ECE 385

Fall 2024

Final Project

3D Rendering & Minecraft

Rohan Das (rohan24)

Kai Karadi (kaick2)

TA: Nuoyan Wang (NW)
12/18/24

Introduction

The goal of this project was to develop an SoC (System-on-Chip) on an FPGA board to perform 3D rendering on a monitor and run a simplified version of the videogame *Minecraft*. The project would take keyboard input and display a 2x2x2 3D rendered Minecraft world on a monitor using 640x480 HDMI video format with 320x240 resolution. The keyboard would give the user complete control over the game's virtual camera to simulate the player/camera moving in all 6 directions in the game (forward, back, right, left, up, and down) as well as rotating the camera left, right, up, and down. The user would also be able to break and place blocks in the simulated world. The project uses triangle rasterization to achieve 3D rendering by converting the world to a series of colored triangles with screen coordinates and then displaying layers of colored triangles such that triangles in the front cover those in the back, achieving depth.

The project consisted of both a hardware component, including a MicroBlaze Block Design, IP blocks, and custom SystemVerilog modules, as well as a software component written in C which runs on the MicroBlaze softcore processor created in the hardware Block Design.

The software stored the composition of the *Minecraft* world as well as the virtual camera's position. Based on user keyboard input, handled by the software with code adapted from Lab 6, the software would change the position of the camera and thereby transform the rendered triangles from world space (in their vertex coordinates are based on xyz coordinates in the virtual *Minecraft* world) to raster space (in which each vertex is mapped to a pixel on the screen), writing the associated data to an index buffer, vertex buffer, and depth buffer in BRAM (the latter of which ended up not being used).

The hardware consisted of a rasterizer module, 2 frame buffers in BRAM, a color mapper, a VGA signal generator, and a VGA to HDMI IP converter. The rasterizer module would read data from the index, vertex, and depth buffers and using the rasterization algorithm write the corresponding data to print to a screen to a frame buffer. Simultaneously, the color mapper would read from the other frame buffer, outputting the color of the pixel specified by the VGA module. After the rasterizer finished writing to its frame buffer, both the rasterizer and color mapper would switch frame buffers. Reading and writing to the frame buffer separately avoided glitches which might occur if the same frame buffer was read and written simultaneously. The resulting VGA module would then be converted to a HDMI signal by the VGA to HDMI IP converter before being outputted to a monitor.

Overall, we were able to achieve most of our goals for this project, being able to fully move the camera and break/place blocks in a virtual Miencraft world, albeit with the world being smaller than originally planned due to bugs

Final Project Description & Features

The project displays a 2x2x2 virtual Minecraft world in 320x240 resolution on a monitor where the user has full control in all directions of the virtual camera and is able to place/break blocks using a keyboard. WASD moves the virtual camera forwards, left, backwards, and right respectively, IJKL rotates the camera down, left, up and right respectively, the space bar moves the camera up (equivalent to flying/jumping in Minecraft), c moves the camera down,] places/replace a stone block at the block location closest to the camera, and [removes the block closest to the camera, replacing it with air. The project supports 6 types of blocks: air (the

absence of a block), grass, wood, stone, leaves, and water, although only grass, water, and stone are actually implemented. Each block has monocolor faces, although the faces for a specific block may be different colors (for grass the top face is green while the side and bottom faces are brown). The world is generated as a 2x2 world of grass and water with a light blue background (similar to the sky in Minecraft) with blocks being able to be placed underneath it (for a full 2x2x2 world) and the user being able to place stone blocks.

3D Rendering Process & Project Design

Converting from World Space to Raster Space

Setup

In the software, each block is assigned an index (ex. air is 0, grass is 1, and water is 5). A 3-dimensional array world_def stores the index of each block based on its xyz location in the Minecraft world. set_up_world() sets world_def as previously described when the program is started. A camera struct, which stores the camera's yaw, pitch, position, and forward, right, and up positions, is then initialized. The program then enters an endless loop during which first the program detects if a key on the keyboard is pressed using code adapted from Lab 6. If WASD, IJKL, or Space Bar or C are pressed, then the camera struct is transformed accordingly using matrix transformations to move/rotate the camera, If [or] is pressed, then the will_remove and will_place values are set high respectively, indicating that a block should be placed/removed. write_vertex_buffer(), which writes to the vertex buffer, is then called.

Writing to the Vertex Buffer

The vertex buffer, stored in BRAM which the Microblaze directly writes to, stores the raster coordinates of every vertex in the Minecraft world (since the world is 2x2x2 blocks, there are 3x3x3 = 27 vertices). write_vertex_buffer() loops through every vertex based on its coordinates in world space (xyz coordinates where each value ranges from 0 to 2) in a predefined row-major order such that the first vertex is vertex 0, the second is 1, and the last is 26. point_index_to_triple() converts the current vertex index which the write_vertex_buffer loops through to its xyz world coordinates. Matrix transformations are then used to transform the current vertex's world coordinates to camera space (which reflects its coordinates from the view of the camera based on its orientation and position) and then clip space, where each dimension's coordinate for the vertex is represented on a scale from -1 to 1 based on its position on the screen such that if the x or y coordinates are not in the -1 to 1 range, then the vertex is not actually currently on the screen and is be rendered. world_to_clip() accomplishes this. clip_to_rast() then uses matrix transformations to map the 3-dimensional clip coordinates to raster space, in which the x value ranges from 0 to 319 and the y value from 0 to 239 (and the z value from 0 to 255), setting the vertex to corresponding to a specific pixel on the 320x240 screen). The x and y raster coordinates are then written to the vertex buffer. The vertex buffer is 32 bits wide with each vertex using one memory address where its x value is written to the lower 16 bits and the y value to the higher 16 bits. The z raster coordinate is not directly used (but must be calculated for the matrix transformations to work).

Writing to the Index Buffer

After write_vertex_buffer finishes writing to the vertex buffer, write_index_z_buffer() is called, which writes to the index and depth buffers. The function loops through every face of every block in the Minecraft world, with each face consisting of two triangles which are rendered. For each of the two triangles per face, get_triangle_verticies() is called. It takes the current cubes' xyz coordinates and the current triangle number per cube and calculates the xyz world space coordinates of all three of its vertices before calling triple_to_point() to convert them to a vertex index. Both the indices and world space coordinates of all three vertices are returned. From the vertex coordinates for both triangles, the four unique vertices which form the faces are averaged using average_vectors() with the resulting point representing the entire face then being converted to clip space and then raster space using world_to_clip() and clip_to_rast(). If the average point's z dimension clip value is between -1 and 1 (indicating it should be rendered on the screen), then triangle_to_word() is called and returns a 32 bit word where the highest bit is empty, the next 27 bits store the indices of the vertices for the triangle (using 9 bits for each index), and the lower 4 bits store the color palette index of the face (calculated using the current block number and id from world_def, face number, and palette indices stored in the block_palette array). This is repeated for both triangles on the face. If the average point's z clip value is greater than 1 or less than -1, then the triangle's color palette index is set to 0 in its 32 bit word. In the original design for the project, the vertex and color index data for each triangle were written to the index buffer and the average point's 8 bit z clip value was written to the depth buffer (such that the z value for four triangles was stored in one 32 bit address space of memory). The rasterizer would then loop through the index buffer, for each triangle looking up its vertex coordinates in the vertex buffer and coloring all of the points in between them in the frame buffer with the specified color and resolve overlap between triangles using the values stored in the buffer. However, the rasterizer module, while it worked correctly in simulation, did not work correctly in actual hardware in correctly interpreting and using data from the depth buffer (although it was able to otherwise render all of the triangles, with overlap occuring in the order in which the index buffer was written). As a result, a workaround was developed in software using the painter's algorithm, an alternative to the z-buffering technique which we originally planned to use. A dist_array was created which stored key-value pairs with 2 keys and a value per array index where the 2 keys were the 32 bit words for the 2 triangles per face and the value was the average point's z clip value for that face. write_index_z_buffer() would add a new key-value pair to this array for every face. The quicksort algorithm (which was taken from online sources) was then called on dist_array to sort it based on each pair's z value such that faces further from the camera were moved to the front of the array. The keys of dist_array were then written to the index buffer (with each triangle's vertex and color index data filling a 32 bit word) in the order of the dist_array. This allowed for triangles which would be covered to be rendered first by the rasterizer such that triangles in the front rendered later would cover them. However, due to the increased time required to sort the dist_array using quicksort, only a 2x2x2 world or smaller could be supported as in a larger world the sorting would not finish quickly enough, leading to glitches in the monitor output. This is what caused us to scale down our project from a 6x6x6 world to a 2x2x2 world.

Placing/Breaking Blocks

After write_index_z_buffer() is called, the program then checks if the will_remove and will_replace values are high. If they are, then rem()/place() is called respectively, determines the xyz coordinates of the block closest to the camera, sets that block in world_def to air/stone, and sets will_remove/will_replace back to low to avoid removing/replacing more blocks. The infinite loop starting with detecting keyboard inputs and then writing to the vertex and index buffers then restarts.

Rasterization

Through the process of rasterization, the implemented hardware takes the data from the vertex, index, and depth buffers, renders it in 3D, and outputs a VGA video stream which is converted to HDMI before being outputted to a monitor to be viewed.

Rasterization is often not an incredibly well defined word but in our case rasterization will describe the process of taking the Vertex, Index and Depth information in the Buffer, reading it and doing the necessary processing required to write to a frame buffer and a z buffer. This frame buffer can then be read directly by SystemVerilog code that can convert a frame buffer to a VGA signal.

Before our discussion of rasterization beings we must quickly explain how the frame buffer and z buffer are laid out in memory. There are two frame buffers, both 32 x 9600 bits and a z buffer of size 64 x 9600 bits. With the frame buffer every 4 bits represents the pallet index of a pixel in row major order. In the z buffer every 8 bits represents the depth of the pixel in row major order, represented as an unsigned 8 bit number. The information is organized such that the one word from BRAM corresponds to 8 pixels on the screen. From the frame buffer each word is 4x8=32 bits and for the z buffer 8x8=64 bits.

Now we can begin with an overview of the rasterization algorithm. This algorithm is separated into different phases beginning with init state. The init state simply sets all variables to their default values.

The next state is the load triangle state. This seeks to, for a triangle at index tri_idx in the frame buffer, load the three x,y positions that represent its vertices, its pallet index, and the average depth of the triangle all from the vertex, index and depth buffers. This works first by reading a triangle from the index buffer, which stores its vertex and color indices. It then reads the vertex coordinate data of the three vertices corresponding to its vertex indices from the vertex buffer. The triangle's average depth value from the depth buffer is also read. This state then calculates the bounding box that surrounds the triangle. The rest of the rasterization involves moving a 1x8 pixel window called a "span", located mostly inside the bounding box. And fills in the frame buffer at the span's location with the correct pixel. Because of this the load triangle state finishes by calculating the x and y of the first span for the current triangle. This state is also responsible for skipping triangles with pallet index 0 (air), and going to the await_buff_swap state if all triangles have been processed.

The next state is the load_cache state. Since the location of the first span is known from the load triangle state, the corresponding pixel indices and pixel depths can be extracted from the frame buffer for the 8 pixels in the span.

