18-545: OpenGL Graphics Acelerator

Andrew J. Lau, Alan X. Zhu, and Nathan L. Wan
{ajlau, axz, nlw}@andrew.cmu.edu
Carnegie Mellon University

December 8, 2010

Contents

List of Figures

List of Tables

1 Introduction

This report discusses an OpenGL graphics accelerator implemented on a Xilinx XUPV5-LX110T FPGA during the Fall 2010 iteration of the Advanced Digital Design capstone course at Carnegie Mellon University. We cover the overall system design, implementation details, and advice for adding further functionality to the system in the future.

2 System Overview

The graphics accelerator can be broken down into five major components:

1. Perl Parser
2. Instruction Assembler
3. Coordinate Transformation Pipeline
4. Rasterization Unit
5. Framebuffer/Display

3 System Specification

3.1 Hardware

- Microblaze processor
- Floating point units
- Coordinate Transformation Pipeline
- Rasterization Unit
- Instruction Cache
- FIFOs
- Framebuffer Controller

3.2 Software

- Perl Parser
- Instruction Assembler (running on Microblaze)

3.3 Development Software

- Xilinx ISE Design Suite 12.2
- Git
- Windows XP
- Ubuntu

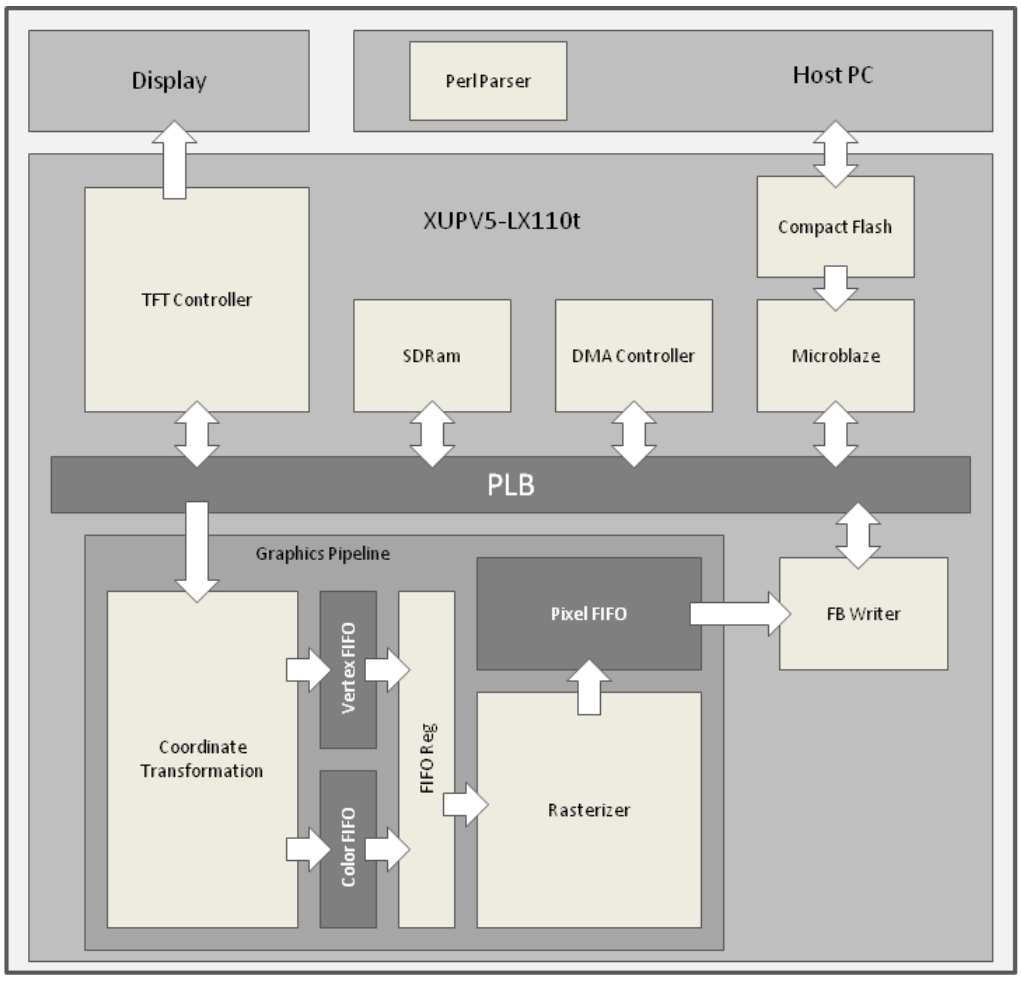

Figure 1: System Diagram

4 OpenGL

4.1 Rationale

We chose to implement an OpenGL pipeline because it is a well known industry standard for 2D/3D graphics. In addition, the pipeline design itself is well documented with numerous existing implementations in both hardware and software. When deciding on the project, an OpenGL pipeline seemed appropriate in scope and difficulty for a semester-long undergraduate course.

4.2 Supported Functions

The pipeline supports the following subset of OpenGL calls:

```
glBegin (void)
glEnd (void)
glVertex (float x, float y, float z)
glColor (float r, float g, float b)
glFlush (void)
```

```
glMatrixMode (enum mode)
glMultMatrix (const float *m)
glLoadMatrix (const float *m)
glPushMatrix (void)
glPopMatrix (void)
glRotate (float angle, float x, float y, float z)
glScale (float x, float y, float z)
glTranslate (float x, float y, float z)
glViewport (int x, int y, int width, int height)
glFrustum (int left, int right, int bottom, int top, int near, int far)
glOrtho (int left, int right, int bottom, int top, int near, int far)
```

4.3 Interface to Pipeline

4.3.1 Perl Parser

In order for the pipeline to accept graphical language calls, a Perl script was written that takes human readable trace of OpenGL calls and outputs a hex representation of the program.

The trace is generated from simple hand-generated C programs. Conditionals, loops, and variables are unrolled and substituted so that what remains is strictly OpenGL calls and data in the form of actual numeric values. These traces correspond to the calls that would be made by a C library such as GLSim.

The main function of the script is to parse each GL call, generate a 32-bit hex value that corresponds to the instruction code defined in the ISA, and output the hex value to a file. If there are argument parameters in the call, each parameter is translated into its hex representation and printed on subsequent lines in the file. The output is an OPNGG hex file (*.gg). Each line of the file is one hex value, either corresponding to an instruction call or one of several 32-bit arguments encoded in hex (single precision floating point , unsigned integer, etc.)

4.3.2 Instruction Assembler

The Microblaze processor is critical for programming the accelerator. To program the instruction cache, the Microblaze reads off a file containing the graphical program and copies in data. The file resides on the Compact Flash and its format guaranteed by the Perl Parser; the assembler does not check the format of the executable. The SDK's XilFATFS is a library that allows the Microblaze to read files on the Compact Flash. For the most part, it follows the C convention for file handling. The instruction BRAM has an interface identical to that of the Xilinx BRAM cores. This is so that a Processor Local Bus (PLB) to BRAM interface core, provided by Xilinx, can be used to allow the Microblaze to write directly into the instruction cache.

5 Instruction Set Architecture

5.1 Specification

The ISA defines a 32-bit instruction word to align with the width of single precision floating point word. The *opcode* field is 8 bits wide, allowing support for up to 256 different routines. This is leaves plenty for room for extending the pipeline to support other some of the 300+ OpenGL calls, as only 17 slots are filled in the current state. The fetch unit addresses the instruction cache at a 32-bit granularity.

