate at the expense of graphics quality.

B. C-type instructions

Camera instructions are used to set and modify the properties of the camera in the 3D scene. Each instruction includes the property being modified and the new value of that property. Each property is a 16-bit value, and for the properties currently implemented, all values are interpreted as 16-bit floating point numbers. The properties of the camera are listed in Figure 4, and we offer brief descriptions below.

Camera properties:

- x-, y-, and z-location determine the location of the camera in 3D space.
- r-, i-, j-, and k-rotation determine the rotation of the camera as a quaternion.
- near-clip and far-clip determine the distances of the near- and far-clipping planes of the camera.
- fov-horizontal and fov-vertical determine the field of view in the camera's horizontal and vertical axes.

We note that the specified rotation and field-of-view of the camera are relative to default camera settings. In particular, the camera points in the negative z direction (i.e. towards $(0, 0, -1)$) by default, with its rotation about the z-axis such that the x- and y- axes of the worldspace are aligned with the horizontal and vertical axes of the screen space, respectively. The default near-clip is 1, and the default camera viewport has a width of 10 and a height of 7.5. The field-of-view and near-clip properties are just used to scale these parameters rather than set them directly.

C. L-type instructions

Lighting instructions are used to set and modify the properties of the various lights in the 3D scene. Similar to camera instructions, lighting instructions require the specification of the property being modified and the new value, but in addition to this, we must also specify the light index, since there are potentially many lights in the scene. The list of currently supported light properties is shown in Figure 4. We offer brief explanations of the properties which are different from the camera properties:

- Source type: the last 2 bits of the value determine the type of light source. We currently support the following:
  - 2'b00 = light off
  - 2'b01 = directional light

  We were not able to implement support for point light sources due to time constraints, but we could they would fit cleanly into our existing system, as the directionality of point light sources can be easily determined given the location of the object and location of the light. After that, point sources would be handled just as directional sources are handled.
- x-, y-, and z- forward determine the direction in which the light points.
- Color: the 16-bit color of the light consisting of a 5-bit red channel, 6-bit green channel, and 5-bit blue channel.
- Intensity: a 16-bit floating point number determining the intensity of the light.

Instruction	Syntax	Description	Type
NR	**nr**	new render	F-type
NF	**nf**	new frame	F-type
ER	**er**	end render	F-type
LR	**lr**	loop render	F-type
CAM	**cam** prop, data	update camera property	C-type
LT	**lt** idx, prop, data	update light property	L-type
SP	**sp** idx, prop, prop2, data, data2	update shape properties	SE-type/SD-type
TR	**tr** idx, prop, prop2, data, data2	update triangle properties	SE-type/SD-type

Fig. 2: Supported instructions and syntax

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	
21'b0																					func		6'b0						opcode			F-type
data																prop					8'b0								opcode			C-type
data																prop					00		index[5:0]						opcode			L-type
index[18:3]																prop					prop2					index[2:0]			opcode			SE-type
data																data2																SD-type

Fig. 3: Instruction encodings

Camera properties

Property index	Name	Data type	Symbolic name	Register name
0	zero	n/a	zero	a0
1	x-location	float16	xloc	a1
2	y-location	float16	yloc	a2
3	z-location	float16	zloc	a3
4	rotation-r	float16	rrot	a4
5	rotation-i	float16	irot	a5
6	rotation-j	float16	jrot	a6
7	rotation-k	float16	krot	a7
8	near-clip	float16	nclip	a8
9	far-clip	float16	fclip	a9

Light properties

Property index	Name	Data type	Symbolic name	Register name
0	source type	int2	src	a0
1	x-location	float16	xloc	a1
2	y-location	float16	yloc	a2
3	z-location	float16	zloc	a3
4	x-forward	float16	xfor	a4
5	y-forward	float16	yfor	a5
6	z-forward	float16	zfor	a6
7	color	int16	col	a7
8	intensity	float16	int	a8