The next state is pineda which executes an algorithm known as pineda algorithm. This algorithm allows, given a single pixel location (x,y) and the (x,y) coordinates of the vertices of a

triangle of a triangle, determines if a point falls within or outside of said triangle. Pinedas is executed in parallel for all 8 pixels. If Pinedas determines that the pixel does fall inside of the triangle and its depth (the depth of the current triangle) is less than the current depth of pixel as stored in the z cache, then the frame cache is updated with the current pallet index and the z cache with the current depth value.

The next stage is store_cache. Now that the frame cache and z cache have been updated to confirm with the current triangle being processed, they are written back into the frame and z buffer memory.

The next stage is known as increment_span. The span window must be moved across the bounding box in row major order. This stage increments the span variables so this behavior is created. If the span has been incremented enough so that the entire bounding box has been covered, then the triangle index is incremented and the state returns to load triangle and this process repeats.

Once all triangles in the index buffer have been processed, the await_buff_swap is called. This state is a waiting state that only transitions to the clear_buff_init state when the vsync pulse comes from the VGA, indicating that it has finished displaying a single frame and that it is time to swap frame buffers. When this happens the framebuffer the rasterizer writes to swaps with that of the color mapper and the clearing of the z and new framebuffer begins with the clear_buff_init state.

In the clear_buff_init all the framebuffer and z buffer BRAM control signals are initialized to set each word in the BRAM to the value 0x0000_0000 (the value of air or empty pallet). The depth buffer signals are initialized to set it to the value 0xFFFF_FFFF_FFFF_FFF (or the furthest depth).

After this the clear_buff_init goes to the clear_buff state which increments the addresses of both buffers so that all values are cleared. This then returns to the load triangle state, restarting the whole process.

Simultaneously to the rasterizer module writing to a frame buffer, the color mapper module reads from the other frame buffer. The VGA module generates a VGA signal with the current pixel being represented by (DrawX, DrawY). The color mapper reads the color index for this pixel from the frame buffer it is reading from and based on the color palette which it stores, outputs the corresponding 12-bit RGB value of that pixel. To avoid the rasterizer and the color mapper writing to/reading from the same frame buffer, which can lead to glitches in the video output, two frame buffers were implemented such that the rasterizer would write to one while the color mapper would read from the other. When the VGA module finishes looping through an entire frame, the rasterizer and color mapper switch frame buffers and continue reading/writing. This creates a constantly updating stream of color data corresponding to the VGA signal, both of which are inputted into the RealDigital VGA to HDMI Converter, which converts the analog VGA signal to a digital HDMI signal which can then be outputted to a monitor.

Modules

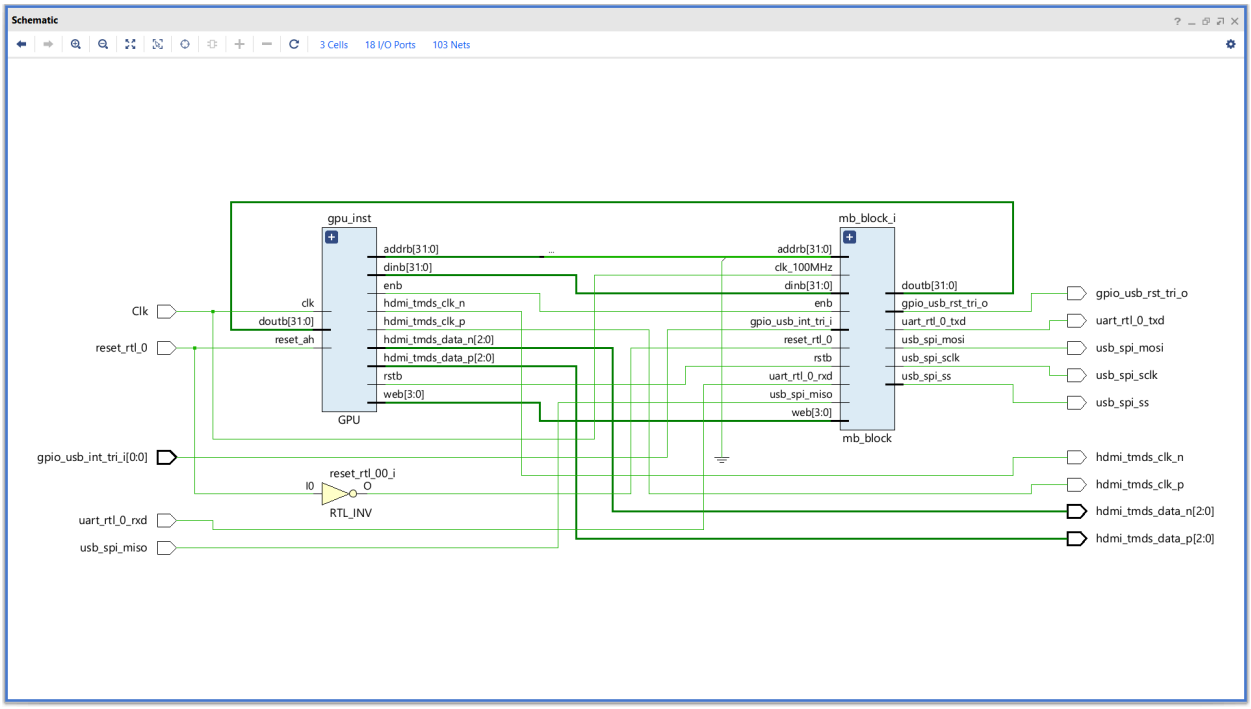

Figure 3. Toplevel RTL Schematic

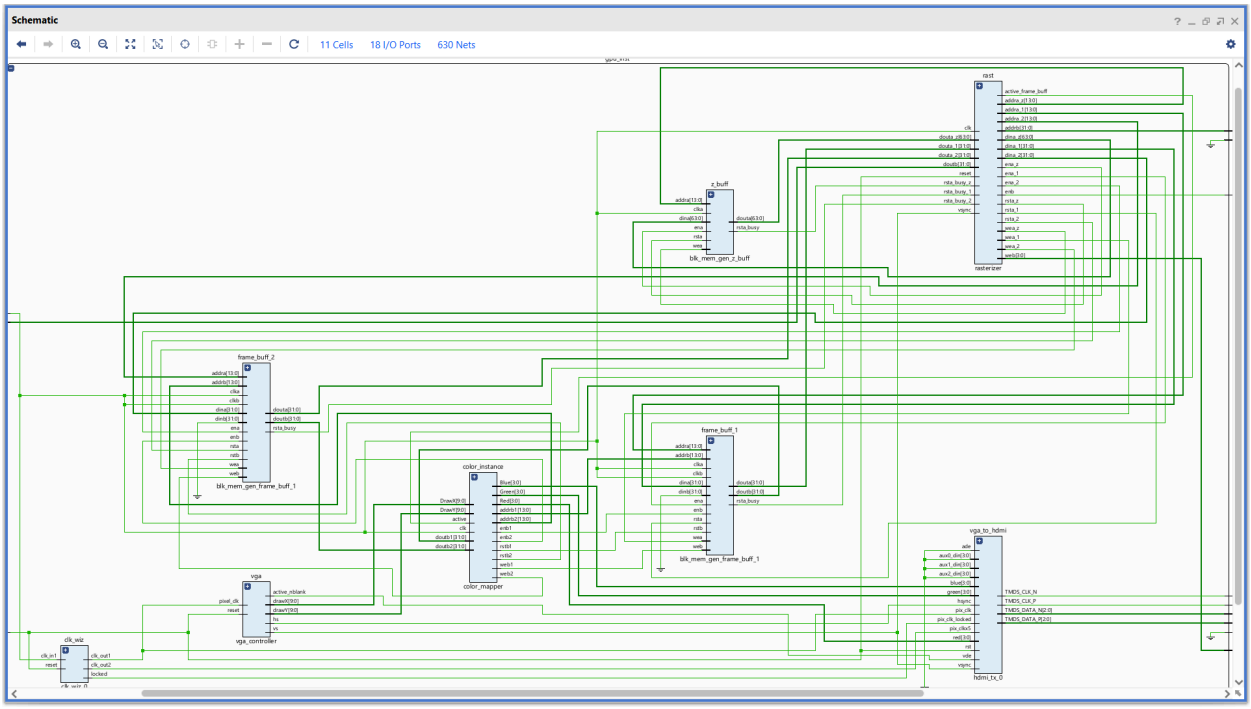

Figure 4. RTL Schematic of the GPU

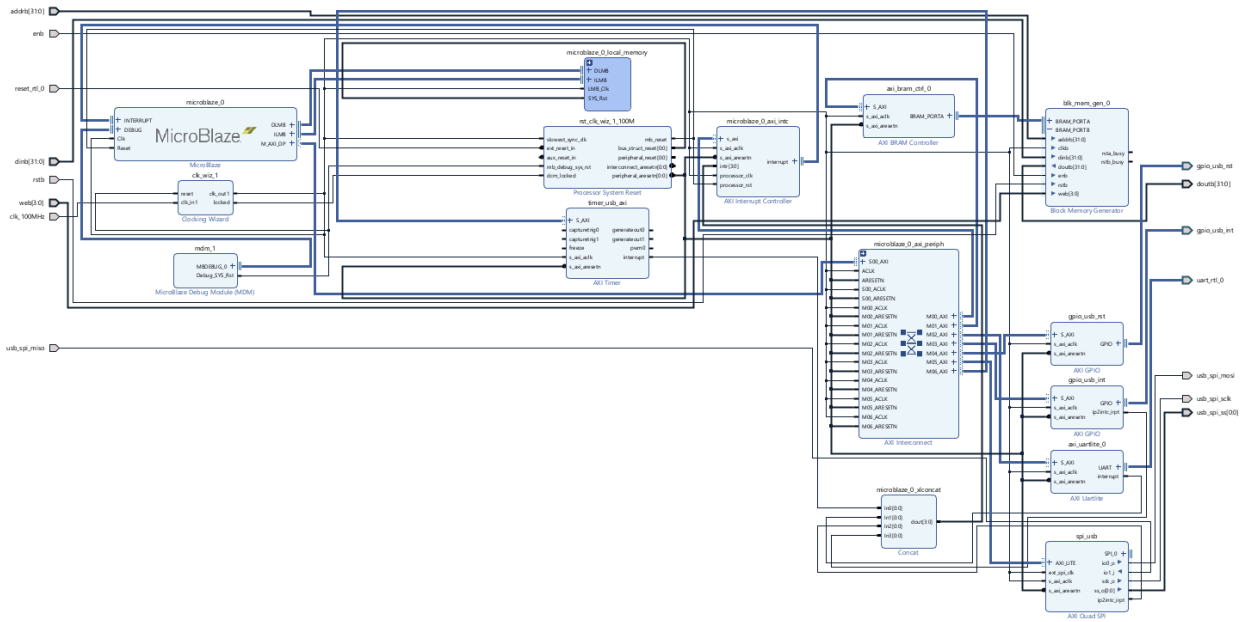

Figure 5. RTL Schematic of the MicroBlaze Block Design

SystemVerilog Modules

Module: `final_project_top.sv (toplevel)`
Inputs: `Clk, reset_rtl_0, [0:0] gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd`
Outputs: `gpio_usb_rst_tri_o, usb_spi_mosi, usb_spi_sclk, usb_spi_ss, uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n,[2:0] hdmi_tmds_data_p`
Description: `final_project_top.sv` is the toplevel for the hardware portion of our final project. It takes in and out all of the external inputs and outputs for the entire hardware: UART (to communicate debug information to Vitis over Micro-USB), USB/SPI (to communicate with the keyboard), HDMI (to communicate to a monitor), and the reset and clock signals. It also instantiates the MicroBlaze Block Design and the GPU module, connecting them with the appropriate ports.
Purpose: This module instantiates the MicroBlaze Block Design and GPU module and connects them with external inputs and outputs (clk, reset, USB/SPI, UART, HDMI signals, and BRAM interfaces).