Each supported function is translated by the Perl parser into the follow instruction words. The *data* field indicates to the fetch unit to expect a certain number of floats following the instruction word if the *type* field is set.

Bits	[31]	[30:8]	[7:0]
Content	*type*	*data*	*opcode*

Table 1: Instruction word

Function	Type	Data	Opcode
`glBegin (void)`	0	X	00000001
`glEnd (void)`	0	X	00000010
`glVertex (float x, float y, float z)`	1	3	00000011
`glColor (float r, float g, float b)`	1	3	00000100
`glFlush (void)`	0	X	00000101
`glMatrixMode (enum mode)`	0	imm	00010000
`glMultMatrix (const float *m)`	0	16	00010001
`glLoadIdentity (void)`	0	X	00010010
`glLoadMatrix (const float *m)`	1	16	00010011
`glPushMatrix (void)`	0	X	00010100
`glPopMatrix (void)`	0	X	00010101
`glRotate (float angle, float x, float y, float z)`	1	4	00010110
`glScale (float x, float y, float z)`	1	3	00010111
`glTranslate (float x, float y, float z)`	1	3	00011000
`glViewport (int x, int y, int width, int height)`	1	4	00011001
`glFrustum (int left, int right, int bottom, int top, int near, int far)`	1	6	00011010
`glOrtho (int left, int right, int bottom, int top, int near, int far)`	1	6	00011011

Table 2: OpenGL routine to opcode mappings

6 Fetch and Decode

6.1 Instruction Cache

The instruction cache consists of 512 entries of 32-bit words and supports 5 reads and 1 write - concurrently serving the fetch, decode, and instruction assembler through a BRAM controller on the PLB. The BRAM interface is the same as the BRAM specified by Xilinx so that we could utilize the BRAM PLB interface provided by Xilinx. It is currently implemented in logic, with a combinational read. Further work would be needed to move the instruction cache into block RAM, requiring a clocked read.

6.2 Fetch Unit

The fetch unit reads one 32-bit word per cycle from the instruction cache, stalling if the vertex and color FIFOs are full and during matrix operations. A PC register holds the address of the current instruction, and gets incremented based on the type of instruction, jumping over the data in the instruction cache to the next instruction.

6.3 Decode Unit

The decode unit generates control signals for the matrix stacks, matrix multipliers, perspective division, and viewport transformation. It also reads 4 32-bit words from the instruction cache for use in the latter pipeline stages. The decode module also contains registers to hold state for current viewport settings, the current matrix mode, current color (red, green, blue).

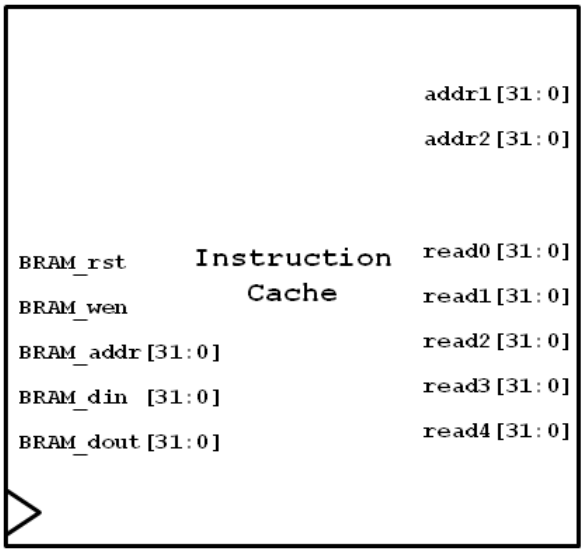

Figure 2: Instruction Cache

7 Matrix Operations

7.1 Matrix Stacks

The pipeline implements two 16x16 matrix stacks - one for Modelview matrices, one for Projection matrices. The bottom of each stack is initialized to the identity matrix by default, with the stack pointer initialized to point at the bottom. The matrix stacks are currently implemented in logic to support combinational reads, but could be moved to BRAM fairly easily.

7.2 Matrix Multiply

Since the coordinate transformation takes relatively short amount of time when compared to the rasterizer, the decision was made to utilize less floating point units and allow a matrix multiply to take 4 cycles per row, or 16 cycles for a 4x4 x 4x4 multiply. Matrices are updated 1 row at a time, with a row update module that utilizes 4 floating point multipliers and 3 floating point adders. The fetch unit is stalled while a matrix multiply occurs, since most of the supported functions utilize the matrix stacks. Because each vertex is multiplied by both the Modelview and Projection matrices, this means that each call to `glVertex` takes 32 cycles on the coordinate transform clock. The decision was made to allow the coordinate transformation to stall during matrix operations because the rasterization unit performs much more work for each set of vertices, and thus would be stalling the coordinate transform pipeline anyway.

8 Coordinate Transformation

The coordinate transformation pipeline utilizes homogenous coordinates to perform the transformations. The rotation, translation, and scaling matrices are pre-calculated by the Perl parser, so they are carried out as matrix multiplies by the pipeline. Each vertex (consisting of 3 floating point numbers corresponding to x, y, z coordinates) that gets passed into Coordinate Transformation goes through a series of transformations so that it ends up in the correct range to be displayed on the screen. The resulting vertices and their corresponding colors are pushed into a dual clocked FIFOs that sit in front of the rasterization unit. A flush signaled by a call to `glFlush` is encoded as a vertex with `0xFFFFFFFF` as its x,y,z coordinates.

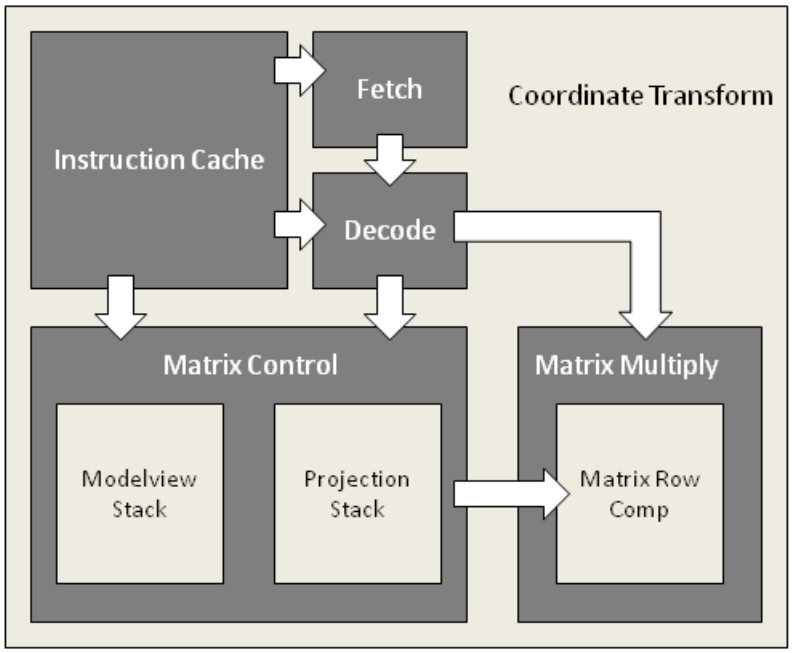

Figure 3: Coordinate Transformation

8.1 Eye Coordinates

The incoming vertex is multiplied by the modelview matrix to produce *eye coordinates*. The modelview matrix is a the combination of the model and view matrices. Since there is no separate camera in OpenGL, the scene must be transformed by the inverse of the view transformation to simulate moving the camera.

$$\begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix} = M_{modelview} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix}$$

Figure 4: Eye Coordinates

8.2 Clip Coordinates

The eye coordinates are multiplied by the projection matrix to produce clip coordinates, in which objects that are not in the viewing frustum are clipped out. The viewing frustum is set by calls to `glFrustum` and `glOrtho` for perspective and orthographic projection, respectively.

