A 3-D Rendering System

Mayur Desai & Benjamin Hebert

6.111 Final Project Report

Teaching Assistant: Charlie Kehoe

May 12, 2005

Abstract

The goal of this project is to design and implement a viable 3-D rendering system onto the Xilinx Field Programmable Gate Array (FPGA) kit. This system takes a 3-D model and proceeds to render, shade it, and output the rendered image onto a VGA monitor. The user is able to rotate the model, zoom in and out, and watch the image update on the fly. The implementation of this project initially consists of the rendering engine, the video controller, and a top-level finite state machine. The rendering engine is responsible for the transformation from the 3-D model file into frames of RGB (red-green-blue) pixel values. The video component is responsible for receiving these pixel values, storing the frame in memory, and then outputting the data to the D/A video converter with the proper timing and setup parameters. Additional features implemented in this system are lighting and a choice of model to display on screen.

Introduction & Design Overview

The design problem of this project is to create usable and feature-rich 3-D rendering system. The resulting design that was implemented in this project is able to load up a 3-D model from memory, render and shade it, and display the resulting image on a VGA monitor. This project is interesting and relevant because an increasing number of applications, electronics, and tools all use 3-D graphics to create a rich and exciting experience. Creating 3-D images is a difficult task that requires a large amount of computation and time – these features also make a hardware rendering engine very attractive. Due to the large number of computations and intensive memory use required by 3-D rendering systems, many modern computers have dedicated video cards to handle the task. Our 3-D rendering system implements many of the same features that are found in video cards. This project exploits the ideas of parallelism and pipelining to render images with relatively high performance. This project was completed using an iterative approach: a simple rendering system without shading was implemented along with video and a simple system FSM. Additional features such as shading, rotation, zoom, lighting, and model switching were then added to the system. This features create a more detailed and interesting experience for a user. We decided to follow a highly modular approach that relies upon the major/minor FSM idea to coordinate between the different units of the system.

Architecture Design & Implementation

The project was implemented via a layered design that consists of many modules, sub-modules, etc. Each module is responsible for a discrete section of the project, and also for coordinating among its sub-modules. This method was chosen because it helps to isolate errors, and it is a convenient way to think about a project that requires many modules. Our project ended up consisting of over 30 different modules, so this report will only give details of the most pertinent ones. The complete code for all modules can be found in the Appendix.

I. Top Level Module & FSM

The top level module is responsible for coordinating the operation of all the units that comprise the 3-D rendering system. The main components that need to be synchronized are the Video Controller and the 3-D Unit. The reason for this is that a double-buffering scheme was used, in order to allow for continuous reading to and

writing from the onboard Zero Bus Turnaround (ZBT) memories. Because of this added complexity, careful coordination was required and accomplished through the use of the major/minor FSM idea. The FSM in the top module plays the role of a Major FSM that arbitrates between units. As seen in the diagram below, the double buffering scheme means that the 3-D unit first writes to a memory, then when it is finished, it begins to write to another one, and the video commences. Once the video has finished reading a frame from memory and the 3-D unit has finished outputting data, then the buffers switch and the 3-D unit begins to render more pixels.