Module: `GPU.sv (toplevel)`
Inputs: `clk, reset_ah, [31:0] doutb`
Outputs: `hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n,[2:0] hdmi_tmds_data_p, [31:0] addrb, [31:0] dinb, enb, rstb, [3:0] web`
Description: `GPU.sv` is the toplevel for the hardware portion of our final project which does all of the hardware computation (the GPU). It takes in and out all of the external inputs and outputs

for the GPU unit including the clock and reset signals, HDMI (to communicate to a monitor), and the Port B interface for BRAM 0 (see IP Modules). It also instantiates Frame Buffer 1, Frame Buffer 2, the Z Buffer, Clock Wizard 0, the VGA Controller, the Color Mapper, the Rasterizer Module, and the Real Digital VGA to HDMI Converter IP Block.
Purpose: This module instantiates Frame Buffer 1, Frame Buffer 2, the Z Buffer, Clock Wizard 0, the VGA Controller, the Color Mapper, the Rasterizer Module, and the Real Digital VGA to HDMI Converter IP Block and and connects them with external inputs and outputs (clk, reset, HDMI signals, and the BRAM 0 Port B interfaces).

Module: `VGA_controller.sv`
Inputs: `pixel_clk, reset`
Outputs: `hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY`
Description: `VGA_controller.sv` controls the VGA output of the hardware and which pixels are currently being written to. It uses a horizontal and vertical counter to increment in raster order through the 800x525 pixels (with 640x480 being visible) with `drawX` and `drawY` being the coordinates of the current pixel being written to. `hs` and `vs` (horizontal and vertical sync) are calculated based on the current pixel position and are used to synchronize when to switch between horizontal lines and start a new frame and art active low. `Active_nblank`, also based on the current pixel's location, is high when the pixel is within the visible part of the VGA screen. `sync` is not used in this implementation.
Purpose: The VGA controller outputs the location of the current VGA pixel being written to as well as horizontal and vertical sync information to control switching between horizontal lines and frames and an active_nblank to indicate whether the current pixel is visible or blanking. It takes in a reset signal to reset the current frame, and a 25MHz clock from the clock wizard to control when each pixel is updated. The VGA data is outputted to the RealDigital VGA to HDMI converter to convert it to HDMI so that it can be displayed on a monitor.

Module: `Color_mapper.sv`
Inputs: `clk, active, [9:0] DrawX, [9:0] DrawY, [31:0] doutb1, [31:0] doutb2`
Outputs: `[3:0] Red, [3:0] Green, [3:0] Blue, enb1, [3:0] web1, [13:0] addrb1, [31:0] dinb1, enb2, [3:0] web2, [13:0] addrb2, [31:0] dinb2,`
Description: `Color_mapper.sv` sets the color of the current pixel based on the data specified by the frame buffer. active controls which frame buffer the color mapper reads from (Frame Buffer 2 if active is 0 and Frame Buffer 1 if it is 1). For a given frame buffer, It calculates the location of the pixel specified by (DrawX, DrawY) in the frame buffer BRAM memory. This includes accounting for the frame buffer corresponding to a 320x240 screen while the VGA module outputting (DrawX, DrawY) sets a 640x480 screen. To resolve this every pixel in the frame buffer corresponding to four pixels on the actual monitor output. It then reads the current pixel's color palette index from the frame buffer BRAM and based on that index and the color palette lookup table which it stores, outputs the corresponding 12-bit RGB color to the VGA to

HDMI Converter. To communicate with BRAM it uses port B of the BRAM block to read data and uses the BRAM communication interface.

Purpose: `Color_mapper` outputs the color of the current pixel in 12-bit RGB format (where every color is 4 bits wide). The current pixel specified by (`DrawX`, `DrawY`) is inputted from the VGA controller, with the color mapper determining the location of the pixel in the current frame buffer and reading it to get its color index from the frame buffer BRAM (using port B). It then outputs the calculated color of the current pixel based on the color palette lookup table which it stores to the VGA to HDMI converter to convert it and the VGA data from the VGA controller to HDMI to be displayed on a monitor. It switches between both frame buffers based on the active signal to avoid it and the rasterizer from reading/writing from the same buffer simultaneously, which could cause glitches.

Module: `rasterizer.sv`

Inputs: clk, [31:0] doutb, [31:0] douta_1, rsta_busy_1, [31:0] douta_2, rsta_busy_2, [63:0] douta_z, rsta_busy_z, reset, vsync

Outputs: active_frame_buff, enb, [3:0] web, [31:0] addrb, [31:0] dinb, rsta_1, ena_1, [0:0] wea_1, [13:0] addra_1, [31:0] dina_1, rsta_2, ena_2, [0:0] wea_2, [13:0] addra_2, [31:0] dina_2, rsta_z, ena_z, [0:0] wea_z, [13:0] addra_z, [63:0] dina_z,

Description: `rasterizer.sv` takes the data from the vertex, index, and depth buffers written to by the MicroBlaze and carries out the rasterization algorithm using triangles, writing the result to a frame buffer which is then converted to a VGA/HDMI signal and outputted. The rasterizer reads from the vertex, index, and depth buffer (stored in the same BRAM block and referenced in IO ports using …b), writes/reads from the z buffer (the BRAM block of which is referenced in IO ports by …_z), and writes to two frame buffers (the BRAM blocks of which are referenced in IO ports by …a_1 and …a_2). The rasterizer uses a state machine, starting with the init state which simply sets all variables to their default values. The next state is the load triangle state. This seeks to, for a triangle at index tri_idx in the frame buffer, load the three x,y positions that represent its vertices, its pallet index, and the average depth of the triangle all from the vertex, index and depth buffers. This works first by reading a triangle from the index buffer, which stores its vertex and color indices. It then reads the vertex coordinate data of the three vertices corresponding to its vertex indices from the vertex buffer. The triangle's average depth value from the depth buffer is also read. This state then calculates the bounding box that surrounds the triangle. The rest of the rasterization involves moving a 1x8 pixel window called a "span", located mostly inside the bounding box. And fills in the frame buffer at the span's location with the correct pixel. Because of this the load triangle state finishes by calculating the x and y of the first span for the current triangle. This state is also responsible for skipping triangles with pallet index 0 (air), and going to the await_buff_swap state if all triangles have been processed. The next state is the load_cache state. Since the location of the first span is known from the load triangle state, the corresponding pixel indices and pixel depths can be extracted from the frame buffer for the 8 pixels in the span. The next state is pineda which executes an algorithm known as pineda algorithm. This algorithm allows, given a single pixel location (x,y) and the (x,y) coordinates of the vertices of a triangle of a triangle, determines if a point falls within or outside of said triangle. Pinedas is executed in parallel for all 8 pixels. If Pinedas determines that the pixel does fall inside of the triangle and its depth (the depth of the current

triangle) is less than the current depth of pixel as stored in the z cache, then the frame cache is updated with the current pallet index and the z cache with the current depth value. The next stage is store_cache. Now that the frame cache and z cache have been updated to confirm with the current triangle being processed, they are written back into the frame and z buffer memory. The next stage is known as increment_span. The span window must be moved across the bounding box in row major order. This stage increments the span variables so this behavior is created. If the span has been incremented enough so that the entire bounding box has been covered, then the triangle index is incremented and the state returns to load triangle and this process repeats. Once all triangles in the index buffer have been processed, the await_buff_swap is called. This state is a waiting state that only transitions to the clear_buff_init state when the vsync pulse comes from the VGA, indicating that it has finished displaying a single frame and that it is time to swap frame buffers. When this happens the framebuffer the rasterizer writes to swaps with that of the color mapper and the clearing of the z and new framebuffer begins with the clear_buff_init state. In the clear_buff_init all the framebuffer and z buffer BRAM control signals are initialized to set each word in the BRAM to the value 0x0000_0000 (the value of air or empty pallet). The depth buffer signals are initialized to set it to the value 0xFFFF_FFFF_FFFF_FFF (or the furthest depth). After this the clear_buff_init goes to the clear_buff state which increments the addresses of both buffers so that all values are cleared. This then returns to the load triangle state, restarting the whole process.
Purpose: The rasterizer converts the vertex, index, and depth data calculated in software and written by the MicroBlaze into a frame buffer which the color mapper can then read and in combination with the VGA module convert to a VGA data stream. It loops through every triangle in the index buffer, getting its vertex coordinates from the vertex buffer and average depth from the depth buffer. It then uses the rasterization algorithm to in the pixel frame buffer fill in the area inside of each triangle with that triangle's color and handle overlap through z-buffering, which involves writing to and reading from the z-buffer (although this function did not end up working correctly in hardware, while it did work correctly in simulation). The final front color value of each pixel is eventually written to the frame buffer, which can then be read by the color mapper to generate a VGA signal with the VGA module. The rasterizer and color mapper also switch frame buffers every VGA frame (indicated by the vsync signal) to avoid writing/reading to the same frame buffer, which could cause glitches.

IP Modules

MB Block

- **MicroBlaze:** The MicroBlaze Processor is the processor which the written software for the final project is run on. It uses a modified Harvard architecture, memory-mapped IO, and includes a full C compiler. It is set to 32-bits in microcontroller mode, is optimized for area, and uses AXI to communicate with peripherals.
- **MicroBlaze Debug Module:** The MicroBlaze Debug Module, which can be enabled/disabled when setting up the MicroBlaze, is used for JTAG-based software bugging of the MicroBlaze processor and can communicate with a laptop/debugging software using AXI UARTLite.

