FPGA 360
6.205 Final Report

Kailas Kahler
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
kailasbk@mit.edu

Ritik Patnaik
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
rik01@mit.edu

***Abstract*—We present a 3-dimensional graphics viewer implemented in FPGA hardware, which uses VGA to output to a monitor and a joystick for camera view control. The graphics module transforms vertex data, maps triangles to pixels via a rasterizer, and shades the 3-D object with a maximum throughput of approximately 95 million pixels per second, though the overall pipeline locked at a frame rate of 50 fps. The control module monitors analog signals from a two-axis joystick, converts them into discrete bits, and transforms the camera's orientation and position. We implement our design on a Nexys A7 FPGA, employing the two on-board XADCs for the joystick, VGA port, and multiple BRAMs. We evaluate the system's performance and quality by rendering custom 3-D models in the OBJ file format, and describe the successful expansion of our design to include low poly art graphics, dual-joystick camera control, and 3-D scene uploads via UART serial communication.**

***Index Terms*—Digital Systems, Field Programmable Gate Array, 3-D Computer Graphics, Joystick, Rasterization, Multiplexer, ADC, GPU**

I. Introduction

3-D Computer Graphics is a research field for digital systems that has given rise to powerful GPUs. An essential application of 3-D graphics is the rendering of custom 3-D models and scenes, such as low poly art. The FPGA 360 is a 3-D graphics viewer based on a rasterization rendering scheme. A user can upload a 3-D model or scene to be displayed over VGA, and control the camera view with joysticks in two modes: gimbal-lock and free.

The following project milestones were achieved for the FPGA 360:

(A) Commitment: Render a scene of cubes with different color faces and control the viewer with buttons in gimbal-lock mode.
(B) Goal: Render hardcoded low-poly art scene with directional lighting and control the viewer with a single joystick in gimbal-lock mode.
(C) Stretch Goals: Upload a 3-D OBJ model or scene from a PC using UART serial communication and control the viewer with two joysticks in either gimbal-lock or free modes.

Ritik designed the camera control module, built the Dual-Joystick controller, and implemented the BRAM storage logic for UART model uploads. Kailas designed and built the graphics pipeline and implemented the UART decoder logic and serial Python script for encoding and sending models. The team members designed, simulated, and deployed their modules on separate hardware before integrating the system together.

II. Control Module - Physical Setup

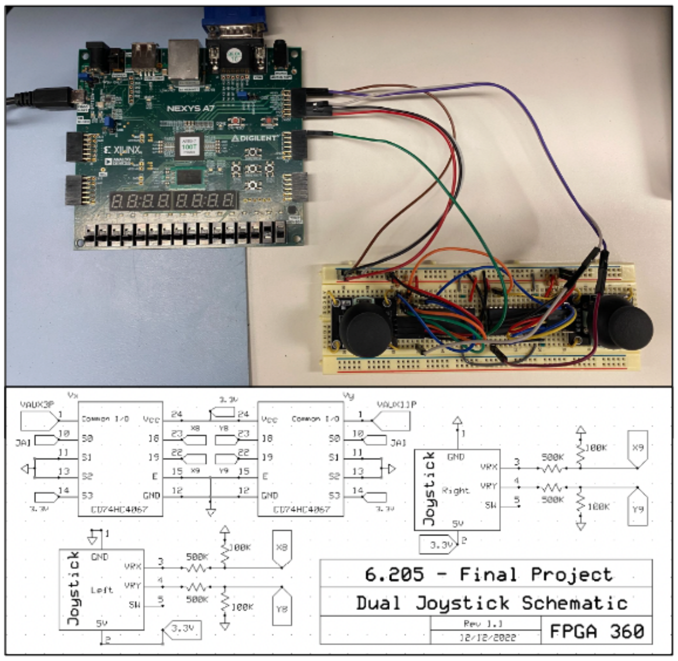


Fig. 1. Breadboard and Schematic for Dual-Joystick Controller

While the majority of the system hardware is integrated in the FPGA, including VGA chipset and Block RAM modules, the control module requires two peripheral joysticks connected to the development board. As shown in Figure 1 schematic, two five-pin Joystick Control Boards (KY-023) and two Analog Multiplexers (TI CD74HC4067) were mounted on a breadboard for easy layout. The power pins, GND and Vcc, respectively connect to the FPGA's power supply pins, GND and 3.3V. While the boards can output up to 5V on the V_x and V_y analog output pins, the FPGA's power supply limits the Joystick analog output voltages to 3.3V.