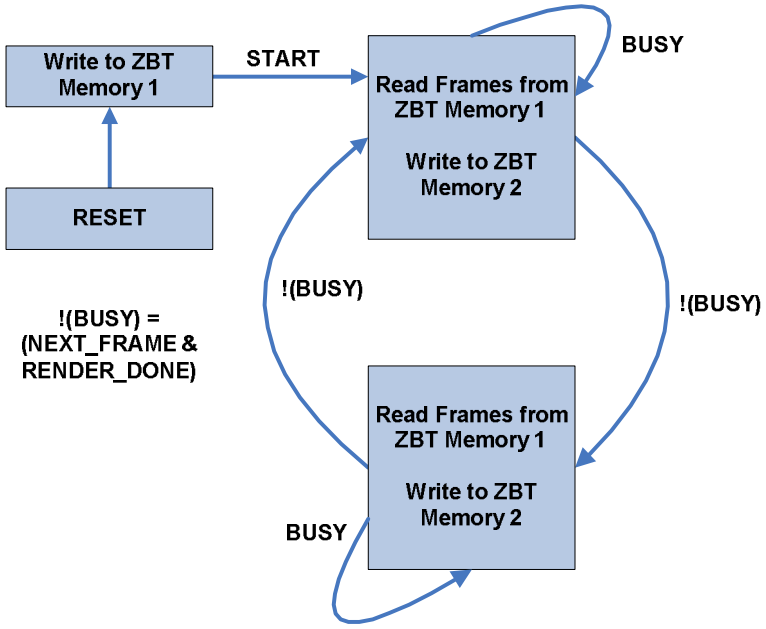


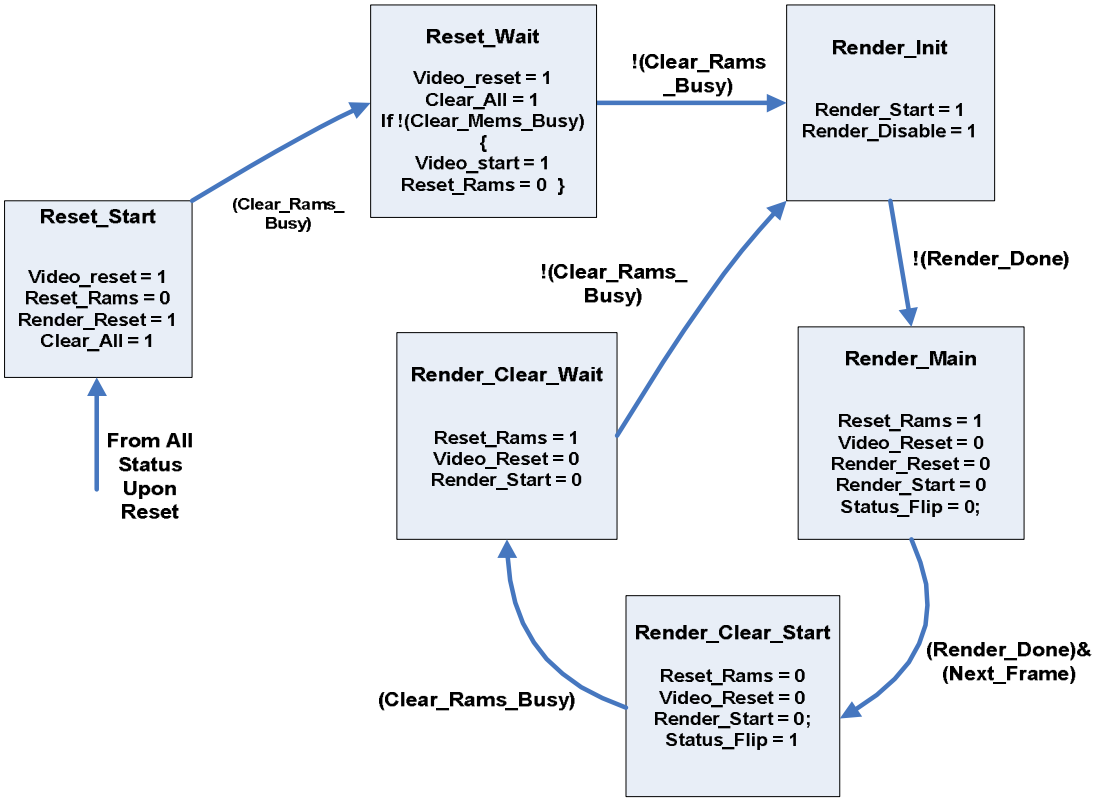
Figure 1; Schematic of Double-Buffering Scheme

The final system consisted of many modules, as stated earlier, and a block diagram is shown in Figure 3 of the final system from a top-level view. The I/O buses used to connect the Video and 3-D components to the memories create a maze of lines, but the idea behind it is relatively straightforward.

The Major FSM that is used to coordinate all of these units has three main functions: it starts the video controller, it clears the memories, and it starts the 3-D unit. The memories are cleared upon every reset and after every frame. This is important because when rotation was implemented onto the system, if the memories were not cleared, the newly rotated object would just be superimposed on the previous image. It implemented this through six states, with the majority of them being the time necessary

to clear the ZBTs. Otherwise, the FSM would "listen" for busy and done signals from the video and 3-D components in order to switch buffers and render the next frame.