- **Local Memory:** The Local Memory is the memory of the MicroBlaze processor and uses BRAM to store data. For the final project it was configured to 128 KB.
- **Processor System Reset:** The Processor System Reset controls the reset signals for all of the IP blocks in the block design. It takes as an input the overall block reset signal and outputs the correct (active low vs active high) reset signals to all IP blocks including the MicroBlaze processor, peripherals, and the debug module.
- **Clocking Wizard 1:** Clocking Wizard 1, located inside the MicroBlaze Block Design, takes in the 100MHz clock input into the Block Design and outputs a 100MHz clock signal to the rest of the IP blocks which can be reset by the Processor System Reset.
- **AXI Interconnect:** The AXI Interconnect connects to all of the IP block peripherals and to the MicroBlaze over separate AXI busses, allowing the MicroBlaze to control and communicate with each of them through a single AXI bus without having to be directly connected to each, minimizing the number of ports on the MicroBlaze processor. It also supplies their reset signals (from the Processor System Reset) and 100MHz clock signals (from the Clocking Wizard). The AXI interface implementation includes an AXI4-Lite control interface. It also has the ability to allow multiple masters to communicate with multiple slaves.
- **AXI Interrupt Controller:** The AXI Interrupt Controller takes as input the 4 bit concatenated input from the Interrupt Concat and outputs the corresponding interrupt signal to the MicroBlaze Processor. It is configured for four peripherals with interrupts and communicates with the MicroBlaze over AXI through the AXI Interconnect.
- **AXI UARTLite:** AXI UARTLite is a peripheral which communicates with the MicroBlaze Processor over AXI through the AXI Interconnect. It is used by the Debug Module to send debug information over UART through the Micro USB cable connected to the FPGA board to a laptop where it can be viewed on a serial monitor.
- **Interrupt Concat**: In part 2 the Interrupt Concat concatenates the 1 bit interrupt signals coming from the AXI Timer, AXI UARTLite, AXI Quad SPI, and USB Interrupt AXI GPIO and outputs the concatenated 4 bit result to the AXI Interrupt Controller. Because in part 1 only one IP block outputs an interrupt (The LED AXI GPIO), it is not needed.
- **AXI Timer:** The AXI Timer provides timing signals for USB communication, which requires many timeouts measured in milliseconds. The AXI Timer allows the MicroBlaze system to keep track of these time periods, allowing for communication to another device over USB. It is connected to the MicroBlaze through the AXI Interconnect.
- **AXI UARTLite:** AXI UARTLite is a peripheral which communicates with the MicroBlaze Processor over AXI through the AXI Interconnect. It is used by the Debug Module to send debug information over UART through the Micro USB cable connected to the FPGA board to a laptop where it can be viewed on a serial monitor.
- **USB Reset AXI GPIO:** The USB Reset AXI GPIO is used as a peripheral of the MicroBlaze processor which communicates with it over AXI through the AXI Interconnect. It is configured as a 1 bit output used to send a reset signal over USB to the connected device (the keyboard).
- **USB Interrupt AXI GPIO:** The USB Interrupt AXI GPIO is used as a peripheral of the MicroBlaze processor which communicates with it over AXI through the AXI Interconnect. It is configured as a 1 bit input used to receive an interrupt signal over USB

from the connected device (the keyboard). The interrupt signal is connected to the Interrupt Concat.

- **AXI Quad SPI:** The AXI Quad SPI is a peripheral of the MicroBlaze Processor which communicates with it over AXI through the AXI Interconnect. It is used to communicate with other devices over SPI and can support up to four devices when set as a master. It is configured to master mode with one slave (the MAX3421E USB chip on the FPGA board). It outputs SCLK (Slave Clock), SS (Slave Select), and MOSI (Master-Out Slave-In) to the MAX3421E and an interrupt to the Interrupt Concat and receives MISO (Master-In Slave-Out) from the MAX3421E through SPI. It can receive memory mapped read and write instructions and convert them to the SPI transactions of the MOSI and MISO.
- **AXI BRAM Controller:** The AXI BRAM Controller is a peripheral of the MicroBlaze Processor which communicates with it over AXI through the AXI Interconnect. It allows the MicroBlaze to write and read from BRAM 0 (containing the Vertex, Index, and Depth Buffers) and connects to its port A using the native BRAM interface. It also sets BRAM Block 0 to be 32 bits wide and have 4096 memory spaces.
- **Block Memory Generator 0:** The Block Memory Generator generates a block of BRAM memory inside the Block Design which stores the Vertex, Index, and Depth Buffers. It stores 4096 32 bit words. The Vertex Buffer starts at address 0, the Depth Buffer at address space 200, and the Index Buffer at address space 1000. It is set to True Dual Output such that it has two IO ports, port A (which is connected to the MicroBlaze AXI module and is used to write and read data), allowing the software the MicroBlaze runs to write read to the buffers, and port B (which is connected to the rasterizer and exclusively used to read data). It communicates with both the AXI module and the rasterizer using the native BRAM interface. The BRAM is also synchronous and takes up to 2 clock cycles to read data (since its output register is disabled).

GPU:

- **Clock Wizard 0:** Clock Wizard 0, instantiated by the GPU, takes as input the FPGA 100MHz and outputs 25MHz and 125MHz clock signals. 25MHz is set as the frequency at which the VGA controller switches pixels while both the 25MHz and 125MHz clock signals are needed for the Real Digital VGA to HDMI IP module (which needs the frequency at which pixels are updated and 5 times that frequency as clock inputs).
- **Frame Buffer 1 Block Memory Generator:** The Block Memory Generator generates a block of BRAM memory inside the GPU which stores Frame Buffer 1. It stores 9600 32 bit words with each group of four bits storing the color index for a pixel on the 320x240 screen (such that 8 pixels are represented per word). It is set to True Dual Output such that it has two IO ports, port A (connected to the rasterizer), allowing the rasterizer to write to the buffer, and port B (connected to the color mapper), allowing the color mapper to read the color index for the pixel specified by the VGA module. The rasterizer and color mapper take turns writing/reading from the frame buffer, such that when one is interfacing with frame buffer 1 the other interfaces with frame buffer 2 before they both then switch. This is done to avoid glitches if both are writing/reading from the same frame buffer. It communicates with both the rasterizer and the color mapper using the

native BRAM interface. The BRAM is also synchronous and takes up to 2 clock cycles to read data (since its output register is disabled).

- **Frame Buffer 2 Block Memory Generator (Part 2):** The Block Memory Generator generates a block of BRAM memory inside the GPU which stores Frame Buffer 1. It stores 9600 32 bit words with each group of four bits storing the color index for a pixel on the 320x240 screen (such that 8 pixels are represented per word). It is set to True Dual Output such that it has two IO ports, port A (connected to the rasterizer), allowing the rasterizer to write to the buffer, and port B (connected to the color mapper), allowing the color mapper to read the color index for the pixel specified by the VGA module. The rasterizer and color mapper take turns writing/reading from the frame buffer, such that when one is interfacing with frame buffer 1 the other interfaces with frame buffer 2 before they both then switch. This is done to avoid glitches if both are writing/reading from the same frame buffer. It communicates with both the rasterizer and the color mapper using the native BRAM interface. The BRAM is also synchronous and takes up to 2 clock cycles to read data (since its output register is disabled).
- **Z Buffer Block Memory Generator:** The Block Memory Generator generates a block of BRAM memory inside the GPU which stores the Z Buffer. It stores 9600 64 bit words. It is set to Single Port such that the rasterizer module uses Port A to both read and write to it using the native BRAM interface. The BRAM is also synchronous and takes up to 2 clock cycles to read data (since its output register is disabled).
- **RealDigital VGA to HDMI Converter:** The RealDigital VGA to HDMI Converter converts the VGA video output with a customizable RGB color format of the SystemVerilog modules into HDMI, which can then be outputted to a monitor. It takes as input the current RGB value to be outputted in a 12-bit format (such that each color is represented by 4 bits) from the color mapper, the horizontal and vertical sync and display VGA signals from the VGA controller, and the 25MHz and 125MHz clock signals from Clock Wizard 0 and outputs the corresponding HDMI signal which can be connected to a monitor. The converter also supports audio, although this feature was not used in this lab.

C Functions

Main calls all of the major functions which set up USB communication and camera and world initialization and contains the infinite loop which detects keyboard input and writes to the vertex and index buffers.

Main

```
int main() {
      init_platform();
      BYTE rcode;
      BOOT_MOUSE_REPORT buf;       //USB mouse report
      BOOT_KBD_REPORT kbdbuf;
```

```
      BYTE runningdebugflag = 0;//flag to dump out a bunch of information
when we first get to USB_STATE_RUNNING
      BYTE errorflag = 0; //flag once we get an error device so we don't
keep dumping out state info
      BYTE device;

      xil_printf("initializing MAX3421E...  \n");
      MAX3421E_init();
      xil_printf("initializing USB...\n");
      USB_init();

      // 3d set up
      init_default_cam(&camera);
      cam_to_clip_mat(cam_to_clip); // defines the perspective matrix

      int will_remove = 0;
      int will_place = 0;

      set_up_world();
      while (1) {



            xil_printf(".");
            MAX3421E_Task();
            USB_Task();

            if (GetUsbTaskState() == USB_STATE_RUNNING) {
                  if (!runningdebugflag) {
                        runningdebugflag = 1;
                        device = GetDriverandReport();
                  } else if (device == 1) {
                        //run keyboard debug polling
                        rcode = kbdPoll(&kbdbuf);
                        if (rcode == hrNAK) {
                              continue; //NAK means no new data
                        } else if (rcode) {
                              xil_printf("Rcode: ");
                              xil_printf("%x \n", rcode);
                              continue;
                        }
                        xil_printf("keycodes: ");
                        for (int i = 0; i < 6; i++) {
                              xil_printf("%x ", kbdbuf.keycode[i]);

                              if(kbdbuf.keycode[i] == 26){
                                    // w
```

```
                        move_cam(&camera,STEP,0,0);
                    }else if(kbdbuf.keycode[i] == 4){
                        // a
                        move_cam(&camera,0,-STEP,0);
                    }else if(kbdbuf.keycode[i] == 22){
                        // s
                        move_cam(&camera,-STEP,0,0);
                    }else if(kbdbuf.keycode[i] == 7){
                        // d
                        move_cam(&camera,0,STEP,0);
                    }else if(kbdbuf.keycode[i] == 44){
                        // _
                        move_cam(&camera,0,0,-STEP);
                    }else if(kbdbuf.keycode[i] == 6){
                        // c
                        move_cam(&camera,0,0,STEP);
                    }else if(kbdbuf.keycode[i] == 12){
                        // i
                        rotate_cam(&camera,0,ANGLE_STEP);
                    }else if(kbdbuf.keycode[i] == 13){
                        // j
                        rotate_cam(&camera,ANGLE_STEP,0);
                    }else if(kbdbuf.keycode[i] == 14){
                        // k
                        rotate_cam(&camera,0,-ANGLE_STEP);
                    }else if(kbdbuf.keycode[i] == 15){
                        // l
                        rotate_cam(&camera,-ANGLE_STEP,0);
                    }else if(kbdbuf.keycode[i] == 0x2f){
                        will_remove = 1;
                    }else if(kbdbuf.keycode[i] == 0x30){
                        will_place = 1;
                    }

                }
                //Outputs the first 4 keycodes using the USB GPIO
channel 1
                //printHex (kbdbuf.keycode[0] +
(kbdbuf.keycode[1]<<8) + (kbdbuf.keycode[2]<<16) + +
(kbdbuf.keycode[3]<<24), 1);
                //Modify to output the last 2 keycodes on channel
2.

                    xil_printf("\n");
            }

            else if (device == 2) {
```

```
                        rcode = mousePoll(&buf);
                        if (rcode == hrNAK) {
                              //NAK means no new data
                              continue;
                        } else if (rcode) {
                              xil_printf("Rcode: ");
                              xil_printf("%x \n", rcode);
                              continue;
                        }
                        xil_printf("X displacement: ");
                        xil_printf("%d ", (signed char) buf.Xdispl);
                        xil_printf("Y displacement: ");
                        xil_printf("%d ", (signed char) buf.Ydispl);
                        xil_printf("Buttons: ");
                        xil_printf("%x\n", buf.button);
                  }
            } else if (GetUsbTaskState() == USB_STATE_ERROR) {
                  if (!errorflag) {
                        errorflag = 1;
                        xil_printf("USB Error State\n");
                        //print out string descriptor here
                  }
            } else //not in USB running state
            {

                  xil_printf("USB task state: ");
                  xil_printf("%x\n", GetUsbTaskState());
                  if (runningdebugflag) {     //previously running, reset
USB hardware just to clear out any funky state, HS/FS etc
                        runningdebugflag = 0;
                        MAX3421E_init();
                        USB_init();
                  }
                  errorflag = 0;
            }

            world_to_cam_mat(world_to_cam, &camera);
            write_vertex_buffer(bram_ptr, world_to_cam, cam_to_clip);
            write_index_z_buffer(bram_ptr, bram_ptr_8, world_to_cam,
cam_to_clip);
            if(will_place){
                  place(&camera);
                  will_place = 0;
            }
            if(will_remove){
                  rem(&camera);
                  will_remove = 0;
```

```
            }
        }
    cleanup_platform();
      return 0;
}
```