The on-board XADCs have an input analog voltage range of 0V to 1V, so a voltage divider is necessary to further step down the output voltage. Voltage dividers with 500K and 100K Ω resistors are built between the V_x and V_y output pins and the Analog Multiplexer input channel pins. The first multiplexer selects either the V_x of the left joystick or the right joystick and the second multiplexer switches between the V_y analog voltages via a FPGA digital input pin on the JA port. The V_x multiplexer common output pin connects to the VAUX3P pin on the XADC port, and the V_y multiplexer pin is wired to the VAUX11P pin.

III. Control Module - FPGA Design

A. XADC Sampling

The XADC IP configured for this system's FPGA is operating in a dual-ADC simultaneous sampling mode. The Joystick V_x and V_y analog outputs are sampled concurrently and their 12-bit values are stored in status registers. These registers can be accessed through a dynamic reconfiguration port (DRP). Figure 2 shows the detailed timing of the DRP. For our purposes of continuously reading from the XADC, the DEN enable signal is always high and the DWE write enable is always low. Ideally, sequential logic in our control module would check for the rising edge of the DRDY data ready signal. At the rising edge, one of the ADCs will have their data updated from the status register read and the DADDR channel address will be switched to the other ADC.

However, detecting the DRDY rising edge worked in simulation but did not work on hardware, so the channel address instead switches every 0.6ms. No crosstalk was captured between the channels. The joystick multiplexer select signal (joystick_select) switches after reading both registers on the XADC. The 0.6ms delay also prevents crosstalk between joysticks. Camera position and orientation logic updates every 20ms based on the joystick data. These latency issues are due to our chosen analog multiplexer and FPGA development board. However, the control module is running at a similar speed as the graphics module, so the aforementioned limitations are not significant enough to effect the overall performance of our project.

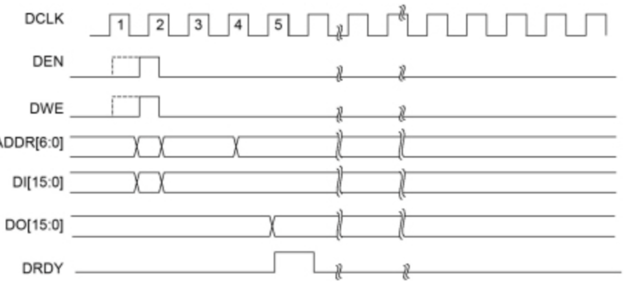


Fig. 2. XADC DRP Timing Example

B. Move, Scale, and Rotate (MSR) Logic

The XADCs convert the V_x and V_y Joystick analog outputs to two 12-bit values (X and Y). Moving the joystick all the way to the left produces 0x000 on the X logic, while a right toggle produces 0x800. For the Y logic, the joystick pulled down results in 0x000 and pushed up outputs 0x800. The MSR logic block in Figure 3 translates the joystick movement to changes in the camera position and direction in the 3-D graphics space.

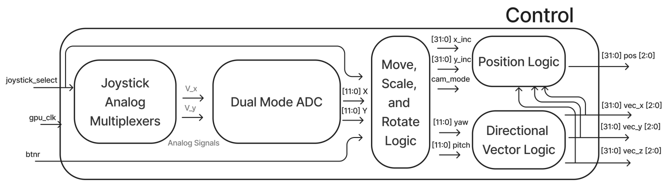


Fig. 3. Control Module Block Diagram

The joystick_select and btnr button input signals control state machines in the MSR logic block that respectively register the input X and Y as left or right joystick values and sets the cam_mode output signal in either a gimbal-lock or free state. In gimbal-lock mode, MSR logic produces a single camera scaling value based on the left joystick Y value and outputs it on both the x_inc and y_inc signals. Threshold ranges are encoded to uniformly increase or decrease the zoom as the joystick moves further up or down. In free mode, the left joystick X and Y values increment the camera's position in the free space based on the same threshold ranges as gimbal-mode. The left joystick X and Y values are calibrated at a resting value of 0x400.

