AC 2012-4308: INTRODUCING GRAPHICS PROCESSING FROM A SYSTEMS PERSPECTIVE: A HARDWARE/SOFTWARE APPROACH

Mr. Michael Steffen, Iowa State University

Michael Steffen is a Ph.D. candidate in computer engineering and NSF Graduate Research Fellow. His research interests include computer architecture, graphics hardware, computer graphics, and embedded systems, and specifically he focuses on improving SIMT processor thread efficiency using a mixture of custom architectures and programming models. He received a B.S. degrees in both mechanical engineering and electrical engineering from Valparaiso University in 2007.

Dr. Phillip H. Jones III, Iowa State University

Phillip H. Jones received his B.S. degree in 1999 and M.S. degree in 2002 in electrical engineering from the University of Illinois, Urbana-Champaign. He received his Ph.D. degree in 2008 in computer engineering from Washington University in St. Louis. Currently, he is an Assistant Professor in the Department of Electrical and Computer Engineering at Iowa State University, Ames, where he has been since 2008. His research interests are in adaptive computing systems, reconfigurable hardware, embedded systems, and hardware architectures for application specific acceleration. Jones received Intel Corporation sponsored Graduate Engineering Minority (GEM) Fellowships from 1999-2000 and from 2003-2004. He received the best paper award from the IEEE International Conference on VLSI Design in 2007.

Prof. Joseph Zambreno, Iowa State University

Joseph Zambreno has been with the Department of Electrical and Computer Engineering at Iowa State University since 2006, where he is currently an Assistant Professor. Prior to joining ISU, he was at Northwestern University in Evanston, Ill., where he graduated with his Ph.D. degree in electrical and computer engineering in 2006, his M.S. degree in electrical and computer engineering in 2002, and his B.S. degree summa cum laude in computer engineering in 2001. While at Northwestern University, Zambreno was a recipient of a National Science Foundation Graduate Research Fellowship, a Northwestern University Graduate School Fellowship, a Walter P. Murphy Fellowship, and the EECS department Best Dissertation Award for his Ph.D. dissertation titled ”Compiler and Architectural Approaches to Software Protection and Security.”

Introducing Graphics Processing from a Systems Perspective: A Hardware / Software Approach

Abstract

Typical courses in computer graphics focus mainly on the core graphics rendering algorithms and software interfaces - hardware and system-level issues are addressed, if at all, through classroom lectures of industrial case studies. We have recently introduced a senior technical elective which introduces graphics processing from the perspective of the software developer, hardware architect, and system integrator. Towards this end, lecture topics are designed for students with no computer graphics background, and focus on solving specific computing problems using skills learned from a variety of computer engineering courses (e.g. digital logic, computer architecture, software design, embedded systems). As part of the laboratory component, students are presented with a series of bi-weekly design challenges that are geared towards implementing a particular module in the 3D graphics pipeline (with both hardware and software support) using an FPGA-based hardware prototyping platform. Although the main focus of the labs is on architectural design, hardware implementation, and hardware / software verification; each assignment also involves both a functional correctness as well as an optional performance optimization component. Only by analyzing the interactions between the graphics application, middleware, architecture, and logic levels can the performance optimization goal be achieved. Each subsequent challenge builds upon those previous, such that by the end of the semester students will have designed and implemented a fully-functional OpenGL-compliant graphics processor, capable of running significant applications. The course was introduced in the Spring of 2011 and the results from the final course project indicated that many of our intended learning objectives were met; student feedback was also positive.

I. Introduction

Computer graphics is becoming increasingly important to modern-day computing. A wide range of computation is being pushed to GPU processors (displays, 3D graphics, image processing, parallel computing).[4] Originally implemented as a fixed-function processor for 3D graphics, GPUs have evolved into a multi-purpose programmable parallel processor. Mastery of modern computer graphics requires more than the knowledge of a 3D graphics API. Today, graphics developers need to understand the underlying GPU architecture, both its strengths and limitations, as well as the interaction between the CPU and GPU, in order to write efficient high-level code. Seniors in computer engineering at Iowa State University (ISU) are exposed to concepts in device interfacing and hardware/software optimization through multiple classes in software development, computer architecture, digital logic and signal processing. A course that focuses on graphics processing and architecture has the potential to nicely tie together several instances of these concepts in an integrated environment.