Figure 2: State Transition Diagram for Major FSM



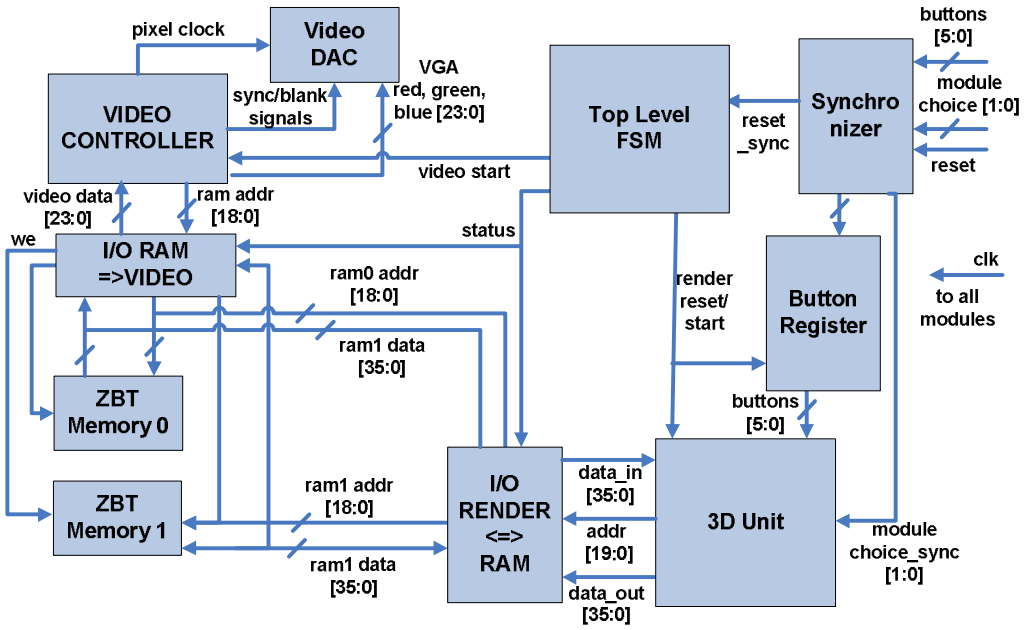


Figure 3: Block Diagram of Whole System

II. 3-D Component

The 3-D Component is responsible for creating the images that appear on the screen. At its top-most level, it takes in a start signal and the input from the user, and outputs data to the memory. The user controls two things: the model (which is chosen via input to the model selector) and the transformation matrix, which the user can manipulate through the push-buttons to rotate and zoom.

The two large sub-modules of the 3-D Component are the renderer and the shader. The renderer transforms the vertices of triangles in the model into coordinates, depth values, and colors on the screen. The shader "fills in" the triangles, computing depth and color for every pixel in the triangle (not just the vertices). The render unit and shader are pipelined—while the render unit is rendering one triangle, the shader can shade another.