For both camera modes, the right joystick controls the camera's orientation, i.e. the direction the camera is pointing in the 3-D space, by translating the X and Y values into yaw and pitch angles, respectively. The angles are encoded into 12-bits, where $90°$= 0x400, $180°$= 0x800, and $270°$= 0xC00. Threshold ranges for X and Y increment or decrement the yaw and pitch angles depending on direction. For example, a slight move to the right (0x500 $< X <$ 0x600) increments the pitch by 0x5 every 20ms, a further push (0x600 $< X <$ 0x700) increases the pitch by 0xA, and a full toggle ($X >$ 0x700) is a pitch delta of 0xF.

C. Directional Vector and Position Logic

The Directional Vector logic block in Figure 3 uses trigonometry to transform the yaw and pitch angular orientation to directional vectors for the camera's x-y-z axes. The math described in Figure 4 require sine and cosine functions to convert the angles to directional vectors. Since there is no in-built representation of trigonometric functions in the FPGA, a custom sine function module was built. The module uses a Python-generated table stored in read-only BRAM that maps the least significant 10 bits in the encoded angles, i.e. [$0°$, $90°$), to 32-bit sine values in the range of [0, 1].

Combinational logic in the module then determines the sign of the sine value using the two most significant bits in the 12-bit encoded angles. The output of the sine function module is

$$\vec{X} = \begin{bmatrix} \cos(yaw) \\ 0 \\ -\sin(yaw) \end{bmatrix}$$

$$\vec{Y} = \begin{bmatrix} -\sin(pitch) * \sin(yaw) \\ \cos(pitch) \\ -\sin(pitch) * \cos(yaw) \end{bmatrix}$$

$$\vec{Z} = \begin{bmatrix} \cos(pitch) * \sin(yaw) \\ \sin(pitch) \\ \cos(pitch) * \cos(yaw) \end{bmatrix}$$

Fig. 4. Math for Transforming Angular Orientation to Directional Vectors

a IEEE-754 floating point 32-bit value representing a decimal value in the range [-1, 1]. Cosine values can be easily read from the BRAM by indexing 0x400 - angle ($90° - \theta$). The Vector Logic block outputs 3 axis vectors with 3 magnitudes (as formulated in Figure 4), fully conveying to the graphics module in what direction the camera is facing. The latency of the Directional Vector logic block is 9 clock cycles, i.e. 2 clock cycles to read from the multiple sine table BRAMs and 7 clock cycles to perform the multiplications.

The Position logic block takes the 32-bit incremental values for each linear axis from the MSR logic and the direction vectors from the Directional Vector logic to fix the position of the camera in the 3-D space. In gimbal-lock mode, the x and y increment inputs have an equal value (a), which is used to scale the vec_z directional vector input (zoom control). The scaled vec_z value is wired to the 3-dimensional 32-bit position vector produced by the control module and fed to the graphics module.

$$\vec{pos} = a\vec{Z}$$

In free mode, the x_inc and y_inc inputs move the camera forward, back, left, and right in the direction it is facing. The x_inc scales the camera's vec_x and y_inc scales the camera's vec_z. The scaled vectors add their components to each other to create a delta vector based on the global coordinate system. In sequential logic, the position vector updates by adding its previous value to the delta vector.

$$\vec{pos} = x_{inc}\vec{X} + y_{inc}\vec{Z} + \vec{pos}$$

The Position logic block has a latency of 25 cycles from the 32-bit floating point add (9-cycle delay) and multiply (7-cycle delay) implementations. The latency of the Directional Vector and Position logic have a minimal difference, and are only read by the graphics module once per frame, so pipelining to synchronize the pos and directional vectors was not implemented as it would have no effect on the performance of the larger graphics system.

