3D Model Interfacing Final Report

Ryan Xiao
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology
Cambridge, MA
ryxiao@mit.edu

Jessica Pan
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
jnpan@mit.edu

Annie Giroux
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
girouxa@mit.edu

***Abstract*—We present a design for a 3D graphics pipeline implemented on a RealDigital Urbana FPGA, capable of rendering and transforming a given 3D model to a HDMI-connected screen. Users are intended to input any low-poly 3D model via an obj file, then interface with the model through translation and rotation commands. The pipeline then performs all necessary calculations to transform, rasterize, color and display the given model. In this report, we discuss design choices for our graphics pipeline, focusing on the general procedure for transformation and rasterization calculations and the decision to use a Z buffer to resolve conflicts. In addition, we explore an evaluation of our project, focusing on potential memory solutions.**

Index Terms— Graphical user interfaces, Graphical models, Field programmable gate arrays, Digital systems

I. Physical Construction

A. Overview of a general 3D Graphics Pipeline

Projecting a 3D model onto a 2D surface requires approximating points and surfaces in the 3D object so that they can be projected onto a 2D plane. 3D models most often store this information as a collection of triangles.

The triangles in a 3D model are rasterized by projecting them onto the 2D viewing plane and coloring them based on their angle to the light source. In the case of two overlapping triangles in the 2D projection, only the triangle closest to the camera should be displayed.

Rendering these triangles requires a lot of low-latency arithmetic that can be done non-concurrently, allowing us to take advantage of the FPGA's parallel processing capabilities. Thus, our FPGA could have a much lower energy consumption than traditional GPUs. However, building a graphics pipeline necessarily involves float calculations, which must all match strict timing requirements.

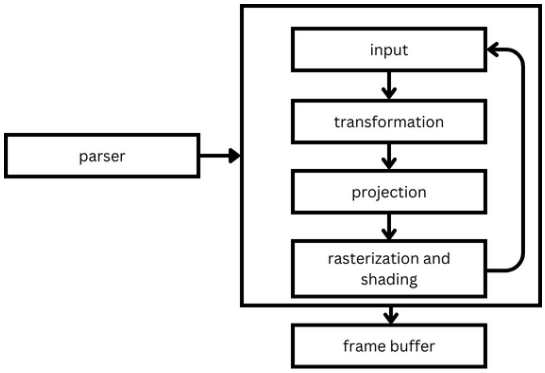

Fig 1. A high-level overview of our graphics system. the full detail diagram is included on the next page

B. Physical System

Our system is intended to be housed entirely within the standard RealDigital Urbana FPGA, yet we were not able to render a full object using only the FPGA. We were able to achieve rotation with our project, but further work would be necessary to render full models.

Although many of our individual modules worked in simulation on their own, modeling on the FPGA itself required too much space for our design scheme. Unfortunately, our group realized this error too late to pivot away to an entirely different scheme, so this report will focus on the tasks we were able to accomplish during our project period.

II. Processing Inputs

A. Inputting the 3D Model

The system takes inputs directly through the switches on the FPGA. The output is displayed on an external monitor through an HDMI cable.