At ISU we have created a senior elective class for teaching graphics processing. While this class is offered as an elective in the computer architecture focus area, course topics are introduced from

the systems perspective, requiring students to use knowledge across multiple computer engineering subfields. For students to acquire a deep understanding of the complexity required to support modern computer graphics, weekly laboratories require students to implement specific graphics processing components, spanning the entire system implementation from software API to the specific GPU architectural components. These architectural components implemented in lab are loosely derived from the conventional OpenGL 3D rendering pipeline. Each lab assignment expands the processing pipeline, refining a stage from the software domain to the hardware domain. By the end of the semester, students are able to run OpenGL applications that are rendered using their graphics processing system. Students are able to prototype their OpenGL pipeline by synthesizing their architecture to an FPGA. By using an existing standard (OpenGL), the implemented system allows for existing graphical applications that would conventionally run on a workstation to also be rendered using the students' designs running on the FPGA.

To attract a wider range of students, no computer graphics prerequisite is required. Due to this, part of the class time is spent on teaching basic computer graphics principles. To differentiate this class from a conventional computer graphics course, we make the trade-off of breath versus depth of knowledge, spending more time on understanding the entire system perspective for the concepts versus presenting a wider range of more advanced algorithmic concepts in the computer graphics domain. Since the entire system perspective is covered from software to hardware design, prerequisites for the class requires strong software and Hardware Description Language (HDL) programming skills, limiting the students to upper-level electrical and computer engineers.

This class was first introduced in Spring 2011 and is being taught again in Spring 2012. The class is offered as a 4 credit (3 hour lecture + 3 hour lab per week) senior elective course. The students' completed laboratory assignments surpassed our expectations in terms of creativity from both the software and the hardware design perspective.

II. Laboratories

A total of 7 laboratory assignments were given during the semester. Students worked in the same group of 3 to 4 students for all laboratories and had two weeks to complete each lab. During the two week time period, students have two 3-hour lab sections, four 2-hour teaching assistant hours in the lab, four 2-hour instructor office hours and an on-line forum related to the course that was monitored frequently by the teaching assistant and instructor. Groups were formed on the first day of class and allowed students to form their own groups with the exception that all students in a group must be in the same lab. To aid in creating groups, students evaluated their experience in three categories: software development, hardware design and computer graphics. Students were advised to form well-balanced groups, since all laboratory assignments require some aspect of all three categories. We did allow teams to re-organize during the semester, based on student feedback.

The first laboratory served as an introduction to the software tools and hardware infrastructure that was common across all labs. The remaining 6 assignments implemented individual pipeline stages for a fixed-function OpenGL-compliant 3D rendering pipeline. We refer to the laboratory assignments as Machine Problems (MP), as they each included significant software and hardware

components. Throughout all the MPs, students were responsible for implementing parts of the software driver for OpenGL API functions, hardware architecture for the OpenGL pipeline, HDL implementation of the architecture, culminating in running the complete system using a workstation connected to an FPGA-based platform. In addition, students were also required to implement their own OpenGL applications for additional debugging and verification. Each MP also had a bonus for either the fastest design or most creative, depending on the requirements for the MP. Each group was only allowed to receive the bonus two times, allowing for each group a chance to receive the bonus.

The course was organized such that by the start of each MP, students had been introduced to the functional requirements for each pipeline stage, as well as a mathematical foundation. Students were allowed to utilize the provided algorithmic solution or create their own, but were required to meet the functional requirements for the component. Since all the MPs build on the OpenGL pipeline, students were provided with the previous labs solution and encouraged to use it for the next MP. This prevented students from falling behind and also simplified debugging of subsequent labs, since the entire class was working from the same basic codebase. To prevent students from falling behind with regards to the course material, each group was required to demonstrate each MP, no matter their degree of completeness. Additional in-person mentoring was provided for groups that had difficulties in understanding key concepts; however, implementation errors were much more common than conceptual errors.

II.A. Laboratory Infrastructure

To allow students to focus more on the specific components being taught in each MP, an initial hardware/software framework was provided. Students then expanded the capability of the framework in each of the MPs. An overview of the provided framework is shown in Figure 1, which supports the following capabilities: source code for an OpenGL API (absent components the students will implement), communication protocol between the PC and FPGA board (software library and hardware interface), FPGA-based hardware Network-on-Chip, FPGA-based DDR memory system, FPGA-based hardware DMA controller and FPGA-based DVI display interface hardware. Providing these capabilities allowed the students to spend their time on implementing the pipeline stages, without worrying about all of the details of building a complete hardware/software system from scratch (which is the focus of other courses at ISU).