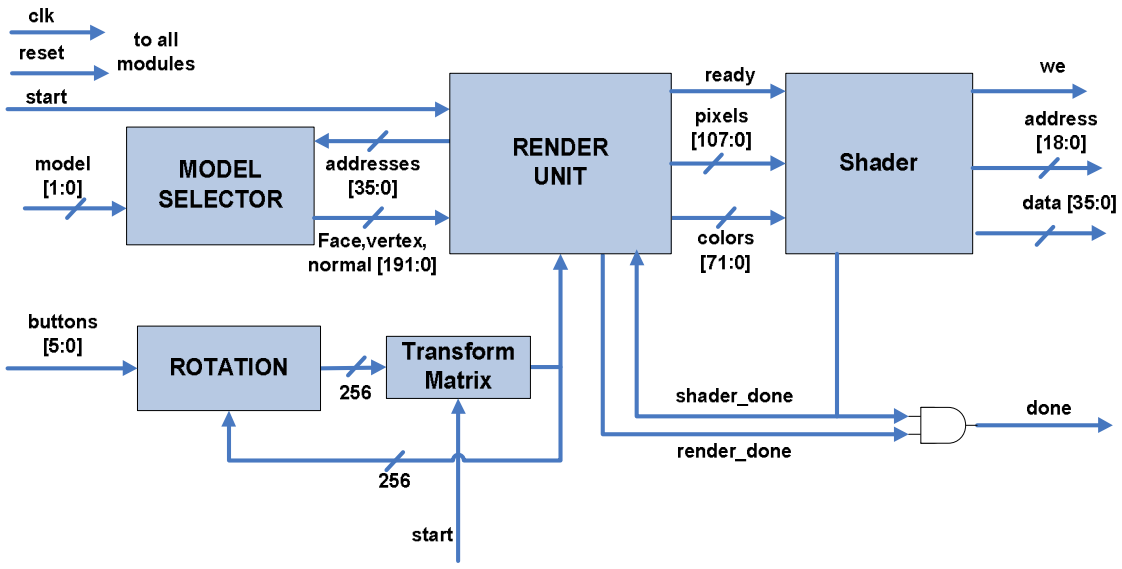


Figure 4: 3-D Unit Block Diagram

III. Rendering Unit

The rendering unit operates using two sub-modules, an FSM and the face renderer. The face renderer is a computational element with start and done signals. The FSM begins at the first triangle in the ROM, runs the face renderer on it, and the proceeds to the next triangle, and so on…, until it reaches the end of the model, at which point it outputs a "done" signal. After every triangle it rendered, the rendering unit outputs a "data ready" signal to notify the shader that a triangle is ready to be shaded.

The number of triangles is stored in the first element of the faces part of the model ROM, so the rendering unit reads that value before it begins the rendering cycle. It then initializes a counter, and counts until it has reached the last face of the model.

A block diagram and state transition diagram for the rendering unit are below.