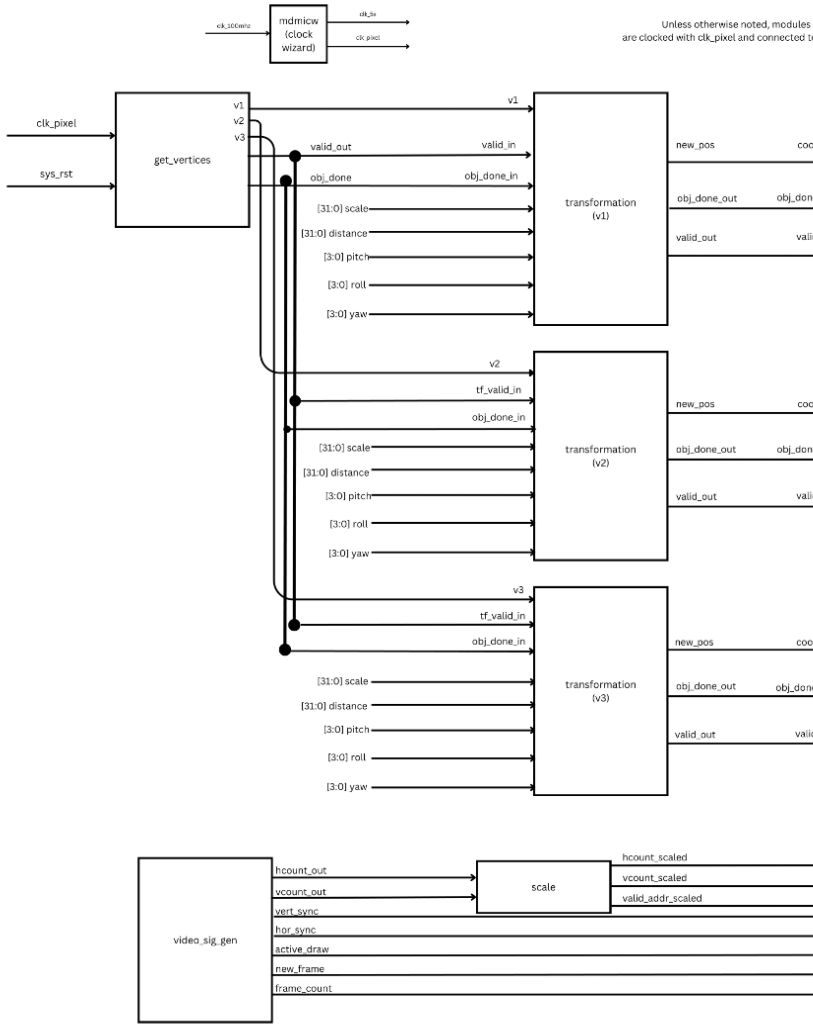

Fig 2. A block diagram for our full process up to the rasterizer (including tri handling video signals)

Our system receives low-poly 3D models via an obj file. Obj files list all of the vertices and facets (triangles) in the object in a human readable format.

We've written a python script to convert the obj file to a format that can be uploaded directly onto the BRAM of the FPGA. This script retains the facet information for the model on the FPGA, in addition to calculating the center of mass of the model for theoretical ease of translation. This script only needs to be run once per 3D model imported into the system. The models we built and tested our project on range from 12 vertices to ~1500 vertices.

B. Real-Time User Inputs

The on-board design is written such that the user would directly give pitch, roll, and yaw instructions on the FPGA. Our design receives input from the buttons and switches; rotation commands (roll, pitch and yaw) are taken from buttons 1 through 3. Though our system did not meet our stretch goal of translation and scaling, we wrote a module that would have been used to parse system inputs (which uses 12 switches on the board, one to increase and one to decrease seven variables — x-axis and y-axis translation, pitch, roll, yaw and scaling).

III. Vertex Transformations

The vertex shader applies transformations to modify the vertices of the displayed model based on real-time input.

A. Rotation

We initially used on-board matrix multiplication to transform the model based on pitch, roll and yaw, however, after our preliminary findings, we realized that we only needed matrices for rotation. To complete rotation, we utilize matrix LUTs stored on the board. We append a *w* coordinate of 1 to our vertex data to convert it to a 4x1 matrix in homogeneous coordinates, so that we can rotate the vertex with respect to the origin by multiplying with a 4x4 rotation matrix. The FPGA has the ability to parallelize matrix multiplication, making it able to rotate objects more efficiently than other computation systems.

B. Translation

We apply translation after rotation (so that rotation can be relative to the origin, simplifying calculations) based on position data updated by our input controls. The vertices will then be translated relative to the object's center of mass (pre-calculated in the vertex BRAM).

For scaling, we multiplied the coordinates by the requisite scale factor

IV. Rasterization

The rasterization pipeline projects each triangle onto the 2D plane and finds pixels within those triangles to pass onto the coloring module.

A. FIFO Buffer

Because the rasterization pipeline takes a variable number of clock cycles, a FIFO buffer would ensure that the system is synchronized and no data is lost. We intended for the buffer to receive and output individual triangles (as a collection of transformed vertices), utilizing Vivado's AXIS Stream Data FIFO IP. However, we had significant difficulty implementing it and ended up deciding to instead pipeline entirely with ready signals, which made effective pipelining difficult.