The open-source project *GLIntercept*[6] was used as a baseline for our OpenGL driver. *GLIntercept* is an OpenGL API function call logger that replaces the workstation's native OpenGL driver. For every OpenGL API function call, an entry in a log file is created. *GLIntercept* also loads the native driver and forwards all API calls to the workstation's GPU (in our case, an NVIDIA GeForce GTX 480 GPU,[3] so that the application will be correctly rendered to the screen. *GLIntercept* was modified to send instructions to the FPGA GPU reference design. Each OpenGL API function call has a function in *GLIntercept* that allows for custom messages to be sent from the workstation to the FPGA framework. By rendering a scene both natively on the workstation as well as using the FPGA-based framework, students are able to visually inspect their solutions to detect discrepancies from the "correct" workstation GPU implementation.

To aid students in implementing driver code and sending instructions between the workstation

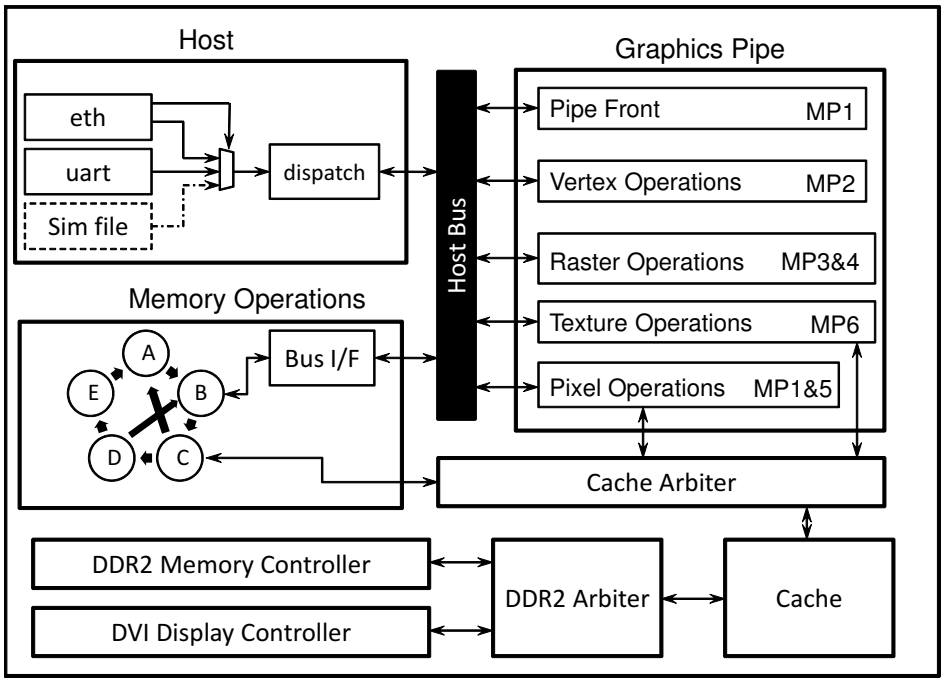

Figure 1: FPGA architecture framework provided to students. Students were responsible for implementing code in the graphics pipe sub-unit.

and the FPGA-based GPU implementation, a protocol was provided along with source code implementation and a corresponding HDL code for the on-chip packet forwarding. The protocol sends messages of 32 bits and must be formated to meet the protocol outlined in Figure 2. This protocol allows for data to be sent from the workstation OpenGL driver to multiple sub-units memory mapped as registers. The `Length` field specifies the number of 32-bit packets that follow the starting packet and allows for custom data formates to be sent to the sub-units. The `OpCode` allows the sub-unit to support multiple instructions and distinguish between them. The `Address` field specifies which of the sub-units to send the messages too. The provided network-on-chip/forwarding hardware interprets the incoming messages and forwards them on to the appropriate sub-unit registers. The physical connection between the workstation and FPGA uses either Gigabit Ethernet or RS232 serial UART. Selecting between the two methods is possible using Linux environment variables, and is fully supported by the framework. Having multiple communication speeds between the workstation and FPGA-based GPU implementation provided additional performance debugging methods.

X	Length																			OpCode								Addr.			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MSB																															LSB

Figure 2: The protocol for sending data from the workstation to the FPGA. This format represents the first 32-bit message in a packet.