Shape properties

Property index	Name	Data type	Symbolic name	Register name
0	zero	n/a	zero	a0
1	x-location	float16	xloc	a1
2	y-location	float16	yloc	a2
3	z-location	float16	zloc	a3
4	rotation-r	float16	rrot	a4
5	rotation-i	float16	irot	a5
6	rotation-j	float16	jrot	a6
7	rotation-k	float16	krot	a7
8	x-scale-inv	float16	xscl	a8
9	y-scale-inv	float16	yscl	a9
10	z-scale-inv	float16	zscl	a10
11	color	int16	col	a11
12	material	int2	mat	a12
13	shape type	int4	type	a13

Triangle properties

Property index	Name	Data type	Symbolic name	Register name
0	zero	n/a	zero	a0
1	x1	float16	x1	a1
2	y1	float16	y1	a2
3	z1	float16	z1	a3
4	x2	float16	x2	a4
5	y2	float16	y2	a5
6	z2	float16	z2	a6
7	x3	float16	x3	a7
8	y3	float16	y3	a8
9	z3	float16	z3	a9
11	color	int16	col	a11
12	material	int2	mat	a12

Fig. 4: Camera, light, shape, and triangle properties

D. S-type instructions

Shape instructions are the most complex in our instruction set due to supporting a large number of shapes. As shown in Figure 3, we support 19-bit shape indices, allowing the programmer to use a total of up to $2^{19} = 524,288$ shapes. However, since our instructions are only 32-bits wide, and since the opcode uses 3 bits, this leaves only 10 bits for specifying the property and value to update for a given shape. We standardize on 16-bit values (especially important for floating-point values to maintain a consistent level of precision), so this presents a problem.

We solve this limited bit-width problem by splitting each shape update instruction into two 32-bit instructions: one SE-type instruction and one SD-type instruction. With 64-bits for shape updates, we have more than enough bits to specify the required values; in fact, we now have enough bits to specify two properties and two 16-bit values. We take full advantage of this gained bit-width to reduce the total number of instructions in any graphics program (which would help improve performance). The SE instruction is responsible for specifying the shape index and the two properties to be updated; the SD instruction simply holds the two 16-bit values for each property.

Note, however, that this means that SD instructions do not have an opcode. In order to ensure that SD instructions are interpreted correctly, we require that SD instructions always immediately follow SE instructions. When our processor encounters an SE instruction (as determined by its opcode), it will always expect the next instruction to be an SD instruction (and will always interpret the next instruction as if it is SD).

Another problem arises with our method of updating two properties at once with SE/SD instructions — what if the programmer only wants to update a single shape property? For this reason, we include the null shape property, as shown in Figure 4. Updating this property will not affect any of the shape's actual properties and allows the programmer to update just one value using the SE/SD scheme.

Note, we actual provide two sets of instruction and property tables for setting shape properties. As described in Section I-C, rasterization rendering and raytracing rendering are handled in very different ways; particularly, rasterization rendering is accomplished using triangles while raytracing is best suited for shapes which can be analytically described. When the processor is run in rasterization mode, it interprets shape data as specified in the Triangle properties table; when the processor is run in raytracing mode, it interprets shape data as specified in the Shape properties table (see Figure 4). As a result, we also provide two separate high-level instructions for setting shape data — `sp` and `tr`, as see as seen in Figure 2. This conveniently allows the programmer to use the appropriate symbolic names when writing programs intended for either rendering mode. Note that both `sp` and `tr` are assembled into SE/SD-type instructions.

We now briefly describe the properties for triangles and shapes. In rasterization mode, triangles are specified by simply specifying their vertices in 3D space (x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3). In raytracing mode, we instead choose from a preset number of analytic shapes, and then transform the shapes as desired. We currently support the following shape types:

- 4'd0 = shape off
- 4'd1 = sphere
- 4'd2 = cylinder
- 4'd3 = cone

These shapes were selected because they can easily be described and solved for ray intersections analytically. We expand on this further in Section IV-B2 and also refer readers to [1] for a more in-depth explanation of how ray intersections are computed on these shapes.

Raytracing shapes have both rotation and scaling, although we require that the programmer provides the inverse scaling, since this significantly simplifies raycasting computations.