B. Triangle Projection

First, we project the triangles in 3D space onto the 2D viewing plane. Because our camera is fixed in space, we simply multiply our vertices in homogeneous coordinates by the perspective projection matrix. Then, we divide by the *w* coordinate to convert back to positional coordinates. The *z* coordinate will now be constant, so the *x* and *y* coordinates will be their positions in the 2D viewing plane. Finally, we map the *x* and *y* coordinates from the domain (-1, 1) to their final positions on the screen in the domain (0, 240). This module contains the bulk of the rasterization pipeline, projecting a triangle's vertices in homogenous 4D coordinates to their positions in the viewing plane. We pass forward both these 2D vertex positions and the distance between the original triangle and the camera to later resolve overlapping triangle conflicts.

C. Determining Pixels Within Each Triangle

To color the pixels within each triangle, we must determine which pixels in the 2D screen are inside of the projected triangle. To do this, we begin by creating a bounding box for the triangle (taking the minimum and maximum value for each dimension) to narrow the search space. Then, we iterate through each pixel in the bounding box and use the Convex Hull method to determine whether or not the current pixel is inside of the triangle of interest.

In this method, v is the current pixel coordinate, v_0 is a vertex of the triangle and v_1 and v_2 are vectors representing the sides of the triangle originating at v_0. We then calculate a and b:

$$a = \frac{det(v\, v_2) - det(v_0 v_2)}{det(v_1 v_2)}$$

$$b = -\frac{det(v\, v_1) - det(v_0 v_1)}{det(v_1 v_2)}$$

We will calculate a and b utilizing Vivado's floating point reciprocal ip, and taking the determinants of 2D matrices will require only multiplication and subtraction. If $a > 0$, $b > 0$ and $a + b < 1$, then the pixel v is within the bounds of the triangle and should be passed to the pixel shader. Because the bounding box is different for every triangle received, the rasterizer has a variable run time, which is most effectively implemented as a state machine. The state machine consists of 6 main stages.

We are using the swapping method for the frame buffers, switching between BRAMs. For this reason, we have an Erase and Next state, which iterates through the entire BRAM after it has been swapped from being read from and clears it to the default value of black color and maximum depth.

Once this is completed, there is a Receive state where the rasterizer is ready to receive a newly projected triangle. The Iter and Check states are where the rasterizer will spend the most time, as it needs to iterate through the entire bounding box in order to get all pixels that are within the triangle and check if they are visible. We do this with the Z buffer method, where the depth is stored with the color in the frame buffer BRAM and if the object has a lower depth than the existing value, it will be more visible, and its value can replace the existing value in the frame buffer.

V. Coloring

The pixel shader receives pixels on the 2D viewing plane that are within the bounds of a given triangle. It

has two roles: coloring these pixels for the final display and maintaining a Z buffer to resolve overlap conflicts.

A. Pixel Coloration

This module receives a triangle and maps it to a single one-byte grayscale color. We consider the light source as coming straight out of the camera.

Because we define light source as constant and uniform from a single direction (similar to sunlight), we will color the pixels according to only the angle between the projected triangle and the light source (Fig. 3). We find the normal vector of the original triangle from the cross product of its sides v_1 and v_2, and then determine the angle between the normal vector and the fixed light source. We define an exponential dropoff of light as the angle gets larger, and if the angle is more than 90° (i.e. the surface is facing away from the camera), the triangle is colored black.

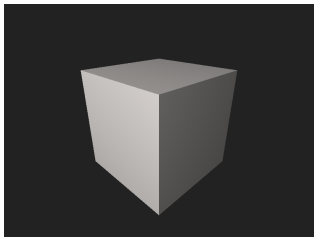

Fig 3. A cube where faces are colored based only on the angle between the face and the light source.

In practice, we implemented this with a pipelined state machine, where each state interfaces with several IP chips (Table. 1). In the first three states, we calculate the vectors from the given coordinates, then calculate their cross product using 6 IP multipliers and 6 IP subtractors. We solve for the cos of this angle using the dot product formula:

$$cos(\theta) = \frac{v_1 \cdot v_2}{|v_1|\,|v_2|}$$

However, because we know the angle of the light source is a fixed vector towards the camera (0, 0, -1), this simplifies. We can also avoid calculating square roots by squaring both sides. If the triangle's normal vector is $< a, b, c >$, then we can use the following formula:

$$cos^2(\theta) = \frac{c^2}{a^2+b^2+c^2}$$

Once we've solved for the normal vector, we square each component and get the magnitude (using 3 IP multipliers and 2 IP adders).