For prototyping the GPU architecture a Xilinx Virtex-5 FPGA XUPV5-LX110T board was selected.[5] The XUPV5-LX110T board has 256MB of DDR2 memory available for a frame buffer and additional data storage. The Virtex-5 FPGA uses two clocks for system components and another one for the graphics pipeline. The graphics pipeline component (where student code was implemented) uses a 100Mhz clock; consequently, students were required to meet this timing requirement for all of their hardware modules. The rest of the system runs at 200Mhz. To simplify the display hardware and clocking, a fixed resolution of 1280x1024 was implemented using the DVI display interface. The memory address for the frame buffer is set through a memory-mapped register, allowing for students to switch between different frame buffers. All memory addresses provided to students are in a virtual address space. The FPGA DDR2 memory uses a word size of 128-bits, equivalent to 4 pixels of 32-bits each.

To simplify the access to the DDR2 memory module, a cache and address translator was provided. The virtual address word size was 32-bits wide, allowing for each unique address to represent a single pixel. The cache provides the address translation by providing two different port sizes. A total of 8 megabytes is allocated for each frame buffer. While the total required memory is 5 megabytes for a tight compaction of data, the equation for computing a pixel address for a tight compaction is: $address = (linewidth) * Y + X = 1280 * Y + X$. Since 1280 is not a power of two, a hardware multiplier would be required. Assuming the width of the display to be the next power of two, allows the base design to use only a shifter to compute the address, rather than a multiplier (the display hardware knows to stop reading at 1280 and not continue to 2048). By padding the framebuffer, we reduce the hardware complexity and simplify the math, at the expense of using more memory. In our framework, reconfigurable FPGA resources are more limited than DDR2 memory.

All but the first MP assignment expands the functionality of this framework. Specifically, MP-1 through MP-6 adds to the graphics pipeline by adding sub-units. Each sub-unit uses the same interface protocol allowing for new sub-units to be inserted between other sub-units. While most of the MPs focus on a specific sub-unit, some assignments span across multiple sub-units. Figure 1 shows the basic OpenGL pipeline sub-units implemented and the corresponding MP assignments.

II.B. Machine Problem 0 (MP-0)

The goal of the first MP was to allow students to become familiar with their new team members and the lab's hardware/software infrastructure. We also wanted to give students a refresher in HDL design methodologies. To ease the transition to future labs, MP-0 required students to implement matrix-vector multiply-accumulation on the FPGA. A constant 4x4 matrix along with 10000 4x1 vectors are used in computing the 4x1 sum. Students were provided with sample code that provided all the initial data stored in FPGA memory and an interface for sending results to the workstation. The MP-0 lab handout walked students thought the process of understanding the provided framework code and how to compile, synthesize, program, and run the code on the FPGA. After running the code, students realize that the output from the provided framework code is not the correct solution (since the vector sum hardware is not implemented). The teams were given two weeks to implement the hardware to perform the matrix-vector multiply-accumulation,

requiring students to read the vector data from on-chip memory and perform all math operations for the matrix vector multiplication and accumulation. As the bonus criteria for this MP is performance, a cycle counter is included in the HDL code to count the number of cycles required for all operations to be performed. Students indicate when all operations are finished by setting a register that stops the counter and triggers the output results to be sent over UART to the workstation for verification. Students were not allowed to modify the provided code with the exception of changing the on-chip (blockram) memory width.

II.B.1. MP-0 Student Results

Common solutions for this problem used 4 multipliers in parallel and then summed the results for computing one of the values in the vector. A Finite State Machine (FSM) is implemented for loading data out of memory and feeding the multipliers and directing the outputs to the right register locations to compute all 4 values of the vector. The initial state of the FSM reads the 4x4 matrix values and stores them in appropriate registers that the multipliers will access. The final two states of the FSM oscillates between loading the vector data from memory and performing the multiplications.

Teams competing for the bonus increased the number of multipliers to perform more operations in parallel. One such implementation added as many multipliers as permitted by the resources on the FPGA. These implementations also utilized an FSM for reading data from the FPGA memory and a separate one for performing the multiplications.