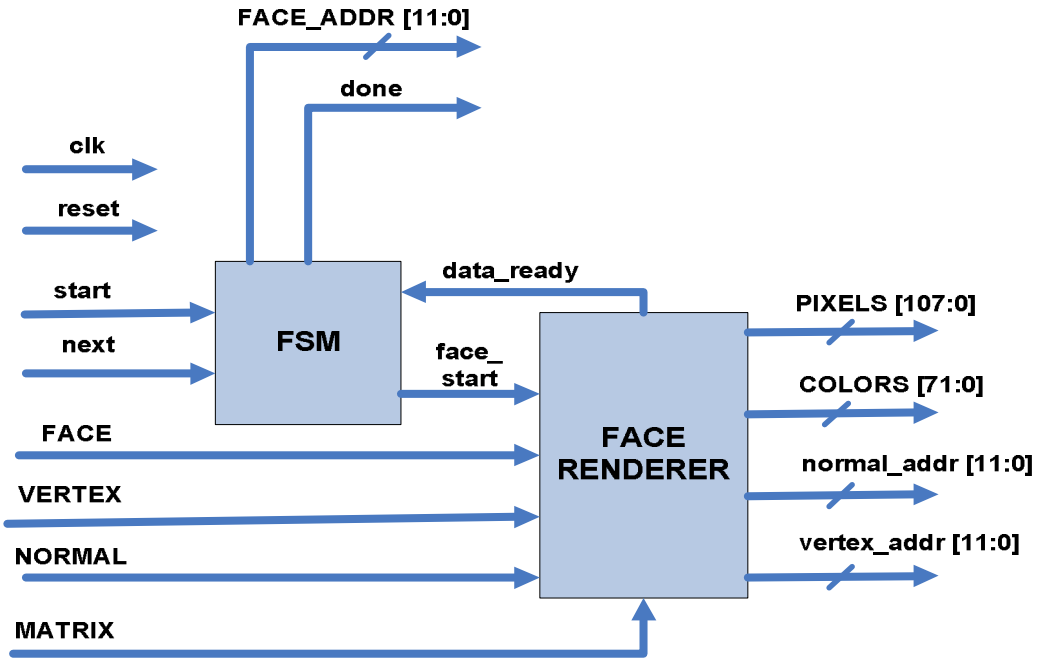


Figure 5: Rendering Unit Block Diagram

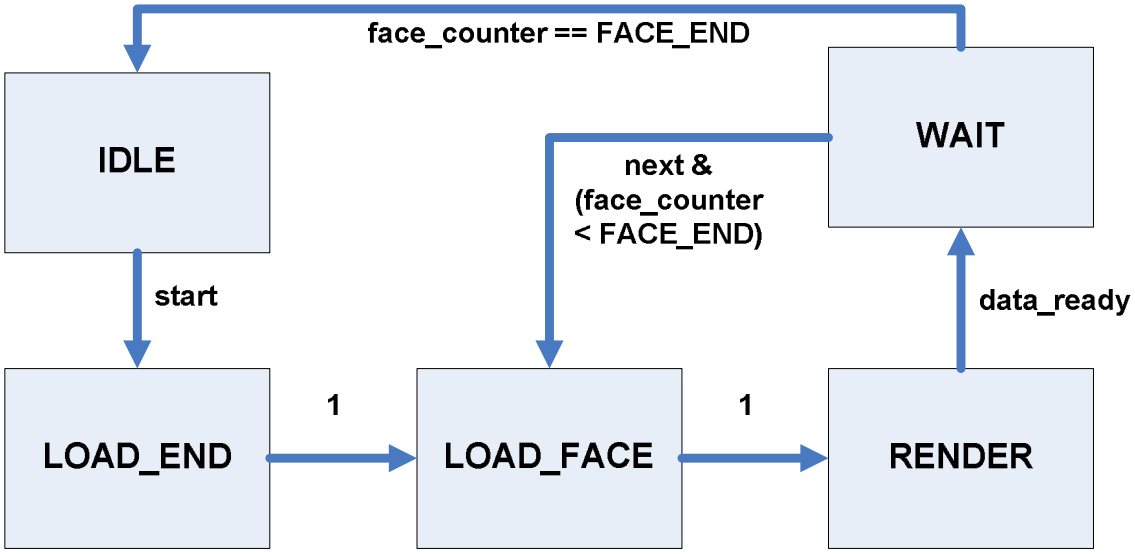


Figure 6: Rendering Unit FSM State Transition Diagram

IV. Face Renderer and Renderer

The face renderer unit consists of an FSM and two parallel computational modules. It selects each vertex from a triangle, and then starts the renderer and lighting

computational modules. When those modules are done, it packs the results into the pixels and colors registers; these registers are input to the shader module.

The lighting and rendering units work in a very similar fashion, so only one is diagrammed below. Both of them are sequential computations, where an FSM iterates through each computational step, and the start and done signals are used to interface with the face renderer module.

The lighting module implements Lambert diffuse lighting, where the intensity of the light at a point is the dot product of the normal vector at that point with the vector to the light source. The renderer implements two 4x4 matrix multipliers, a transform matrix that rotates and translates the model, and a perspective matrix that transforms 3-D coordinates to the 2-D screen.

Block diagrams and state transition diagrams for the renderer and face renderer modules are below.