Color for both triangles and shapes is as specified for lights. The final property for shapes is the material property, which selects a preset material type (which can be used to describe reflectivity, refractivity, etc.). We don't implement different material types in this report (due to time constraints) but allow for future implementations by including in the ISA.

E. Assembler

We also provide an assembler to assist with programming for our processor. The assembler is a Python script which parses the human readable instructions shown in Figure 2 and writes out the equivalent binary instructions. The assembler gives a one-to-one instruction translation for the program given by the user, with the exception of `sp` and `tr` instructions. As discussed in Section II-D, shape updates are actually split across two 32-bit instructions, so the assembler accordingly assembles `sp` and `tr` instructions into one SE instruction and one SD instruction (in that order, as required by the processor).

III. Instruction Processor and Control

The instruction processor reads instructions from the instruction bank, decodes them from the 32-bit encoding, and executes them appropriately. The processor executes C-type, L-type, and S-type instructions by simply updating the specified properties in the memory bank by writing to the appropriate address (as determined by the object type and the object index).

F-type instructions allow for control over the system, and are therefore more complicated to handle. First, we note that the instruction processor should not update 3D objects while the rendering controller is reading from the memory bank and performing computations. This could lead to undesired behavior in which the rendered performs computations on data that has only been partially modified. In particular, the renderer should only perform computations after all updates for a given frame have been made (since no guarantee is made about the order of updates or the order of computations scheduled by the controller).

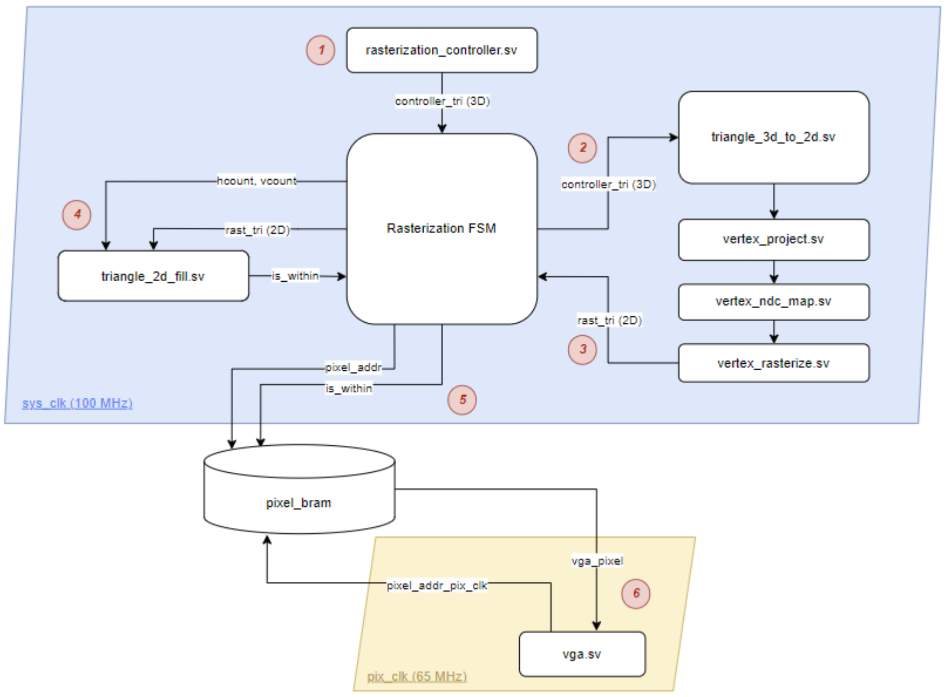

Fig. 5: Block diagram for rasterization process

So, a state machine is implicitly implemented in which the system is either in update mode or in compute mode at any given time. When the processor executes a `nf` command, it puts the system into compute mode until the controller signals that computation is completed, after which the system is restored to update mode. While in compute mode, all stages of the instruction processor stall to prevent updates until computation is completed.

When the processor receives a `nr` command, it clears the memory bank, and signals to the controller to clear the frame buffer. When the processor receives a `lr` command, it jumps to the top of the instruction bank, effectively looping the specified render program. Finally, When the processor receives an `er` command, it permanently stalls and does not consume any additional instructions.