3d.c

Performs the matrix multiplication <M><in> = <out> where M is a 4 by 4 matrix of floats and and in and out are both 4 dimensional floats vectors.

```
void linear_transform(float M[4][4], float in[4], float out[4]) {
    out[0] = in[0] * M[0][0] + in[1] * M[1][0] + in[2] * M[2][0] + in[3] *
M[3][0];
    out[1] = in[0] * M[0][1] + in[1] * M[1][1] + in[2] * M[2][1] + in[3] *
M[3][1];
    out[2] = in[0] * M[0][2] + in[1] * M[1][2] + in[2] * M[2][2] + in[3] *
M[3][2];
    out[3] = in[0] * M[0][3] + in[1] * M[1][3] + in[2] * M[2][3] + in[3] *
M[3][3];
}
```

Given a camera struct and its yaw and pitch angles, recalculates the forward vector.

```
void set_cam_fwd(Camera* camera){
    camera->fwd[0] = cosf(camera->yaw) * cosf(camera->pitch);
    camera->fwd[1] = sinf(camera->yaw) * cosf(camera->pitch);
    camera->fwd[2] = sinf(camera->pitch);
}
```

Given a camera struct and a change in the yaw (dyaw) and change in the pitch (dpitch) rotates the camera, alongside updating the camera basis vectors.

```
void rotate_cam(Camera* camera, float dyaw, float dpitch){
    camera->pitch+=dpitch;
    camera->yaw+=dyaw;
    set_cam_fwd(camera);
    set_cam_basis(camera);
}
```

Given a camera struct and a displacement, written in terms of two of the camera basis vectors and the world up vector (dfwd, dright, dup) moves the camera position.

```
void move_cam(Camera* camera, float dfwd, float dright, float dup){
    float world_up[3] = WORLD_UP;
    // a planar_fwd_norm is required as fwd projected onto the plane is
not necessarily going to be normalized
    float planar_fwd_norm = sqrt(camera->fwd[0] * camera->fwd[0] +
camera->fwd[1] * camera->fwd[1]);
```

```
    // scales each of the displacements with the appropriate scalars
(dfwd, dright, dup)
    // renormalized the x displacement so it doesn't depend on the pitch
    camera->position[0] += dfwd * camera->fwd[0] / planar_fwd_norm;
    camera->position[1] += dfwd * camera->fwd[1] / planar_fwd_norm;
    camera->position[0] += dright * camera->right[0];
    camera->position[1] += dright * camera->right[1];
    camera->position[2] += dup * world_up[2];
}
```

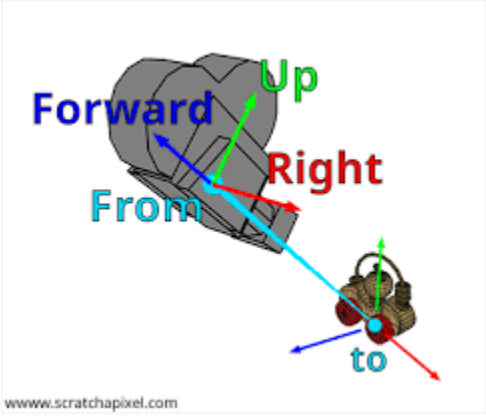

Figure 6. Forward, Right, and Up Directions of a Virtual Camera

Given a camera struct, calculates the camera right and camera up basis vectors in relation to the camera fwd basis vector.

```
void set_cam_basis(Camera* camera){
    float world_up[3] = WORLD_UP;
    cross(camera->fwd, world_up, camera->right);
    normalize_vec_3(camera->right, camera->right);
    cross(camera->right, camera->fwd, camera->up);
    normalize_vec_3(camera->up, camera->up);
}
```

Initializes a default camera with pitch of 0 radians, a yaw of pi radians, a position of (2,0,0) = (x,y,z).

```
void init_default_cam(Camera* camera){
    camera->pitch = 0;
    camera->yaw = M_PI;
    camera->position[0] = 2;
    camera->position[1] = 0;
    camera->position[2] = 0;
    set_cam_fwd(camera);
    set_cam_basis(camera);
}
```

Given a world to camera affine transformation matrix, a camera to clip space perspective matrix this function converts a point (vector 3) in world space to a (vector 3) in clip space.

```
void world_to_clip(float world_to_cam[4][4], float cam_to_clip[4][4],
float world_point[3], float clip_point[3]){
    // vector3s are promoted to vector4s which are required for affine and
perspective transforms
    float world_vec[4] = {world_point[0], world_point[1], world_point[2],
1};
    float cam_vec[4];
    float clip_vec[4];
    // applying the matrix portion of the affine and perspective transform
    linear_transform(world_to_cam, world_vec, cam_vec);
    linear_transform(cam_to_clip, cam_vec, clip_vec);
    // concluding the perspective transform with the necessary perspective
divide
    clip_point[0] = clip_vec[0] / clip_vec[3];
    clip_point[1] = clip_vec[1] / clip_vec[3];
    clip_point[2] = clip_vec[2] / clip_vec[3];
}
```

Given a world to camera affine transformation matrix, a camera to clip space perspective matrix this function converts a point (vector 3) in world space to a (uint16 vector 3) in raster space.

```
void world_to_rast(float world_to_cam[4][4], float cam_to_clip[4][4],
float world_point[3], uint16_t rast_point[3]){
      float clip_point[3];
      world_to_clip(world_to_cam, cam_to_clip, world_point, clip_point);
      clip_to_rast(clip_point, rast_point);
}

// perspective transformation matrix formulas
// [1/(tan(fov/2)*aspect), 0, 0, 0]
// [0,1/tan(fov/2),0,0]
// [0,0,-(far+near)/(far-near),-2*far*near/(far-near)]
// [0,0,-1,0]
```

Generates a 4x4 perspective transformation matrix which is used for converting camera space vector4s into raster space vector4s. The function depends on the defined arguments ASPECT for the aspect ratio, FOV for the field of view, NEAR for the distance to the near clipping plane and FAR for the distance to the far clipping plane.

```
void cam_to_clip_mat(float M[4][4]){
    M[0][0] = 1/(tanf(FOV/2) * ASPECT); M[0][1] = 0;             M[0][2] =
0;                          M[0][3] = 0;
    M[1][0] = 0;                          M[1][1] = 1/tanf(FOV/2); M[1][2] =
0;                          M[1][3] = 0;
    M[2][0] = 0;                          M[2][1] = 0;             M[2][2] =
-(FAR+NEAR)/(FAR-NEAR);     M[2][3] = -1;
```

```
    M[3][0] = 0;                          M[3][1] = 0;              M[3][2] =
-2 * FAR * NEAR/(FAR-NEAR); M[3][3] = 0;
}

// [r.x,u.x,-f.x,0]
// [r.y,u.y,-f.y,0]
// [r.z,u.z,-f.z,0]
// [-p5.Vector.dot(r,p),-p5.Vector.dot(u,p),p5.Vector.dot(f,p),1]
```

Generates a 4x4 affine transformation matrix which is used for converting world space vector4s into camera space vector4s. The function depends on the camera struct, of which the camera basis and positions are used to obtain the affine transformation matrix as defined in the above formulas.

```
void world_to_cam_mat(float M[4][4], Camera * camera){
    world_to_cam_mat_cam_basis(M, camera->right, camera->up, camera->fwd,
camera->position);
}

 // /[PX*(xprime+1)/(2),PY*(1-yprime)/(2),FAR*NEAR/2*zprime+(NEAR+FAR)/2]
```

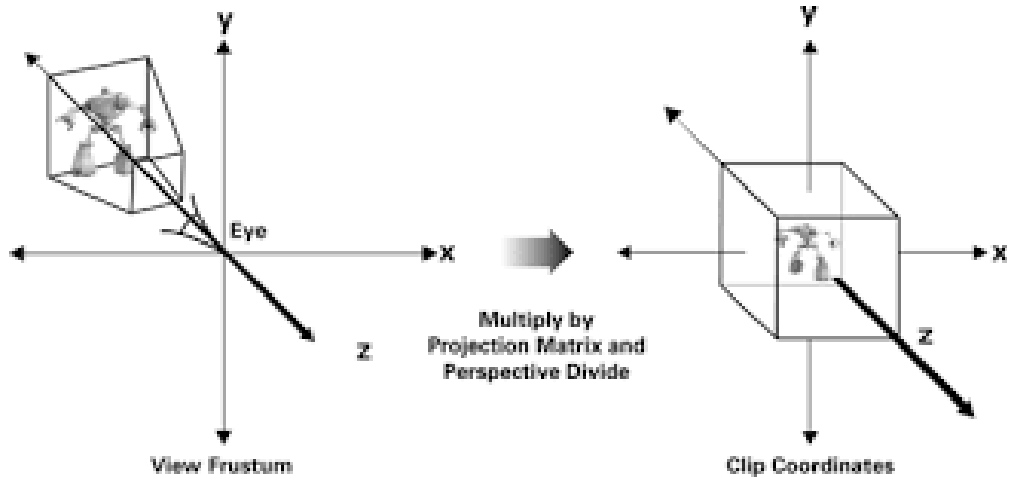

Figure 7. Transforming from Camera to Clip Coordinates (https://developer.download.nvidia.com/CgTutorial/cg_tutorial_chapter04.html)

Converts a vector3 in clip space to a uint16 vector 3 in raster space. This is done by matching the [-1,1]x[-1,1]x[-1,1] volume that is the clip space to a new discrete space which is a [0,2^16-1]x[0,2^16-1]x[0,255] dimensional vector which represents the x,y and z positions. The x and y positions are given more accuracy as they must describe the pixel coordinates on the screen. The z position is only used to properly order triangles in the scene.

```
void clip_to_rast(float clip_point[3], uint16_t rast_point[3]){
    rast_point[0] = (uint16_t)((clip_point[0]+1)/2*HALF_SCREEN_WIDTH);
    rast_point[1] = (uint16_t)((clip_point[1]+1)/2*HALF_SCREEN_HEIGHT);
    rast_point[2] = (uint16_t)((clip_point[2]+1)/2*256);
}
```