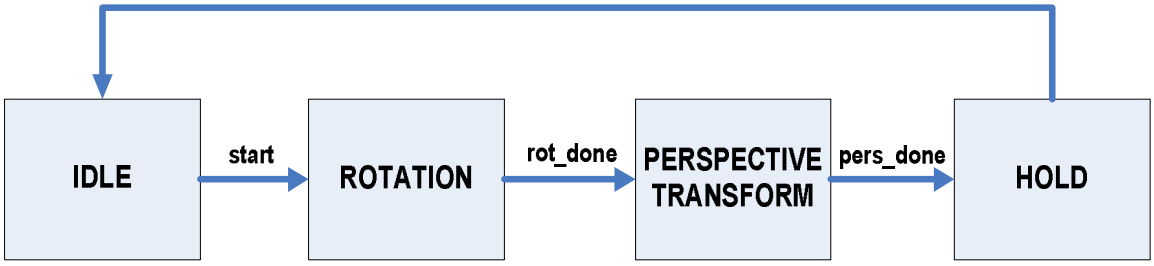


Figure 7: State Transition Diagram for Render Unit

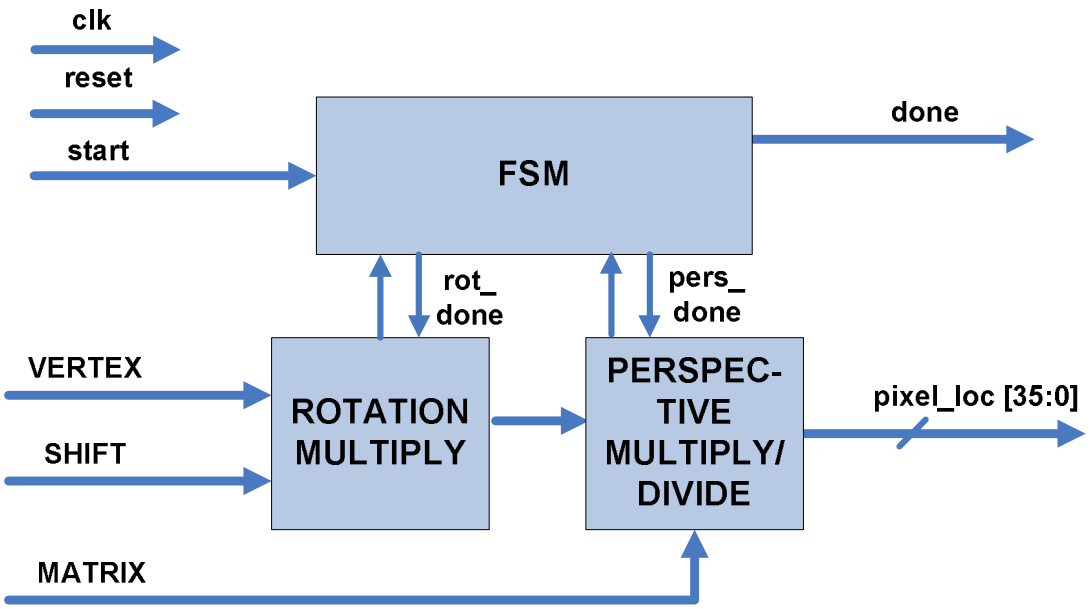


Figure 8: Block Diagram for Renderer Unit

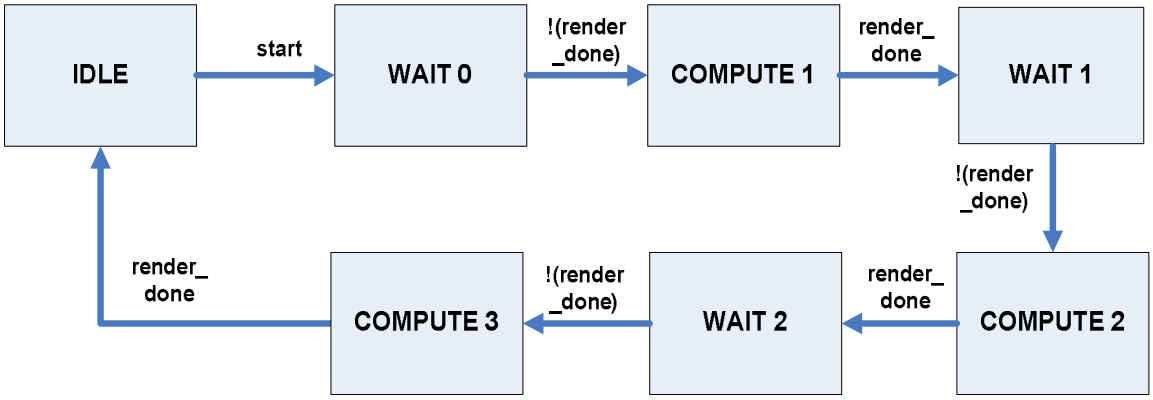


Figure 9: State Transition Diagram for Face Renderer

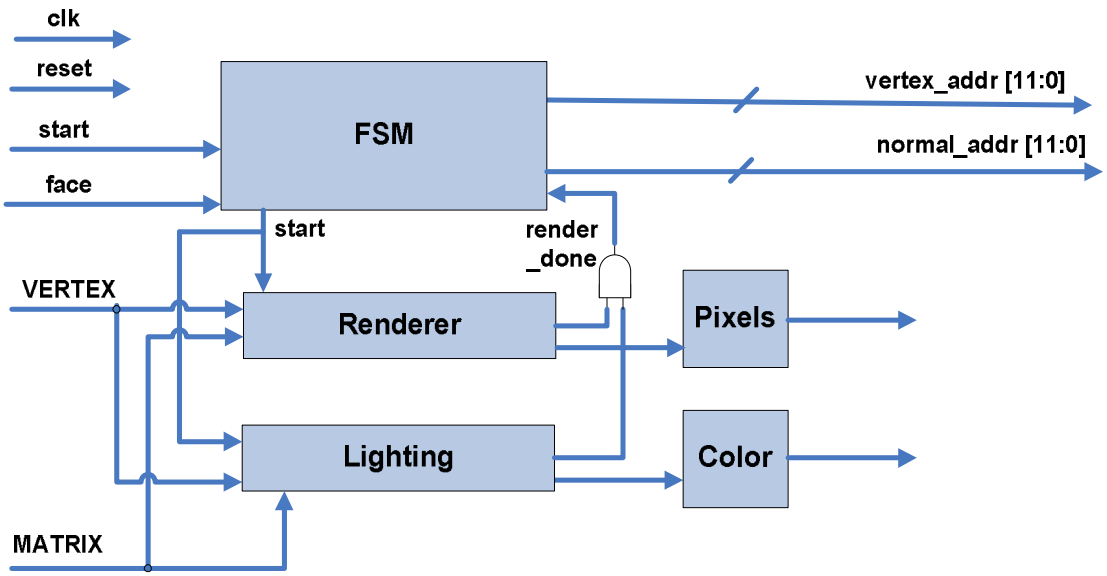


Figure 10: Block Diagram for Face Renderer

V. Shader Unit

The shader unit takes the pixel locations and color values of the vertices from the renderer, and writes all of the pixels in the triangle to memory. It implements Gouraud shading, a common interpolation scheme used in 3-D rendering. It also implements a Z-buffer, to ensure that only the front-most polygons are drawn to the screen.

The shader has two steps: computing the derivates and coefficients required for Gouraud shading, and scanning the boundary box of triangle to write each pixel. The computation of the coefficients is a serial computation in some respects similar to rendering. Once the coefficients are computed, the shader scans the boundary box of the triangle (from min(x), min(y) to max(x), max(y)). It uses the coefficients to determine of the pixel is inside the triangle, and if so to interpolate a color value and a depth value. The depth value is compared to any existing pixel data stored at that location, and the pixel is only written if its depth is smaller than the existing value.

The pixel pipe performs these interpolations and comparisons. Each pixel requires a read from memory and (possibly) a write to memory, if the color is being updated.

These operations are interleaved, so that the pixel pipe can process one pixel every two clock cycles.

VI. Model Format

For ease of implementation, the 3-D model format used in this system is the Alias Wavefront OBJ format, which is a very simple and powerful way to describe 3-D models. The most basic 3D shape, a tetrahedron (in a triangle-based system), is easily described as a set of geometric vertices and corresponding faces. Figure 3 shows a sample OBJ file that translates into a tetrahedron.

Figure 11: Simple 3-D Tetrahedron Model in OBJ Format

```
g tetrahedron

v 1.00 1.00 1.00
v 2.00 1.00 1.00
v 1.00 2.00 1.00
v 1.00 1.00 2.00

f 1 3 2
f 1 4 3
f 1 2 4
f 2 3 4
```