IV. Rendering

A. Rasterization mode

1) Scheduling: When in rasterization mode, the controller schedules computations one triangle at a time. In particular, the controller iterates through all triangles and passes each to the renderer. Since the renderer may take many cycles to produce the output for a single triangle, the renderer passes back a `pause` signal to the controller, signifying when it is ready to consume the next triangle in the scene.

2) Data transformations: As mentioned previously, triangles are stored as vertex triples in a buffer within the BRAM as half precision (16 bits) floats. All math is done with half precision floats. We use the floating point Vivado IP, to perform operations on floats. In the previous stage (the instruction parser), the high level information about the shape is interpreted. From this information, we use the position and orientation of the shape to calculate the 3D triangle array for a specific shape. In addition, we store the shape's desired color into a BRAM. We then attach the address of the shape's color to the triangle array, which is used when determining the color of a triangle before the corresponding projection is calculated. This process creates an array of data that is described by Figure 6.

Next, we stream the 3D triangle array (and its attached address) into a module that will transform it into the representation given by Figure 7. For each triangle of each shape we can calculate its associated surface normal, which is the vector perpendicular to the plane coincident with the triangle. This comes from the vector cross product of any two edges of the triangle. Note that there are actually two surface normals, collinear with each other and pointing in opposite directions. The order of the vertices when determining the triangles thus matters, and must be properly specified to the controller. We can then take the dot product of the surface normal with the direction vector to any lights in the scene to determine how much a surface is "looking at" a particular light source. Then we mask the defined color of the shape with a lighting layer to get the final color of the triangle. When this step is over, we will have turned an array of shapes, as shown in Figure 6, into an array of triangles, as shown in Figure 7.

3) Projection: Notice at this point, we have dropped the concept of shapes, and are now working with just triangles

```
- info_address: 0x123
- triangles_3d:
    - { 0.0, 1.0, 2.0 }
    - ...
```

Fig. 6: Data format for 3D vertex array in DDR2 as shapes

```
- vertices_3d: { 1.0, 2.0, 3.0 }
- color: 0x123
```

Fig. 7: Data format for raw 3D vertex array as triangles

and their colors. With the associated position and orientation information given by the camera command, it is possible to know how a shape will appear in 2D space. For each 3D triangle, we project each of its vertices onto the plane associated with the given camera command. If the triangle lies at least partially on the screen, it is saved to a 2D vertex buffer that is within the DDR2 RAM along with its associated z-index. This "z-index" is actually the distance of the position of the centroid of the 3D triangle to the camera. The subsequent 2D triangle array contains items as shown in Figure 8.

```
- vertices_2d: { 10.0, 20.0, 30.0 }
- z-distance: 5.0
- color: 0x123
```

Fig. 8: Data format for projected 2D vertex array

4) Painting: To paint, the module does checks on each triangle in the 2D vertex buffer. A state machine is used to order the triangles by z-distance in descending order. Then for each triangle, a 2D bounding box is calculated around the shape of the triangle. This allows us to do checks on the pixels only within the bounding box, so as to not waste time on every other pixel in the scene. For each 2D pixel in the bounding box, we check if that pixel exists within that triangle. If so, we assign the pixel to the color of the triangle. All other pixels default to a color of black. With this information, we have created an entire frame of data, which is saved to a frame buffer implemented with BRAM.

B. Raytracing mode

1) Scheduling: For raytracing, we compute each pixel value one at a time. Computing the value of a single pixel involves finding the first intersection for the ray which originates at the camera focus and passes through the virtual pixel in the 3D viewport. However, in order to compute the nearest intersection, we must intersect the ray with every shape in the scene[2]. So, our controller performs a raycast for every shape for every pixel.

[2]There are heuristics which could be employed to skip raycast computations for certain shapes, but since our raycaster is fully pipelined, this would provide little to no benefit, since applying the heuristic on a given shape would take more time than simply raycasting onto the shape

In order to compute lighting and more advanced material effects, recursive raytracing is required. Particularly, after computing the intersection for a pixel ray, computing the lighting requires scheduling rays from the intersection point towards each of the light sources to determine interference from other shapes. And, computing for reflective materials just requires scheduling a reflection ray from the intersection in the appropriate direction. We don't implement advanced material effects, but we do implement lighting effects.