II.C. Machine Problem 1 (MP-1)

The second MP is the first assignment in developing the 3D OpenGL rendering pipeline. Students are also exposed to the framework that they will be using for the remainder of the class. Since the provided framework is very large and complex, a majority of the assignments is understand the capabilities of framework and how it is all implemented. The MP-1 handout guides students through all the software libraries and the provided HDL code. In addition, students are also introduced to all the protocols that they will have to use. The lab handout asks students to evaluate different parts of the framework and describe how it works in their report.

In addition to understanding the provided framework, students are required to implement a portion of the driver and HDL code. The driver implementation requires students to implement the API function that clears the entire screen. Implementing the clearing instruction requires students to send messages from the workstation driver to the FPGA framework memory controller to write zeros to a range of memory addresses that correspond to the frame buffer address. Correct implementation requires 4 32-bit messages to be sent to the FPGA, with the challenge being understanding how to use the software driver, workstation to FPGA protocol, functionality of provided HDL sub-units and the FPGA memory space.

The second portion is to draw pixels to the screen. The driver code for sending OpenGL vertices from the workstation to FPGA was provided to students. The format for sending vertices to the FPGA is done using multiple lists queues outlined in Figure 3. The advantage for of using list queues reduces the amount of data that is required to be sent between the workstation and FPGA

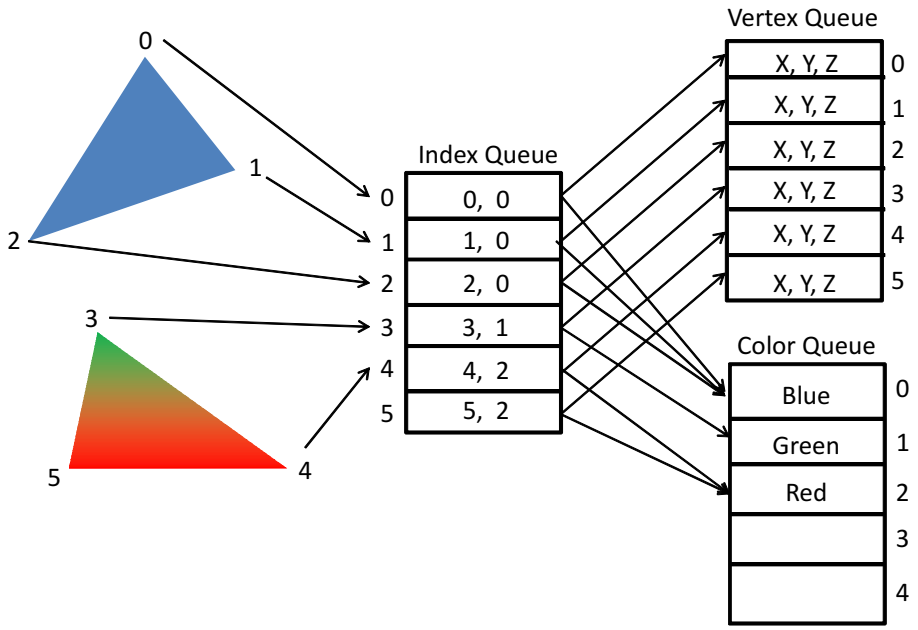

Figure 3: Format of how vertex attributes are sent to the FPGA and then stored in memory. The index queue stores address for the vertex and color queue for each vertex.

for large complex scenes. Students had to implement the HDL code to read from the different list queues to get the X, Y locations for the vertices and the correct color for each vertex. Once the X,Y vertex location and color are known (no transformations were required for this lab) the framebuffer memory address is computed. Once the address is known, students interface with the provided memory controller to write the color value to the address in memory. To verify their code, students also had to implement their own OpenGL application that drew pixels. The bonus for this MP is how creative students can be in creating an application using only pixels. Students then had to demo their OpenGL application being rendered with the FPGA.

II.C.1. MP-1 Student Results

Two similar solutions for reading vertex data were implemented. One method implements the process of reading the data from the queues as a pipeline. A counter is used to increment through the index queue and feed the pipeline with the data results from the index queue. The second stage uses the data in the first pipeline stage as addresses to read from the remaining index queues. The output from these queues is then used for computing the memory address.

The second method implements a FSM. The different stages of the FSM reads values from the different queues. While the FSM transitions between states in order decreasing performance, students find implementing state machines easier then implementing multiple pipeline stages.

A sample OpenGL application developed by students drawing individual pixels is shown in Figure 4. Students implemented a fully functional 2D video game with multiple moving characters. Other student applications drew their name or other basic shapes.

