IMPLEMENTATION OF A 3D GRAPHICS RENDERING PIPELINE
USING A FIELD PROGRAMMABLE GATE ARRAY

A graduate project submitted in partial fulfillment of the requirements
for the degree of Master of Science
in Electrical Engineering

By

Vahe Jabagchourian

December 2011
The graduate project of Vahe Jabagchourian is approved:

________________________________________  ____________________________
Xiaojun (Ashley) Geng, Ph.D.  Date

________________________________________  ____________________________
Shahnam Mirzaei, Ph.D.  Date

________________________________________  ____________________________
Ramin Roosta, Ph.D.  Date

________________________________________  ____________________________
Ronald W. Mehler, Ph.D., Chair  Date
Acknowledgements
# Table of Contents

Signature Page................................................................................................................................................. ii  
Acknowledgements ....................................................................................................................................... iii  
List of Figures ................................................................................................................................................ vi  
List of Tables............................................................................................................................................... viii  
Abstract ......................................................................................................................................................... ix  

Chapter 1: Introduction ................................................................................................................................... 1  
  1.1: Fundamentals of GPUs ..................................................................................................................... 2  
  1.2: Components of Graphics Rendering Systems ................................................................................... 2  
  1.3: Processing Steps of the Graphics Rendering Pipeline ...................................................................... 3  

Chapter 2: Project Specifications .................................................................................................................... 5  
  2.1: Overview ........................................................................................................................................... 5  
  2.2: Input Format ..................................................................................................................................... 5  
  2.3: Dataset Details .................................................................................................................................. 6  
  2.4: Coordinate Format ............................................................................................................................ 6  
  2.5: Selection of a Hardware Platform ..................................................................................................... 7  
  2.6: Project Goals ..................................................................................................................................... 9  
  2.7: Document Organization ........................................................................................................................ 9  

Chapter 3: Functional Units .......................................................................................................................... 10  
  3.1: Overview......................................................................................................................................... 10  
  3.2: Trackball Movement Module .......................................................................................................... 11  
    3.2.1: Mouse Protocol..................................................................................................................... 12  
    3.2.2: Implementation ..................................................................................................................... 13  
    3.2.3: Verification ........................................................................................................................... 15  
  3.3: Vertex Fetcher ......................................................................................................................................... 21  
  3.4: Coordinate Rotator .......................................................................................................................... 24  
    3.4.1: Mathematics of Rotation......................................................................................................... 24  
    3.4.2: Center Of Rotation ................................................................................................................. 26  
    3.4.3: Fixed Point Arithmetic .......................................................................................................... 27  
    3.4.4: Verification ........................................................................................................................... 27  
  3.5: Projector and Viewport Mapper ...................................................................................................... 30  
    3.5.1: Overview ................................................................................................................................... 30  
    3.5.2: Mathematical Foundation ....................................................................................................... 31  
    3.5.3: Implementation ..................................................................................................................... 33  
    3.5.4: Design Challenges ................................................................................................................ 35  
  3.6: Line Drawer Module ....................................................................................................................... 38
3.6.1: State Machine .......................................................................................................................39
3.7: Frame Buffer ..........................................................................................................................44
3.8: Video Generator .....................................................................................................................45
Chapter 4: Results .........................................................................................................................48
Chapter 5: Conclusion ...................................................................................................................51
Chapter 6: Future Work ..................................................................................................................52
References ..........................................................................................................................................53
Appendix A: GPU Source Code ......................................................................................................55
Appendix B: Modelsim Virtual Signal Commands ........................................................................66
Appendix C: Fixed Point Generation Script ...................................................................................67
List of Figures

Figure 1: Top level block diagram of the Graphics Rending System ............................................................. 1
Figure 2: Five Components of a Graphics Renderer [1] ................................................................................ 3
Figure 3: GPU Functional Unit Breakdown [2] ............................................................................................. 4
Figure 4: OFF Dataset Syntax and Teapot Dataset Snippet .......................................................................... 6
Figure 5: 13-Bit X and Y coordinates (Fraction 1.99 in Sign Magnitude Fixed Point) ....................................... 7
Figure 6: 13-Bit Depth (Z) Coordinate (Fraction 1.99 in Unsigned Magnitude Fixed Point) ............................. 7
Figure 7: System Level Block Diagram of 3D Graphics Rendering Pipeline .................................................. 10
Figure 8: Top Level Schematic of the Original PS/2 Tx/Rx Unit Shown as Separate Blocks .......................... 11
Figure 9: Device to Host Communication (Data Bit Read on Rising Edge of Clock) [13] ............................. 12
Figure 10: Host to Device Communication (Data Bit Read on Falling Edge of Clock) [13] ............................ 12
Figure 11: PS/2 Communication Protocol [14] ............................................................................................. 12
Figure 12: Depiction of 4 byte packets transmitted from mouse to host FPGA [14] ....................................... 14
Figure 13: Simplified Mouse Initialization/Streaming Controller State Machine ........................................ 15
Figure 14: LED arrangement on the XUPV5-LX110T for debugging direction and position .......................... 16
Figure 15: User Constraints File for PS/2 Module ......................................................................................... 16
Figure 16: Mouse Initialization Behavioral Simulation .................................................................................. 17
Figure 17: Mouse Initialization (Part 2) Behavioral Simulation ...................................................................... 18
Figure 18: Streaming Packets from PS2 Mouse (Y Movement) ..................................................................... 19
Figure 19: Streaming Packets from PS2 Mouse (X Movement) ..................................................................... 20
Figure 20: Vertex Fetcher Controller Block Diagram (Inputs on Left, Outputs on Right) ................................. 21
Figure 21: Original Static Pyramid – Shape Generator State Machine without clear states ............................ 22
Figure 22: Original Static Four Line Shape Generator State Machine with Clear States ............................... 22
Figure 23: Vertex Fetcher State Machine ..................................................................................................... 23
Figure 24: Coordinate Rotator Block Diagram ............................................................................................ 24
Figure 25: Summary of rotation options for trackball input device .............................................................. 26
Figure 26: Behavioral Simulation (Analog Step) of Sin/Cos ROM ............................................................... 28
Figure 27: Behavioral Simulation (Analog Step) of Rotator Multipliers ......................................................... 29
Figure 28: One point perspective convergence [19] ..................................................................................... 30
Figure 29: Viewport Transformation and Scaling [20] .................................................................................. 30
Figure 30: 3D Pyramid alongside its projected representation ....................................................................... 33
Figure 31: Initial Integration of Coordinate Generator, Viewport Projector and Rasterizer ......................... 34
Figure 32: Projector / Viewport Mapper Block ............................................................................................ 34
Figure 33: Internal Blocks of the Coordinate Projector and Viewport Mapper ............................................. 34
Figure 34: High Speed Division by Reciprocal Multiplication [23] ............................................................... 35
Figure 35: Perspective Divider (using Reciprocal Lookup Multiplication Method) ........................................ 35
Figure 36: Behavioral Simulation of Fetcher/Projector with Several Line Vertices Generated ................. 36
Figure 37: Behavioral Simulation of Fetcher/Projector with a single line endpoint shown ...................... 37
Figure 38: Depiction of a Rasterized Line with a positive slope [24] .......................................................... 38
Figure 39: Bresenam Rasterizer pseudo code [25] ...................................................................................... 39
Figure 40: State Machine for Line Generator Block .................................................................................. 40
Figure 41: Top Level Block Diagram of Line Generator ............................................................................ 41
Figure 42: Implementation tasks for the Static Line Generator ................................................................. 41
Figure 43: Implementation tasks for the Dynamic Line Generator ............................................................. 42
Figure 44: Results From Cursor Position Based Line Generator .............................................................. 42
Figure 45: Behavioral Simulation of Single Line Generator ................................................................. 43
Figure 46: SRAM Interface versus DPRAM Module .................................................................................. 44
Figure 47: I2C Protocol [26] ....................................................................................................................... 45
Figure 48: Video Timing Diagram ............................................................................................................. 46
Figure 49: Video Generator ...................................................................................................................... 46
Figure 50: Pixel Multiplexer (Input to Chrontel 7301C) .......................................................................... 47
Figure 51: Orthographic (parallel) projected teapot .............................................................................. 48
Figure 52: Perspective (convergent) projected teapot ............................................................................... 48
List of Tables

Table 1: Hardware Requirements Compliance Matrix ................................................................................... 8
Table 2: GPU System Level Breakdown ......................................................................................................... 10
Table 3: Trackball Controller States and Transmission Values [14] ............................................................. 13
Table 4: Depiction of when to add 2.5 to Z (Refer to Figure 24 for red and green signal locations) .............. 26
Table 5: Summary of rotation options for trackball input device [18] ............................................................. 27
Table 6: Summary of parameters for viewport transformation ..................................................................... 32
Table 7: Line Generator State Description ................................................................................................... 40
Table 8: Video Parameters ............................................................................................................................ 45
Table 9: Summary of Object Parameters .................................................................................................... 48
Table 10: Application of rotation transformation with various angles ......................................................... 49
Table 11: Rotation across both x and y ......................................................................................................... 50
The objective of this project was to produce a working design of a 3D graphics rendering pipeline using an FPGA (Field Programmable Gate Array) and to develop proficiency in the development of a digital design using Verilog Hardware Description Language. The FPGA is loaded with the bitstream of the GPU (Graphics Processing Unit) which produces an object on an LCD monitor. The final deliverable is Verilog source code describing the GPU which has been implemented on a Virtex-5 LX110T FPGA development board. The input to the FPGA is a bidirectional serial PS/2 signal which passes data back and forth between FPGA and trackball. The output from the FPGA is a digital video signal which displays a projected object on a monitor whose orientation can be changed when a new dataset configuration is loaded. Finally, the design provides a method to load in new datasets and initialize on-chip Block RAM content with the datasets. The primary functional units of the 3D rendering pipeline include the vertex fetcher unit, rotation unit, mouse movement unit, projection unit, perspective division unit, viewport mapping unit, line drawing unit and video generation unit.
Chapter 1: Introduction

Computer graphics describes the process by which points are converted into pixels. Specialized hardware called Graphical Processing Units (GPU) convert 3D geometric objects into a 2D representation that is displayed on a monitor. GPUs are manufactured on Application Specific Integrated Circuits (ASICs) commercially and on Field-Programmable Gate Arrays (FPGAs) for rapid-prototyping and hardware emulation. FPGAs are hardware chips that contain specialized blocks of logic, and specialized hardware components for performing virtually any type of logic function. The advantage of FPGAs over ASICs is that FPGAs can be re-programmed many times which makes them customizable and well suited for academic and research projects. Since FPGAs are reprogrammable and have a simpler design cycle than ASICs, they are used in rapid prototyping applications. Figure 1 shows the FPGA development board (XUPV5-LX110T) that has been selected for this project. It contains the Xilinx Virtex-5 FPGA, which is used in high speed embedded applications. It can be configured to run a standalone digital design that connects to on board peripherals or run a hardware/software co-design with C code alongside an embedded (on-chip) processor.

![Digital Monitor][6] ![Xilinx Virtex-5 FPGA Board][7] ![Trackball Mouse][8]

Figure 1: Top level block diagram of the Graphics Rending System

Information about the board provided from www.xilinx.com includes the following common applications for the XUPV5-LX110T FPGA

1. Digital Design
2. Embedded Systems
3. Digital Signal Processing and Communications
4. Computer Architecture
5. Operating Systems
6. Networking
7. Video and Image Processing
8. High Speed Serial I/O Transceivers

---

1.1: **Fundamentals of GPUs**

A GPU is a commonly used acronym for Graphics (or Graphical) Processing Unit. GPUs are specialized primarily used for rendering and visual applications such as medical visualization or high performance computing. A special type of GPU called GPGPU (General Purpose GPU) is used to refer to Graphical Processing Units that can perform a non-specific function compared to a classical GPU. The focus of this project is to develop a prototype of a classical single core graphical processor for display applications.

Computer graphics is the process of displaying information. Displaying information means representing 3D information on a 2D space such as a computer monitor. The primary input to a computer graphics system is a set of coordinates. The main use of a computer graphics system can be seen in applications such as display systems, design of interactive handheld devices, smart phones and human computer interaction systems. In each of these applications the user of the system needs to see a visual representation of the data to facilitate the interaction process. In medical imaging systems, for example, the medical technician is working to interpret a 3D graphical representation of a dataset. The visualization system is producing high quality images at a high speed using modern graphics rendering hardware. In other applications such as general purpose computing, or simulation, the end user is using the visual display as a guide to be able to control and see what is going on with his/her input commands. When pilots train for flight they often rely upon virtual reality systems to help simulate conditions that are present in the real flight. These virtual reality systems, computer platforms and medical systems represent a sample of the applications seen in computer graphics. The goal of any graphics system is to convert points to pixels which provide a visual way of interpreting large amounts of numerical data.