The data is contained in an easily understandable text format. The lines starting with a "v" specify a vertex and its x-, y-, and z-coordinates. The lines with "f" specify faces between the vertices (the vertices are indexed at 1 and increment downward) and that is all that is required for a simple 3-D image. This format allows for much more detail when using more complex models, as different parameters such as normal vertices, texture vertices, points, curves, color, etc. can be specified in this format. This allows for the sequential addition of features to our system so that we arrive at real results within the time constraints. The model data is going to be implemented as a ROM on the FPGA using a Perl script to create a Verilog file from the OBJ file.

A model switcher unit was implemented that basically instantiates different models as ROMs on the FPGA and then can switch through them using the input switches built onto the lab kit. To convert between the OBJ files to a COE file used by the Xilinx Coregen or a Verilog that can infer a ROM (through a case statement), a fairly extensive perl script was written. This script extracts the data from the OBJ file, computes the

normal vectors for every face or vertex, converts the formats, and generates the appropriate file. This script can be found at the end of the appendix.

VII. Video Component

The video component is responsible for reading the data in from the ZBT RAMs and outputting it to the Video DAC with the appropriate sync and blank signals. The goal of this portion of the project is to display the 3-D model at a high frame rate and resolution, in order to have a smooth picture on the monitor. To maximize throughput and minimize delays, the output from the rendering engine is first sent to a ZBT memory. Once a whole frame has been written to that memory, the video controller initializes and starts to read data from that memory, as described earlier and shown in Figure 1. Meanwhile, the rendering engine continues to process the model and outputs the frame to the second ZBT memory. The VGA video is at a resolution of 640x480 and runs at 60 Hz. This corresponds to a 25 MHz pixel clock. This was implemented in a fairly simple way that used a counter and customizable parameters (i.e. the values for HSYNC, VSYNC, etc.) that be easily changed to accommodate a different resolution or refresh rate.