IV. Graphics Module Design

A. Floating Point Modules

Much of the graphics module, and parts of the control module, is implemented using floating point arithmetic, which is useful because numbers with different, and inconsistent, magnitudes are used together. Vivado provides IP for floating point units, but for simulation compatibility reasons and for intellectual curiosity reasons, we are using our own floating point units. These abide by the IEEE-754 spec in all regards except for the handling of sub-normal numbers, which are always rounded to 0. Given that sub-normals almost never occur, and handling them requires significant additional LUT logic, this is a nice saving for our purposes. The floating points units implemented include 32-bit floating point add (with 9-cycle latency), multiply (with 7-cycle latency), and divide (with 30 cycle latency). All of the floating points units are fully pipelined to allow for a throughput of a calculation each cycle. Only multiply uses DSP resources, which is 2 DSP48's per multiply module. On a similar note, there is also a fixed point divider module which is using by the floating point divide as well as the barycentric conversion.

B. Graphics Pipeline Overview

The general diagram of the graphics module pipeline as a whole is shown in Figure 5. The different modules vary in

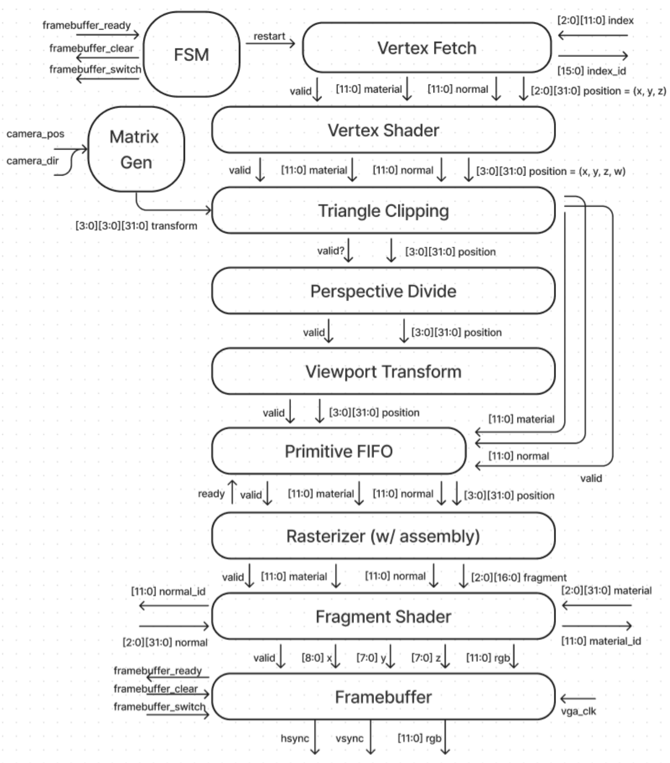


Fig. 5. Graphics Pipeline

complexity, with additional focus being given to the rasterizer and framebuffer, which are the most essential modules.

- The model memory module (not pictured) holds all of the model data, such as vertex positions and normal vectors, as well as the index buffer and the colors of the materials. The vertex fetch and fragment shader modules get index, position, normal, and material data from this module. This module can also be written to over UART, which is detailed in Section V, and allows for rendering of multiple different models.
- At the top of the pipeline, the vertex fetch module fetches the index (which is packed array of a indices for each of the position, normal vector, and material) from the model memory, and then uses that to fetch the vertex position data, and sends that, along with the normal and material indices, to the vertex shader. To render the next frame, there is a restart signal which signals the fetch to restart from the first index. This restart is triggered on a timer, which is set to 50 times each second, but could be altered to a different frame rate, and does not need to match with the VGA frame rate of 60 frames per second.
- The vertex shader multiplies the vertex positions by the transform matrix, which uses 16 floating point multiplies and 12 floating point adds in total. The vertices (and their triangles) outside of the bounds of their w coordinates are clipped.
- The perspective divide divides each of the x, y, z coordinates by w, and outputs $1/w$ as the fourth coordinate, using a total of 4 floating point divides.
- The viewport transform, which has 3 floating point adds and 3 floating point multiplies, converts the coordinates from ranges of [-1, 1] to [0, 320), [0, 240), and [0, 1) for the x, y, and z coordinates, respectively.
- The primitive FIFO stores the vertex positions, and the material and vertex normal indices, so that they can be given to the rasterizer when it is ready, as the raster can spend up to 10's of 1000's of cycles on a given triangle. Due to some issues with simulating Vivado IP, this FIFO was implemented ourselves. It uses one 96-bit width by 8192 entry BRAM for the positions and a 12-bit by 8192 entry BRAM for each of the vertex material and normal indices. Each BRAM has a write pointer which is incremented when the vertex transform produces a vertex, and a single read pointer which is incremented when the rasterizer consumes a vertex. The FIFO uses 1,245 Kb of BRAM, which ends up being a significant contributor to the total BRAM usage of the overall design.
- The rasterizer consumes vertices from the FIFO and links them together into triangles. It determines the pixels inside each triangle, which are then sent to the fragment shader.
- The fragment shader fetches the normal vector and material color from the model memory for each pixel and computes the lighting of each pixel with the following formula: $rgb = rgb_{material} \cdot \max(1, \min(\vec{light\_dir} \cdot \vec{normal\_vec}, 0) + .1)$. The color, and the screen space position of the pixel, are then sent to the framebuffer.