Our ultimate goal is to map the angle to a color, so we want to map the float cos^2 values to integer indices. We do this by multiplying by 16 and rounding to the nearest integer. To parallelize computation, we can multiply c^2 by 16 while taking the reciprocal of the magnitude, then multiply and round the final result.

Finally, we use a lookup table to map the value of cos^2 to one of 16 one-byte grayscale color values. Once we know the cos^2 of the angle, we can approximate the angle itself, and hardcode a value of gray for each rounded cosine result.

State	IP Modules Used
RECEIVE	N/A
VECTOR_CALC	6 subtractors
NORMAL_CALC_MULT	6 multipliers
NORMAL_CALC_ADD	3 subtractors
SQUARE_NORMAL	3 multipliers
MAGNITUDE	1 adder (used twice)
RECIP	1 reciprocal
COS_SQUARED	1 multiplier
ROUND	1 float-to-fixed
COLOR	N/A

Table 1. IP modules utilized in each stage of the pixel shader.

B. Outputting to a Z Buffer

The pixel shader also resolves conflicts that arise when two 3D triangles map to the same pixel location when projected onto the 2D plane. We do not want to display triangles that are hidden from view, so we maintain a Z buffer as we iterate through each triangle. For each pixel in the final output image, the Z buffer holds the pixel color and distance between the displayed triangle and the camera. Thus, if two triangles map to the same pixel in the 2D image, the output can display only the triangle closer to the camera and discard the value of the triangle obscured from view.

Maintaining the Z buffer was very memory intensive. The system stores 240 x 240 pixels, each of which requires two bytes — one for the grayscale color of the pixel and one for the distance to the camera to resolve conflicts. This is a total of 115,200 bytes of memory. The FPGA has 75 BRAMs, each of which stores 36 kbits, for a total of 337,500 bytes of memory in the FPGA BRAMs. This leaves only 222,300 bytes for the rest of our processing, buffering, and data storage, which was sadly not enough for the final iteration of the project.

VI. Output

Finally, our Z buffer is passed to a frame buffer to be displayed. Though we originally created a double buffer in which one BRAM would accept downstream data and copy it to another BRAM, we

elected to change this process to utilize a ping pong buffer, in which two BRAMs can be both read from and written to.

Using this approach, after receiving an obj-done signal, the buffers will be swapped such that the buffer that was being written to is now being read from and the buffer that was being read from is now being written to. Although this requires more complexity to calculate where to read and write, it allows dynamic flexibility in sending and receiving data, in addition to lending itself well to an efficient buffer clearing strategy.

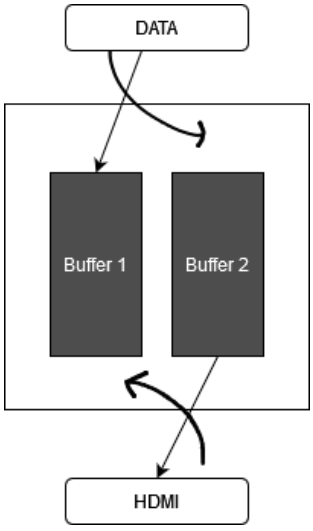

Fig. 4: Ping Pong Buffer

This process works using two BRAMS with 240*240 17-bit entries each. In each entry, the most significant 8 bits are the greyscale color, while the least significant 8 bits store the depth. This output is projected onto an external monitor through the HDMI protocol. Since the HDMI output is only read from the video buffer, editing the frame buffer will not cause artifacts. The goal was for the final screen to be 360 by 360 pixels in dimension, however, space constraints limited us to 240 by 240.

VII. Evaluation

A. Memory and Timing

Our extensive float arithmetic cost significant overhead. In order to perform costly operations such as in pixel shading, we chose to use 6 identical IP modules in parallel instead of in series. This caused the pixel shader alone to utilize 26.67% of the DSPs on the FPGA. We feel that this choice was justified by the decreased latency. The multiplier IP, for instance, takes 12 clock cycles to complete, so running the six multipliers in series would require 72 clock cycles to complete the multiplication step while also requiring additional logic and storage. We were able to parallelize the triangle projection and transformation modules, one for each of the vertices in the incoming triangles. Each module consisted of multiple floating point IPs. In total, we used 65 of the 120 available DSPs, which means theoretically, we would be able to double our throughput given more parallelization.

We were limited by the amount of BRAM available on the FPGA. When doing early calculations, we underestimated the amount of BRAM required for the logic and double buffer, so we had to cut the output resolution to 240 by 240 to be able to synthesize on the FPGA (from 360 by 360 in our original design). As described earlier, the FPGA has 337,500 bytes of on-board BRAM space. In addition to the Z-buffer, which takes up 115,200 bytes, the buffering scheme results in an additional 122,400 additional bytes of space as there are two BRAMs in our final buffer.

We did not give enough emphasis in our preliminary report that the buffering mechanics of the system would also cause a significant amount of overhead. The Z-buffer, frame buffer, LUTS, and large amounts of on-board memory required for large variables and instruction storage resulted in a project that was unable to fit on the FPGA. One point of insight is that we did not necessarily need to have 32 bit precision in our floats. These values take up a lot of space and given our eventual mapping to an integer coordinate on the screen, this preciseness of a coordinate value was not entirely needed. Utilizing a 24 bit precision float would free up more BRAM storage for us to increase our resolution, which was another point of issue. Especially with smooth objects, there are many more smaller triangles on the object surface, so projecting these small triangles may yield smaller and finer areas, which low resolutions simply would not be able to pick up. There is a possibility that this had an affect on our final results.

Our current project iteration has 1.571ns of slack from the pixel clock with period 13.468ns. Our logic was mostly split IPs and accessing BRAMs. However, calculations for the Convex Hull method required more expensive calculations, necessitating us to split those calculations into separate stages. Notably, determinant calculations require two steps of multiplication, which is already an expensive operation to carry out in hardware.

Because of the difficulties with triangle projection and rasterization, it is difficult to determine if other modules are functioning as intended. With the additional resources necessary to create a fully rotating rendered object, we speculate that the design could be easily modified to include translation and scaling capabilities. In addition, it would be relatively easy to add more complicated 3D models, since each model only needs to be processed once to create a parsed vertex file. With a larger amount of polygons, however, we would have to be careful to adhere to latency constraints.

B. Iteration and Process

Although our full project did not end up functioning, our process built heavily upon what was taught

during the lab portion of the class. We created, simulated, and tested nine different modules within a single project, gaining new experience with 3D rendering, function LUTs, parallelizing matrix operations, and frame buffering. In addition, we got extensive experience with integrating IP modules (with modules like the pixel shader interfacing with 14 IP modules), pipelining finite state machines, and writing test benches (both in Manta and system Verilog). Through the process of implementation, we also created several modules we did not end up including in the final build scheme.

VIII. Possible fixes and future iterations

As discussed in evaluation, we realized after Thanksgiving break that we could not fit 360*360 video on our FPGA without breaking space limitations, and decided as a result to attempt to scale down the video to 240*240. An ideal fix would reintroduce our initial plan to set up an SD card or off-board storage. Using this method, we could store a portion of the project that required a large amount of memory, like the Z-buffer, and load it into a smaller cache BRAM as needed. In doing so, however, we would introduce additional timing concerns into the main pipeline.

Further iterations could include the use of 2 port BRAMs to store vertices. Our current iteration only used ROM through mem files exclusively. This is simple, but requires us to apply each transformation to every vertex for every render, which is very inefficient. A future idea would be to only accept one user inputted transformation at a time and once transformed, we send the vertices both forward through the pipeline as well as back into the BRAM.

VIII. Individual Contributions

Ryan took point early on on researching 3D graphics methods, and developing our transformation system, triangle projection system and rasterizer. Jessica designed and implemented our pixel shader and input system, helped with transformations and took point on our written reports. Annie investigated SD card caching, researched and implemented a double buffer, assisted with the pixel shader and wrote the shader test bench and simulation script, assisted with signal integration, and took point on system block diagrams and state machine graphs.

IX. Source Code

Our code repository can be found at https://github.com/ryxiao0/FPGA-3D.