Creates the basis for the world to camera transformation matrix.

```
void world_to_cam_mat_cam_basis(float M[4][4], float r[3], float u[3],
float f[3], float p[3]){
    M[0][0] = r[0]; M[0][1] = u[0]; M[0][2] = -f[0]; M[0][3] = 0;
    M[1][0] = r[1]; M[1][1] = u[1]; M[1][2] = -f[1]; M[1][3] = 0;
    M[2][0] = r[2]; M[2][1] = u[2]; M[2][2] = -f[2]; M[2][3] = 0;
    M[3][0] = -dot_vec_3(r,p); M[3][1] = -dot_vec_3(u,p); M[3][2] =
dot_vec_3(f,p); M[3][3] = 1;
}
```

Calculates the dot product of its two inputs.

```
float dot_vec_3(float a[3], float b[3]) {
    return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}
```

Calculates the cross product of its two inputs.

```
void cross(float a[3], float b[3], float out[3]) {
    out[0] = a[1] * b[2] - a[2] * b[1];
    out[1] = a[2] * b[0] - a[0] * b[2];
    out[2] = a[0] * b[1] - a[1] * b[0];
}
```

Normalizes the inputted 4-vector.

```
void normalize_vec_4(float v[4], float out[4]) {
    float norm = sqrtf(v[0] * v[0] + v[1] * v[1] + v[2] * v[2] + v[3] *
v[3]);
    out[0] = v[0] / norm;
    out[1] = v[1] / norm;
    out[2] = v[2] / norm;
    out[3] = v[3] / norm;
}
```

Normalizes the inputted 3-vector.

```
void normalize_vec_3(float v[3], float out[3]) {
    float norm = sqrtf(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
    out[0] = v[0] / norm;
    out[1] = v[1] / norm;
    out[2] = v[2] / norm;
}
```

Prints the inputted matrix.

```
void print_mat(float M[4][4]) {
    printf("<<<mat 4x4>>>\n");
    printf("[%f, %f, %f, %f]\n", M[0][0], M[0][1], M[0][2], M[0][3]);
    printf("[%f, %f, %f, %f]\n", M[1][0], M[1][1], M[1][2], M[1][3]);
    printf("[%f, %f, %f, %f]\n", M[2][0], M[2][1], M[2][2], M[2][3]);
    printf("[%f, %f, %f, %f]\n", M[3][0], M[3][1], M[3][2], M[3][3]);
}
```

Prints the current state of the camera struct.

```
void print_cam(Camera *camera){
    printf("<<<<<<Camera>>>>>>\n");
    printf("POSITION\n");
    print_vec_3(camera->position);
    printf("PITCH\n");
    printf("%f\n",camera->pitch);
    printf("YAW\n");
    printf("%f\n",camera->yaw);
    printf("FORWARD\n");
    print_vec_3(camera->fwd);
    printf("RIGHT\n");
    print_vec_3(camera->right);
    printf("UP\n");
    print_vec_3(camera->up);
}
```

Prints the inputted 4-vector.

```
void print_vec_4(float v[4]) {
    printf("<<<vec 4>>>\n");
    printf("[%f, %f, %f, %f]\n", v[0], v[1], v[2], v[3]);
}
```

Prints the inputted 3-vector.

```
void print_vec_3(float v[3]) {
    printf("<<<vec 3>>>\n");
    printf("[%f, %f, %f]\n", v[0], v[1], v[2]);
}
```

Prints the inputted raster space 3-vector.

```
void print_rast_vec_3(uint16_t v[3]){
    printf("[%u, %u, %u]\n", v[0], v[1], v[2]);
}
```

Determines if the inputted point in clip space is out of bounds.

```
int in_clip_volume(float v[3]){
    if(v[0] > 1 || v[0] < -1){
        return 0;
    }
    if(v[1] > 1 || v[1] < -1){
        return 0;
    }
    if(v[2] > 1 || v[2] < -1){
        return 0;
    }
    return 1;
}
```

write_buffers.c

Sets up the world_def matrix by setting the ID of each block.

```
void set_up_world(){
	for(int z = 0; z < LENGTH; z++){
		for(int y = 0; y < LENGTH; y++){
			for(int x = 0; x < LENGTH; x++){
				world_def[x][y][z] = 0;
			}
		}
	}
	world_def[0][0][0] = 1;
	world_def[1][0][0] = 5;
	world_def[0][1][0] = 5;
	world_def[1][1][0] = 1;
}
```

Given a triangle on a cube returns its vertex indices and world coordinates.

```
void get_triangle_verticies(int cube_x, int cube_y, int cube_z, int
triangle, int triangle_indicies[3], float triangle_points[3][3]){
	int cube_point_index[3];
	for(int i = 0; i < 3; i++){
		cube_point_index[i] = CUBE_POINT_INDICIES[triangle*3+i];
		int x = cube_x+CUBE_POINTS[cube_point_index[i]][0];
		int y = cube_y+CUBE_POINTS[cube_point_index[i]][1];
		int z = cube_z+CUBE_POINTS[cube_point_index[i]][2];
		triangle_indicies[i] = triple_to_point_index(x,y,z);
		triangle_points[i][0] = x;
		triangle_points[i][1] = y;
		triangle_points[i][2] = z;
	}
}
```

Writes the raster coordinates of each vertex to the vertex buffer.

```
void write_vertex_buffer(volatile uint32_t *bram_ptr, float
world_to_cam[4][4], float cam_to_clip[4][4]) {
	int num_verticies = (LENGTH+1)*(LENGTH+1)*(LENGTH+1);
	for(int i = 0; i < num_verticies; i++) {
	  float world_point[3];
	  point_index_to_triple(i, world_point);

	  float clip_point[3];
	  world_to_clip(world_to_cam, cam_to_clip, world_point, clip_point);

	  uint16_t rast_point[3];
	     clip_to_rast(clip_point,rast_point);
```

```
            buffer_ctrl->VERTEX[2*i] = rast_point[0];
            buffer_ctrl->VERTEX[2*i+1] = rast_point[1];
    }
      printf("FINISHED WRITING VERTEX BUFFER\n");
}
```

Creates a 32-bit word for a triangle to write to the Index Buffer storing its vertex coordinates and color index.

```
uint32_t triangle_to_index_word(int index_0, int index_1, int index_2,
uint8_t pallet) {
      return ((pallet&0xF) << 27) + ((index_2 & 0x1FF) << 18) + ((index_1
& 0x1FF) << 9) + (index_0 & 0x1FF);
}
```

Averages the inputted three 3-vectors.

```
void average_vectors(float a[3], float b[3], float c[3], float d[3], float
average[3]){
      average[0] = (a[0] + b[0] + c[0] + d[0])/4;
      average[1] = (a[1] + b[1] + c[1] + d[1])/4;
      average[2] = (a[2] + b[2] + c[2] + d[2])/4;
}
```

Swaps the two inputted values in an array.

```
// Swap two key-value pairs
void swap(KeyValuePair* a, KeyValuePair* b) {
    KeyValuePair temp = *a;
    *a = *b;
    *b = temp;
}
```

Creates a partition as required by the quicksort algorithm (taken from online sources).

```
// Partition function for quicksort
int partition(KeyValuePair* arr, int low, int high) {
    // Choose the rightmost element as pivot
    float p = arr[(low+high)/2].value;
    int i = low;
    int j = high;

    while (i < j) {

        // Find the first element greater than
        // the pivot (from starting)
        while (arr[i].value <= p && i <= high - 1) {
            i++;
        }

        // Find the first element smaller than
```

```
        // the pivot (from last)
        while (arr[j].value > p && j >= low + 1) {
            j--;
        }
        if (i < j) {
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[low], &arr[j]);
    return j;
}
```

Sorts the inputted array using the quicksort algorithm (taken from online sources).

```
// Quicksort recursive function
void quickSort(KeyValuePair* arr, int low, int high) {
    if (low < high) {
        // Find the partitioning index
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Loops through every triangle and writes its vertex and color indices ot the index buffer.

```
void write_index_z_buffer(volatile uint32_t *bram_ptr, volatile uint8_t
*bram_ptr_8, float world_to_cam[4][4], float cam_to_clip[4][4]) {
      int triangle_index = 0;
      KeyValuePair dist_array[LENGTH*LENGTH*LENGTH*6];

      for(int z = 0; z < LENGTH; z++){
            for(int y = 0; y < LENGTH; y++){
                  for(int x = 0; x < LENGTH; x++){
                        for(int face = 0; face < 6; face++){
                              int triangle_0_indicies[3];
                              int triangle_1_indicies[3];

                              float triangle_0_points[3][3];
                              float triangle_1_points[3][3];

                              float average_point[3];
                              float average_point_clip[3];
                              uint16_t average_point_rast[4];
```

```
                              get_triangle_verticies(x,y,z,face*2,triangle_
                              0_indicies,triangle_0_points);

                              get_triangle_verticies(x,y,z,face*2+1,triangl
                              e_1_indicies,triangle_1_points);


                              average_vectors(triangle_0_points[0],triangle
                              _0_points[1],triangle_0_points[2],triangle_1_
                              points[1], average_point);

                              world_to_clip(world_to_cam, cam_to_clip,
average_point, average_point_clip);
                              clip_to_rast(average_point_clip,
average_point_rast);

                              uint32_t a;
                              uint32_t b;
                              if(-1 <= average_point_clip[2] &&
average_point_clip[2] <= 1){
                                    char block_id = world_def[x][y][z];

                                    a =
triangle_to_index_word(triangle_0_indicies[0],triangle_0_indicies[1],trian
gle_0_indicies[2],block_pallet[block_id][face]);
                                    b =
triangle_to_index_word(triangle_1_indicies[0],triangle_1_indicies[1],trian
gle_1_indicies[2],block_pallet[block_id][face]);
                              }else{
                                    a =
triangle_to_index_word(triangle_0_indicies[0],triangle_0_indicies[1],trian
gle_0_indicies[2],0);
                                    b =
triangle_to_index_word(triangle_1_indicies[0],triangle_1_indicies[1],trian
gle_1_indicies[2],0);
                              }

                              dist_array[triangle_index/2].key = a;
                              dist_array[triangle_index/2].key2 = b;
                              dist_array[triangle_index/2].value =
average_point_clip[2];
                              triangle_index+=2;
                        }
                  }
            }
      }
```

```
}
```

“Places a block” by finding the block nearest to the camera and updating its ID to stone.

```
void place(Camera* camera){
	float min_d = 1000000;
	int min_x = 0;
	int min_y = 0;
	int min_z = 0;
	for(int z = 0; z < LENGTH; z++){
		for(int y = 0; y < LENGTH; y++){
			for(int x = 0; x < LENGTH; x++){
				float d =
(z-camera->position[2])*(z-camera->position[2])+

(y-camera->position[1])*(y-camera->position[1])+

(x-camera->position[0])*(x-camera->position[0]);
				if(d < min_d){
					min_d = d;
					min_x = x;
					min_y = y;
					min_z = z;
				}
			}
		}
	}
	world_def[min_x][min_y][min_z] = 3;
}
```

“Removes a block” by finding the block nearest to the camera and updating its ID to air.