Graphical Processing Units that output graphical data operate on large amounts of vertex points which are sent off to a monitor or host processor for display or processing. GPUs differ from CPUs in that they are inherently designed to operate on large amounts of data in parallel. GPUs can achieve greater speed than CPUs because they are designed with great amount of functional units and dedicated memory. Because of the sheer number of parallel processing units on GPUs as compared to CPUs there is a great performance disparity when running instructions on GPUs that are programmable rather than conventional CPUs.

1.2: **Components of Graphics Rendering Systems**

Computer graphics renderers display information. The hardware that makes a graphics system work is similar to the five components of a computer (Memory, Input, Output, ALU and the CPU). The components of a graphics system are depicted in Figure 2. The input device can be a mouse, trackball,
sensor, or even a camera or any device that acquires information. The memory block is used to store the vertices of the object. The GPU is the focus of the project and it consists of functional units which alter the points passed through the pipeline. The frame buffer receives the data from the GPU and sends it to an output device. An output device can be a monitor, for example.

![Five Components of a Graphics Renderer](image)

**Figure 2: Five Components of a Graphics Renderer [1]**

1.3: **Processing Steps of the Graphics Rendering Pipeline**

A classical rendering pipeline begins with the acquisition of data points (vertices) and ends with the production of pixels on a screen. The internal components of the pipeline are what constitute the GPU. The first stage in the classical rendering pipeline is the transformer. The transformer receives a set of vertex coordinates and performs mathematical transformations such as translation, rotation or scaling. There are other transformations including shearing that alter the original shape of the object. A transformation matrix which contains the information for transforming input coordinates to output coordinates is the basis for many of the hardware blocks in the graphics system.

After the object is modeled (translated, rotated, scaled) it is then viewed which means that the object changes representation from world coordinate system to the eye or camera coordinate system. This usually involves projecting the object onto a 2D space where the object undergoes further manipulations. Transformation, which consists of modeling and viewing, embodies the first of the four transformations in the classical rendering pipeline. The next step is called the clipping stage. A clipper trims portions of the object which are not in the viewing space (the 2D plane of projection). There are many algorithms to accomplish the clipping process and they usually consist of a pipeline which performs a left trim, a right trim, and finally a top and bottom trim. At each trim stage the coordinates of intersection are calculated...
and then used to reshape the object to have edges that are the same as the bounding edges. After an object is clipped, it is projected onto the device space that serve as the viewing space for the object.

There are different types of projections and they include Orthographic Projections and Perspective Projections. The Orthographic Projection consists of parallel projection whose center of projection is at infinity. Perspective projections have their center of projection at a finite distance from the viewing plane. It is easier to perform orthographic projection because there is no need to perform perspective correction. The projection matrix for an orthographic matrix converts the depth (z coordinate) value to zero and preserves the x and y coordinates of the shape. After projected onto the projection space the rasterizer performs the main drawing function of the renderer. Figure 3 breaks down the functional units of the graphics rendering pipeline.

Four processing steps of a graphics renderer include:

1. Application – Reads memory of object vertices and sends to transformation unit
2. Geometry – Performs geometric transformations on object and maps to screen space
3. Rasterization – Calculates line shape pixels, fills object with a solid or shaded color
4. Display – Reads from frame buffer generates video synchronization and multiplexed RGB data

<table>
<thead>
<tr>
<th>Application</th>
<th>Geometry</th>
<th>Rasterization</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read in 3D Data</td>
<td>Geometric Transform</td>
<td>Clipping/Scissoring</td>
<td>Read Frame Buffer</td>
</tr>
<tr>
<td>Write to Registers</td>
<td>Perspective Division</td>
<td>Line Generation</td>
<td>Generate HV Sync</td>
</tr>
<tr>
<td>Stream to Geometry</td>
<td>Screen Mapping</td>
<td>Store in Frame Buffer</td>
<td>Multiplex RGB Data</td>
</tr>
</tbody>
</table>

Figure 3: GPU Functional Unit Breakdown [2]
Chapter 2: Project Specifications

2.1: Overview

The objective of this project was to produce a GPU design in Verilog and implement the design on an FPGA. The GPU accepts a 3D dataset at compile time (during behavioral elaboration) and outputs a projected image on the monitor. The first requirement is to select an appropriate FPGA development board with sufficient resources. The second requirement is to provide capability to easily import in different dataset (binary valued lookup tables using a real to fixed point converter function). In addition, the design supports the storage of a relatively complex dataset. It also renders the object on a digital monitor. Finally, the object is able to be rotated using compile time rotation equations which show the rotated object on the monitor. Use of Verilog Hardware Description Language was the main component of the project. In addition, use of binary conversion scripts in Verilog and other scripting languages have aided in producing a quick memory initialization file for testing the design on the FPGA. Tools used in the project include Matlab, Microsoft Excel, Xilinx synthesis, design entry and simulation tools to verify and implement the design on the Virtex-5 board.

Summary of Requirements

1. Utilize an FPGA with sufficient resources to run the GPU design
2. Employ chip Block Rams as a Frame Buffer for Wireframe rendering
3. Perform computations using pre-computed fixed-point lookup tables
4. Execute rotation transformation with ability to modify functionality for transformation
5. Provide for addition of extra modules such as double-buffering to reduce flickering
6. Allow entry of arbitrary dataset during design and implementation

2.2: Input Format

 Originally, a simple 4 sided pyramid of triangles was hard coded into a state machine and it consisted of four states. The state machine was a four vertex pyramid. The state machine proved to be a quick way of generating the object. Though this method is quick to write, it was later improved and even scaled by segregating the data from the logic. Eventually, a reasonably sized dataset was located and a vertex fetching state machine was created and two blocks of memory (object vertices and triangle vertex coordinates) was created to provide an easy way to change the object. Verilog and especially Xilinx XST (synthesis tool) allows the initialization of Dual Port Block Rams using memory (mem) files for FPGA applications. The dataset used for testing the GPU consists of two files (“vertices.mem” and “triangles.mem”) created from an OFF dataset. The original memory file was an object file format based dataset which is used in a UNIX graphical rendering program called GeomView.
2.3: Dataset Details

Object File Format (OFF for short) is a 3D data format that specifies the x, y, and z coordinates of an object and it consists of the file extension followed by the number of vertices, triangles, lines. The data that follows the number of vertices and triangles followed by vertex coordinates and vertex indices. The file representing the object does not need the off keyword and it has been removed from the file. Figure 4 shows the OFF syntax and a sample of the data file at the second half of the figure. The sample data shown below corresponds to a single teapot object used for the test case for the graphics pipeline design.

```
OFF
NUM_VERTICES  NUM_TRIANGLES  NUM_LINES  #size
X1  Y1  Z1    #coordinates
   .
   .
XN  YN  ZN    #last coordinate
V1  V2  V3 [V4]    #face (3, 4 Vertices)
   .
   .
```

```
OFF
480  448  926
0.00000000  0.00000000  0.488037
0.00390625  0.0421881  0.476326
0.00390625 -0.0421881  0.476326
0.01074220  0.00000000  0.575333
0.01250000  0.0562508  0.450561
0.01250000 -0.0562508  0.450561
0.01953120  0.00000000  0.413654
0.02109380  0.0421881  0.424797
0.02109380 -0.0421881  0.424797
0.02500000  0.00000000  0.413086
0.03875000  0.1962500  0.488037
0.03875000 -0.1962500  0.488037
```

Figure 4: OFF Dataset Syntax and Teapot Dataset2 Snippet

2.4: Coordinate Format

Before displaying objects on a monitor it is necessary to represent the object vertices with fractional vertices (endpoints) in either fixed or floating point format using three coordinates (x, y, z). The chosen format for the implementation was fixed point because it simplified the vertex representation and hardware construction of the transformation, projection and viewport mapping modules. Figures 5 and 6 depict the allocation of bits for the x coordinate, y coordinate and the depth (z) coordinate. Note that the depth (z) coordinate assumes only positive values (bits 1 and 0) while the x and y coordinates are signed.

---

2 Teapot dataset is modified from http://www.holmes3d.net/graphics/teapot/teapot.off
2.5: Selection of a Hardware Platform

There were several factors that were considered when selecting an appropriate FPGA board. Memory and clock speed were primary factors to consider. Because of previous experience with Xilinx based tools in past projects, Xilinx boards were used in the comparison. A total of four boards were compared and the following measures were used to evaluate capabilities of the boards. The important measures used to evaluate the boards were memory capacity, mouse interface support, and digital video output support. The first measure used to determine a suitable FPGA board was memory capacity. Sufficient memory was needed for the frame buffer, vertex memory and pipeline components.

A comparison matrix was made between four Xilinx FPGA boards. The four FPGAs were the Virtex-5 (XCV5LX110T), the Virtex-4 (XC4VFX12), the Spartan 3E 1600, and the Virtex-II Pro (XC2VP30). The V5LX110T had a DDR2 (Double Data Rate 2) SDRAM along with an included 1GB compact flash card. The compact flash can be used to store an ASCII text file consisting of 3D vertices in an OFF format. In addition to having support for extended memory, the Virtex-5 FPGA also had 9.4 Megabits of SRAM which can also be used to store one frame of video.

In comparison to the Virtex-5 board, the remaining three boards either had no support for PS/2 or had less memory and hardware capacity for supporting the 3D graphics rendering pipeline. After comparing all three boards for basic requirements the decision was made to select the Virtex-5 LX110T [3] board because of its resource availability and overall capabilities. Table 1 shows the hardware comparison matrix used to select the board.

---

Table 1: Hardware Comparison Matrix

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 5: 13-Bit X and Y coordinates (Fraction 1.99 in Sign Magnitude Fixed Point)

Figure 6: 13-Bit Depth (Z) Coordinate (Fraction 1.99 in Unsigned Magnitude Fixed Point)

---

3 S is for Sign bit, M is for Magnitude bit(s), F is for Fractional Bits
<table>
<thead>
<tr>
<th>Attributes</th>
<th>Requirement</th>
<th>Virtex-5</th>
<th>Virtex-4</th>
<th>Spartan-3E</th>
<th>Virtex-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>XC5VLX110T</td>
<td>XC4VFX12</td>
<td>S3E-1600</td>
<td>XC2VP30</td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>[8]</td>
<td>[9]</td>
<td>[10]</td>
<td>[11]</td>
<td></td>
</tr>
<tr>
<td>Mem Type</td>
<td>DDR2</td>
<td>DDR</td>
<td>DDR</td>
<td>DDR</td>
<td></td>
</tr>
<tr>
<td>RAM Size</td>
<td>RAM Type</td>
<td>64-bit wide</td>
<td>32 MB DDR SDRAM</td>
<td>6MB SDRAM</td>
<td>Unsure</td>
</tr>
<tr>
<td>RAM Size</td>
<td>256Mbyte</td>
<td>DDR</td>
<td>DDR</td>
<td>DDR</td>
<td></td>
</tr>
<tr>
<td>Flash</td>
<td>512MB</td>
<td>1GB Included</td>
<td>4 MB FLASH</td>
<td>Not Clear</td>
<td>Unsure</td>
</tr>
<tr>
<td>PS2 Interfaces</td>
<td>One</td>
<td>Two PS2</td>
<td>None</td>
<td>One PS2 Interface</td>
<td>One PS2 Interface</td>
</tr>
<tr>
<td>Mouse Interface</td>
<td>One</td>
<td>Keyboard, Mouse</td>
<td>No PS2</td>
<td>PS/2 Keyboard</td>
<td>PS/2</td>
</tr>
<tr>
<td>Slices</td>
<td>Not Specified</td>
<td>17,280</td>
<td>5472</td>
<td>14,752</td>
<td>13696</td>
</tr>
<tr>
<td>BRAM</td>
<td>2Mbits</td>
<td>5.328Mbits</td>
<td>0.648Mbits</td>
<td>0.648Mbits</td>
<td>2.448Mbits</td>
</tr>
<tr>
<td>Logic Cells</td>
<td>Not specified</td>
<td>Not Sure</td>
<td>12312</td>
<td>33,192</td>
<td>30,816</td>
</tr>
<tr>
<td>DSP48E</td>
<td>32 Blocks</td>
<td>64 DSP Blocks</td>
<td>32 DSP Blocks</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Multipliers</td>
<td>32 Blocks</td>
<td>None</td>
<td>None</td>
<td>36</td>
<td>136</td>
</tr>
<tr>
<td>Multiplier Width</td>
<td>18 X 18</td>
<td>25bit X 18bit</td>
<td>18bit X 18bit</td>
<td>18x18</td>
<td>18x18</td>
</tr>
<tr>
<td>Heat sync</td>
<td>Possibly Needed</td>
<td>Extra</td>
<td>None Indicated</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Core Clock</td>
<td>300MHz</td>
<td>550 MHz</td>
<td>500 MHz</td>
<td>300 MHz</td>
<td>100MHz</td>
</tr>
<tr>
<td>On-Board LCD</td>
<td>Not Crucial</td>
<td>16X2 LCD</td>
<td>128 x 64 Display</td>
<td>Optional LCD</td>
<td>Unsure</td>
</tr>
<tr>
<td>Price of Board</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Price</td>
<td>$750</td>
<td>$350</td>
<td>$225</td>
<td>$299</td>
<td></td>
</tr>
<tr>
<td>Cable Price</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Price of Cable</td>
<td>Included</td>
<td>$225</td>
<td>Unsure</td>
<td>Unsure</td>
<td>Unsure</td>
</tr>
<tr>
<td>Total Price</td>
<td>Based on Features</td>
<td>$750</td>
<td>$575</td>
<td>$225</td>
<td>$299</td>
</tr>
</tbody>
</table>

Table 1: Hardware Requirements Compliance Matrix
2.6: Project Goals

The first goal of this project was to improve proficiency developing a modular digital system using Verilog and work with Xilinx tools to produce a digital design (GPU). The second goal was to develop an understanding of the function of the blocks for a graphics rendering pipeline. In addition to understanding the graphics rendering pipeline, the ultimate objective was to actually produce the design in Verilog while gaining an understanding of the process of prototyping digital designs on an FPGA. Verification of the design was an integral part of the project.

2.7: Document Organization

The rest of the document details the implementation and verification of each of the blocks of the graphics rendering pipeline starting with the vertex fetcher, trackball movement module, object rotation module, projection and viewport mapping module, line drawer module, pixel reader, and frame buffer as well as the video generator. The peripherals include the PS/2 trackball and the video initialization module, which is directly on the board. Finally, output results are presented, followed by the conclusion, and appendix.
Chapter 3: Functional Units

3.1: Overview

This section describes the main functional units of the graphics rendering pipeline. Figure 7 and Table 2 provide more detail. The mouse movement generator is the first module that receives the input from the user and sends mouse displacement information to the coordinate rotator. Simultaneously, the Vertex Fetcher fetches the object vertices from the pre-initialized block rams and sends them off to be rotated in the rotation module. The projection module takes the 3D coordinates and converts them into 2D by feeding the x and y parts into the perspective correction using the reciprocals generated from the reciprocal ROM. The line generator, frame buffer, pixel reader and video generator are the final blocks.

![3D Graphics Rendering Pipeline Core – XCV5-LX110T FPGA](image)

Figure 7: System Level Block Diagram of 3D Graphics Rendering Pipeline

<table>
<thead>
<tr>
<th>Unit</th>
<th>Short Name</th>
<th>Speed(MHz)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Mouse Mover</td>
<td>100/25</td>
<td>Generates axis of movement and displacement value</td>
</tr>
<tr>
<td>B</td>
<td>Rotator</td>
<td>100</td>
<td>Performs rotation multiplication and addition/subtraction</td>
</tr>
<tr>
<td>C</td>
<td>Triangle RAM</td>
<td>100</td>
<td>Stores the object vertices (x,y,z) and the triangle obj. addr</td>
</tr>
<tr>
<td>D</td>
<td>Vertex RAM</td>
<td>100/100</td>
<td>Fetches triangle coordinate address and extracts (x,y,z)</td>
</tr>
<tr>
<td>E</td>
<td>Projector</td>
<td>100</td>
<td>Maps 3D coordinate data to a 2D space by collapsing Z</td>
</tr>
<tr>
<td>F</td>
<td>Reciprocal ROM</td>
<td>100</td>
<td>Stores a fixed point binary reciprocal table [1.00:3.99]</td>
</tr>
<tr>
<td>G</td>
<td>Divider</td>
<td>100</td>
<td>Converts mouse data into x,y,z positional values</td>
</tr>
<tr>
<td>H</td>
<td>Viewport</td>
<td>100</td>
<td>Translates world to screen space</td>
</tr>
<tr>
<td>I</td>
<td>Line Generator</td>
<td>100</td>
<td>Draws a line between start/end values (x,y)</td>
</tr>
<tr>
<td>J</td>
<td>Frame Buffer</td>
<td>100/100</td>
<td>Stores a single frame of data (1024X1024)</td>
</tr>
<tr>
<td>K</td>
<td>Pixel Reader</td>
<td>100</td>
<td>Reads the frame buffer contents and outputs to video gen</td>
</tr>
<tr>
<td>L</td>
<td>DVI Video Gen</td>
<td>100/200</td>
<td>Sends data and synchronization signals to DVI Tx</td>
</tr>
</tbody>
</table>

Table 2: GPU System Level Breakdown
3.2: Trackball Movement Module

The implementation of the GPU began with the development and verification of the input interaction block. The user interaction block is a PS/2 compatible scroll wheel mouse controller which was modified slightly from Dr. Pong Chu’s Verilog example source as described in his text “FPGA prototyping by Verilog examples Xilinx Spartan -3 version” [12]. The PS/2 controller core was designed using three state machines. The first state machine describes the current state of the mouse packet transmission and reception to/from the mouse. The main module containing the described state machine configuration (packet controller created from scratch and the serial transmitter, and serial receiver used with slight modification from Dr. Pong Chu’s example) has been consolidated and integrated with the packet controller. The bit level state machines have been interconnected together structurally. They are depicted as two separate modules in Figure 8. In the final design there was a problem using them separately because of feedback errors in synthesis and as a result they were consolidated into one module to allow for design logic partitioning during implementation.

![Figure 8: Top Level Schematic of the Original PS/2 Tx/Rx Unit Shown as Separate Blocks](image-url)
3.2.1: Mouse Protocol

The mouse or trackball needed to be initialized before it can begin transmitting mouse movement to the FPGA. In order to achieve successful initialization a top level controller has been created which is capable of running the low level serial transmit receive state machines and allow them to send data back and forth between the host FPGA and the mouse peripheral. To understand the PS/2 protocol it is important to review the main steps involved in transmission or reception of data. The PS/2 is a serial bidirectional protocol used to interface keyboards and mice to a host processor. The FPGA in these waveforms is the device which has control over the data line and must send a request to initialize the mouse for receiving displacement information. To request control of the data line the host pulls clock low and then data low and then releases clock. The PS/2 mouse oscillates the clock in order to transmit or receive packets of data. The protocol is depicted in Figure 9 and Figure 10.

1. Host brings the clock line low for at least 100 microseconds
2. Bring the data line low
3. Release the clock line
4. Wait for the device to bring the clock line low
5. Set/reset the data line to send the first data bit
6. Wait for the device to bring clock high
7. Wait for the device to bring clock low
8. Repeat steps 5-7 for the other seven data bits and the parity bit
9. Release the data line
10. Wait for the device to bring data low
11. Wait for the device to bring clock low
12. Wait for the device to release data and clock

Figure 9: Device to Host Communication (Data Bit Read on Rising Edge of Clock) [13]

Figure 10: Host to Device Communication (Data Bit Read on Falling Edge of Clock) [13]

Figure 11: PS/2 Communication Protocol [14]
Communication with the trackball was achieved by writing a top level state machine that interfaces with the transmitter and receiver. The purpose of the state machine is to orchestrate the transmission and reception of command/data codes sent to and from the trackball. The primary states include the reset, acknowledge, identification, transmit and packet states. A detailed summary of the states is shown in Table 3.

<table>
<thead>
<tr>
<th>Sending Device</th>
<th>State Name</th>
<th>Sending Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>RESET_STATE</td>
<td>FF</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_ACK_STATE</td>
<td>FA</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_BAT_STATE</td>
<td>AA</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_ID_STATE</td>
<td>00</td>
</tr>
<tr>
<td>FPGA</td>
<td>SET_SAMPLE_RATE_200_STATE</td>
<td>F3</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_ACK_STATE</td>
<td>FA</td>
</tr>
<tr>
<td>FPGA</td>
<td>SEND_DECIMAL_200_STATE</td>
<td>C8</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_ACK_STATE</td>
<td>FA</td>
</tr>
<tr>
<td>FPGA</td>
<td>SET_SAMPLE_RATE_100_STATE</td>
<td>F3</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_ACK_STATE</td>
<td>FA</td>
</tr>
<tr>
<td>FPGA</td>
<td>SEND_DECIMAL_100_STATE</td>
<td>64</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_ACK_STATE</td>
<td>FA</td>
</tr>
<tr>
<td>FPGA</td>
<td>SET_SAMPLE_RATE_80_STATE</td>
<td>F3</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_ACK_STATE</td>
<td>FA</td>
</tr>
<tr>
<td>FPGA</td>
<td>SEND_DECIMAL_80_STATE</td>
<td>50</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_ACK_STATE</td>
<td>FA</td>
</tr>
<tr>
<td>FPGA</td>
<td>READ_DEVICE_TYPE_STATE</td>
<td>F2</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_ACK_STATE</td>
<td>FA</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_ID_STATE</td>
<td>03</td>
</tr>
<tr>
<td>FPGA</td>
<td>SET_EN_STATE</td>
<td>FA</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_ACK_STATE</td>
<td>FA</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_PACKET1</td>
<td>Sign and Button</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_PACKET2</td>
<td>X Value</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_PACKET3</td>
<td>Y Value</td>
</tr>
<tr>
<td>Mouse</td>
<td>GET_PACKET4</td>
<td>Z Value</td>
</tr>
</tbody>
</table>

Table 3: Trackball Controller States and Transmission Values [14]

3.2.2: Implementation

Because this was the first design that was implemented on the board, learning the features of the board took a great deal of time. Considerable time and effort was spent on producing a module to work in conjunction with a PS/2 mouse. Additionally, because this was a first time using an FPGA it became a
challenge to overcome learning curve to develop further designs. Moreover, it became clear in hindsight that the main problem in developing this module was in mixing behavioral descriptions with module instantiations and not dividing the combinational logic into separate functions. This approach proved very difficult to debug. In fact, generated schematic showed a great deal of overlap of gate level components and hierarchical blocks which took more time to troubleshoot than expected. The lesson learned from this block was not to mix module instantiations and behavioral code in one block for synthesis. This caused difficulties during block integration and added more time to debug the design. Although the mouse was working, it became clear that there was a need to segregate all components using modules and functions.

The primary means for debugging the mouse module was to observe its direction of motion using the on board LEDs. Before the motion counters were able to be received, the first step was to verify that the mouse was in streaming (transmitting) mode. Streaming mode means that the mouse delivers packets to the FPGA represented as 9 bit signed 2’s compliment values for X and Y, and 4 bit 2’s complement Z values. Figure 12 summarizes the packet level protocol for a PS/2 mouse. The packets are transmitted to the host (FPGA) during streaming mode.

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y overflow</td>
<td>X overflow</td>
<td>Y sign bit</td>
<td>X sign bit</td>
<td>Always 1</td>
<td>MiddleBtn</td>
<td>Right Btn</td>
<td>Left Btn</td>
</tr>
<tr>
<td>Byte 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 2</td>
<td>X Movement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 3</td>
<td>Y Movement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 4</td>
<td>Z Movement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 12: Depiction of 4 byte packets transmitted from mouse to host FPGA [14]

The initial goal of the mouse module stage (Block A) was to produce a cursor capable of being displayed on a monitor and this goal has been achieved. Through many trial and error loops the first cursor was displayed on the screen. The next step was to move the cursor using a standard Microsoft PS/2 mouse. Majority of the learning took place in experimenting with the board features and implementing simpler modules on the FPGA. Eventually a trackball was purchased to give the user the ease of traversing three orthogonal axes of movement.

The mouse packet controller state machine was created incrementally. First, the state and transmission values were used to create parameters in Verilog along with a list of parameterized command values sent to and from the FPGA.

After setting out all of the parameters, the next step was to write the transition conditions and develop the state machine in Verilog. When the mouse is powered on the first step is to wait until the controller
sends the reset mouse command. The mouse responds with a basic assurance test OK code of 0xAA at which point it sends its standard identifier as a PS/2 mouse. The FPGA requests the mouse to stream position information to the FPGA by sending an enable command. The mouse responds with an acknowledge signal after the FPGAs command request (and after each command sent from the FPGA). Finally the FPGA receives packets 1 through 4 and then back to 1 so long as there is still power applied to the mouse. The FPGAs next job is to decode the position into an angle, displacement or scaling value and apply it to the object that is loaded into the FPGA. Figure 13 depicts a simplified diagram of the described packet level protocol.

![Figure 13: Simplified Mouse Initialization/Streaming Controller State Machine](image)

3.2.3: Verification

After the module was written, the next step was to verify its functionally. This proved to be the most time consuming activity among all of the modules primarily because there were logic errors present in the design and because this was the first time working with an FPGA board. Many times the logic was stuck at a particular state and only by outputting the state to the board LEDs was the state fault resolved. The first step to verify the design was to create a test fixture and run a behavioral simulation on the mouse. The test fixture sent responses to the DUV (design under verification) and function as a simulated mouse. The waves included the names of the states along with the bidirectional lines which were optionally pulled high using a pull up resistor in Verilog. A depiction of the LED arrangement on board the V5LX110T board is

15
shown in Figure 14. These LEDs were ultimately be used to determine the state of the mouse and calibrate the state machine to output the proper packets to the FPGA. Additionally, the User Constraints File (UCF) is shown in Figure 15. The purpose of the UCF is to connect the internal logic of the FPGA to any peripheral device connected to the FPGA pins. Finally, Figures 16 through 19 show the initialization and movement of the mouse in behavioral simulation.

Figure 14: LED arrangement on the XUPV5-LX110T for debugging direction and position

```
# Timing Constraints
NET "user_clk" PERIOD = 9.7 ns; #100MHz

# Pin Constraints
# PS2 Mouse
NET "PS2C" LOC="R27"; # Bank 15, Vcco=1.8V, DCI using 49.9 ohm
NET "PS2D" LOC="U26"; # Bank 15, Vcco=1.8V, DCI using 49.9 ohm
NET "user_clk" LOC="AH15"; # Bank 4, Vcco=3.3V, No DCI

# Debugging LEDs
NET "led<12>" LOC="H18"; # Bank 3, Vcco=2.5V, No DCI
NET "led<11>" LOC="L18"; # Bank 3, Vcco=2.5V, No DCI
NET "led<10>" LOC="G15"; # Bank 3, Vcco=2.5V, No DCI
NET "led<9>" LOC="AD26"; # Bank 21, Vcco=1.8V, DCI using 49.9 ohm
NET "led<8>" LOC="G16"; # Bank 3, Vcco=2.5V, No DCI
NET "led<7>" LOC="AD25"; # Bank 21, Vcco=1.8V, DCI using 49.9 ohm
NET "led<6>" LOC="AD24"; # Bank 21, Vcco=1.8V, DCI using 49.9 ohm
NET "led<5>" LOC="AE24"; # Bank 21, Vcco=1.8V, DCI using 49.9 ohm
NET "led<4>" LOC="E8"; # Bank 20, Vcco=3.3V, DCI using 49.9 ohm
NET "led<3>" LOC="AG23"; # Bank 2, Vcco=3.3V
NET "led<2>" LOC="AF13"; # Bank 2, Vcco=3.3V
NET "led<1>" LOC="AG12"; # Bank 2, Vcco=3.3V
NET "led<0>" LOC="AF23"; # Bank 2, Vcco=3.3V
```

Figure 15: User Constraints File for PS/2 Module
Figure 16: Mouse Initialization Behavioral Simulation
Figure 18: Streaming Packets from PS2 Mouse (Y Movement)
Figure 19: Streaming Packets from PS2 Mouse (X Movement)
3.3: Vertex Fetcher

The next module in the pipeline is the vertex fetcher. The purpose of the vertex fetcher is to retrieve vertices from the Block RAMs where the vertices are stored. As mentioned in Chapter 2, the file format used is OFF, whose coordinate values are represented as base 10 decimal fractions. Using a conversion script, the OFF vertices were converted into a binary lookup table which was suited for processing on the FPGA. Prior to using an external data set, a hard coded state machine with vertex coordinates was used and the module was called the shape generator. It consisted of four states, one state per each of the four lines, and it was used to generate a for line object such as an “x” whose lines met in the center of the screen and whose center was controlled by the mouse trackball. The state machine controller block diagram is shown in Figure 20.

As its name suggests, the vertex streamer acquires points from the vertex memory (FPGA dual-port block rams) and feeds them into the graphics pipeline. They can also be manipulated using the rotation module which, one by one, alters the coordinates fetched from memory. The original contents of memory remain un-altered. The vertex streamer is a revision from the original shape generation module which consists of a statically hard-coded object in a state machine (one state per edge of the object). The state machine is depicted in Figure 21. It consists of six depicted states - reset, idle, line1, line2, line3, and line4.

Not shown in the state diagram is the line clearing states which essentially draw the background color to the frame buffer. Though inefficient in hindsight, hard coding the lines, proved to be a quick way to produce a four lined object using a small amount of memory content and only hard-coded vertex values. Before each draw cycle it was important to clear the screen by un-drawing the lines. More detail about the video generation and line drawing process is discussed towards the end of Chapter 3. Also, the creation of

---

4Conversion script modified from http://www.easysurf.cc/conver17.htm#b10to2
the line on the screen produced artifacts when the FPGA was reset which were cleared when a new line was drawn over the original line in the buffer.

The vertex fetcher was developed in conjunction with the projection module is covered towards the end of this Chapter. It fetches data concurrently while the FPGA initializes the mouse and receives packets. Below are the state machine diagrams with and without clear states.

Figure 21: Original Static Pyramid – Shape Generator State Machine without clear states

In Figure 21 the rasterizerDone transition condition was shown between Line 1 State and Line 2 State, as well as Line 4 State and Idle State. It is not shown between Line2 and Line 3 State nor is it shown between Line 3 State and Line 4 State. Figure 22 depicts the full state machine with the clear states. Note that the state transition equations are not shown in the figure below.

Figure 22: Original Static Four Line Shape Generator State Machine with Clear States
Once the four line generator was created, the next step was to create a new method for generalizing the state machine for any dataset. This meant that the vertex fetcher was capable of handling any amount of data within limits specified in the parameter definitions of the vertex fetcher module. The vertex fetcher takes triangles from the dataset and fetches all the vertices. It is also responsible for coordinating the projection of the coordinates and writing the pixels to the frame buffer. Once the triangle is done, the state machine receives a new triangle from the dataset and fetches all coordinates. This processing repeats until the shape is finally done at which point the object is finished. A mouse movement triggers a new coordinate to be drawn. The way a new set of coordinates are streamed through the pipeline is when the user moves the mouse. The state machine shown is shown in Figure 23.

Figure 23: Vertex Fetcher State Machine
3.4: Coordinate Rotator

The purpose of the coordinate rotator is to take unaltered (no change from original dataset) coordinates and perform the rotation operation on each coordinate, one at a time. The method of rotation is done by adding two operands or subtracting two operands. Each of the operands that are added or subtracted are single dimensional rotations (the operands of the add/subtract) values. When added or subtracted together the rotation forms a 2D rotation. To achieve 2D rotation, simply repeat the process for the next orthogonal pair of axes that are not on the rotation axis. Keep the axis of rotation unaltered. This operation completes the 3D rotation about a single axis for an arbitrary rotation. The significance of keeping rotation axes separate for composite operations is that they can be performed independent of the other rotations. If a rotation operation rotates 30 degrees across the x axis, followed by 30 degrees on either y or z axis the operations do not interfere. Rotation in 3D is a composite operation of arithmetic (addition/subtraction of 2D rotations) and multiplies and trigonometric coefficient computations (1D rotation using a sine/cosine ROM). To compose 3D rotations from one axis to another axis, take the first axis and perform the rotation then store the angle of rotation to perform the second rotation. This method of compositing rotations works well with a mouse interface which outputs each axis of movement on successive bursts. Figure 24 depicts the block diagram of the coordinate rotator for a rotation about the x axis.

![Coordinate Rotator Block Diagram](image)

Figure 24: Coordinate Rotator Block Diagram

3.4.1: Mathematics of Rotation

When an object is rotated about any axis, the values of all points on the axis remain constant while the other two axis values change. If an object were to be rotated about the z axis, the x, and y values change according to the rotation angle theta and its original value phi. Equation 3.1 shows the mathematics for a rotation about the z axis using the sum of angles formula in trigonometry. Phi ($\phi$) is the original angle from the reference axis, theta ($\theta$) is the angle of rotation, and rho ($\rho$) is the radius. Once the equations are determined then the next step is to insert the components into the rotation matrix. Equation 1 presents 2D
rotation equation about the z axis (which affects the x and y coordinates). The equations contain the
sinusoid functions which form the coefficients of the 2D rotation matrix. In 3D computer graphics it is
common to represent the transformations (translation, rotation, scaling and shearing) using matrices.

\[
x = \rho \cos \phi \\
y = \rho \sin \phi \\
x' = \rho \cos(\theta + \phi) \\
y' = \rho \sin(\theta + \phi)
\]

\[
x' = \rho \cos \phi \cos \theta - \rho \sin \phi \sin \theta = x \cos \theta - y \sin \theta \\
y' = \rho \cos \phi \sin \theta + \rho \sin \phi \cos \theta = x \sin \theta + y \cos \theta
\]

Equation 1: 2D Rotation Equations (X, Y) About The Z Axis [15]

The rotation matrix is composed of matrix coefficients. Once the equations are determined then the
signed coefficients of x and y are inserted into the rotation matrix. Equation 2 shows the rotation matrix.
Finally, Equations 3 and 4 show the vector notation and the rotation result in 2D homogenous form.

\[
p = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad p' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \quad R_z = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}
\]

Equation 2: Original, New Coordinate and 2D Rotation Matrices [16]

\[
p' = R_z p \\
x' = x \cos \theta - y \sin \theta \\
y' = x \sin \theta + y \cos \theta
\]

Equation 3: 2D Rotation about Z in Vector Notation and Component Form [16]

\[
R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_z = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

Equation 4: 3D Rotation Matrices about x, y, and z axes [17]

Finally, equation 3.3 summarizes the 3 rotation matrices for performing rotation in 3 dimensions. Each
rotation matrix is performed separately one axis at a time when the user rotates the trackball. When using
a 3D rotation matrix the coefficients along the main diagonal that are 1 make their corresponding rotation
coordinate stay the same. Whichever coordinate has a 1 is the coefficient is the coordinate about which
rotation is performed.
3.4.2: Center Of Rotation

The final consideration when performing rotation is to observe that the object is centered at the midpoint of the viewing frustum (the active object region). The viewing frustum is a depth range from 1 to 3.99 (which is approximately 3.00 in total). The viewing frustum is divided into two and the center of the viewing frustum is where the object lies. In order to perform a rotation about the center of the object is necessary to translate the object to the origin so that its center is at the origin. Then rotation in 3D is performed. Once the rotation is complete then the object is moved back to the center of the viewing frustum where it was originally placed. Adherence to this protocol ensures that the object is rotated about its center and not about the origin. Because there is one generic rotation module (which consists of multipliers, subtractors and adders) the method to determine if a particular instance in Verilog needs to shift the z coordinate back to the origin is if the rotation axis is (x or y) and if the parameter designating that z is affected is set to true (it is passed in as true to the module using parameterization in Verilog). A summary of this method of conditionally adding the center of the viewing frustum (2.5) to z is shown in Table 4. For every z input to the rotator, first subtract 2.5 and then add 2.5 after rotation.

<table>
<thead>
<tr>
<th>Z Changes</th>
<th>Rotation Axis</th>
<th>Red Signal</th>
<th>Green Signal</th>
<th>Add 2.5 to Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>X</td>
<td>Y_Rotated</td>
<td>Z_Rotated</td>
<td>YES</td>
</tr>
<tr>
<td>YES</td>
<td>Y</td>
<td>X_Rotated</td>
<td>Z_Rotated</td>
<td>YES</td>
</tr>
<tr>
<td>NO</td>
<td>Z</td>
<td>X_Rotated</td>
<td>Y_Rotated</td>
<td>NO</td>
</tr>
</tbody>
</table>

Table 4: Depiction of when to add 2.5 to Z (Refer to Figure 24 for red and green signal locations)

Figure 25: Summary of rotation options for trackball input device
3.4.3: Fixed Point Arithmetic

Rotation involves the addition or subtraction of fixed point numbers. When adding or subtracting fixed point numbers with each other, it is common to follow rules for determining the sign of the operands based on the comparison of the magnitude values. For example, if the first operand is less than the second operand then based on the operation performed the rule is to determine the sign of the operands and then perform either an addition or subtraction based on the weight of the magnitudes. Table 5 summarizes the rules in tabular form.

<table>
<thead>
<tr>
<th>Operation</th>
<th>ADD Magnitudes</th>
<th>SUBTRACT Magnitudes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+A) + (+B)</td>
<td>+ (A + B)</td>
<td>A &gt; B</td>
</tr>
<tr>
<td>(+A) + (-B)</td>
<td></td>
<td>A &lt; B</td>
</tr>
<tr>
<td>(-A) + (+B)</td>
<td>- (A + B)</td>
<td>A = B</td>
</tr>
<tr>
<td>(-A) + (-B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+A) - (+B)</td>
<td>+ (A - B)</td>
<td></td>
</tr>
<tr>
<td>(+A) - (-B)</td>
<td>+ (A + B)</td>
<td></td>
</tr>
<tr>
<td>(-A) - (+B)</td>
<td>- (A + B)</td>
<td></td>
</tr>
<tr>
<td>(-A) - (-B)</td>
<td>- (A - B)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Summary of rotation options for trackball input device [18]

3.4.4: Verification

The next two pages (Figures 26 and 27) show the results of the output of the sine / cosine ROM along with the rotation multipliers. A single coordinate at (0,0,1.5) was used in the rotation equation and its rotation is depicted as an analog step in the waveforms. The coordinate is rotated about x, y and z axes and it is shown on the next two figures. Its behavioral simulation is graphically presented in the next two figures. The ROM simulation clearly shows a phase shift in the outputs while the rotator multiplier results show a 1D rotation of the coordinates (x, y or z).
Figure 26: Behavioral Simulation (Analog Step) of Sin/Cos ROM
Figure 27: Behavioral Simulation (Analog Step) of Rotator Multipliers
(Rotate x,y,z = 0,0,1.5 about each axis)
3.5: Projector and Viewport Mapper

3.5.1: Overview

The projector and viewport mapper are responsible for taking a 3D coordinate and converting it into a 2D coordinate that is ready for display on a monitor. Also, the viewport mapper assures that the object is scaled to the screen space which starts from the top left hand corner to the bottom right hand corner using positive whole numbers. Furthermore, the module also performs perspective correction on the object. Perspective correction assures that points further away along the z axis converge to the center of the screen space (midpoint of the screen) or they converge to the center of x and y (where x and y are both zero). A graphical depiction of one-point perspective convergence is shown in Figure 28.

![ONE-POINT PERSPECTIVE](image)

Figure 28: One point perspective convergence [19]

As its name implies, one point perspective correction makes all points in the viewing frustum converge to a point situated at \((x,y) = (0,0)\). For objects further on the x or y axis from \(x = 0\) and \(y = 0\) objects have significantly higher noticeable distortion. Points closer in depth to the vanishing point appear closer together than points further in depth to the vanishing point.

The next step in the pipeline is the viewport transformer, whose job it is to perform translation from world to screen coordinates. Figure 29 shows the transformation of a triangle from world to screen coordinates. The world coordinates represent the coordinates where the points are active and the screen coordinates represent the coordinates \((0 \rightarrow n)\) for the screen where \(n\) is one less than the horizontal or vertical resolution of the monitor.

![Viewport Transformation and Scaling](image)

Figure 29: Viewport Transformation and Scaling [20]
Mathematical Foundation

The first step in projecting a point from the world space to the camera space is to either discard the z coordinate and form an orthographic (parallel) projection matrix or to form a perspective (convergent) projection matrix where the depth information divides the horizontal and vertical information. Equation 5 presents both projection matrices. The main difference is that in the left matrix the z component drops out entirely whereas in the right matrix the z component does not drop out immediately but is rather used to divide into the x and y components.

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1/d & 0
\end{bmatrix}
\]

Equation 5: Orthographic and Perspective Matrices [1]

After projection and perspective correction the point undergoes viewport transformation. During viewport transformation the point undergoes a translation to the origin, a scale from world to normalized device coordinate space and then a translation back to the monitor origin to complete the screen mapping process. Equation 6 shows the viewport mapping matrix (3x3).

\[
V = \begin{bmatrix}
\frac{u_{\text{max}} - u_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} & 0 & -\frac{x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} + \frac{u_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \\
0 & \frac{v_{\text{max}} - v_{\text{min}}}{y_{\text{max}} - y_{\text{min}}} & -\frac{y_{\text{min}}}{y_{\text{max}} - y_{\text{min}}} + \frac{v_{\text{min}}}{y_{\text{max}} - y_{\text{min}}}
\end{bmatrix}
\]

Equation 6: Viewport Matrix [21]

When the actual matrix multiplication is performed to form the algebraic equalities, each component of the transformation matrix is multiplied one at a time with the input coordinate matrix and then the products are added together to form respective cell values of the result matrix. When multiplying a series of 3 matrices together, as in Equation 7, the result matrix is a result of the accumulated products in each row cell combination. The product is evaluated between two matrices and then the resulting matrix is multiplied with the third matrix. After the equation is presented the next step is to evaluate the matrix expression to derive the general equation for viewport transformation. The convention is to use three values for the coordinate (2D) which are x, y and w, which correspond to homogenous matrices. Once the homogenous matrices are evaluated, the last component (w) is used to alter the result coordinate. The result matrix is formed by multiplication of the matrices in Equation 7 which results in a matrix shown in Equation 8. This matrix is then multiplied by the input coordinate to form the preliminary equation for the viewport.
\[ \mathbf{V} = (\text{Translate to Center})(\text{Scale to Viewport})(\text{Translate to Viewport}) \]

\[ \mathbf{V} = \begin{bmatrix} 1 & 0 & u_{\text{min}} \\ 0 & 1 & v_{\text{min}} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{u_{\text{max}} - u_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} & 0 & 0 \\ 0 & \frac{v_{\text{max}} - v_{\text{min}}}{y_{\text{max}} - y_{\text{min}}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_{\text{min}} \\ 0 & 1 & -y_{\text{min}} \\ 0 & 0 & 1 \end{bmatrix} \]

Equation 7: Viewport Transformation Matrix Determination [22]

\[ \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \mathbf{V} \mathbf{P} = \begin{bmatrix} \frac{u_{\text{max}} - u_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} & 0 & -x_{\text{max}} \\ 0 & \frac{v_{\text{max}} - v_{\text{min}}}{y_{\text{max}} - y_{\text{min}}} & y_{\text{max}} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} \]

Equation 8: Transformation Matrix Determination [22]

The viewport along the y axis is inverted initially. A flip operation is required to take the inverted axis of y whose midpoint is 511 and subtract from the midpoint the inverted expression for the y axis. Equation 9 below summarizes the result of the matrix multiplication from the above matrices.

\[ x_{\text{Viewport}} = x_{\text{World}} \left( \frac{u_{\text{max}} - u_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \right) + \left( -x_{\text{min}} \frac{u_{\text{max}} - u_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} + u_{\text{min}} \right) \]

\[ y_{\text{Viewport inveted}} = y_{\text{World}} \left( \frac{v_{\text{max}} - v_{\text{min}}}{y_{\text{max}} - y_{\text{min}}} \right) + \left( -y_{\text{min}} \frac{v_{\text{max}} - v_{\text{min}}}{y_{\text{max}} - y_{\text{min}}} + v_{\text{min}} \right) \]

Equation 9: Viewport Transformation Equations

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Orientation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u_{\text{max}})</td>
<td>1023</td>
<td>Horizontal</td>
</tr>
<tr>
<td>(u_{\text{min}})</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(x_{\text{max}})</td>
<td>1.999</td>
<td></td>
</tr>
<tr>
<td>(x_{\text{min}})</td>
<td>-1.999</td>
<td></td>
</tr>
<tr>
<td>(v_{\text{max}})</td>
<td>1023</td>
<td>Vertical</td>
</tr>
<tr>
<td>(v_{\text{min}})</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(y_{\text{max}})</td>
<td>1.999</td>
<td></td>
</tr>
<tr>
<td>(y_{\text{min}})</td>
<td>-1.999</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Summary of parameters for viewport transformation
Once the viewport mapping equations are evaluated with the parameters in Table 6 then they are ready for implementation. The resulting viewport equations are shown in Equation 10.

\[
\begin{align*}
x_{\text{Viewport}} &= 255 \ast x_{\text{World}} + 512 \\
y_{\text{Viewport}} &= 128 \ast y_{\text{World}} + 255 \\
y_{\text{Viewport}} &= 511 - y_{\text{Viewport}^{-1}} \\
x_{\text{Viewport}} &= 255 \ast x_{\text{World}} + 512 \\
y_{\text{Viewport}} &= 511 - 128 \ast y_{\text{World}}
\end{align*}
\]

Equation 10: Final viewport equations (1024x1024 screen resolution)

3.5.3: Implementation

The final goal of the projection module is to start with a static data set (initially a pyramid) and generate the lines on the screen corresponding to the edges of the pyramid. An example of how projection works is graphically depicted in the following figure. Projection starts with 3D coordinates in the world space. Using a projection matrix multiplication the 3D points are projected onto a projection plane. After the points are projected they undergo a perspective division step where they are scaled to the screen in proportion to their distance from the projection plane. The z axis position of the points determines the points scaling factor on the XY plane. After the perspective correction step, the final step is to map the points to the monitor. Figure 30 shows a graphical representation of projection.

![Figure 30: 3D Pyramid alongside its projected representation](image-url)
When testing the initial module it was necessary to send in the four coordinates (xStart, yStart, xEnd, yEnd) to the rasterizer in parallel. This meant creating 4 instances of the projection, perspective division and viewport mapping blocks. This approach ensures that the signals arrived to the line drawer (rasterizer) at the same time allowing the rasterizer to perform its rasterization. It became clear in implementation that while this method worked in simulation it did not synthesize for a larger design with a larger lookup table. Additionally, because of the use of a combinational lookup table needed to synthesize such a design routing the design became a problem for the place and route stage of the implementation.

Figure 31: Initial Integration of Coordinate Generator, Viewport Projector and Rasterizer

The solution to the problem became reducing the four instances into only one instance and the first step in designing the projector and viewport module is to layout the block diagram and its inputs/outputs. The top level module is synchronous and contains a coordAxis, dataValid, coordValue, depthValue and coordNumUnderProcess as inputs. The outputs are the window coordinates which are sent to the line drawer (covered in the next section).

Figure 32: Projector / Viewport Mapper Block

Figure 33: Internal Blocks of the Coordinate Projector and Viewport Mapper
3.5.4: Design Challenges

The first challenge that was overcome was determining how to divide two fixed point numbers. Before finding a suitable solution the first method was to try to divide two numbers by using an unsigned binary divider. Unfortunately, because both binary numbers were fixed point representation the division operation produced incorrect results due to differences from fractional representation as compared to binary whole numbers. The solution to this problem was to either use a fixed point algorithm or to use a division method called multiplication by the reciprocal of the divisor which is shown schematically in Figure 34 and 35. This method seemed most feasible for the division operation that needed to be implemented. To test this solution a binary look-up table was created which included all possible combinations of bits. Initially the weighted sum of each bit multiplied by its fractional representation was computed and then stored and the closest matching value was used. After this method was used the more reliable method of using a base 10 fraction to base 2 fixed point converter (presented in the Appendix C) was used. The code was modified to work with an array of numbers in base 10 fixed point precision format.

![Figure 34: High Speed Division by Reciprocal Multiplication [23]](image1)

The next step was to construct the look up table. The lookup table began with 2999 entries. A case statement based LUT was created which was then converted to a block ram with initialized contents.

![Figure 35: Perspective Divider (using Reciprocal Lookup Multiplication Method)](image2)
Figure 37: Behavioral Simulation of Fetcher/Projector with a single line endpoint shown.
3.6: **Line Drawer Module**

The purpose of the line drawer module is to take a start and end coordinate and map it to the frame buffer for each pixel coordinate (memory address). The start and end coordinates are generated from the vertex fetcher which passes the coordinates through the projector (coordinate space) and viewport mapper (screen space). Once in the screen space the lines are drawn in a circular fashion every time the frame address counter goes back to the first address. Each pixel that is written to the frame buffer needs to be converted to an address. The formula for finding the address is the multiply the number of pixels per line (1024) by the line number (yOut) and add the pixel offset (xOut). Once the formula is written then the final step is to represent the equation for memory address access as a sum of shifts by different constants.

The method by which the line is drawn is through the Bresenham algorithm. The Bresenham rasterization algorithm is an algorithm that determines which pixels are active on raster (two dimensional pixel bitmap) display as shown in Figure 38. The pixels that are active correspond to the pixels that approximate the line containing the two end points specified on the input end of the rasterization block.

The objective of the rasterization block was to produce a module capable of generating pixels on a monitor at certain coordinates. Each coordinate with an active pixel is a coordinate that is generated using the Bresenham line rasterization algorithm. Bresenham’s algorithm is a powerful algorithm because it generates coordinates for a line without using expensive floating point multiplication or division and instead uses incremental addition or subtraction computations to calculate successive coordinates for a line. The result of the incremental computation is a line that is rendered on a bitmap pixel matrix. The first step in implementing the rasterization block is to conceptualize the top level block with required inputs, outputs and bidirectional lines. The rasterizer can be developed independent of its screen mapping module predecessor.

![Figure 38: Depiction of a Rasterized Line with a positive slope](image-url)
3.6.1: State Machine

The state machine for the line generator consists of 6 states (idle, setup1, setup2, setup3, rasterization, and done). Originally the state machine was written with two states for each pixel generation. The final state machine consists of one state per pixel. Pseudo code for the state machine is depicted below (Figure 39).

```java
public void lineBresenham(int x0, int y0, int x1, int y1, Color color) {
    int pix = color.getRGB();
    int dy = y1 - y0;
    int dx = x1 - x0;
    int stepx, stepy;

    if (dy < 0) { dy = -dy; stepy = -1; } else { stepy = 1; }
    if (dx < 0) { dx = -dx; stepx = -1; } else { stepx = 1; }
    dy <<= 1;  // dy is now 2*dy
    dx <<= 1;  // dx is now 2*dx

    raster.setPixel(pix, x0, y0);
    if (dx > dy)
    {
        int fraction = dy - (dx >> 1); // same as 2*dy - dx
        while (x0 != x1)
        {
            if (fraction >= 0)
            {
                y0 += stepy;
                fraction -= dx;  // same as fraction -= 2*dx
            }
            x0 += stepx;
            fraction += dy;   // same as fraction -= 2*dy
            raster.setPixel(pix, x0, y0);
        }
    }
    else
    {
        int fraction = dx - (dy >> 1);
        while (y0 != y1)
        {
            if (fraction >= 0)
            {
                x0 += stepx;
                fraction -= dy;
            }
            y0 += stepy;
            fraction += dx;  // same as fraction -= 2*dx
            raster.setPixel(pix, x0, y0);
        }
    }
}
```

Figure 39: Bresenham Rasterizer pseudo code [25]
The next step after understanding the pseudocode was to convert it to a state machine in Verilog. The first step was to convert the pseudocode to distinct states. The states are shown in Figure 40 and Table 7 below. A description of the function of each state is provided alongside the states. As mentioned before the first step was to receive the reset signal at which point the first setup state was reached. This setup state takes the input coordinates and performs initial delta calculations on the coordinates. Additionally, it moves to second setup state which finds the two delta values. Any sequential code presented in the previous page that requires a computed result to be put into a secondary setup state. Finally, the third setup state then calculates the actual next line. In the rasterization state, the line is incremented in accordance with the delta calculations performed in the previous steps. When the coordinate counters reach the final coordinate (end) value the rasterization done signal is issued. The loop back arrows in the figure show the preservation of the state variables. The only time the state is issued a state transition request is when the state calculations and conditions are complete.

![State Machine for Line Generator Block](image)

<table>
<thead>
<tr>
<th>State Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE_STATE</td>
<td>Waits for new line pulse to go to first setup state</td>
</tr>
<tr>
<td>SETUP_STATE_1</td>
<td>Calculates delta values of input coordinates</td>
</tr>
<tr>
<td>SETUP_STATE_2</td>
<td>Calculates 2delta values of input coordinates</td>
</tr>
<tr>
<td>SETUP_STATE_3</td>
<td>Calculates slope based on delta values</td>
</tr>
<tr>
<td>RASTERIZATION_STATE</td>
<td>Calculates successive pixel coordinates until end</td>
</tr>
<tr>
<td>RASTERIZATION_DONE</td>
<td>Asserts finished signal and de-asserts writing signal</td>
</tr>
</tbody>
</table>

Table 7: Line Generator State Description
The last step was to connect the line generator to the video generation module and verify that a line could be drawn. The primary steps consist of generating X,Y pairs and sending them to the frame buffer (dual port block ram) which is configured to work as a circular buffer. The final step is to stream to pixel data and color data to the LCD monitor. The formula for finding the address in block ram for an input x, and y coordinate is address = 1024y + x. This equation can be hardware optimized to simply perform a shift by the log base 2 of the constant coefficient multiplier. The final optimized equation is a = yout << 10 + x. This results in a savings of a multiplier (assuming there is no optimization during elaboration of the RTL code). The block diagram is shown below.

![Block Diagram of Line Generator](image)

The first integration task was to connect a static line generator to the video generator and frame buffer and observer the monitor draw a static line on the screen. The next step was to create a pattern of four lines on the screen and alter the pattern based on the mouse position. The initial and final versions of the system tasks are depicted Figure 42 and Figure 43.

![Implementation tasks for the Static Line Generator](image)
After creating the static line generator the final step was to create an animated line generator based on the position of the mouse (displacement values) and to demonstrate that the lines moved in accordance with the coordinate values.

Figure 43: Implementation tasks for the Dynamic Line Generator

The initial output results of the line generator are shown below. They consist of two cursor positions (center and left). The left image depicts a centered cursor position while the right image depicts a left aligned cursor position. As the cursor moves the four lines stretch and move to form the appropriate shape.

Figure 44: Results from Cursor Position Based Line Generator

The rasterizer module takes in a start and end coordinate which is sent in by an internal state machine and generates the lines corresponding to the start and end points. The state machine draws and undraws 4 lines when the mouse changes position. The rasterizer connects to the screen map module, the DVI transmitter module and the PS/2 mouse module.

In summary, the bresenham algorithm was converted from pseudo code to a Verilog state machine. The state machine generates coordinates that send the color values of each coordinate to a corresponding to the Block RAM frame buffer. The frame buffer is a 1024 X 1024 with a single bit depth of color. A color depth of 1 corresponds to a monochrome image with only two colors. A behavioral simulation of the single line (addresses) is presented in the following page. It starts from the origin to the maximum address in the frame buffer (1024x1024).
Figure 45: Behavioral Simulation of Single Line Generator
3.7: Frame Buffer

The purpose of the frame buffer is to accept color and coordinate values and input them into a block of memory for one frame of video. From there the pixel reader circularly reads the pixels and output them to the monitor. The frame buffer stores 1024x1024 (~1 Million) monochrome (color depth of 1) pixels. Dual port ram is used to display the contents of the graphical object as pixels. Additionally, dual port ram can read and write at the same time and is used for bridging two clock regions. For purposes of this project the frame buffer is used to display single pixels per address but can be configured to display higher pixel color depths for specific applications. A suitable SRAM controller was located from OpenCores.org (developed by Victor Lopez Lorenzo) before the decision was made to use exclusively on-chip memory. The SRAM and DPRAM are shown in Figure 46. The on-chip memory contains space for approximately 5 megabits of data whereas the off chip SRAM contains approximately 9.4 megabits of data supporting a higher resolution of pixels. The need to shift from SRAM to BRAM was due to the fact that the SRAM controller did not properly send data to the FPGA at the 100MHz speed requested. Eventually, the SRAM controller was abandoned and a dual port block ram method was used to communicate video frames to the monitor and store the frames. Because DPRAM is directly embedded in the FPGA, it can be run at much faster speeds than the SRAM. More importantly, because SRAM is not on the FPGA and has no ability to allow for simultaneous reads or writes, the SRAM is less suitable for using for a frame buffer that requires concurrent reads and writes. This limitation resulted in use of the DPRAM on the FPGA.

Figure 46: SRAM Interface versus DPRAM Module
3.8: Video Generator

The final stage in the pipeline design is the video generator and pixel multiplexer. It takes in RGB data values across three 8 bit channels and converts them into a 12 bit multiplexed data which combine to produce a 24 bit RGB data. Since the models use single bit color and the fact that the design is running at 1024x1024, this means that the module needs to extend the color bits. Each 12-bit burst of data consists of the RGB data repeated (if less than 8 bits per color channel) to fill the 8 bit channel with. The chip responsible for the transmission of video synchronization and data signals is the Chrontel 7301C transmitter. The transmitter can be configured to run in DVI or VGA bypass modes, in which the output data is configured to output in a single analog port for each color channel. When configured for DVI mode, the transmitter sends a data stream that directly compatible for a digital monitor.

The slave (CH7301C) at address 0x76 (on an I2C line) is initialized from the FPGA using I2C bus master. I2C is a two wire serial protocol similar to the mouse protocol in that it contains two signals (clock and data). When the master is ready to transmit data to the slave it does so by initializing the address of transfer. The I2C slave responds with “acknowledge” if the address transferred matches the address value of the slave device. Once an acknowledge is passed to the master on the data line then the master starts transmitting 8 bit data values after it indicated the write status to the slave. Figure 47 shows the I2C timing diagram.

![Figure 47: I2C Protocol](image)

Once the FPGA initializes the DVI transmitter for the appropriate mode (DVI or analog bypass), the DVI transmitter begins streaming video data to the monitor. The primary signals needed for a DVI transmission are the pixel clock, horizontal sync, vertical sync, data, data enable. The video controller parameters and I2C initialization were modified from existing video parameters that were part of two separate existing modules which were combined to produce initial test patterns on the monitor. Video timing parameters for the video generator were created from the XFree86 modeline generator [27].

<table>
<thead>
<tr>
<th>Video Parameter</th>
<th>Horizontal Pixels</th>
<th>Vertical Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Width</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Front Porch</td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>Synch Width</td>
<td>376</td>
<td>11</td>
</tr>
<tr>
<td>Back Porch</td>
<td>32</td>
<td>21</td>
</tr>
<tr>
<td>Total</td>
<td>1432</td>
<td>1076</td>
</tr>
</tbody>
</table>

Table 8: Video Parameters
The timing diagram below shows the monitor signals generated by the CH7301C. The vertical synchronization pulse is not shown because it does not fit within the confined view. The vertical synchronization pulse needs 1024 lines which go beyond the figure width. A summary of the parameters used for the video generator is shown below the figure. The columns in the table depict the parameters, the horizontal and vertical parameters.

The next step in the design was to construct the primary logic for the video module. The video module (Figure 48) outputs red green and blue signals based the counter values presented by the synchronization generator.

Figure 48: Video Timing Diagram

Figure 49: Video Generator
Figure 50: Pixel Multiplexer (Input to Chrontel 7301C)
Chapter 4: Results

This section presents the frame buffer results after applying rotation transformation to the object. The primary means of extracting the frame buffer contents for a stationary non-moving object was to export the contents of the memory array from Modelsim. The contents of the frame buffer are displayed on the monitor. First the orthographic projection result is shown on Figure 51 which does not rely upon the reciprocal ROM to compute the projected coordinates. Figure 52 shows the results of projection of the object which undergoes perspective distortion and therefore shows points closer together which have a higher depth coordinate.