2) Raycaster: Most of the computation is handled by the raycaster module. Each raycast involves determining the intersection between a 3D ray (specified by a source and direction) and a given shape. As mentioned in Section II-D, we currently only implement spheres, cylinders, and cones. We chose these specific shapes because for each, computing a ray intersection involves solving a quadratic equation of very similar form.

For each shape type, the quadratic is simplest when the shape is assumed to be in normal form; i.e. centered at the origin, with unit size, and standard rotation. Thus, to perform raycast computation, we need to be able to convert between the normalized spaced and the real world space. Every shape X can be written in the form $X = TRS\hat{X}$ where T, R, and S represent translation, rotation, and scaling transformation respectively, and where $\hat{X}$ represents the normal form of the shape. The normal forms of our implemented shapes are:

- Sphere: Unit-radius sphere centered at origin.
- Cylinder: Unit-radius infinite sphere aligned along the z-axis centered at $(x, y) = (0, 0)$.
- Cone: Infinite double-cone defined by the curve $z^2 = x^2 + y^2$.

Note that, for each shape, T, R, and S are specified by the programmer's setting of shape properties.

Rather than transforming the shapes (and thus the quadratics), we instead transform the ray into the normalized space of the shape. Particularly, for source S and direction D in the real space, we compute $\hat{S} = S^{-1}R^{-1}T^{-1}S$ and $\hat{D} = S^{-1}R^{-1}D$. Then, we perform the raycast in the normalized space before transforming the result back to the real world space. For more mathematical details, we encourage readers to refer to [1].

So, overall, the raycaster first transforms the ray into the shape's normalized space, then it generates the appropriate quadratic coefficients (which are based on the ray parameters) dependent on the shape type, then it solves the generated quadratic for the smallest real solution, and finally it computes the related intersection point, distance, normal vector, etc. to give as output.

In the future, we could easily implement coordinate-based clipping for the analytical shapes, since this would just require checking the coordinates of each intersection point to see if it was within the specified bounds. This would allow for finite (open-ended) cylinders and cones along with other interesting objects.

V. System Evaluation

We define the following parameters for discussion of our system's resource usage and performance:

- `NUM_INSTRUCTIONS` = number of instructions stored in bank.
- `NUM_LIGHTS` = number of lights in the 3D scene.
- `NUM_TRIANGLES` number of triangles in rasterization mode.
- `NUM_SHAPES` = number of shapes in raytracing mode.
- `SCREEN_WIDTH` = width of virtual screen (in pixels).
- `SCREEN_HEIGHT` = height of virtual screen (in pixels).
- `NUM_PIXELS` = total number of pixels, `SCREEN_WIDTH * SCREEN_HEIGHT`.
- `CLK_PERIOD` = period of the system clock used to drive all computation and rendering.

We kept `SCREEN_WIDTH` = 512, `SCREEN_HEIGHT` = 384, and `CLK_PERIOD` = 10 ns (corresponding to a clock speed of 100MHz) constant throughout evaluation.

A. Memory

To store the vertices used for our rendering, we first considered using DDR2 RAM. This affords us high capacity memory at relatively high speeds with only the minor inconvenience of actually interfacing with the internal DDR2 module. The internal module in question is from Vivado IP called the Memory Interface Generator (MIG). Using Vivado's wizard, we configured the MIG to our needs. The internal module has its own clock for the physical memory, which we have chosen to be 325 MHz, which both divides cleanly from our system clock of 100MHz, allowing for very fast physical memory. The controller clock is at a 4:1 PHY, so it runs four times slower than the memory clock, at 81.25MHz. This necessitates clock domain crossing for reading and writing data.