8.3 Perspective Division

Perspective division yields coordinates known as *normalized device coordinates*, with their range normalized to $(-1, 1)$ for all 3 axes.

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = M_{projection} \cdot \begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix}$$

Figure 5: Clip Coordinates

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{pmatrix}$$

Figure 6: Perspective Division

8.4 Viewport Transformation

Viewport transformation scales and translates the normalized device coordinates to fit the rendering screen. The results (x_w, y_w, z_w) are passed to the rasterizer. The transformation is given by:

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{w}{2} x_{ndc} + (x + \frac{w}{2}) \\ \frac{h}{2} y_{ndc} + (y + \frac{h}{2}) \\ \frac{f-n}{2} z_{ndc} + \frac{f+n}{2} \end{pmatrix}$$

Figure 7: Viewport Transformation

With some factoring, this is implemented using three floating point multipliers and five floating point adders.

9 Rasterization

The purpose of rasterization is to take sets of vertices from coordinate transform, and figure out which pixels on the screen should be drawn which color so that the correct primitive gets displayed on the screen in the correct position.

In our project, we dealt only with triangles. This means the rasterizer would dequeue a set of 3 vertices from the vertex FIFO and a set of 3 color objects from the color FIFO that the coordinate transform pipeline fills. Each vertex dequeued from the vertex FIFO is a 96-bit value which has has three 32-bit coordinates corresponding to x, y, z. Each color object dequeued from the color FIFO is a 96-bit value which has three 32-bit coordinates corresponding to values for red, green, and blue, respective to the vertex that was also dequeued at the same time. Both the 32-bit colors and the 32-bit coordinates are in IEEE single precision floating point format in correspondence with the OpenGL function calls.

Once three vertex objects and three color objects have been dequeued, the first step is finding the bounding box that the rasterizer must iterate through to decide which pixels correspond to the triangle described by the vertices and thus, which pixels are actually drawn onto the screen. After the bounding box is determined, the rasterizer must scan through each pixel within the bounding box and decide whether it is part of the triangle being drawn or not. This bounding box is 2-dimensional since the display is only 2-dimensional so it only checks the pixel against the x, y values of the vertices.

To determine the color of pixels that should be drawn, the rasterizer uses color interpolation, since only the RGB values for the 3 vertices are given from the color FIFO. The RGB value for each pixel in the triangle depends on where the pixel lies in the triangle in relation to the vertices since only the three vertices have defined colors.

The process that the pixel color is determined by also determines the z value of the pixel being drawn. The z value associates the pixel with a certain depth, which allows the frame buffer writer to determine which pixels belong in front and which pixels are in back if the x and y coordinates overlap. That is they don't need to be drawn since something is already being drawn in the same place in front of them.

When the rasterizer finishes processing a pixel that needs to be drawn to the screen, it packages the coordinates of the pixel and color it that pixel should be drawn into a 96-bit value that gets queued onto the FIFO between the rasterizer and the frame buffer.

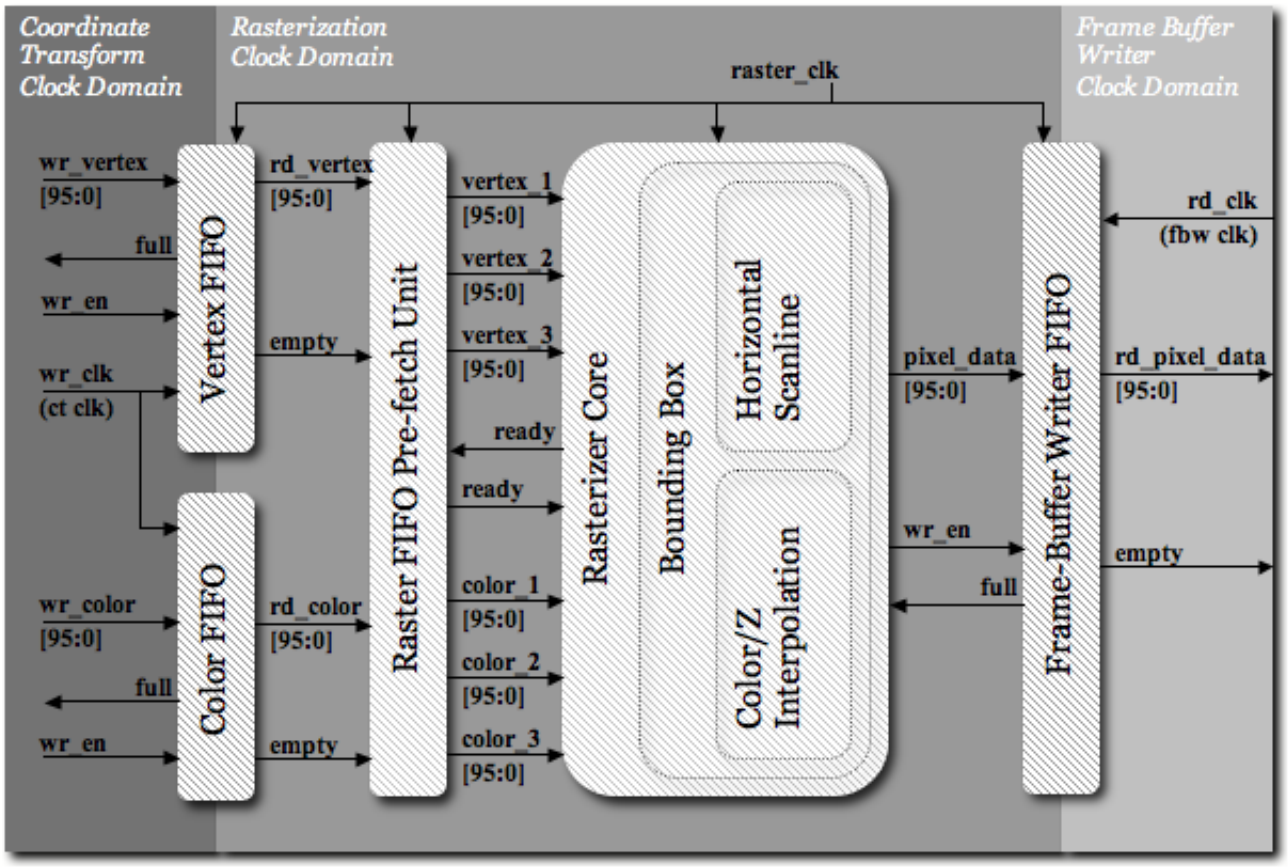

Figure 8: High-level Diagram of Rasterization Module and Related Interfaces

9.1 Pre-Fetching

The high level rasterizer diagram shows the interface with the coordinate transform and frame buffer writer via FIFOs. One critical observation of the rasterizer in terms of implementation is the fact that the FIFOs are only 96 bits wide. This means on any one read, only one vertex and one color object are dequeued. Since the core calculations in the rasterizer operate on three vertices and three color objects, the rasterizer must wait until 3 reads are completed.