Figure 51: Orthographic (parallel) projected teapot

Figure 52: Perspective (convergent) projected teapot

<table>
<thead>
<tr>
<th>GPU Parameter</th>
<th>Numerical Value</th>
<th>Maximum (Adjustable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_VERTICES_IN_OBJECT</td>
<td>480</td>
<td>512</td>
</tr>
<tr>
<td>NUM_TRIANGLES_IN_OBJECT</td>
<td>880</td>
<td>1024</td>
</tr>
</tbody>
</table>

Table 9: Summary of Object Parameters
<table>
<thead>
<tr>
<th>Rotation about Y Axis (0 Degrees)</th>
<th>Rotation about X Axis (0 Degrees)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
</tr>
<tr>
<td>Rotation about Y Axis (30 Degrees)</td>
<td>Rotation about X Axis (30 Degrees)</td>
</tr>
<tr>
<td><img src="image3.png" alt="Image" /></td>
<td><img src="image4.png" alt="Image" /></td>
</tr>
<tr>
<td>Rotation about Y Axis (60 Degrees)</td>
<td>Rotation about X Axis (60 Degrees)</td>
</tr>
<tr>
<td><img src="image5.png" alt="Image" /></td>
<td><img src="image6.png" alt="Image" /></td>
</tr>
<tr>
<td>Rotation about Y Axis (90 Degrees)</td>
<td>Rotation about X Axis (90 Degrees)</td>
</tr>
<tr>
<td><img src="image7.png" alt="Image" /></td>
<td><img src="image8.png" alt="Image" /></td>
</tr>
</tbody>
</table>

Table 10: Application of rotation transformation with various angles
After applying the rotation of the four different angles across two axes, the next step was to check how the rotation across two axes displayed on the frame buffer. Two rotation angles were used (30 and 60 degrees) and they were applied across x and y axes. The final results are shown in Table 11.

<table>
<thead>
<tr>
<th>30 Degree Rotation (x then y)</th>
<th>60 Degree Rotation (x then y)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
</tr>
</tbody>
</table>

Table 11: Rotation across both x and y
Chapter 5: Conclusion

The project was a success because the primary goals of learning the graphics pipeline and designing the system were achieved. There were difficulties encountered during the project. The first difficulty was that there was a considerable learning curve that needed to be overcome since this was the first time developing on an FPGA. Additionally, familiarizing with the 3D pipeline was challenging in the beginning. To overcome this challenge, an incremental approach was taken to familiarize with the pipeline. The mouse module was the first module that was created, which was followed by the video generator. Outputting to the on board LEDs proved to uncover several cases where the state machine was remaining in the same state. There were modules in the design that were not used in the end and one of them was the SRAM frame buffer interface. As covered before, the SRAM frame buffer was not able to output back values to the graphics video generator. A decision was made to abandon the SRAM module and instead use the Dual Port RAMS on the FPGA. Finally, the projection module started off by drawing random lines. After several iterations it was discovered that the timing was the main issue for this. To resolve timing, delay registers were placed to allow for the signals to line up. Additionally, the perspective divider needed to be corrected to produce the proper product (quotient).
Chapter 6: Future Work

There are several areas that can be explored to further enhance the project. A primary enhancement is to continue automating the generation of the binary fixed point values. When using the original fixed point binary generation script there was a need to export the binary data manually and combine the results. The binary conversion script is shown in Appendix C. There was an initial development on automating the acquisition and generation of the binary fixed point values from the input dataset. Unfortunately, XST (Xilinx Synthesis Tool) does not support real data types in synthesis even if the intent is to use the real data types for only elaboration and file generation. To overcome this limitation, a temporary solution was found which was to take the data and to run it in behavioral simulation. Afterwards, the non-synthesizable instructions were comment out the code that is not supported by XST and use $readmemb system task to read in the data generated by behavioral elaboration. Tcl (Tool Command Language) could have been used to enhance the automation of this fixed point generation problem but, due to time constraints, this method was not explored in the project.

The next development applicable to the project is to employ a shader which requires an exponentiation look up table and a pixel color interpolator. A first development can be to design a flat (single color) shading and then to change it to a smooth shader. The mouse trackball can be used to rotate the light source to change the position of the light aiming on the object. Finally, clipping is needed to prevent the rasterizer from drawing unnecessary lines that are not part of the object. This can be done using either scissoring (image space clipping) directly on the frame buffer or clipping (object space vertex removal) which is done prior to the object being displayed on the screen.

The design is currently a static pipeline. It could be converted to a programmable pipeline which can run alongside an embedded processor (Microblaze) using C instructions. This hardware/software co-design option was explored but not implemented because it did not prove to directly add any measurable difference to the desired results.
References


[16] 2D Rotation Notes [Accessed September 25, 2011.]
http://www.siggraph.org/education/materials/HyperGraph/modeling/mod_tran/2drota.htm

http://www.siggraph.org/education/materials/HyperGraph/modeling/mod_tran/3drota.htm


http://articles.katorlegaz.com/quicktipsinartanddesign/depth/0802one-point.jpg

http://www-users.aston.ac.uk/~cornford/cs2150/pdf/window2vp_lec_8up.pdf


http://cs.gettysburg.edu/~skim/cs373/lectures/02_ScanLineConversion_Line.pdf


ftp://ftp.gwdg.de/pub/misc2/opencores/cores/i2c/i2c_rev03.pdf

http://xtiming.sourceforge.net/cgi-bin/xtiming.pl
Appendix A: GPU Source Code

coordinateRAM.v

`timescale 1ns / 1ps

/**********************************************************************
* Module Name: coordinateRAM
* File Name: coordinateRAM.v
* Author: Vahe Robert Jabagchourian (CSUN)
* References:
* http://www.cse.lehigh.edu/~caar/marnold/papers/sanjose_hdlcon.doc
* [2] Inferring Dual Port Block Rams [Retrieved 7/1/2011]
* [3] Verilog File I/O Routine
* Creation Date: July 1, 2011
* Description: Loads a compile time rotated dataset using sin/cos
  mathematical functions
* Revision History
* July 1, 2011 Parameters altered for calculated sizes
* August 22, 2011 Added initialization memory content for vertex memory
**********************************************************************/

module coordinateRAM
    (clk,
     wr_en,
     wr_addr,
     wr_data,
     rd_addr,
     rd_data,
     rd_addr2,
     rd_data2);

    parameter ADDR = 9;    //512 points total (2^9)
    parameter DATA = 39;    //13 bit coordinate data for each x, y, z
    parameter IN_FILE = "teapot.off";
    parameter MEM_FILE = "teapot.mem";

    parameter SIGNED = 1;
    parameter UNSIGNED = 0;
    parameter X_AXIS = 2'b00;
    parameter Y_AXIS = 2'b01;
    parameter Z_AXIS = 2'b10;

    input wire clk;
    input wire wr_en;

    // write port 0
    input wire [ADDR-1:0] wr_addr;
    input wire [DATA-1:0] wr_data;
// read port 0
input wire [ADDR-1:0] rd_addr;
output reg [DATA-1:0] rd_data;

// read port 1
input wire [ADDR-1:0] rd_addr2;
output reg [DATA-1:0] rd_data2;

reg [DATA-1:0] mem[0:(2**ADDR)-1];
real pi = 3.14159265;
real xCoordinateRotated, yCoordinateRotated, zCoordinateRotated;
real deltaAngle = 0.00613592315154256;
real angleOfRotation;

//Retrieved from [1]
function real sin;
    input x;
    real x;
    real x1,y,y2,y3,y5,y7,sum,sign;
begin
    sign = 1.0;
    x1 = x;
    if (x1<0)
    begin
        x1 = -x1;
        sign = -1.0;
    end
    while (x1 > 3.14159265/2.0)
    begin
        x1 = x1 - 3.14159265;
        sign = -1.0*sign;
    end
    y = x1*2/3.14159265;
    y2 = y*y;
    y3 = y*y2;
    y5 = y3*y2;
    y7 = y5*y2;
    sum = 1.570794*y - 0.645962*y3 + 0.079692*y5 - 0.004681712*y7;
    sin = sign*sum;
end
endfunction

function real cos;
    input x;
    real x;
begin
    cos = sin(x + 3.14159265/2.0);
end
endfunction

//Rotator Task
task rotate; // (xCoord, yCoord, zCoord, axisRot, theta)
    input real xCoordReal;
    input real yCoordReal;
    input real zCoordReal;
endtask

56
input reg [1:0] rotationAxis;
input real rotationAngle;
beginn
case (rotationAxis)
//Rotation about x
//x'=x
//y'=y(cosθ) - z(sinθ)
//z'=y(sinθ) + z(cosθ)
X_AXIS:
begin
//Hold X
//Transform Y and Z
//x'=x
xCoordinateRotated = xCoordReal;
//y'=y(cosθ) - z(sinθ)
yCoordinateRotated = (yCoordReal * cos(rotationAngle)) -
((zCoordReal - 2.5) * sin(rotationAngle));
//z'=y(sinθ) + z(cosθ)
zCoordinateRotated = (yCoordReal * sin(rotationAngle)) +
((zCoordReal - 2.5) * cos(rotationAngle)) + 2.5;
end
//Rotation about y
//x'=x(x(cosθ) - z(sinθ)
//y'=y
//z'=z(x(sinθ) - y(sinθ)
Y_AXIS:
begin
//x'=x(x(cosθ) - z(sinθ)
xCoordinateRotated = (xCoordReal * cos(rotationAngle)) +
((zCoordReal - 2.5) * sin(rotationAngle));
//y'=y
yCoordinateRotated = yCoordReal;
//z'=y(sinθ) + z(cosθ)
zCoordinateRotated = -(yCoordReal * sin(rotationAngle)) +
((zCoordReal - 2.5) * cos(rotationAngle)) + 2.5;
end
Z_AXIS:
begin
//Rotation about z
//x'=x(x(cosθ) - y(sinθ)
xCoordinateRotated = (xCoordReal * cos(rotationAngle)) -
(yCoordReal * sin(rotationAngle));
//y'=x(sinθ)+y(cosθ)
yCoordinateRotated = (xCoordReal * sin(rotationAngle)) +
(yCoordReal * cos(rotationAngle));
//z'=z
zCoordinateRotated = zCoordReal;
end
endcase
end
task
real angle;
//Count up to the desired index
//Compute the angle based on the equal increment deltaAngle
//Equal increment is deltaAngle
function real indexToAngle;
    input reg [9:0] index;
begin
    indexToAngle = deltaAngle * index;
end
endfunction

//Begin decimal to binary conversion functions

//Returns integer component of the fractional decimal in base 10
function integer truncate;
    input real fractionalDecimal;
begin
    truncate = $rtoi(fractionalDecimal);
end
endfunction

function integer greatestExponentBase2LessThan;
    input integer decimalInteger;
    integer binaryExponent;
begin
    binaryExponent = 0;
    //Check the next value to see if it as a power of two is less
    //than or equal to binaryInteger
    while (2**(binaryExponent+1) <= decimalInteger)
    begin
        binaryExponent = binaryExponent + 1;
    end
    greatestExponentBase2LessThan = binaryExponent;
end
endfunction

function [1:0] decimalToBinary;
    input integer decimalInteger;
    integer decimalValueRemainder;
    //Result of subtraction of decimalInteger from
    //greatestPowerOf2LessThan
    integer greatestExponentOfBase2LessThanDecimalValue;
begin
    decimalToBinary = 2'b00; //Initialize
decimalValueRemainder = decimalInteger;
    //Iteratively subtract the greatest power of two less than the
    //decimal integer until the result of subtraction is equal to 0
    while (decimalValueRemainder > 0)
    begin
        greatestExponentOfBase2LessThanDecimalValue =
            greatestExponentBase2LessThan(decimalValueRemainder);
decimalValueRemainder = decimalValueRemainder -
        2**greatestExponentOfBase2LessThanDecimalValue;
decimalToBinary[greatestExponentOfBase2LessThanDecimalValue-1]=1;
end
endfunction
//convert fractional signed magnitude number to binary
function [1:-11] binaryFraction;
  input real fractionalNumber;
  input fractionType;
  real currentFractionalNumber;
  real fractionalComponent;
  integer binaryFractionIndex;
  reg [1:-11] binaryFractionVar;
  integer integerComponent;
begin
  fractionalComponent = 0;
  //Start index at the position behind the decimal
  binaryFractionIndex = -1;
  //Initialize the currentFractionalNumber
  currentFractionalNumber = fractionalNumber;
  binaryFractionVar = {11{1'b0}};