However in practice, managing the DDR2 module proved to be extremely difficult. DDR2 is fast, but it was slow enough that it took a couple hundred cycles before it could fully fulfill a request. From an initial implementation, this caused pixels to be offset from where they were requested. In addition, managing DDR2 came with the responsibility of an added clock domain to the project. This would mean we would have to manage crossing clocks between the trio of our system clock (which makes write requests), the DDR2 clock, and pixel clock (which makes read requests and controls writing to the screen). The extensive use of BRAMs to weave information between these clock domains in conjunction with the inferred BRAM usage by the controller for buffering the reading and writing of pixels pushed our BRAM utilization to its limit. We needed to be able to buffer the pixels coming from DDR2 because of the variable response cycle count to ensure we put each pixel in the right place. However, we were already at 88% BRAM usage, which made it impossible to buffer pixels at any reasonable rate.

We thus decided to drop DDR2 from our project entirely. We shrunk our target resolution from 1024×768 to 512×384, a four-fold reduction. We now use a simple dual port BRAM for our frame buffering purposes, and scale the pixels up to the original resolution 1024×768 size. Had we more BRAM resources on our board, we would have been able to properly buffer the pixels coming out of the DDR2 before painting them, which would have allowed us to paint at the full 1024×768 resolution, but switching to BRAM was the best possible course of action at the point we made the decision. We ended with 53% usage using BRAM for our frame buffer (for rasterization mode), saving precious space and expanding the amount of instructions and triangles that we store.

B. DSP and LUT

We make extensive use of Vivado floating-point IP in our design to perform geometry and rendering calculations which dominate the compute resources. For the `float_add_sub` and `float_multiply` and floating-point modules, Vivado provides the option to synthesize with no DSP usage (i.e. all LUT), half DSP usage, or full DSP usage. As a result, we see a clear trade-off in the DSP/LUT resource utilization of our system.

For each of `float_add_sub` and `float_multiply`, we consider synthesizing using no DSP usage or full DSP usage, resulting in four total combinations for each of our designs.

1) Rasterization: Synthesizing and implementation complete with no issues for both the full-DSP and no-DSP variants of `float_add_sub` and `float_multiply`.

`float_add_sub`	`float_multiply`	LUT usage	DSP usage
LUT	LUT	10 %	8%
LUT	DSP	N/S%	N/S%
DSP	LUT	N/S%	N/S%
DSP	DSP	8 %	20%

TABLE I: LUT and DSP utilization for rasterization renderer design for various implementation choices of floating point modules. N/S indicates not synthesized.

2) Raytracing: The DSP/LUT utilizations for the raytracing rendering mode are shown in Table II. We found that the raytracing design did not synthesize when we attempted to use DSP for both `float_add_sub` and `float_multiply`.

`float_add_sub`	`float_multiply`	LUT usage	DSP usage
LUT	LUT	60.13 %	1.02%
LUT	DSP	45.87%	58.75%
DSP	LUT	80.42%	48.66 %
DSP	DSP	N/A	N/A

TABLE II: LUT and DSP utilization for raytracing renderer design for various implementation choices of floating point modules. N/S indicates not synthesized. N/A indicates synthesis failed due to lack of resources.

As seen in TableII, we do not make full use of the compute resources available with our current raytracing design. We could better utilize the resources available to us by increasing the number of raycast modules, since this would increase the throughput of our raytracing computations proprtionally. This would require more overhead to manage a more complex controller, as well as requiring more read ports on the memory bank to read shapes and lights at an increased rate to pass to the raycasters.

We synthesize just the raycaster to getter a better sense of the maximum possible throughput of our system. As seen in Table III, if we use LUT for `float_add_sub` and DSP for `float_multiply`, we could fit an additional raycaster onto our raytracing renderer, thereby doubling the throughput of our design. As discussed in V-C2, this would result in halving the render time for a single frame (assuming that raycasts could be scheduled efficiently with a more advanced rendering controller).

`float_add_sub`	`float_multiply`	LUT usage	DSP usage
LUT	LUT	40.60%	0%
LUT	DSP	24.24%	37.08%
DSP	LUT	34.24%	57.50%
DSP	DSP	17.82%	94.58%

TABLE III: LUT and DSP utilization for raycaster module for various implementation choices of floating point modules.

C. Performance

We measure the performance of our system based on the number of geometric objects being rendered, the rendering mode, and the time to render a single frame. We provide performance estimates based on calculations, and in cases where measurement was possible, we provide actual render times as well.