One of the implementation decisions made was to put a pre-fetching unit between the rasterizer core and the FIFO. The pre-fetching unit completes 3 reads, and then signals to the rasterizer core that the 3 vertices and colors are ready for processing. While the rasterizer core begins processing that set of 3 vertices, the pre-fetching continues to complete 3 more reads, and then waits for a ready signal from the rasterizer before

starting the process over again. This way, the cost for dequeuing 3 items from the FIFO is amortized over the time it takes for the rasterizer to iterate through the bounding box.

9.2 Bounding Box

The bounding box is actually only an optimization, and not necessary for correctness, although the performance increase is too great to avoid. When the rasterizer is determining where to draw a triangle (3 vertices + color objects), it could potentially scan through the entire screen and for each pixel, determine whether it should be drawn and if so what color and what depth. It could do this for every triangle and the result would be the same as using a bounding box.

However, simply by finding the minimum and maximum x, y values of the entire triangle, The percentage of the screen that the rasterizer must process decreases dramatically, namely to the size of the smallest rectangle with integral coordinates that can contain the triangle being drawn. The minimum and maximum x, y values would still take a good number of floating-point comparators to resolve since there are 3 x, y pairs, but luckily we don't have to use any floating-point comparators, because the calculations for horizontal scanline plus some bit checking can easily tell us the minimum and maximum x and y values.

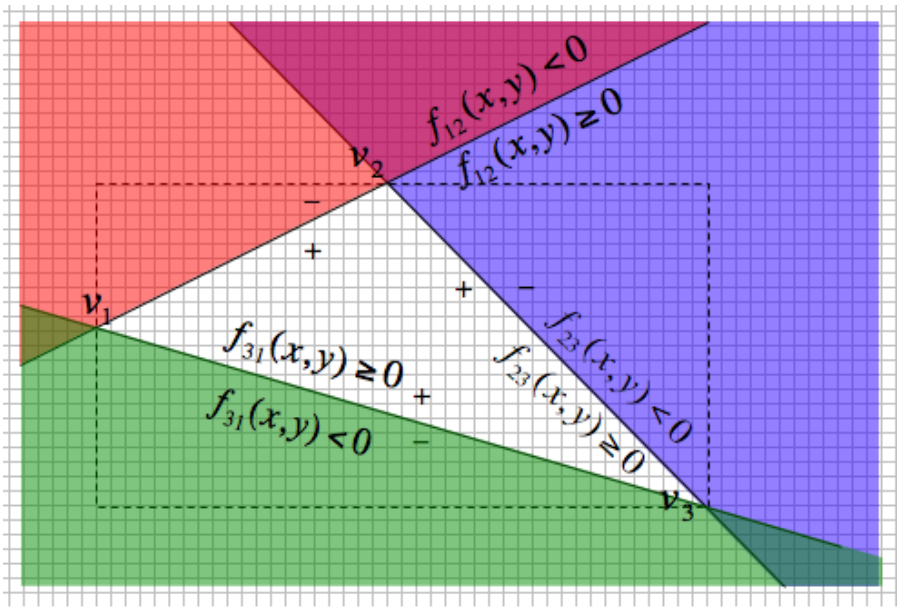

Figure 9: Bounding Box and Half Functions

9.3 Horizontal Scanline

So once given the bounding box, there is still the task of how to determine for each pixel in the bounding box, whether it belongs to the triangle or not and thus should be drawn or not? For each edge of the triangle we define a half-function. A half-function is a line-based function such that, given a pair of coordinates (x, y), the function evaluates non-negative for all input points on a certain line and on one side of that line, and negative for input points on the other side of the line. Since the triangle has 3 edges which correspond to 3 different lines, 3 half-functions can be formulated such that if all 3 evaluate to non-negative values, the point in question is guaranteed to be inside the triangle bounded by the 3 lines of the 3 functions.

For vertices a, b the corresponding half-function is defined as follows:

$$\begin{aligned} f_{ab}(x,y) &= (x_b - x_a) * (y - y_a) - (y_b - y_a) * (x - x_a) \\ &= (x_b - x_a) * y - (y_b - y_a) * x - (x_b - x_a) * y_a - (y_b - y_a) * x_a \\ &= constant_1 * y - constant_2 * x - constant_3 \end{aligned}$$

Since we have a set of 3 vertices, we have 3 such functions, f_{12}, f_{23}, f_{31}. For each pixel (x, y) we must evaluate all three of the functions to determine whether it lies in the triangle or not. With the equation written as is, this equates to 2 multiplies and 2 subtractions per pixel. This is because x_a, x_b, y_a, y_b don't change for any set of 3 vertices so any part of the equations using only those variables need only be calculated once per triangle. Notice however that with some simple mathematical manipulation, we can manipulate the function such that even less computation per pixel is required.

$$\begin{aligned} f_{ab}(x,y) &= constant_1 * y - constant_2 * x - constant_3 \\ f_{ab}(x+1,y) &= constant_1 * y - constant_2 * (x+1) - constant_3 \\ f_{ab}(x+1,y) &= constant_1 * y - constant_2 * x + constant_2 - constant_3 \\ f_{ab}(x+1,y) &= f_{ab}(x,y) + constant_2 \end{aligned}$$

Thus between any pixel (x, y) being processed and the next pixel $(x + 1, y)$ (with respect to the positive x direction), we only have to do one addition operation. We can also do the same for when we are at the edge of the bounding box and starting a new line, meaning the transition from (x, y) and $(x, y + 1)$:

$$\begin{aligned} f_{ab}(x,y) &= constant_1 * y - constant_2 * x - constant_3 \\ f_{ab}(x,y+1) &= constant_1 * (y+1) - constant_2 * x - constant_3 \\ f_{ab}(x,y+1) &= constant_1 * y + constant_1 - constant_2 * x - constant_3 \\ f_{ab}(x,y+1) &= f_{ab}(x,y) + constant_1 \end{aligned}$$

So now we only need to compute the above constants once per triangle, and then for each pixel in the bounding box, if we traverse from the top left of the box to the bottom right, we only need to do one addition for each pixel.

If we exam the constant parts of the half-functions, as labeled above, we observe for every f_{ab} for vertices a, b:

$$\begin{aligned} constant_{1ab} &= x_b - x_a \\ constant_{2ab} &= y_b - y_a \end{aligned}$$

Since we develop half-functions for all 3 vertices, namely $f_{12}, f_{23}, f31$, we essentially have the differences between all possible x and y values. Now we can easily find the bounding box by just doing a little logic on these differences. Recall that floating-point format stores a sign bit at the most significant position. Since we only need to know whether each difference is positive or negative to determine which x and y are smallest and largest, we only need to check the sign bit of these differences. In the implementation, a separate module to handle this logic was developed that sat inside the rasterizer core. The logic in the module is all bitwise operations and ternary statements, which ideally saves significant costs compared to using full-functionality floating point comparators.