This module was actually an extension of the design used in one of the problem sets, and had to be modified to read from memory as well. The task of reading from memory was simplified because a scheme whereby each line of a frame was able to have its own 1024 memory locations. This was convenient because it allowed for the pixel counter (used to generate the horizontal syncing and blanking signals) to be used as the lower order bits of the memory address and the line counter (used to generate the vertical sync and blanking signals) to be used as the higher order bits.

IIX. I/O Units

The I/O units that connect the RAMs to the 3-D unit and video component were a crucial part of the total system. These units used a status bit that controlled to which memory a bus from either of the units went to. It was hard-coded that the unit that connected the 3-D unit was always connected to the memory other than that which the video controller was hooked up to. This is simple in theory, but actually proved to be the cause of a lot of our initial problems. In our first simulations, we found that there were many sources of bus contention that we needed to track down before we could proceed to burn onto the

FPGA chip. As bus contentions can be potentially crippling to the FPGA, it was very important that we made sure these I/O units were working perfectly before implementing them onto the actual hardware.

Testing and Debugging

Our primary test mechanism for the project was ModelSim. Every sub-module had associated unit tests, and we used these unit tests to ensure that each module was operating properly. Additionally, we had a test bench for the entire labkit.v file (the outermost file in the project) to do integration testing.

Aside from ModelSim, we used the ChipScope utility to extract runtime data from the chip. This was useful for fixing bugs that were based on accumulated error—for example, at certain angles the shader did not work properly. To debug the problem, we used ChipScope to read the current transformation matrix, then built a testbench that used that matrix in ModelSim and found the problem.

Our major hurdles had to do with the memory. We made two significant errors: first, we did not realize that the byte-write-enable signals also had to be low to enable a write on the memory. Secondly, we connected the shader to the memory only for output (and not input). The second error was fixed by having separate data-in and data-out buses leading to the I/O render->ram unit. These two errors were very time-consuming to detect and fix, but aside from them the rest of our system was comparatively bug-free and issues only arose when integrating the final system.

Another headache in the testing and debugging process, although unavoidable, was the large amount of time necessary to synthesize and implement our project onto the FPGA. Due to the large number of gates used (around 40% of the slices), huge buses, and many multipliers (120 used out of approx 140 available), the Xilinx application definitely took a considerable amount of time to compile. Initially, this caused for a large amount of wasted time, but it really taught us the importance of simulating thoroughly before synthesizing. In the end, we feel that our system was designed and implemented more efficiently because we were forced to try and catch bugs early.

Conclusion & Last Thoughts

Overall, this project was an incredible learning experience for both of us. The process of specifying, designing, and implementing a large-scale project of this nature was definitely a challenge. The preceding labs definitely helped prepare us in terms of design methodology, debugging strategies, etc. but there are definitely issues that arise when designing such a complex project. The most important thing that we have taken away from this project is that unexpected things definitely happen in a complex system. When different components are put together, even if they seem to work fine independently, we found that emergent properties in the form of bus conflicts, timing glitches, etc. can arise. The debugging and testing that are involved in tracking down these errors was definitely the most time-consuming part, yet also the most satisfying when the working product appeared.

We had several design trade-offs that were necessary in order for the project to work in the time allotted. Our initial plan was to have a high resolution, high refresh rate image on the screen, but found that it was untenable in practice. Three-dimensional image rendering and processing is usually done by dedicated hardware that has been optimized and contains millions of carefully placed gates. We realized that same performance would not be attainable on our lab kits. Also, we were limited by the amount of memory we could initialize in ROMs on the chip; we had initially planned to use the CompactFlash interface to load up models, but that also proved not to be achievable in our time limits.

In the end, we had a great time working on the project and were definitely satisfied by the results. We would like to acknowledge our Teaching Assistant, Charlie, for his invaluable assistance and advice. We also want to thank Nathan Ickes for his expertise on the intricacies of how the new lab kits work, and we definitely pestered him a bit when it came to issues such as DCMs.