1) Rasterization: The rasterization pipeline is split into two sections. One section performs the hard calculations which transforms the three-dimensional triangles to the two-dimension space mapped to our screen. The second section performs the half-plane check, determining if a triangle is visible for any given pixel position on the screen. The first section has a latency of 63 cycles for every triangle calculated. The second has a latency of four cycles and needs to be run for every pixel address in our resolution space before it is complete, and for every triangle calculated. Our resolution space is $512 * 384 = 196608$ in size, so this totals to $63 + 4 + 196608 = 196675$ cycles before it can write a full triangle's worth of information to the frame buffer. Using a clock period of 10ns, this equates to a latency of 1.96ms for calculating the frame buffer for single triangle, linearly scaling the number of triangles in the frame.

2) Raytracing: We were able to make performance estimates for raytracing mode based on the specified behavior of our system. Our implementation of the fully-pipelined raycaster module has a latency of 205 cycles. So, intersecting a single ray with all shapes requires `NUM_SHAPES` + 205 cycles (plus some small number of cycles to tabulate results). And, thus computing the value of a single pixel requires (`NUM_SHAPES` + 205) * (`NUM_LIGHTS` + 1) cycles since we require one initial ray and `NUM_LIGHTS` lighting rays after the intersection is computed. So, the total number of cycles to render a single 3D frame is approximately (`NUM_SHAPES` + 205) * (`NUM_LIGHTS` + 1) * `NUM_PIXELS`. If material effects were implemented, this would simply add another multiplicative factor to the cycle computation to process the recursive raytraces. We fix `NUM_PIXELS = 512 * 384 =` 196608. Using a clock period of 10 ns, we produce estimates for frame render times, as shown in Table IV.

`NUM_SHAPES`	`NUM_LIGHTS`	Estimated render time (s)
1	0	0.405
16	0	0.435
256	0	0.906
4096	0	8.456
1	1	0.810
16	1	0.869
256	1	1.813
4096	1	16.912

TABLE IV: Estimated raytracing render times (in seconds) for a single frame as `NUM_SHAPES` and `NUM_LIGHTS` vary.

As seen in Table IV, the constant factor in running time dominates for small `NUM_SHAPES` (e.g. ≤ 16), but for larger values (e.g. ≥ 256), the linear term dominates. The running time grows linearly with the number of lights.

We note that the multiplicative factor for lighting is a worst case estimate; currently, our design does not execute lighting raytraces if no shape is hit for a given pixel. Additionally, in the case that lighting must be calculated, the multiplicative factor for lighting could be improved by terminating lighting raytraces whenever a shape is hit, rather than waiting to intersect the ray with all shapes. This is because once a shape is hit, we know the light source is blocked, so we don't need to continue raycasting. While the worst case runtime would be the same, this heuristic would strictly improve the average runtime since it amounts to early termination of a raytrace. We don't implement this second early termination of lighting heuristic in our present design.

We also measure the actual render times with one light and 2 shapes. The estimated number of cycles (using our formula) for this setting is 81,395,712 $\approx 8.1 \cdot 10^{10}$ and the estimated runtime is 0.814 seconds. The actual measured number of cycles is 45,766,350 $\approx 4.5 \cdot 10^{10}$, corresponding to a run time of 0.458 seconds. We think the number of cycles is lower than estimated due to the fact that not all pixel rays hit a shape, and so the number of lighting rays is significantly less than computed, as explained above.

VI. Implementation

Our implementation is hosted at https://github.com/vedadehhc/rend3r. Note that raytracing is on a separate branch of the repository while rasterization is on the main branch.

References

[1] Dodgson, Neil. "Ray Tracing Primitives." Ray Tracing Primitives, University of Cambridge, 29 Oct. 1999, https://www.cl.cam.ac.uk/teaching/1999/AGraphHCI/SMAG/node2.html.

[2] Scratchapixel. (2015, January 25). Rasterization, A Practical Implementation. Rasterization: A practical implementation (an overview of the rasterization algorithm). Retrieved December 14, 2022, from https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation