HoloForge: A Hand Gesture-Controlled 3D Model Viewer

Mena Filfil
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
menaf@mit.edu

Yeabsira Hawaz
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
yhawaz@mit.edu

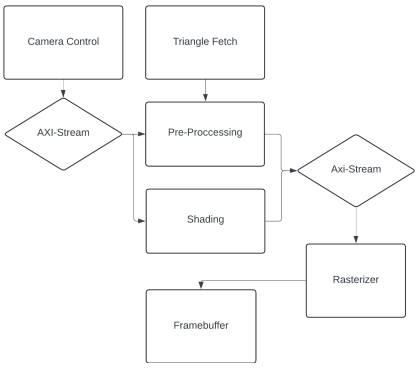

Fig. 1. High-Level Design Diagram

***Abstract*—We present Holo-Forge a 3-D graphic model viewer implemented on an FPGA, with a camera control module that calculates the virtual camera's position based on the center of mass of the user's hand. A graphics module that takes in triangles projects them to 2D and maps the pixels via a rasterizer, and a shader that determines the color and the intensity of the color based on light. And a Frame buffer that allows us to interface with the off-board DDR RAM. We'll implement our design on an Urbana board using a Xilinx Spartan-7 XC7S50-CSGA324 FPGA.**

***Index Terms*—Graphics, Geometry Processing, Gesture Tracking, Field Programmable Gate Arrays, Texture Shading, Rasterization, Dynamic RAM**

I. Overview

In this project we attempt to implement holoforge a 3D model viewer that uses camera inputs to view a 3D model. Utilizing the resources of a FPGA to efficiently achieve the heavy computation required for such a task. The development of Holoforge consists of four core components: Gesture Tracking and Camera Control, Graphics Pipeline, Texture Shadding, and Framebuffering/Interaction with DRAM.

II. High-Level Design

Our design consists of three main disjoint stages that rely on each other:

- Camera Control Signals
- Graphics Pipeline (Triangle Preprocessing, Shading, and Rasterization)
- Frame Buffering

The camera control signals interface with the I/O camera we are using for this project and perform center-of-mass and area calculations that are used to adjust the virtual camera position (more on that in the environment section).

The graphics pipeline consists of 3 substages. In the first one, triangle preprocessing, we take the 3 3D vertex points we received from our triangle fetcher and project them onto the virtual camera plane defined by the information we received from the camera control module. In parallel with this projection, we also fetch normal and color information about the same triangle and perform backface culling and lighting calculations to add the shading effect on the colors of each triangle depending on its angle from the single light source originating from the camera that we are working with. The final stage in the graphics pipeline is the rasterization, where we take 3 2D points with their depth information (distance from the virtual camera), and iterate through a bounding box for that triangle in tandem with interpolating the depth information (to get an estimate of the distance from the camera for every single point inside the triangle).

This pixel-by-pixel information from the rasterizer is then fed into the frame buffer which is responsible for stacking the memory write requests for efficiency and making sure we are only writing to pixels that are closer to the camera.

III. Project Milestones

A. Milestones

Below is an overview of the main project milestones followed and how different parts of these components fit within our roadmap:

- **Commitment**: Render a static scene loaded into BROM, initially without user-controlled interaction(no camera control)
- **Objective**: Enable custom model loading via UART and implement virtual camera control through the cam-control module
- **Stretch Goal**: Increase utilization of DRAM by moving our depth buffer there, and add texture shading.

IV. Environment Assumptions

To simplify our calculations, we decided to use fixed-point math to represent all of our information. Before loading a mesh on the FPGA we normalize it using a Python script to fit in a 1x1x1 3D bounding box. This allows us to compactly represent the mesh in a known range of numbers with a lot of precision ($Q2.14$ fixed-point numbers) in our case. The object is also shifted to be centered at the origin of our 3D world with the camera constituting a sphere around it to allow viewing from different angles.

Similarly, for the camera coordinates, we have constrained the camera distance from the origin to always adhere to a sphere radius $r \in [2, 5]$. Our sin/cos lookup tables are defined in terms of the possible COM values coming from our camera control modules to avoid unnecessary scaling at runtime. Also, our camera and single light source (at least for the initial iteration) are assumed to be at the same position and have the same normal for simplicity's sake. Also, as a starting point, we're working with a frame resolution of 320x180 as a starting point and we intend to increase this if we're able to move the depth buffer (discussed below from BRAM to DRAM).

Throughout most of our modules, we also pre-calculate quantities like half of the viewport width and provide them as equivalent numbers with enough precision to make incrementing values and scaling operations related to projecting our 3D points onto the screen faster. Since these are parameters that are defined at build-time and only relate to the known resolution and viewport of our graphics pipeline, it's unnecessary to make these values dynamic if they only change between different builds.

V. Camera Control

Our camera control module takes in the X_{COM}(COM being the center of mass), Y_{COM}, and the area of the thresholded pixels from the camera. We would then equate the X_{COM} and Y_{COM} as θ and ϕ respectfully, and the area as R. By doing this we can now obtain the spherical coordinates of the matrix, and the vectors of the plane from the tangential derivatives. The position can then be found by

$$(r \cdot sin(\phi)cos(\theta)w, r \cdot sin(\phi)sin(\theta), r \cdot sin(\phi))$$

, and the u,v vectors that represent the plane can be found by

$$u = \begin{bmatrix} -\sin(\phi)\sin(\theta) \\ \sin(\phi)\sin(\theta) \\ 0 \end{bmatrix}, \quad v = \begin{bmatrix} -\cos(\phi)\cos(\theta) \\ \cos(\phi)\sin(\theta) \\ -\sin(\theta) \end{bmatrix}$$

We can save ourselves the computation of finding sine and cosine each time by just using a lookup table. Since we only really need to store sine since $cos(\theta) = sin(90-\theta)$, so we just need to adjust how we index into the look-up table in BRAM, and we have a working way to find sin and cosine of angels, and hence a working way to find out Camera_position, and u, v vectors from camera input. We can also pass the normal, by just giving the camera position, but not multiplying each value by r (or i.e. the area of our camera blob).

VI. Graphics Pipeline

A. 3D to 2D Projection

Our projection consists of 3 individual vertex projections running in parallel with short-circuiting logic between them to allow us to reset the projection if any of them are out of bounds. To save resources for the projection, we simplified the projection logic to be parameterized in terms of constant known at compile-time such as half the viewport width and height as well as the ratio of our viewport to our resolution. This simplification reduced unnecessary multiplies and renormalization and simplified the steps after the projection to 2 simple additions and divides for normalizing the x and y coordinates.

Below is the math that we do for projecting the vertices onto our camera plane. Our camera information is determined by a camera center C, and 3 basis vectors u, v, n that represent a 3D coordinate system relative to the camera. For the outputs let us define dot_x, dot_y, dot_n, and D as intermediary variables.

$$\vec{D} = \vec{P} - \vec{C}$$

$$dot_x = \vec{D} \cdot \vec{u}$$

$$dot_y = \vec{D} \cdot \vec{v}$$

$$dot_n = \vec{D} \cdot \vec{n}$$

$$x_{renorm} = dot_x / dot_n$$

$$y_{renorm} = dot_y / dot_n$$

$$2d_{projcord} = \{x_{renorm}, y_{renorm}, dot_{n_z}\}$$

Where we perform a boundary check on x_{renorm} and y_{renorm} where we ensure they're withing $[-h/2, h/2]$ and $[-w/2, w/2]$

B. Shading

Since our camera and light normals are the same from our environment assumptions, our shading logic is responsible for both calculating the light intensity for a given triangle (a value with which we'll scale the RGB color for that triangle to simulate shading), as well as the directionality of the triangle with respect to the camera which is used for backface culling. Backface culling is essentially the process of eliminating triangles that are facing away from them since they constitute non-surface information (rendering does not really show the internals of a mesh of triangles).

Let $\vec{N}$ be our camera/light normal, $\vec{n}$ be our triangle normal, and I be our light intensity.

$$I = min(max(\vec{N} \cdot \vec{n}, 0), 1)$$

. $I > 0$ implies a triangle facing away from our camera which would constitute a backface cull, allowing the pipeline to skip rasterizing and move on to the next triangle in our mesh. The reason we clip the values between 0 and 1 is to avoid overflows when scaling our 565 RGB color representation.

C. Rasterizer

The rasterizer consists of two main components:

- Bounding Box Generator & Incrementer
- Barycentric Interpolation

The first stage is responsible for taking the 3 2D vertices and calculating the $x_{min}, y_{min}, x_{max}, y_{max}$ boundaries as well as their pixelized $hcount, vcount$ equivalents to iterate through all the pixels in the bounding box around the triangle.

These pixels corresponding x, y values (within the coordinate system of the viewport, not the resolution) are then fed into a barycentric interpolation module along with the 3 depth values at the 3 2D vertex points making up the triangle.

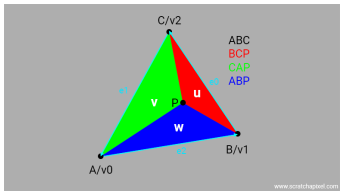

Fig. 2. Barycentric Interpolation

Barycentric interpolation as shown in the diagram above involves taking a weighted average of the 3 depths based on the weighted areas of the 3 triangles that are formed between the point x, y inside the triangle and the vertices inside, making up the whole triangle. The exact math for the interpolation would look like this:

$$z_{interp} = u \cdot z_1 + v \cdot z_2 + w \cdot z_3$$

$$u = D_1/D, v = D_2/D, w = D_3/D$$

Where D is the total area of the triangle, D_1, D_2, D_3 are the areas of the triangles formed between x, y and each of the original triangle's 3 edges, and z_1, z_2, z_3 are the depths for each vertex in our triangle.

One nice property about this interpolation method is the fact that the 3 coefficients, u, v, w making up the areas of the 3 triangles are always guaranteed to have the same sign when an x, y point truly lies inside the triangle. This allows us to easily eliminate points outside of the triangle if this invariant is violated.

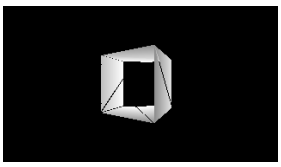

Fig. 3. Rasterizer Example 1

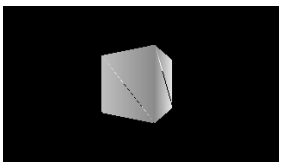

Fig. 4. Rasterizer Example 2

Figures 3 and 4 show some examples of interpolated depth maps generated by a combination of our rasterizer and pixel preprocessing components of the graphics pipeline during the rasterization in simulation.

D. Whole Graphics Pipeline

Once each of these sections was tested and verified in isolation, the next task was to combine them, this meant creating a module that ran the shader and pre-processing simultaneously, then communicating the output of both of these modules via AXI Stream to the rasterizer.

VII. Framebuffer

The FrameBuffer comprises two 320 x 180-pixel color frames each with a 16-bit color width, stored in DRAM. Additionally, a 320 x 180 x 16-bit depth buffer is stored in the BRAM. The system alternates between two color frames: one frame being read out of the DRAM and being sent out via HDMI to the screen, and the other frame currently being written based on inputs from the Rasterizer. When the framebuffer receives clear signals from the graphics pipeline, they switch roles, successfully allowing for there to always be a fully rendered frame on screen. It will also clear the new write frame, and clear out the depth ram before allowing for new input from the graphics pipeline.

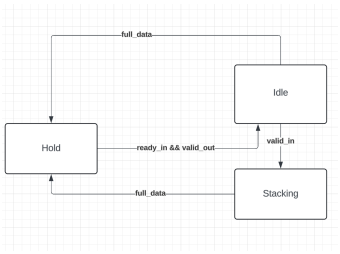

Fig. 5. Pixel FSM

A. Depth Check/Clearing Frame

Once we receive the hcount, vcount, and depth values, the system checks whether the depth for the hcount and vcount is less than or equal to the depth stored in the Depth BRAM at the corresponding address. If the incoming depth exceeds the stored depth, a mask signal is sent to the MIG req gen, signaling it to disregard the data associated with that hcount and vcount—ensuring that only closer pixels are prioritized for inclusion in the frame. However, if the incoming depth is closer or equal to the stored depth, the mask signal is not triggered, allowing the new data to be stacked and sent to the frame.

All of the input values are then pipelined for 2 cycles in order to account for the stall of the BRAM. There's also a stage of muxing before the values get to interface with the MIG, these values are decided based on whether or not we're clearing. If it's the first 10 cycles from when we received the clear signal(which we keep track of with a counter based on the ready_out signal coming from the mig). We set the strobe_in and write enable signal of the BRAM both to low, since the purpose of these writes is to guarantee that the last pixel was successfully put in the DRAM FIFOs with the correct frame associated with it.

B. Interactions with MIG and DDR3 RAM

Due to a non-negligible portion of our data coming out of our rasterize being out of order, reusing the DRAM interface from lab06 wasn't feasible. So we opted to implement a different IP to interact with the MIG. The AMBA AXI Protocol gives us a wrapper around the memory interface with 5 ports(but only 4 that we care about) a Read Address Channel, a Read Data Channel, a Write Address Channel, and a Write Data Channel. The IP will handle the arbitration for us and has a strobe signal for writing data, which is very useful for our out-of-order data processing. In order to stack out-of-order data, we have a module called "Pixel_stacker" that takes in 16-bit color values along with their addresses. It handles misalignment (meaning two consecutive values aren't in the same 128-bit stack) by just sending out the data we currently have stacked, and setting the strobe signal of the renaming data associated with that chunk with 0. This allows us to still stack to 128 bits when we can, but not allow the misaligned addressing to cause inconsistencies in the DRAM. We also have a wrapper file around the MIG(that we named DDR_Whisperer), that successfully handles the address generations based on the AXI signals coming out of the MIG(including the frame), and handles all the AXI protocol and clock crossing using the AXI Fifo IP. The state machine for the pixel stacker can be seen in Figure 5, this design was simplified heavily from the previous version we proposed in our preliminary report, mainly to simplify debugging while integrating.

C. Implementation

VIII. Results

We were able to successfully rasterize and render a cube using the DDR framebuffer to store and align the pixels and write and read from two different frames successfully. This graphics pipeline did not end up using the shader module since we had trouble synchronizing the backface culling with the rest of our pipeline (our AXI connections would result in a deadlock that's really hard to debug in that case). The DRAM was rendered at the same resolution except it was placed in the top corner of the screen since we were focusing mostly on getting correct rendering behavior before zooming it in to fill the whole screen.

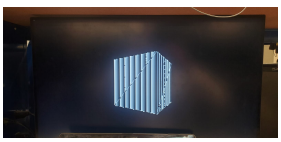

Fig. 6. Test Cube Rendered from BRAM

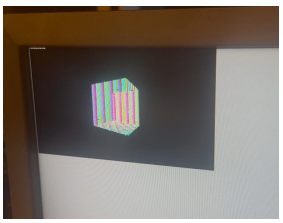

Fig. 7. Test Cube Rendered from DRAM

IX. Resource Utilization

We tried to minimize redundant logic in our design, especially for cycle-intensive operations like division. For example, our rasterizer does a single division by pre-calculating the value of $1/D$ to avoid the need for multiple dividers and all of the extra logic and resources that come with it.

To that end, we've synthesized some of the main modules in our graphics pipeline with the following resource utilization on the board:

- Triangle Preprocessing (30 DSPs)
- Rasterizer (25 DSPs)
- Shader (5 DSPs)
- Camera Control (estimated 15 DSPs since it's not yet implemented)

With the current design, we currently have 10% of the BRAM with memory capacity for about 1000 triangles.

Breakdown of BRAM memory usage:

- Mesh with 2,000 triangles about **0.42Mb**
- Color Palette about **4Kb**
- Depth buffer (320 x 180) (16-bits for depth) **0.88Mb**
- *Remaining* 2.7Mb - (above) = **1.3Mb** free for use with FIFOs and buffering

X. Timing Constraints

During the initial implementation of the graphics pipeline, we ensured correct pipelining for data-intensive sections like the rasterizer and the shader. With this, we implemented a pipelined rasterizer with a throughput of a single pixel per clock. The overall critical path delay of our whole graphics pipeline was about 7.503ns and with this, we ensured that we met timing on the main 100mhz clock driving our pipeline with some additional slack. We tried to integrate the pipelining into our design from the beginning and there are minor opportunities that we found for additional pipelining to drive the delay down and allow for potential higher frequency driving the rasterizer, but we did not have time to implement these changes, and test them thoroughly. To ensure that our pipelined rasterizer is also able to abide by AXI we created a modified freezable pipeline module that let us hold all values static to allow for correct AXI protocol without sacrificing throughput or dropping data packets.

We currently drive the system of 3 separate clocks, a 100mhz clock to drive the Graphics Pipeline, a 200 mhz clock to drive the camera(and to serve as reference for the DDR RAM), and a 74.5mhz clk to drive the HDMI signals out. We also have a 325mhz clk being produced by the DDR RAM, which is also the clock at which it produces data. This means we need clk_crossing between the DDR and the rest of the system.

XI. Retrospection

A. Design Flaws

A lot of design flaws weren't revealed in design but rather during implementation which really slowed down the progress of the project. A good example is the clearing state and

FSM, the entire functionality of this wasn't considered until we were already halfway through integrating the Graphics Pipeline with the FSM, this also really halted debugging since a lot of time debugging would turn into us questioning our design rather than debugging for functionality while already having a design in mind. Another one that came up was we didn't realize that our pipelining logic had to abide by AXI, so halfway through development we had to implement the freezable pipeline mentioned, which once again doubled the difficulty of debugging given bugs since it was a lot harder to track points of trouble.

B. Integration

The integration was a huge part of the reason we didn't make as much progress on the project as expected. Despite simulating every module we could use CocoTb and Iverilog, and stress testing our systems very hard with those tests. The integration between various internal aspects of the framebuffer to make the whole framebuffer, and the integration between the framebuffer and the graphics pipeline caused us to spend way more time debugging than planned. These bugs were also just so hard to track since we aren't really able to simulate any aspect of the project that interfaces with the DRAM. A big lesson learned here is we need to synthesize things in Verilog much earlier in the process. Thankfully we didn't violate timing or use too many resources on the board, the main issue was just not being able to debug the on-board bugs we couldn't find in simulation in time. Another aspect of integration that would've helped complete more of the project sooner was just synthesizing on the board during the earlier states. We had a few bugs revolving around misconceptions/typos around which clocks drove which aspects of the project, which took a considerable chunk of time and would've greatly helped in debugging integration if done sooner.

C. Framebuffer

This gets its own section since in retrospect the framebuffer should've been the first thing implemented. We went into this project planning to do all the simulatable aspects(most of our math and the graphics pipeline) first then working on the framebuffer after those aspects were done and fully tested. However we really should have worked on the framebuffer earlier, that way we could have centered our project around that. Instead, we had a fully working(at least in simulation) graphics pipeline that we just tried to attach to the end of the framebuffer, which didn't quite end up working. Also, a lot of the design flaws with the framebuffer like the state machine for the out-of-order data stacker and clearing state, would've been much easier to debug if we had solidified the design in the earlier stages of the project. Utilizing a different IP, and taking out a lot of the support offered from deviating from lab 6, made the framebuffer incredibly time-consuming and challenging to debug.

However all and all this project was an incredibly educational experience, and a great mix of fun and challenging. And if we were to have any advice for future teams looking to do something similar it would be to start synthesizing on the board.

ACKNOWLEDGMENT

We'd like to thank the entire teaching team for all the help, guidance, debugging, and fun times in the lab. We'd also especially like to thank Kailas, Jan,and Kiran(but especially Kailas) for all the advice,debugging,and support throughout this project, and Joe Steinmeyer for all the great lectures.

REFERENCES

Code Repository: GitHub Repo: holoforge.