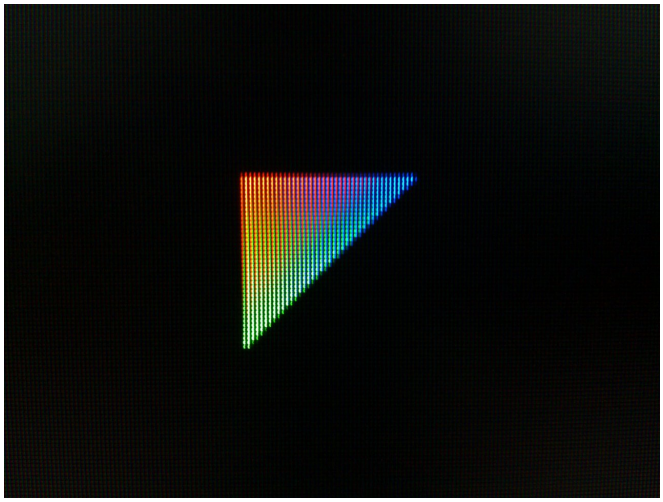

Figure 10: Simple triangle with three different colored vertices and resulting color interpolation

9.4 Color/Z Interpolation

Both color and depth interpolation utilize what are called barycentric coordinates. The barycentric coordinates consist of three constants α, β, γ. They are defined by the following equations:

$$\alpha = \frac{f_{23}(x, y)}{f_{23}(x_1, y_1)}$$
$$\beta = \frac{f_{31}(x, y)}{f_{31}(x_2, y_2)}$$
$$\gamma = \frac{f_{12}(x, y)}{f_{12}(x_3, y_3)}$$

Although we will not explain the math here, the barycentric coordinates have some basic characteristics. First, the barycentric coordinates may hold negative or positive values corresponding to the specific half function ratio they represent. Notice that each of the barycentric coordinates are simply the half function itself, normalized by the value of the function defined at the vertex not included in that specific half function. If the half function evaluates to a negative value, then the corresponding barycentric coordinate will also be negative. The barycentric coordinates also have one important invariant:

$$\alpha + \beta + \gamma = 1$$

This is good for checking correctness during implementations. On every pixel, to check whether it should be drawn, we now check if all three of α, β, γ are non-negative which corresponds to all three unique half functions evaluating to non-negative values. In terms of deciding whether or not the pixel is in the triangle, we could obtain the same result from just evaluating the half functions themselves. The critical aspect of the barycentric coordinates is to help us determine the color of the pixel that will be drawn.

We define the color of the pixel being processed as:

$$c_r = \alpha * r_1 + \beta * r_2 + \gamma * r_3$$
$$c_b = \alpha * b_1 + \beta * b_2 + \gamma * b_3$$
$$c_g = \alpha * g_1 + \beta * g_2 + \gamma * g_3$$

Note that we need to do this 3 times, 1 time for each of red, green, and blue, since each vertex has a corresponding red, green, and blue component. This actually proves to be a computationally intensive step compared to the others, and could really be improved by some pipelining and non-combinational logic. Since we only consider color when the pixel being processed is inside the triangle, all three coefficients will be non-negative when the color values actually matter, thus the color for valid pixels is guaranteed to be non-negative.

In a similar fashion, we use the same constants α, β, γ to interpolate the z value of a pixel being drawn. Again, the z value for valid pixels is guaranteed to be non-negative by the same argument given for color values. Given a pixel that passes as valid in the triangle, its corresponding depth is given by:

$$z = \alpha * z_1 + \beta * z_2 + \gamma * z_3$$

Once we have the depth value of the pixel, we have all the information the frame buffer writer logic needs and so we pack the pixel according to an agreed format with the frame buffer writer and enqueue it onto the FIFO that sits between the rasterizer and the frame buffer writer. In our project, the z-buffer is implemented on the frame buffer logic side, although depending on implementation it can also be done in the rasterizer, likely requiring some caching depending on the precision of the color and the size of the display. The output format is as below:

Bits	[95:89]	[88:80]	[79:74]	[73:62]	[61:54]	[53:48]
Content	`7'b0`	`y`	`6'b0`	`x`	`8'b0`	`red`
Bits	[47:46]	[45:41]	[40:39]	[38:34]	[33:32]	[31:0]
Content	`2'b0`	`green`	`2'b0`	`blue`	`2'b0`	`z`

Table 3: Raster Output

Some things to notice are that the RGB values are truncated to 6 bits each. The intensity/contribution from each color is scaled so that 6-bit 0 is no contribution and 6-bit all 1's is maximum contribution. This is simply following the specifications for the Xilinx TFT controller module. So what ended up happening in implementation is that we took the floating point color value, which was normalized to some value between 0 and 1 by barycentric coordinates, multiplied it by $2^7 - 1$ and then converted it to integer format.

Also note that since our display is only 640 x 480, we do not need the entire 32-bits anymore for our x, y values so we truncate them respectively to the least number of bits that can represent the full range of values within the dimensions of our display, namely 10 and 9 in this case.

Last thing to note is that based on the definitions of α, β, γ, that these constants have to be recomputed with each pixel, and actually, this is probably the bottleneck if any in the rasterizer implementation. As mentioned later on, there is code corresponding to a more efficient pipelined implementation of the rasterizer that was not able to be completed on time, but the source is also included in the project repository.

10 Framebuffer/DVI Controller

10.1 DVI Controller

The reference design for the XUPV5-LX110T contained a DVI controller that displays a frame buffer on DDR2 SDRAM. The core outputs a 640 by 480 video over the DVI port of the board. The core reads a line of the SDRAM frame buffer over the PLB bus and stores it locally in BRAM. Since the frame buffer is any 2MB region in the PLB address space, the frame buffer does not actually have to reside in SDRAM. Any mapped address can be used as a frame buffer, which demonstrates some of the flexibility of the PLB. The DVI controller has control registers mapped on the DCR bus interface. In order for other elements in the system to configure the controller, there is a PLB/DCR bridge what maps the DCR space to a part of the system PLB address space. The design is nearly identical to the TFT LCD[3] data sheet.

10.2 PLB IPIF

The reference design, and the Xilinx norm, is to utilize the Processor Local Bus (PLB) and modules that interface (IF) with it. Using the PLB actually offered great flexibility in the interconnect of cores, even though it restricted the project in some ways, requiring the use of the particular cores. Xilinx provided PLB slave and master cores that simplify bus interfacing for custom peripherals. The documentation is mostly accurate according to the data sheets[4], but looking at the DVI controller also helped. This was difficult to test based on the limited methods for we had no way of simulating a PLB bus.

10.3 Frame and Z Buffers

Each buffer consists of a contiguous 2MB region. The frame buffer region is specified by the TFT LCD datasheet. The Z buffer must be of equal size as it stores a z value for every pixel being displayed. Notice that a change between frame and z buffer consists only of a bit flip. The line and column remain exactly the same for every buffer is 2MB aligned. The same is true to change buffers for double buffer: flipping the next significant bit changes the set of z and frame buffers being edited and displayed. Our flush is combined with a buffer swap operation. To reduce displayed tearing (horizontal scan lines on the screen), the frame buffer being displayed is not the same frame buffer being written to by the accelerator. A flush operation consists of clearing out data in the frame buffer as well the z buffer. A buffer swap is a write into the DCR space that controls the DVI controller. By altering the frame buffer base address, the frame buffers are effectively swapped.

10.4 DMA

The DMA module was a late addition to the project. It is an effort to increase the speed of the flush operation. Previously, the fbwriter would zero out each pixel individually, requiring far more PLB transactions than necessary. The fbwriter can program the flush operation to the DMA, which in turn will flush not only the frame buffer but the z-buffer, since they are contiguous in memory. This meant a flush operation became a series of writes to program the DMA controller.

11 Development Software

11.1 FPGA Tool Chain

The obvious, and apparently only, choice was Xilinx ISE Design Suite 12.2. This is not an endorsement, for it is definitely worth the effort to consider an alternative.

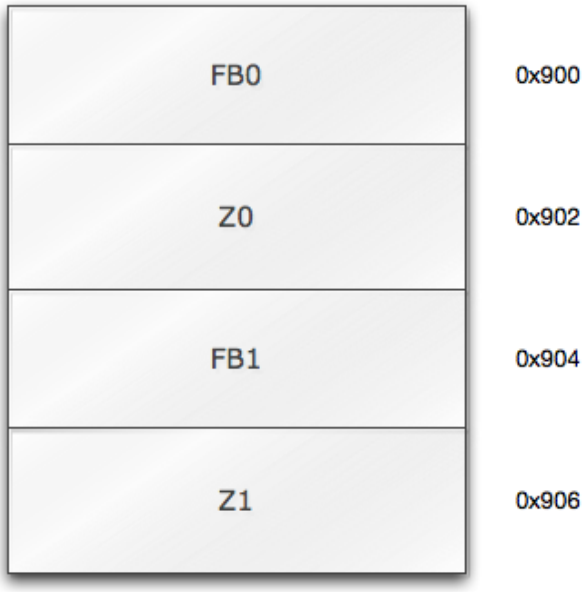

Figure 11: Organization of Buffers

11.1.1 EDK and SDK

The reference design came as a Platform Studio and Embedded Development Kit (EDK) project. To gain familiarity with this duo, consider Xilinx documentation "OS and Libraries Document Collection" and "EDK Concept, Tools, and Techniques"[5]. The EDK allowed us to organize our system level design, make the appropriate connections and build the bitstreams to download to the board. Our graphics pipeline was seen as a custom peripheral that could be imported to projects. This became the crux of our tool chain problems. Discovering the file organization and the methodology of the EDK eventually allowed us to develop a simple routine for getting our descriptions on the board.

The SDK allowed up us to write all the software the Microblaze processor might run. This was used in developing the web-server, when that was part of the plan, and the cf2bram program.

11.1.2 ISE

We used the ISE to manage multiple hardware projects. The core had it's own project, which EDK uses to import as a peripheral. There was also a project that contained all the component to simulate integration testing. Also, each component had a testbench project which was used for simulations. This contributed to our unit testing ability.

11.1.3 CoreGEN

Another component of the Design Suite used to generate cores. It was useful in many ways to construct basic hardware elements; see gripes in other sections. We used CoreGEN to generate all the floating point units, as well as the FIFOs.

11.1.4 PlanAhead

PlanAhead is another component of the Design Suite. We used it to generate most of our ChipScope modules from netlists. The netlists comes from the EDK builds and PlanAhead allows us to add ChipScope modules that sample particular signals on real hardware. It does require a separate bit file and thus, an additional build, but the information is invaluable; this can solve so many of your bugs. Getting this up early would also yields exponential benefits.

11.2 Git

A project without source control is simply unwieldy. Two group members have had relatively enjoyable experiences with Git, and it was simpler to initialize and set-up than svn, the other likely alternative. The distributed style allowed for great flexibility moving central repositories and asynchrony of the group's work style. Our Git repository is online at `https://github.com/lzw545`

11.3 Windows XP

Operating system that existed on the lab machine and had a licensed installation of the Design Suite was fairly convenient. Machine was fairly slow, but static ip address and domain name given by the department made it easy to work remotely.

11.4 Ubuntu

Initially, wanted machine to be primary machine for development. Unfortunately, the usb drivers for the JTAG cable never came into fruition, so board programming was relegated to the Windows machine. Much of the simulation and core development still happened on the Ubuntu box. Also held a git repository that was later moved to github. It was used as an easy way to transfer files between group members because Windows networked file system is unusable.

12 Testing Methodology

Our testing methodology consisted of a combination of simulation, hardware analyzer, and a little bit of trial and error.

12.1 Unit Testing

The majority of our custom components were unit tested by simulations. Using iSim, the included simulation program in the ISE Design Suite, we simulated individual parts of the design to ensure correctness. Most of the testbench projects are unit tests for each component. When changing individual components, running the components testbench is fair smoke test. The PLB was probably most difficult to test, as the Bus Functional Model (BFM) required ModelSim (CMU does not have a license for Windows). Realistically, the only way to test PLB operation was trial and error, and testing after synthesis. Other than that, simulating to specification was the furtherest unit testing possible.

12.2 Integration Testing

We also used simulations in integration testing to ensure the functionality of the pipeline. It was also helpful in ensuring connections were correct. Another tool useful for integration testing was the hardware logic analyzer, ChipScope Pro. It allows us to monitor perticular signals on the synthesized hardware, and we found the most convenient way to setup the trigger modules was through PlanAhead, a netlist and implementation tool. The XMD debugger tool was also very useful in poking around the FPGA once the bitstream was loaded. It can be found in both the EDK and the SDK.

13 Major Design Decisions

13.1 Floating Point vs. Fixed Point

One of the main decisions we had to make was whether to use floating point or fixed point math throughout the pipeline. Fixed point has the advantages of lower board utilization and faster performance, at the cost of precision. Floating point eventually won out because of the higher precision and the fact that we were not really limited by board space, and the CoreGEN'd floating point units could utilize the onboard DSP48 slices which reduces LUT usage even more.

13.2 Synchronization

With the division of work, and division of operational units of the pipeline, we clearly needed an intuitive, robust mechanism to synchronize the RTL. fbwriter needs to operate on the PLB bus, so it conformed to the clock of the PLB bus. Note also that per OpenGG instruction, the rasterization unit has more work than the coordinate transform. The rasterization unit needs at least one clock cycle for each pixel in the bounding box in the process of horizontal scanline; that means it can vary on the size of the triangle. The transformation unit has a fixed number of cycles per instruction; it does not vary on the size of the triangle drawn. Clocking these units independently became extremely convenient. Using clock independent FIFO's, data can be passed across clock domains safely and efficiently. The unit will stall only when the queue is full, which is as good as can be expected.

13.3 External Cores

It soon became apparent we could not do everything. The decision to conform to the Xilinx tool chain stemmed partially from the convenience of provided Xilinx cores and generated cores from CoreGEN. The framework provided by the Xilinx EDK had some advantages. It has a way to run C code a soft processor, the Microblaze. It provides the interconnecting bus structure, PLB. The reference design also contains the Multi-Port Memory Controller[6] (MPMC) which provides an accessible interface for the DD2 SDRAM.

13.3.1 CoreGEN

CoreGEN allows us to generate different types of modules. It is a tool provided by Xilinx included in their design suite and provides blackbox *.v and *.ngc files for usage in your projects. They provide a variety different, commonly used cores, that is, if it is not a custom type of hardware, you can probably find it in CoreGEN. Most of the modules in the reference design actually comes from CoreGEN, including the versatile MPMC. The floating point units used throughout our design were generated to specific parameters. It was useful to specify usage of DSP48E slices instead of logic slices. CoreGEN also allowed us to specify latency, which became extremely important. The FIFO's were also generated by CoreGEN. It was nice to be able to specify what kind of hardware resources each would consume, whether Block RAM, Distributed RAM or built-in units. Note that the built-in units may not simulate as easily; simply generate Block RAM modules for simulation and regenerate the cores with the same project file when moving back to synthesis.

14 Individual Contributions

14.1 Andrew Lau

My primary responsibilities for this project were the implementation of the vertex transformation pipeline and the integration and testing of the various parts of the pipeline. As the only team member who had taken Computer Graphics, much of the graphics design was deferred to me.

The first few weeks of the project were spent working closely with Alan to design the front end of the pipeline, specifically how to use GLSim to talk to the pipeline. This idea was quickly squashed as

we found that GLSim was deprecated and had not been updated since 2002. The next few weeks were spent designing the ISA, looking for suitable floating point units for use in the pipeline, and investigating alternatives to GLSim, which included Mesa. We ultimately decided to write a primitive Perl script that would parse OpenGL calls and assemble the byte code. Some time was also spent generating and checking the CoreGEN'd floating point units. At first they did not seem to be working in simulation, so I spent some time looking for alternatives. A while later, when we revisited the CoreGEN option, we realized that those floating point units had the option to be optimized for latency or for throughput and that the output was being delayed by some 20 cycles and thus lead us to believe they were not functioning correctly.

After the floatin point units were settled, Alan and I worked closely to develop the matrix multiply and matrix stack control modules. These modules implement the 4x4 x 4x4 and 4x4 x 1x4 matrix multiply functionality as well as read in the operands from the correct places. I also wrote the testbenches for these modules, and tested them in simulation to verify correctness.

I was mainly responsible for the coordinate transformation pipeline, as well as the fetch, decode units. The remainder of the semester was spent implementing, testing and integrating this part with the other parts of the pipeline. The coordinate transform was fairly straightforward to implement once I understood everything that needed to be done. The stalling logic and control signals needed to control the matrix operations took the most time to implement. Like Alan, one of the main problems with the coordinate transform was the use of combinational reads for the instruction cache. Since Xilinx BRAMs do not support combinational reads, the instruction cache and matrix stacks ended up being implemented in LUTs rather than BRAM, and thus took up a large portion of the available resources on the board. This was ok for our demo, as we used about 96% of the board resources- any additional floating point units and we would have had to move the the instruction cache and matrix stacks into actual BRAM.

In terms distribution of work goes, I feel like the project was divided along very natural boundaries that ended up dividing the work pretty evenly as well. In the end, everything worked out fairly well, as we were able to get a working pipeline up for the demo, spinning an octahedron at a decent framerate, even though the rasterizer and coordinate transform were only clocked at 8 MHz. We definitely could have had a more interesting demo, but we were pushing right up against the demo time in our last build, and did not have enough time to hand code a more complicated program to run.

14.2 Alan Zhu

My responsibilities consisted of implementing the Perl parser, designing and implementing the matrix multiply and matrix stack units, and designing and implementing the rasterization module.

At the beginning of the project, I worked closely with Andrew in attempting to us the GLSim frontend to communicate to the pipeline. After a short period of successive failures and finally accepting that the already-broken source code had been sitting in the dust for 8 years, we decided do build everything from the ground up, which would start with implementing our own instruction set that would reflect a primitive version of basic OpenGL commands.

This also prompted a hunt for critical components that we would need which would be too much overhead to implement (i.e. floating point ALUs), and a search other alternatives to GLSim (which turned up with only huge libraries we didn't have time to deal with). At this time we also had to consider whether we wanted most of the pipeline to work with floating point or fixed point values. Here, the Xilinx tools proved fairly useful, as they included a Core Generator which could generate adders, subtracters, multipliers, and dividers that were all IEEE 32-bit floating point compliant. In hindsight, if proper fixed point arithmetic units could be correctly written and implemented, it would ultimately be best to redesign our project using fixed point inputs, due to the fact there are optimizations limited to fixed point, as well as the fact that the precision offered by floating point is much more than actually necessary.

As our ISA came together, I started putting together the primitive parser that would allow us to write basic programs that the pipeline could understand. Perl seemed like an ideal language not only because of my familiarity with it, but its nature as a scripting language and its excellent manipulation of regular expressions allowed a simple implementation that could potentially make our project a lot less complicated than necessary. Things like converting decimal floating point to binary floating point and calculating sine

and cosine values helped us push some of the work to the software side, which would definitely help us focus on the more critical parts of the project. I would find myself adding/modifying the perl script regularly whenever we needed to push some abnormally difficult or confusing functions to the side, most notably calculating and arranging various matrices based on OpenGL specifications.

Much of Andrew and I then proceeded to design and implement the matrix control modules, namely those for managing the matrix stack and the matrix multiply functions. This was really our big first step into the project, and so we played a lot of our design decisions conservatively (namely, using as little floating point multipliers as necessary). Working together proved useful as we were able to bounce ideas off each other as well as correct one another's logic. Since time wasn't a luxury, after the basic matrix functionality was completed, Andrew took onthe rest of the coordinate transform pipeline, including more complicated testing procedures with our baseline implementation, and I moved on to other necessary components, namely the rasterizer.

The other bulk of my efforts went in to designing the rasterization unit. The rasterization unit proved to be fairly intuitive to implement, and in hindsight, not overly complicated nor difficult if the high-level idea is clear. I actually thought the harder part included interfacing the rasterizer with the coordinate transform FIFO. Since coordinate transform and rasterization are done in separate clock comains, the state machine that controls the timing is everything. The acutal raster calculations were very easy to debut and pretty much worked after one or two edits to the first draft of the code.

One thing I really regret about this portion of the project is that we decided to use combinational logic, and did not pipeline the rasterizer from the very beginning. The group was pressed for time near the end of the project, so our focus was more on getting the entire system to work, rather than tuning performance without guaranteeing correctness. In hindsight, the straightforward combinational approach makes implementation much simpler and eaiser to understand, but at a substantial cost in performance.

The Xilinx Core Generator can generate floating point units with user-specified latency so combined with a fixed-point pipeline implementation, the rasterizer (normally considered the computational bottleneck) has significant potential for high performance. Of course, the caveat here is that much more logic is required due to the increased control inherent in pipelining and added latency. Also, some optimizations can be achieved simply by having more floating point multipliers and dividers available.

Nevertheless, our rasterizer performed significantly well considering we added little optimization into the design. In fact, it's likely that in the end the rasterizer may not have been the bottleneck, since we also made many design sacrifices in the other components to balance resource utilization. In general, our project was a success, in the sense that any improvements we could have made would mostly be bells and whistles.

As far as contributions go, we all took on separate responsibilities at certain points in the project, and I feel like the work distribution was extremely well balanced. Having been good friends prior to the class, our team dynamic proved invaluable later on. Although we did not know every intricate detail of what each person was doing, we were always able to just bounce design and implementation decisions off one another, as well as occasionally help one another debug. Working together at the same times in lab also helped set aside dedicated time to work on the project. Of course, being ambitious seniors we all had significant coursework outside of the class which taxed our project noticeably.

I was quite motivated after seeing decently successful results of our project to pipeline the rasterizer but given the amount of work I had in other courses after the final demonstration, I was not able to fully implement a pipelined rasterizer. Another part of the rationale was that it probably isn't the current bottleneck in our project. The code repository includes a semi-completed version of the rasterizer and corresponding FIFO management unit that I began to pipeline at the end of our project, but was not able to complete.

14.3 Nathan Wan

My main responsibilities involved the interfaces for the pipeline and the tool set.

I was mainly in charge of any software that ran on the Microblaze processor. When we were considering a web-server front-end for our accelerator, I was working with the lwip library on the SDK trying to get the web server running. This involved learning the SDK interface along with the EDK interface; initially this

was annoying as the design suite was in a transitionaly phase separating functionality between the two. In fact, the EDK still provided some software configuration deprecated functions meant for the SDK, so there was an overlap of functionality.

The current design uses an assembler to get data from Perl Parser to the pipeline. Now the Microblaze runs a program that copies the executable from the Compact Flash to the instruction cache.