```
void rem(Camera* camera){
	float min_d = 1000000;
	int min_x = 0;
	int min_y = 0;
	int min_z = 0;
	for(int z = 0; z < LENGTH; z++){
		for(int y = 0; y < LENGTH; y++){
			for(int x = 0; x < LENGTH; x++){
				float d =
(z-camera->position[2])*(z-camera->position[2])+

(y-camera->position[1])*(y-camera->position[1])+

(x-camera->position[0])*(x-camera->position[0]);
				if(d < min_d){
					min_d = d;
					min_x = x;
```

```
                        min_y = y;
                        min_z = z;
                    }
                }
            }
        }
        world_def[min_x][min_y][min_z] = 0;
}
```

Converts a vertex index to its world-space xyz coordinates.

```
void point_index_to_triple(int index, float world_point[3]) {
        int n = LENGTH + 1;
        world_point[0] = index%(n);
        world_point[1] = (index%(n*n))/(n);
        world_point[2] = index/(n*n);
}
```

Converts a vertex's world-space xyz coordinates to its index.

```
int triple_to_point_index(int x, int y, int z) {
        int n = LENGTH + 1;
        return (z)*(n*n) + y*n + x;
}
```

Simulation

load_tri,
load_cache,
pineda,
increment_span,
store_cache,
done,
await_buff_swap,
clear_buff,
Clear_buff_init

Rasterizer - Introduction

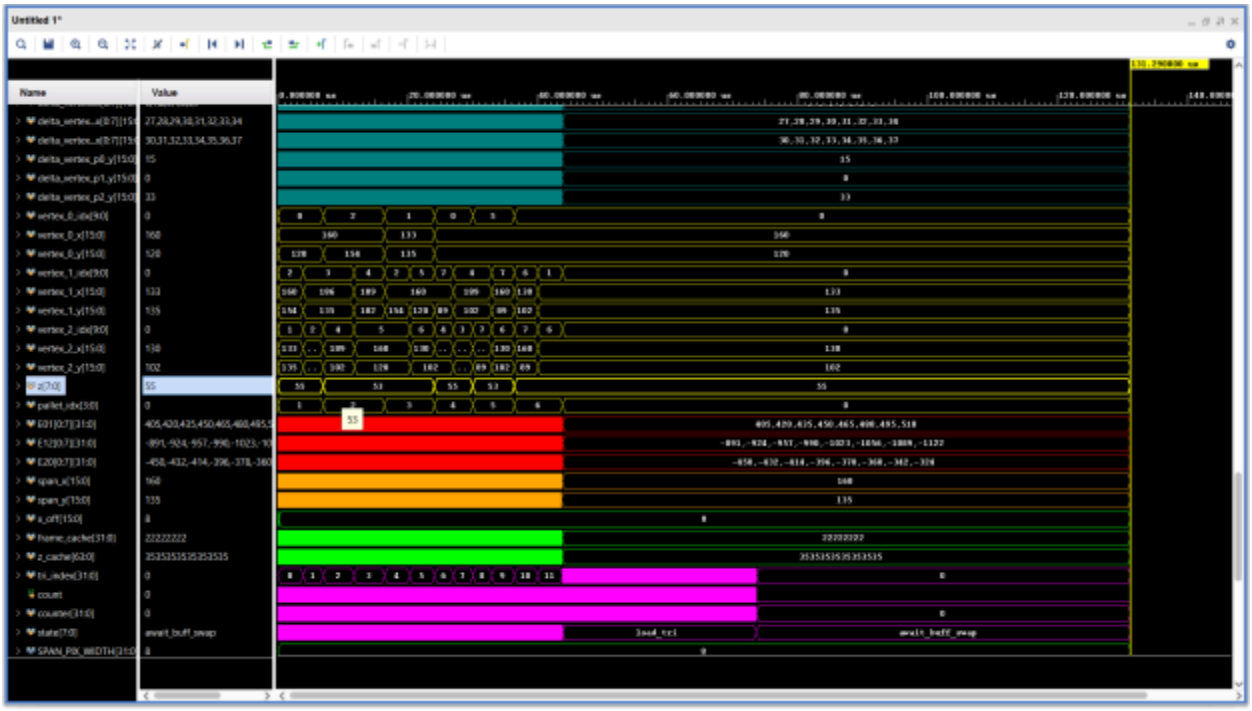

Figure 8. Rasterizer Simulation

This serves as an introduction to the simulation of the rasterizer. Given the complexity of the operations the rasterizer completes, the simulations will be divided into smaller segments which describe the operation of each of the different states within the rasterizer. This simulation above shows one cycle through all operations which are required to draw the following single cube into the frame buffer. This frame buffer was then read using SystemVerilog code and converted to a bitmap as shown below.

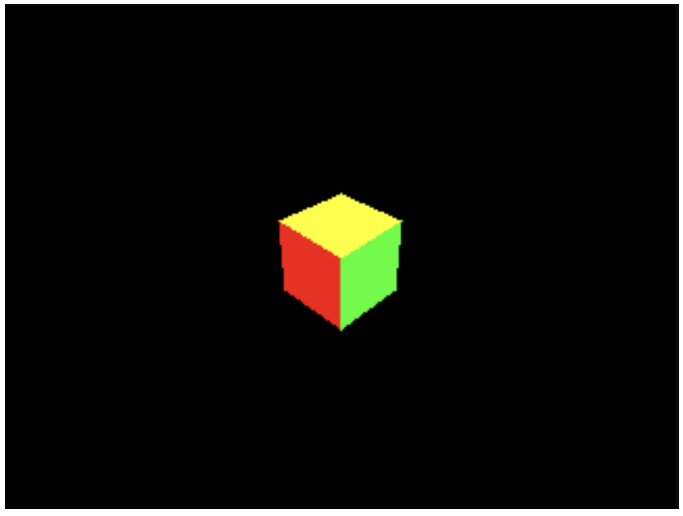

Figure 9. BMP Image of a Rasterized Cube from Simulation (Z-Buffering is Working Correctly)

Rasterizer - load_tri

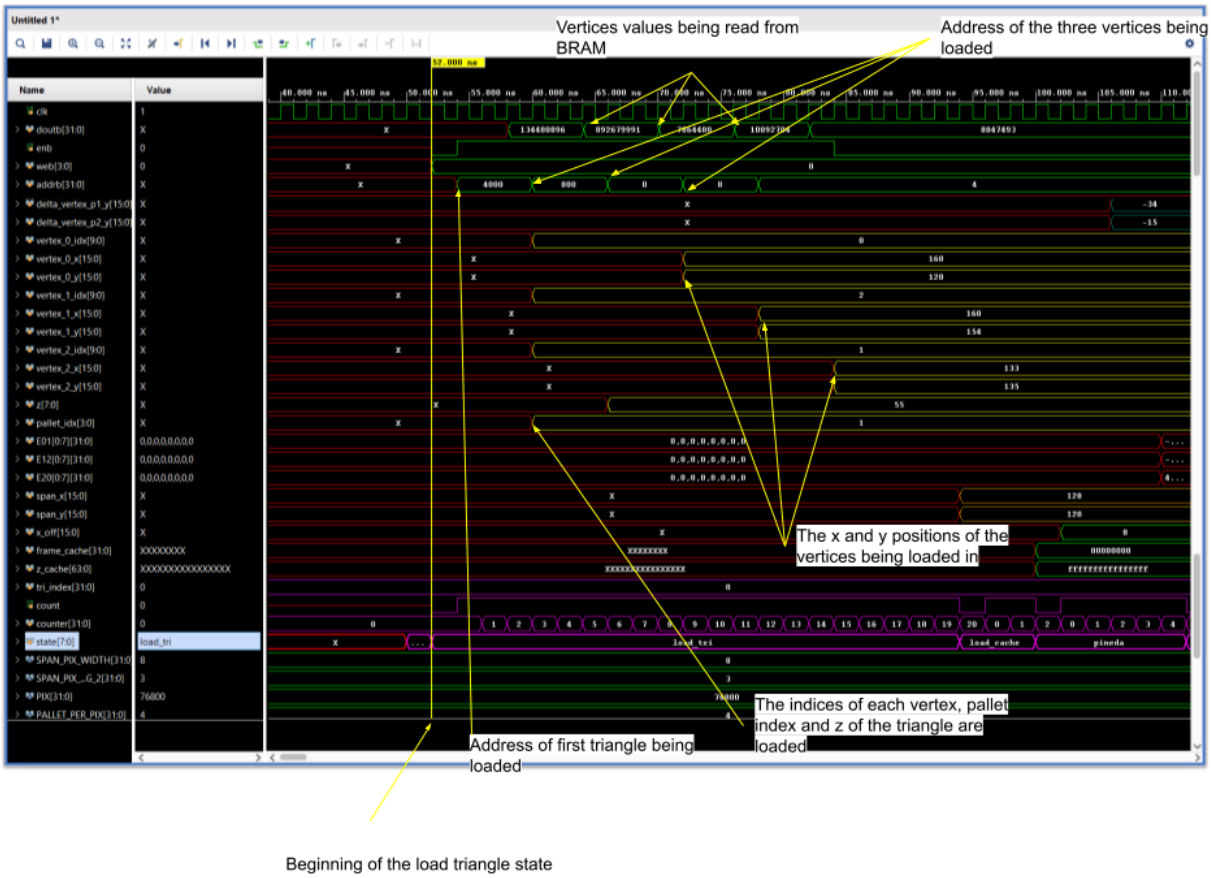

Figure 8. Rasterizer Simulation - load_tri state.

During the load_tri state, the rasterizer loads all of the data written by the MicroBlaze for a specific triangle. It first reads from the first address (triangle 0) in the Index Buffer at address line 4000 (in hardware the BRAM is treated as having a word width of 8 bits compared to 32 in software so all of the addresses are what they would be in software multiplied by 4). From this it gets the 3 vertex indicides (0, 2, and 1), and color index (1) for the triangle. It then reads the corresponding depth value (55) for the triangle from the depth buffer at address 800. Finally, it reads the vertex coordinates from the vertex buffer for the three vertices that form the given triangle based on their indices ((160, 120) for v_0, (160,154) for v_1, and (133,135) for v_2) at memory address 0, 8, and 4.

Rasterizer - store_cache, increment span, and load_cache

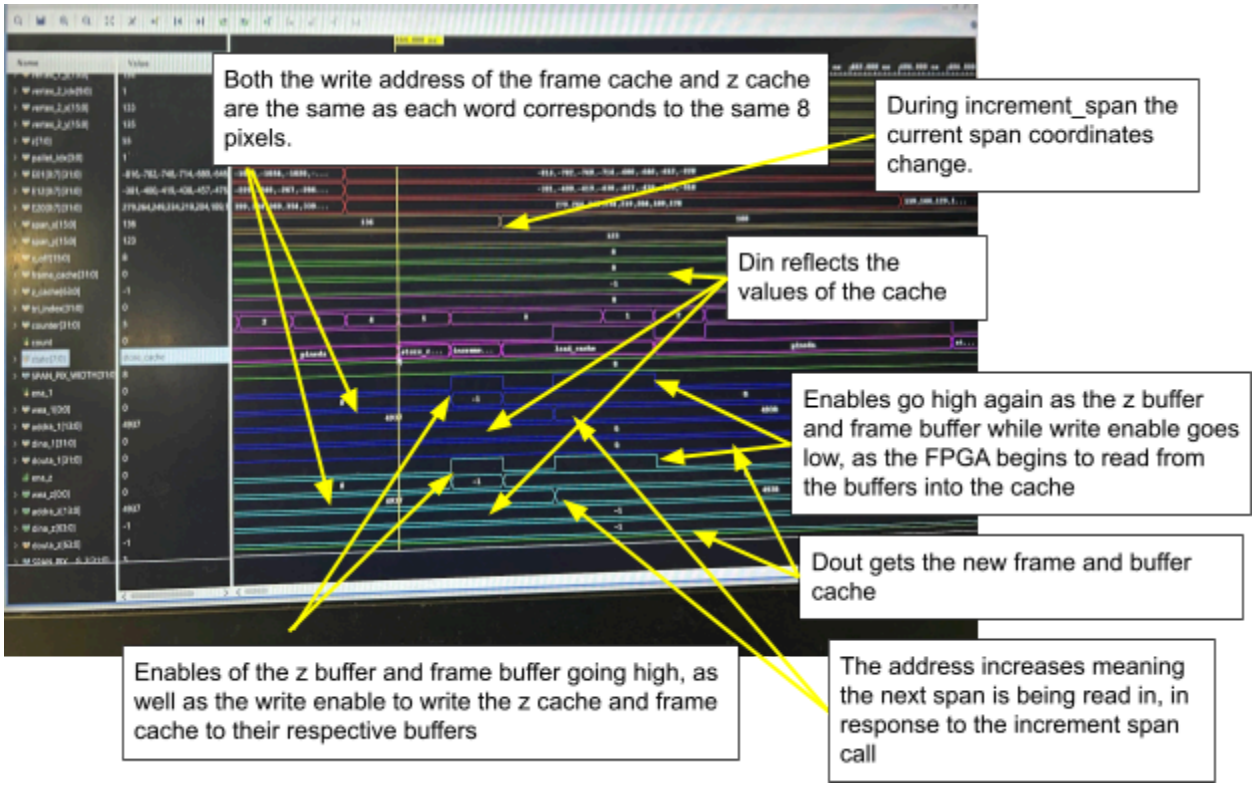

Figure 9. Rasterizer Simulation - store_cache, increment span, and load_cache states.

During the store_cache state, the rasterizer writes frame and z cache data for the current 1x8 span (at coordinates (136, 123)) in the bounding box of the current triangle to the frame and z buffers (this data is calculated in the pineda state). A given span has the same address in both the z and frame buffers (in this case 4937) and both ena_1 and wea_1 (in this case the rasterizer is currently writing to the first frame buffer) and ena_z and wea_z are set high to perform the writes. As the writes are happening the state briefly goes to increment_span, where the current span is incremented to the next span in the current triangle's bounding box and its coordinates are updated accordingly (in this case to (144, 123)). The state then becomes load_cache, where the z and frame buffer for the new span (this time at the next address 4398 based on the new span coordinates) are read as ena_1 and ena_z are driven high, with the outputs being loaded into douta_1 and douta_z and then frame_cache and z_cache respectively (in this case the loaded cache values are the same as the previous cache values so no noticeable change occurs). The first time that the current state becomes load_cache is after load_tri (where load_tri sets the span address), after which it is entered after increment_span.

Rasterizer - pineda

Figure 11. Rasterizer Simulation - pineda state.

Pineda's algorithm can be used to determine if a given point is inside or outside of a given triangle. During the pineda state, the algorithm is used concurrently on the 8 points in the current span for the current triangle, with 3 edge functions being generated (one for each triangle edge). If a point has the same sign when evaluated by all three edge functions, then it is inside the triangle, otherwise it is not. During the pineda state after the 8 points in the span are evaluated by the edge functions, the frame and buffer cache are updated with them if they are inside the triangle and are in front of all other points at the same pixel, otherwise the frame and buffer cache stay the same. The state then becomes store_cache to store these updated cache values back to the frame and z buffers.