  //Convert the sign and magnitude of the signed integer
  if (fractionType == SIGNED)
  begin
    if (fractionalNumber < 0)
    begin
      binaryFractionVar[1] = 1;
      binaryFractionVar[0] = (fractionalNumber <= -1)? 1'b1: 1'b0;
    end
    else if (fractionalNumber > 0)
    begin
      binaryFractionVar[1] = 0;
      binaryFractionVar[0] = (fractionalNumber < 1)? 1'b0: 1'b1;
    end
    else if (fractionalNumber == 0)
    begin
      binaryFractionVar[1] = 0;
      binaryFractionVar[0] = 0;
    end
  end

  //Convert only the magnitude of the unsigned integer
  else if (fractionType == UNSIGNED)
  begin
    integerComponent = truncate(currentFractionalNumber);
    if (integerComponent == 0)
    begin
      binaryFractionVar[1:0] = 2'b00;
    end
    else if (integerComponent == 1)
    begin
      binaryFractionVar[1:0] = 2'b01;
    end
    else if (integerComponent == 2)
    begin
      binaryFractionVar[1:0] = 2'b10;
    end
  end
endfunction

//convert fractional signed magnitude number to binary

end
else if (integerComponent == 3)
begin
  binaryFractionVar[1:0] = 2'b11;
end
end

//Compute Fractional Component
if (currentFractionalNumber < 0)
begin
  //Get the fractional part and handle negative numbers by
  //negating //them to positive values
  fractionalComponent = -currentFractionalNumber -
    truncate(-currentFractionalNumber);
end
else if (currentFractionalNumber > 0)
begin
  fractionalComponent = currentFractionalNumber -
    truncate(currentFractionalNumber);
end
else if (currentFractionalNumber == 0)
begin
  fractionalComponent = 0;
end

//Perform the conversion
//Keep multiplying the fraction by 2 and make sure that the
//fractional component is non-zero
while (fractionalComponent != 0 && binaryFractionIndex >= -11)
begin
  fractionalComponent = fractionalComponent * 2;
  //If the magnitude of 2*fractionalComponent is greater than 2
  //Then assign a 1 to the binaryFraction vector at the
  binaryFraction index position
  binaryFractionVar[binaryFractionIndex] =
    (fractionalComponent >= 1)? 1'b1 : 1'b0;
  //Update values
  binaryFractionIndex = binaryFractionIndex - 1;
  currentFractionalNumber = fractionalComponent -
    truncate(fractionalComponent);
  fractionalComponent = currentFractionalNumber;
end

  binaryFraction = binaryFractionVar;
end
endfunction

//-----------------------------------------------------------------------------------
//End decimal to binary conversion functions
//-----------------------------------------------------------------------------------

parameter EOF = 32'hffff_ffff;
parameter NULL = 0;
parameter MAX_LINE_CHARS = 100;
integer lineIndexParsed = 0;
//Begin memory initialization block

//File reader modified from –
//http://chris.spear.net/pli/fileio.htm [3]
task initializeLUT;
integer numVertices; //Defined in OFF (object file format) dataset
integer numTriangles; //Defined in OFF (object file format) dataset
integer numPrimitiveVertices;  //3 for triangle, 4 for quad, etc...
integer numLines;

//File variables
integer inputFile; //file handle
integer outputFile; //file handle
integer outputFile2;
integer line;  //line handle

real xCoordinate, yCoordinate, zCoordinate;
integer vertexIndex1, vertexIndex2, vertexIndex3;
begin

//Open input OFF file for reading
inputFile = $fopen(IN_FILE, "r");
//Open output MEM file for writing
outputFile = $fopen(MEM_FILE, "w");
//outputFile2 = $fopen("teapotX30.off", "w");

//Check for existence of file
if (inputFile == NULL)
begin
    $display("Input file %s not found!", inputFile);
end

//OFF FORMAT
//Line Number   Line Content
//Line 1:      [NUM_VERTICES] [NUM_TRIANGLES] [NUM_LINES]
//Line 2:    [X_COORDINATE] [Y_COORDINATE] [Z_COORDINATE]
//...
//Line2+NUM_VERTICES:[X_COORDINATE][Y_COORDINATE][Z_COORDINATE]
//Line2+NUM_VERTICES+1:"4"[INDEX_1][INDEX_2][INDEX_3][INDEX_4]

//Read first line values
line = $fscanf(inputFile, "%d %d %d", numVertices, numTriangles, numLines);
lineIndexParsed = lineIndexParsed + 1;
//$display("%d", numVertices);
while (line != -1) //while (!$feof(infile))
begin
    if (lineIndexParsed <= numVertices)
    begin
        //Use fscanf to read the formatted data from filestream as
        //opposed to sscanf to read data from string
line
= fscanf(inputFile,"%f %f %f",xCoordinate, yCoordinate, zCoordinate);

//Convert the numerical index(10'd85) into the delta theta
//into an angle (30 degrees)
angleOfRotation = indexToAngle(10'd85);
rotate(xCoordinate, yCoordinate, zCoordinate, Y_AXIS, angleOfRotation);
rotate(xCoordinateRotated, yCoordinateRotated, zCoordinateRotated, X_AXIS, angleOfRotation);

fwrite(outputFile, "%b",
{binaryFraction(xCoordinateRotated, SIGNED),
binaryFraction(yCoordinateRotated, SIGNED),
binaryFraction(zCoordinateRotated, UNSIGNED)});

fwrite(outputFile, "\n");
end
else if (lineIndexParsed > numVertices)
begin
    line = fscanf(inputFile, "%d %d %d %d",
numPrimitiveVertices, vertexIndex1, vertexIndex2, vertexIndex3);
end

//Go to next line index
lineIndexParsed = lineIndexParsed + 1;
end

//initialize rest of memory content to 0
for (lineIndexParsed = numVertices+1; lineIndexParsed < (2**ADDR); lineIndexParsed = lineIndexParsed+1)
begin
    fwrite(outputFile, "%b\n", {39{1'b0}});
end
fclose(outputFile);
fclose(inputFile);

//$display("%d", lineIndexParsed);
$display("Vertex Memory Content Loaded!");
//Load memory content
$readmemb(MEM_FILE, mem);
//$display("TESTING");
//$display("%13b %13b ", binaryFraction(1.5, UNSIGNED), binaryFraction(-//1.5, SIGNED));
end
task

initial
begin
    initializeLUT;
end
always @(posedge clk)
begin
    rd_data    <= mem[rd_addr];
    rd_data2   <= mem[rd_addr2];

    if(wr_en)
    begin
        mem[wr_addr]    <= wr_data;
    end
end
endmodule
timescale 1ns / 1ps

module lutDivider(
    dividend,
    divisor,
    signDividend,
    signDivisor,
    clock,
    result,
    coordNumUnderProcess,
    dataValid
);

parameter UNSIGNED = 0;
parameter SIGNED = 1;

input wire clock;
input wire [0:-11] dividend; //10 fractional bits
input wire [1:-11] divisor; //10 fractional bits
input wire signDividend;
input wire signDivisor;

**********************************************************************
* Module Name:       lutDivider
* File Name:         lutDivider.v
*
* Author:   Vahe Robert Jabagchourian
* California State University, Northridge
*
* Creation Date:    June 20, 2011
*
* Description: Look up table for pre-computed reciprocals
*    Inputs to reciprocal lookup are 1.00->3.99
*
* Modifications Made:
* June 20, 2011 Initial module created with lookup values
* imported from Excel
* June 23, 2011 Conversion script modified to produce
* LUT values from
* http://www.easysurf.cc/cnver17.htm#b10tob2
* June 25, 2011 Basic 2 bit integer encoding accounted for in z
* axis
* z = [-1.99:+1.99]
* x = [-1.99:+1.99]
* y = [-1.99:+1.99]
* 400 X 400 pixels / frame
* Added description on how to convert from z
* coordinate to encoded z coordinate
* July 15, 2011 Resolution changed to 511 X 511 with
* x,y,z at 2 bit (sign, integer) and 10 bit
* fractional
**********************************************************************
input wire dataValid;
input wire [2:0] coordNumUnderProcess;
output wire [1:-11] result;

reg [-1:-14] reciprocalDivisor; //14 bits wide
reg [0:-11] dividendReg;
reg dataValidReg, dataValidReg2;
reg [25:0] product, productReg;
reg signProduct, signProductReg;

parameter MEM_FILE = "reciprocalROM.mem";

//Originally 2999 max for mem upper limit
reg [-1:-14] mem [0:6142];

//Memory content generation
initial
begin
    //Read in the reciprocal ROM conent
    $readmemb(MEM_FILE, mem);
end

always @(posedge clock)
begin
    reciprocalDivisor<=mem[divisor[1:-11]-13'b0100000000001];
    //offset = 13'b0100000000001
    //Output Registers (Perspective Division)
    //Perspective Projection
    product<={1'b0,dividendReg}*((divisor[1:-11]==13'b0100000000000)?15'b01000000000000:
    {1'b0,reciprocalDivisor});
    //Orthographic Projection
    //Product <= {1'b0, dividendReg} * 15'b010000000000000;
    signProductReg <= signDividend ^ signDivisor;
    dataValidReg <= dataValid;
    dataValidReg2 <= dataValidReg;
    dividendReg <= dividend;
    signProduct <= signProductReg;
end

assign result = (dataValidReg2)? {signProduct, product[25:14]} :
13'b0000000000000;
endmodule
Appendix B: Modelsim Virtual Signal Commands

```
vsim> virtual signal {((sim:/sinCosROM_TB_v/cosine[1])? (-1.0):(1.0))*
((sim:/sinCosROM_TB_v/sine[0])? (1.0) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-1])? (0.5) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-2])? (0.25) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-3])? (0.125) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-4])? (0.0625) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-5])? (0.03125) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-6])? (0.015625) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-7])? (0.0078125) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-8])? (0.00390625) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-9])? (0.001953125) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-10])? (0.0009765625) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-11])? (0.0004882815) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-12])? (0.000244140625) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-13])? (0.0001220703125) : (0.0)) +
((sim:/sinCosROM_TB_v/sine[-14])? (0.00006103515625) : (0.0))}
```

```
sineFixed
```

```
vsim> add wave /sineCosine_ROM_TB_v/sineFixed
```

```
vsim> virtual signal {((sim:/sinCosROM_TB_v/cosine[1])? (-1.0):(1.0))*
((sim:/sinCosROM_TB_v/cosine[0])? (1.0) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-1])? (0.5) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-2])? (0.25) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-3])? (0.125) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-4])? (0.0625) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-5])? (0.03125) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-6])? (0.015625) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-7])? (0.0078125) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-8])? (0.00390625) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-9])? (0.001953125) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-10])? (0.0009765625) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-11])? (0.0004882815) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-12])? (0.000244140625) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-13])? (0.0001220703125) : (0.0)) +
((sim:/sinCosROM_TB_v/cosine[-14])? (0.00006103515625) : (0.0))}
```

cosineFixed

```
vsim> add wave /sineCosine_ROM_TB_v/cosineFixed
```
Appendix C: Fixed Point Generation Script

<html>
<head>
<script type="text/javascript">
//Fractional Base 10 Decimal to Binary Fixed Point Converter
//Source: http://www.easysurf.cc/cnver17.htm#b10tob2
//Modified by: Vahe Robert Jabagchourian
//Modifications: Comma Separated Quoted Numbers
//Language: JavaScript/HTML
//Directions:  Save code as .html and open with browser
window.onload = generateNumbers;

function convert(val, arrayIndex) {
    var pw3 = parseInt("2", 10);
    var sNumber = parseFloat(stripBad(val));
    var toHex=sNumber.toString(pw3);
    toHex=toHex.toUpperCase();
    document.write(toHex + "<br/>");
}

function stripBad(string)
{
    for (var i=0, output='',
        valid="eE-0123456789.";i<string.length;i++)
        if (valid.indexOf(string.charAt(i)) !=-1)
            output += string.charAt(i)
    return output;
}

function generateNumbers()
{
    /******************************************************************************
    * Insert Comma Separated Quoted Numbers
    * Within the array declaration
    * Replace numbers below with desired list of number
    ********************************************************************************/
    var numbers = Array(
        '0.00613588464761194',
        '0.012271538282635');

    //Added to generate list of numbers
    for (i = 0; i< numbers.length; i++)
    {
        convert(numbers[i], i);
    }
}
</script>
</head>
<body>
<br/> <!--Converted output goes here -->
<!--Copy output into a text editor, remove dashes, periods, align outputs, and paste into a .mem file -->
</body>
</html>