C. Framebuffer

The framebuffer was the first graphics module to be implemented, as it is required to display pixels on the screen and do any verification on the FPGA of the function of the other modules. The framebuffer used 3 BRAMS, which were generated by Vivado IP because of an issue where Vivado would use more block RAMs than was needed when inferring block RAMs. There are two 320x240 12-bit entry BRAMs which store the two frames in 12-bit RGB color, and an 14-bit 320x240 entry BRAM for the depth buffer. This makes for a total usage of about 2,918 Kb of BRAM usage, which is almost two thirds of the BRAM resources on the FPGA. While one frame is being written to, the other one is being scaled and read out through the VGA port at 640x480 resolution at 60 FPS. When a valid pixel is written in at the input to the framebuffer, the z-value is checked against the depth buffer for the pixel at the same position. If the depth at the input is less than the depth stored in the depth buffer, then the pixel is considered valid and the new color value is written to the target frame and the new depth value to the depth buffer. The framebuffer also has an FSM to switch between this normal operating mode, and the clearing mode, where it ignores all inputs and writes the maximum depth of 0xFFFF (clipped to 14 bits) and the color gray (0x222), to every pixel. This clearing period generally coincides with the switching of the read and write target frame BRAMs. The diagram of the framebuffer module is seen in Figure 6:

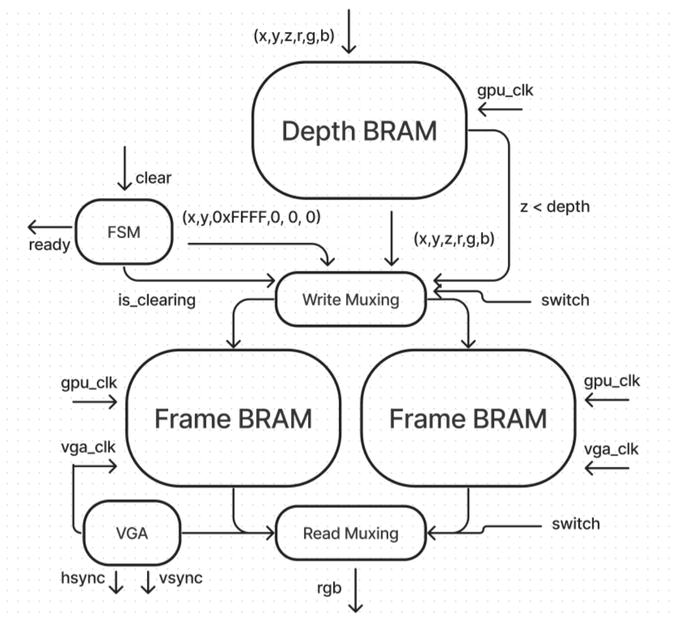


Fig. 6. Framebuffer Module Block Diagram

D. Rasterizer

The rasterizer module is grouped into 3 main portions, the FSM portion, the barycentric conversion portion, and the interpolation portion. This is shown in Figure 7. The barycentric conversion portion and interpolation portion are