I also worked mainly on the fbwriter, the module that took the pixel data output by the rasterizer and wrote it the frame buffer. This was mostly where the PLB interfacing was important. Also, the fbwriter is critical in implementing the flush command. Adding the DMA module was the final part of this.

As the build engineer, I was left with most of the tedious task of building the bit streams. By the end of the project, I was corralling code changes to get into each build. Since each build took so long, it was necessary to make every build count. It was not sensible to force too many group members to learn the Xilinx tool chain, so I was akin to a sacrificial lamb. The other group members were relatively removed from the Xilinx peculiars, in the hopes that they would be more productive that way. For the most part, this was a good decision in dividing the nuisances. I would say that most of my time had been committed to understanding the tool chain. Any guidance in this area would directly benefit me, and probably left me with more time and patience to devote to other aspects of the project.

With my role removed from the OpenGG ISA, I became in charge of the entire system. Most of the system level design decisions came through me since I was in charge of the synthesis and the Microblaze project. Also since I was in charge of most of the interfacing, this was fairly convenient to work on most the system design and the endpoints.

From my perspective, I oversaw most of the progress of the project so that I could create something synthesizable; we wanted to get something new on the board as often as possible. On the final day, with pressure to bring down EDK utilization, I took SRAM out of the project. Unbeknownst to me, SRAM actually contained configuration data necessary for the Microblaze to run. That left us with a broken build pretty close to demo time. The lesson here is to ensure you know exactly what you're cutting out of the reference design, or know as much as possible.

15 Words of Wisdom / Lessons Learned

15.1 Tool Chain

If I recall correctly, no guidance on the Xilinx Design Suite because we were encouraged to look for alternatives. That is, we were not to be restricted to Xilinx tools. Yet, no alternatives were found, none were found by other classmates. Getting a tool chain or build process up early is a huge key to success. This should be a priority for every group. Decide what tools you want to use, pick a lint tool as Xilinx tools are useless for finding inferred nets, etc.

15.2 Stay on Schedule

When setting your schedule, try to parallelize as many things as possible. If something can be started at the beginning of the semester (i.e. does not depend on anything else), you should make every effort to do so. In our schedule, we planned to get the front end done in the first 4 weeks, and start the pipeline after that. What we should have done was start the pipeline earlier, and do that in parallel with the front end. Everything takes longer than you'd expect, and just getting started on it early gives you more time to think through any issues you come across.

16 Status and Future Work

16.1 Status

At the time of demo, we had a working pipeline, with a spinning octahedron running at a decent framerate. However, with no z-buffer or double buffering the image had issues with tearing and the triangles that were drawn were not necessarily the triangles that were closest to the camera. Since the demo, we have added an attempt at both a z-buffer and double buffer. The results can be seen in the following videos, where we are running the same program as on the demo day, except with incorrect z-buffer. It's possible the DMA flushing is occurring too slowly, and that it is actually overwriting actual values, since DMA flush is a lazy flush.

- `http://www.youtube.com/watch?v=kBDY8LMxK2k`
- `http://www.youtube.com/watch?v=75B-Qa6SDFc`

16.2 CPU Integration

The front end interface to the CPU is an area for improvement. Our initial plan was use GLSim, a software implementation of OpenGL, and replace the calls to the functions we planned to implement with code that would issue those calls to our hardware pipeline. When this plan fell through, we opted to use a Perl script that translates hand coded programs that only used the functions we support, forgoing a running C program altogether.

A plan we considered but never got around to implementing involves using the instruction cache as a ring buffer for instructions called from a running OpenGL program running on the Microblaze. We would implement an OpenGL library similar to GLSim, that would translate all OpenGL calls to instructions for our pipeline and put those into the instruction cache. Whenever the cache is full, the Microblaze would have to wait until the pipeline catches up. In this way, we would be able to run actual OpenGL programs, ignoring all calls that we don't support and hardware implementation of calls we do support.

16.3 Shader

A very interesting project would be to work on a graphics shader. OpenGL ES 2.0 relies primary on the shader to replace many fixed function units in the pipeline. Although most of those units are effect units, and unimplemented in our implementation, the project would be interesting as it is the main unit in GPGPU Frameworks. It also requires a microkernel to run in hardware, which would make for a interesting fusion of OS and hardware.

16.4 MPMC

There are many alternatives to the main PLB bus organization we used. Multi-Port Memory Controller (MPMC) has an attached DMA unit that can operate closer to SDRAM, in place of the central DMA unit that can operate on any address. This could have potentially saved some more logic utilization as well as improved performance. The MPMC was also capable of a Point-to-Point connection, where there would only be exactly one master and one slave on the PLB bus. This would also benefit performance, if the need arose. There are other interfaces to the MPMC besides PLB that would work well for our needs. The rasterizer outputs lines of pixels that are actually contiguous in the frame buffer, because every row is stored together. Finding a way to optimize writing of the entire line might also be ideal.

16.5 vsync Timing

One further refinement would be to time buffer swaps to the display. We would wait until the monitor is ready to display something new before changing the buffers. This would make for an even smoother image or scene.

17 Class Impression/Improvements

17.1 Tool Chain Frustration

We feel like the course simply didn't provide adequate support for proper use of the tools provided. Students were left basically to read through the entire Xilinx documentation, and there was no previous code or anyone with much previous experience that could shed light on how to properly set up the tool chain. Because of this, throughout the entire project we found ourselves struggling with the tools simply because we were pretty much feeling around in the dark and just breaking/modifying things until they seemed to work.

The fact that one of our group members basically spent the entire span of the project trying to hack together our tools just to make sure they weren't going to crash on us, says something about maybe what should be done for the class in the future. It was our impression that this class should be about digital design, and that any overhead in learning the tool chain should be, if not minimal, certainly straightforward with proper guidance. There really should be some examples that have been tested by course staff to be known to function properly. When students were stuck with tool chain problems that stalled their entire project, they were left only to Xilinx forums and documentation, which, although usually ended up providing some sort of hack to allow continuation of the project, is not the ideal way to learn things.

17.2 Too much Independence / Lack of Feedback

Another aspect of the class that probably could have been improved is the amount of feedback from course staff about the progress of the project. Status reports were written on a weekly basis, and presentations given every so often. However, it seemed that there was no real response or real evidence that something useful was being done with our reports. It just felt like for the most part that each group knew more about the projects of other groups than the professor or the TAs knew; we feel that perhaps the professor and the TAs should be only second in knowledge of each group's project, with that group of course being first.

18 Credits

- Professor Bill Nace and course staff for running this semester's capstone course, a graduation requirement
- Professor Ken Mai for an initial meeting and steering our project from the beginning
- Google Docs for making collaboration simple, but kinda ugly
- Tumblr for host our blog and recorded missteps

References

[1] Ahn, Songho, *OpenGL Transformation*. 2008, `http://www.songho.ca/opengl/gl_transform.html`.

[2] Capens, Nicolas, *Advanced Rasterization*. 2004, `http://www.devmaster.net/forums/showthread.php?t=1884`.

[3] Xilinx, Inc. *Advanced XPS Thin Film Transistor (TFT) Controller (v2.01a)*. Product Specification DS695. May 03, 2010.

[4] Xilinx, Inc. *PLBV46 Master Single (v1.01a))*. Product Specification DS536. May 14, 2008.

[5] Xilinx, Inc. *EDK Concepts, Tools, and Techniques v11.1*. UG683. December 02, 2009.

[6] Xilinx, Inc. *Multi-Port Memory Controller (MPMC) (v6.01.a)*. Product Specification DS643. July 23, 2010.