Rasterizer - await_buff_state, init_clear_buff, clear_buff

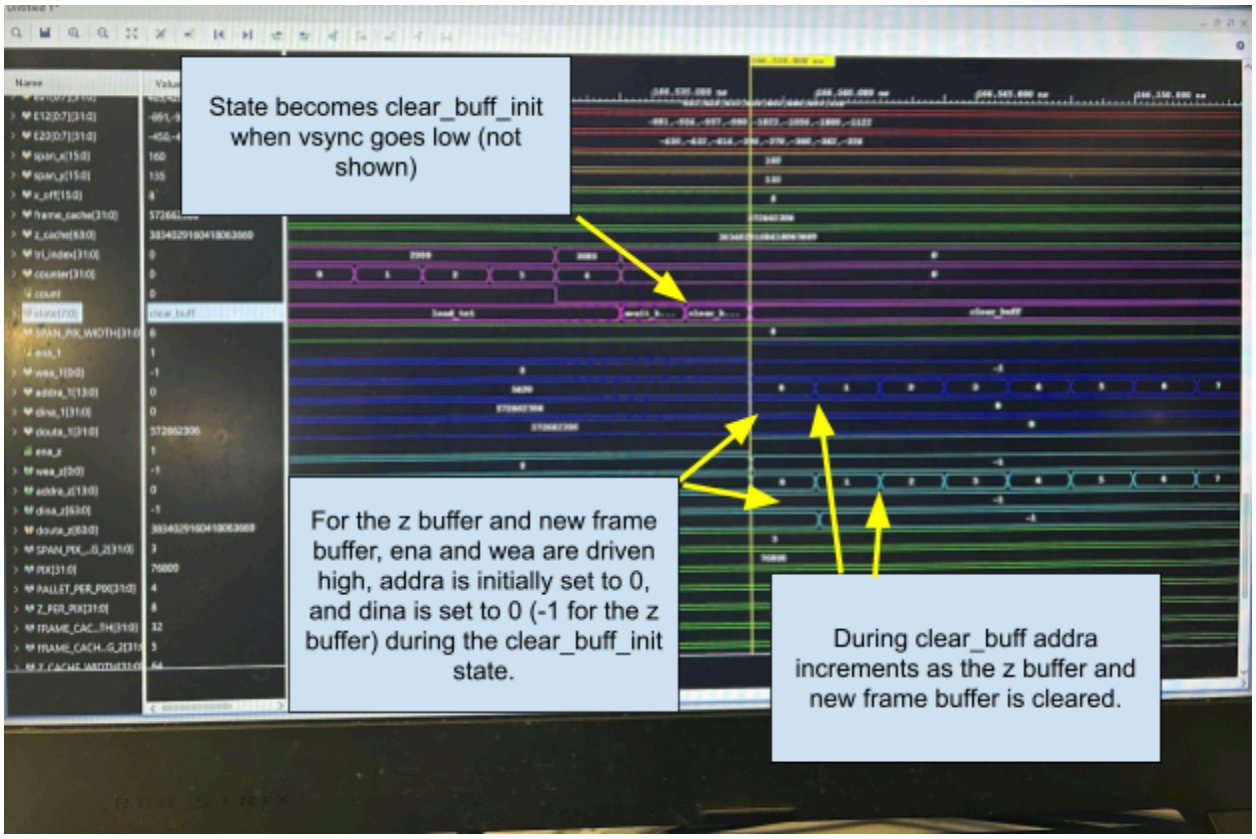

Figure 12. Rasterizer Simulation - await_buff_state, init_clear_state, clear_buff state.

After all of the spans for a triangle's bounding box have been iterated through, the next triangle is loaded in during a load_tri state. After all triangles have been integrated through, the state machine waits in an await_buff_swap state until the VGA module finishes the current frame (marked by vsync going low), after which it swaps frame buffers and goes to the init_clear_buff state. During the init_clear_buff state, the rasterizer prepares to clear the z buffer and the new frame buffer by setting ena_1 (in this case the first frame buffer is being cleared), wea_1, ena_z, and wea_z high, setting addra_1 and addra_z high, and setting dina_1 to 0 (to clear the frame buffer to 0) and dina_z to 2^64 - 1 to clear the z buffer to a high positive value (this value is unsigned but is interpreted as a 2s complement number by the simulator, which is why it shows -1). The state then becomes clear_buff, during which it progressively iterates through the z buffer and frame buffer, clearing each word so that they can be rewritten to.

Design Statistics

Table 1. Implementation Statistics

Criteria	2x2x2 World Minecraft Rendering
LUT	5684
DSP	54
Memory (BRAM)	72.5
Flip-Flop	4138
Latches	0
Frequency (MHz)	101.92
Static Power (W)	0.076
Dynamic Power (W)	0.444
Total Power (W)	0.52

Conclusion

Overall, our final project met most of our goals. It was able to achieve full 3D rendering and display a 2x2x2 Minecraft world while providing the user with full control over the virtual camera and the ability to place/break blocks. This included being able to move the camera in all 6 directions (forward, backward, left, right, up, and down), and rotate it in four directions (up, down, left, and right) with a good framerate. The software component of the project was able to successfully store the Minecraft World and based on user keyboard input and the resulting changes to the virtual camera position and orientation, output triangle vertex and color data to the vertex and frame buffers. For the most part, the hardware was able to take that data and through the rasterization algorithm write it to a frame buffer, which was simultaneously through double frame buffering read and outputted to a monitor in HDMI format. However, we were not able to render a 6x6x6 world as originally planned due a bug in the hardware rasterizer's implementation of z-buffering and the painter's algorithm instead used for depth control taking too long timewise for worlds larger than 2x2x2.

The rasterizer, including z buffering, worked correctly in simulation but not on hardware, most likely to either how the testbench was set up or due to how non-synthesizable constructs were used in the rasterizer module. While we were able to overcome this bug by using the painter's algorithm in software to achieve visibility control, this reduced the maximum possible world size (although it also reduced the amount of BRAM used as a depth buffer was no longer needed).

Our initial plan also involved using the off-chip DDR3RAM on the FPGA board to store the frame buffers instead of BRAM, which would allow for 640x480 resolution. However, we failed to be able to successfully write and read from the DDR3RAM and so transitioned to

BRAM, reducing our resolution to 320x480 to account for the reduced amount of memory. It would be helpful if more examples of using the MIG for DDR3RAM specifically using the RTL interface were provided in the future (as the provided example wrote/read the entire DDR3RAM memory at once while ours would do it incrementally, and we had trouble adapting the code), in addition to a provided testbench to test a DDR3RAM write/reader (we attempted to adapt the IP example project's testbench but do not know if we did so correctly).

Possible extensions of this project include fixing the z-buffering bug in the rasterizer to allow for bigger worlds to be rendered, allowing the user to change what block they place, and if DDR3RAM is used, adding texture mapping instead of the monocolor face approach currently used.

Overall, we enjoyed this project and learnt a lot from it including how 3D rendering and the rasterization algorithm works and gained experience working on a larger SoC/FPGA project in Vivado/Vitis, further developing our hardware design/testing and SystemVerilog/C skills.