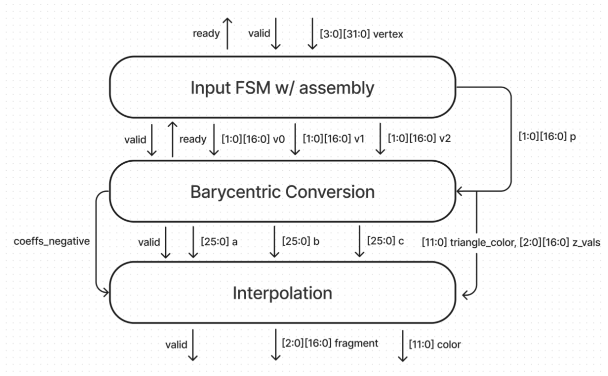


Fig. 7. Rasterizer Module Diagram

fully pipelined, allowing for a maximum throughput of 1 pixel per clock cycle. However, due to the time needed for framebuffer clearing in the parent module and the latency of the FSM setup pipeline, the maximum throughput of rendering is closer to .95 pixels per clock cycle. The FSM diagram is shown in Figure 8. The behavior of each state is the following:

- Ready: the FSM waits for a valid vertex, and when it receives one, it stores it and moves to the Convert state.
- Convert: the vertex's coordinates are converted from 32-bit floating point to 17-bit fixed point with a shift. The FSM then moves to the Store state.
- Store: the fixed point vertex is stored, and the x and y coordinates of the bounding box of the triangle are updated. If it is the 3rd vertex, the FSM goes to Assemble, otherwise it goes back to Ready.
- Assemble: the first sampling point of the rasterization sequence is set to be the upper left corner of the bounding box. Then the FSM transitions to Raster.
- Raster: the FSM sends the current sampling point to the barycentric conversion module, then increments the point in the x direction. If it passes the x bound of the bounding box, it wraps back around to the left of the bounding box after incrementing the point in the y direction. If both the x and y coordinates have passed the bounding box, then rasterization is over and the FSM goes back to Ready with 0 vertices.

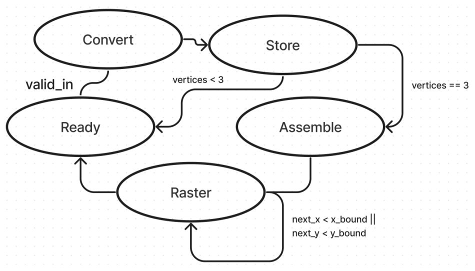


Fig. 8. Rasterizer FSM Diagram

From the FSM, samples are sent to the barycentric conversion module, where each is converted to barycentric coordinates, meaning that each point is rewritten as a coefficient times each of the 3 vertices. This is done by calculating the area of the 3 triangles inside the original triangle made by connecting 2 vertices and the sample point, and dividing those areas by the area of the entire triangle. These calculations are done using the DSP multipliers (which is the reason for moving to 17-bit fixed point), with each area calculation taking 6 DSP48 multiplies for a total of 24 for the entire conversion. If all of the coefficients are positive and less than or equal to 1, then the fragment is considered valid and given to the interpolation section. In addition, a fragment is considered valid only if it is front-facing, which is generally called backface culling, and removes some graphical artifacts occurring when faces shared an edge, shown in Figure 9. Using the coefficients given by

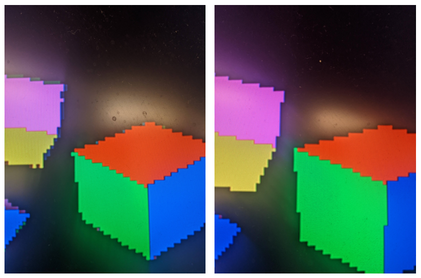


Fig. 9. w/o culling on the left v. w/ culling on the right

the barycentric conversion, the fragment's attributes can be interpolated. For example, the depth value is interpolated as follows, where a, b, c are the barycentric coefficients.

$$z = a \cdot z_a + b \cdot z_b + c \cdot z_c$$

V. UART and Model Memory Module

A target stretch feature for the design was to have a method of importing external models. Originally, importing from an SD card was proposed, but upon advice, this was changed to be from a PC over a serial connection. Thus, the design allows for the models to be uploaded to the FPGA over a UART serial connection, which utilises the USB to serial bridge on the Nexys A7. This required that the UART module had a pathway to write to the 4 BRAMs which held the model data: the indices BRAM, the positions BRAM, the normals BRAM, and the materials BRAM (in reality because this was small, it was implemented as registers). Thus, the required BRAMs are all placed in a single model memory module, with read ports for access. The model memory has a 8196 entry 36-bit width index buffer, along with a 2048 entry position and normal buffer, each with 96-bit width, totalling 721Kb of BRAM usage. In addition, the module exposes a UART Rx and Tx line which is handled by an internal UART transceiver operating at 115200 baud. A diagram for the model memory module is given in Figure 10. Because the UART controller runs at 115200 baud, it takes about 6.259 seconds to transmit the model bits and about 7.823 seconds when accounting for the additional start and stop bits, to rewrite all of the BRAMs entirely. Through Vivado has UART IP available, a controller with a simpler interface was created. When the Rx line goes

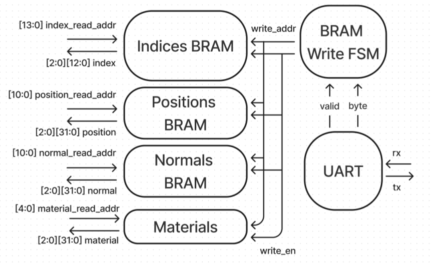


Fig. 10. Model Memory Module Diagram

low, the controller starts a 115200 baud ”clock” which is created using a counter based on the reference GPU clock of 100MHz. Then, the 8 data bits are sampled on the rising edge of the clock, which is timed to fall in the center of each bit’s transmission. After the ending high bit is received, the controller goes back to idle and emits a 1 cycle valid signal along with the received byte. As a part of debugging, it was also made to echo this byte back along the Tx line, though this is not a required functionality. The BRAM writing logic then takes the bytes from UART controller and links them together into valid entries to the BRAMs (which are written upon receiving 5 bytes per index entry, and 12 bytes for the other BRAMs). When the FSM receives an entry of all 1’s, this acts as the stop sequence and the controller goes to writing to the next BRAM in the order of indices, positions, normals, material, then back to indices and so on.

On the PC end, there is a Python script with parses the .obj file and then sends the bytes to the FPGA using Pyserial.

VI. Evaluation and Alternative Implementations

The design can be evaluated in two main facets: the smoothness and the quality of rendering.

In this design, speed of the rendering and control logic, and thus smoothness, was not an issue, as the modules run at comparable speeds to the VGA output. The vertex transform stage has a throughput of 1 vertex per cycle, thus given the maximum model size of 8196 indices, vertex processing is completed in a small fraction of the 1.66 million cycles of rendering time given even when targeting 60 frames per second which is the maximum for the 640x480 VGA standard. On the rasterization side, it takes 10 cycles to setup each triangle, and then outputs 1 pixel per cycle, thus the rasterizer can process, each rendering cycle, about 20 times the pixels in a single 320x240 frame. Though overrasterization is possible, where the same pixel is rasterized multiple times (e.g. when triangles overlap), it is unlikely to happen to this degree. From a timing perspective, the main logic runs on an 100MHz clock, and has 1.979 nanoseconds of slack after synthesis, which is reduced to 0.529 nanoseconds after routing, so there is not much immediate room to increase operating frequency.

Quality can be assessed in multiple regards, but the main components are generally resolution, triangle count, and (lack of) visual artifacts. With regard to resolution and triangle count, the final design uses 97.78 percent of the BRAM on the FPGA, thus there is not space to increase these metrics, though the rendering pipeline could keep up. It should also be noted, that the design only uses 41.62 percent of slice logic and 44.17 percent of the DSPs, so the rendering pipeline couple also be theoretically duplicated to double throughput. There are many possible visual artifacts in graphics, but the three following issues, and possible solutions, are discussed below:

1) Z-fighting, where the depth buffer precision is not sufficient, is pictured in Figure 11. The main method of fixing this is having a higher precision depth buffer, as most GPUs use 24 or 32-bit depth values as opposed to the 14-bit value we used. Given there are not extra BRAMs on the board, this would require augmenting the design to use the DDR.
2) The rendering also suffers from some amount of aliasing, which is expected in rasterized graphics, and is seen in the jagged edges of triangles. Modern GPUs support multisampling, where multiple points per pixel are sampled and then blended before displaying the frame to remove such edges. Like z-fighting, this requires additional memory which is not present on the FPGA without using the DDR.
3) Though it behaves as designed, the way the triangle clipping works in the design can lead to subpar rendering in some cases, where is looks like there are segments of the model missing. This is because triangles are clipped if any of their vertices lay outside the clipping volume. In most graphics applications, however, new vertices are inserted into the pipeline so that an equivalent in-bounds triangle is rendered, which gives smoother clipping. There are resources remaining to implement this on the FPGA, but it would significantly complicate the vertex transform stages and require stalling signals in the entire pipeline.

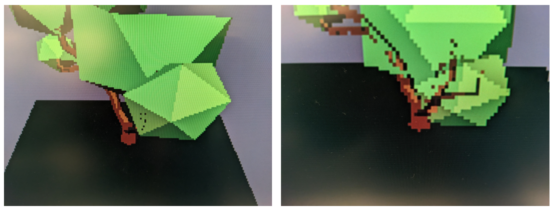


Fig. 11. z-fighting when rendering a model tree

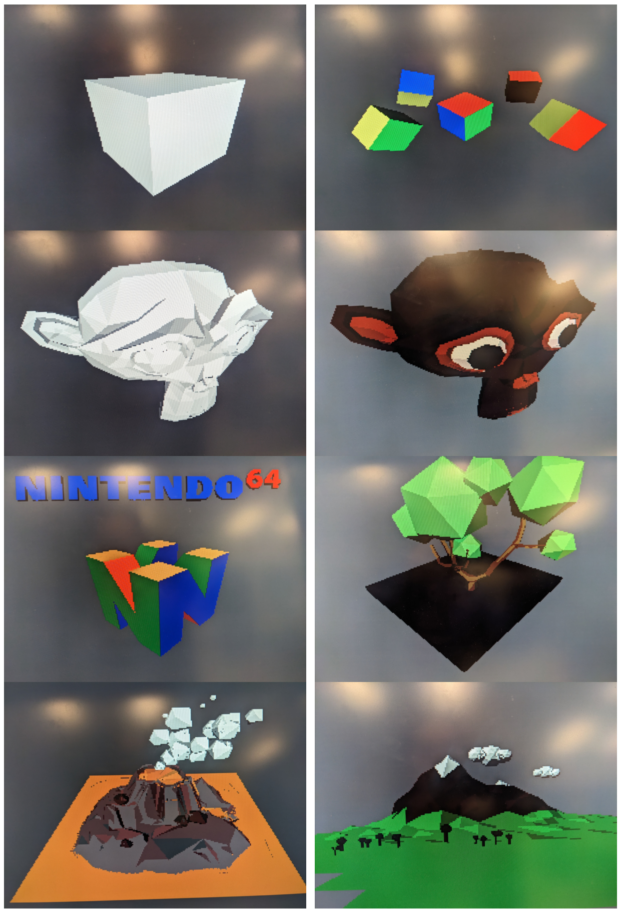
Appendix

A. Source Code

The source for the project can be found at the following repository: https://github.com/kailasbk/fpga-360. All source code used in the project is included, with the following exceptions: 1) Verilog modules given in class (`debouncer`, `xilinx_single_port_ram`, `xilinx_dual_port_ram`), and 2) Vivado generated IP modules, which is the `clk_wiz`, `depth_ram`, `frame_ram`, and `xadc_fpga360` modules.

B. Renderings

As a final inclusion, this appendix includes renderings of all models which were tested on the board. All modules were uploaded to the design over the UART connection (as opposed to being hard-coded).



Credits for the models, from top left to bottom right:

- The top 4 models were created by Kailas using Blender.
- ”N64 Logo” (https://skfb.ly/6TuRp) by Zero One Designs is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).
- ”Low poly tree 3D model” was created by Simon Telezhkin (https://hum3d.com/free/low-poly-tree/).
- ”the volcano” (https://skfb.ly/oyQOx) by Cozmoth is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).
- ”Low Poly Mountain scene” (https://skfb.ly/FtxN) by David Junghanns is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).