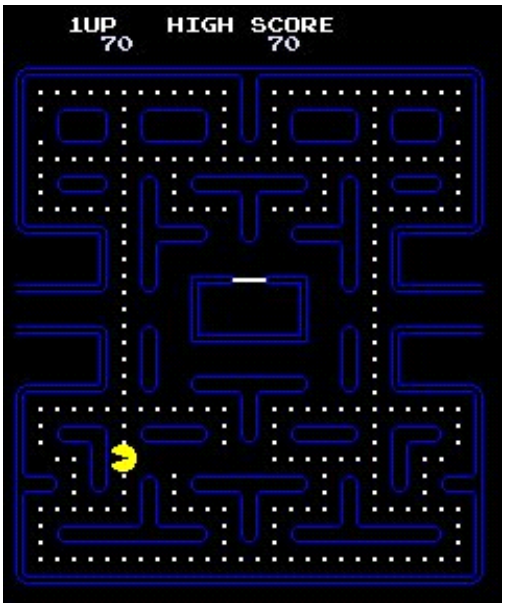

Figure 4: Sample student OpenGL application that draws a 2D video game using pixels.

II.D. Machine Problem 2 (MP-2)

The mapping of 3D points defined in the OpenGL coordinate system to the 2D screen is accomplished through the application of multiple transformation matrices. OpenGL uses 2 4x4 matrix transformations, one vector division and a 2x2 matrix transformation for converting 3D coordinates to 2D screen coordinates.[1] The transformation process is shown in Figure 5. MP-2 requires students to implement the hardware for all of these matrix multiplications and division. No software driver implementation is required, due to the complexity of the hardware implementation. Students are also required to describe how the different matrices are sent to the FPGA in their lab writeup. Due to limited resources on the FPGA only 4 64-bit multipliers, a single 64-bit divider and half of the available DSP slices could be used. Since floating-point operations are not efficient on FPGAs, fixed-point notation is used. Conversion to fixed-point notation is done inside of the workstation driver code.

The fixed-point notation for input vertices is Q32.32. The two 4x4 matrix transformations are also

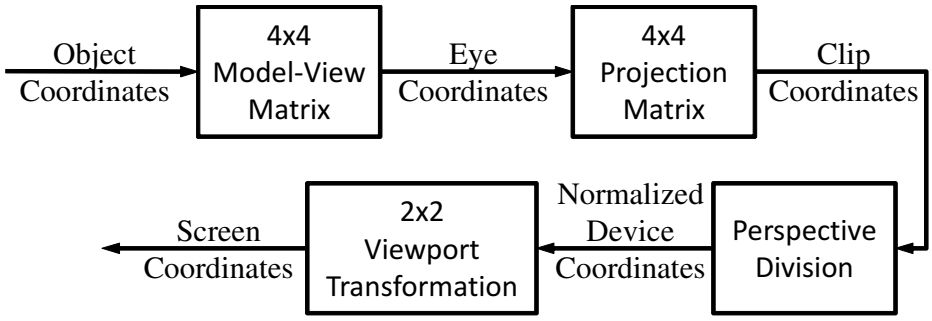

Figure 5: Transformation process for converting 3D OpenGL coordinates to screen coordinates.

stored in Q32.32. The division operation is used for inverting vertex values requiring no fixed-point notation. The 2x2 matrix transformation is composed of 11-bit unsigned integers and the final vertex output corresponds to the pixel location requiring two 11-bit unsigned integers for a 1280x1024 resolution. When designing their architecture for MP-2, students were required to understand the precision requirements needed for each multiplication operation to implement the correct size multiplier. In this lab, the bonus is a function of performance divided by area. Performance is measured for the cycles required to render a specific benchmark application divided by the number of FPGA LUTs used.

II.D.1. MP-2 Student Results

Students implemented the first two matrix transformations in similar method as MP-0. The four 64-bit multipliers are used for the transformations and the FSM is slightly modified to perform the 4x4 matrix vector multiplication twice. The results after the two 4x4 matrix transformations are written into a FIFO. The FIFO is drained by a second FSM that reads one value every 3 cycles. The 3 cycle delay allows for 3 of the vector results to go through a pipelined divider implemented using the FPGA DSP slices. Results from the divider are regrouped back into a vector and stored into a FIFO. The final transformation is implemented using 4 11-bit multipliers implemented in a pipeline method, since only 4 multipliers are needed for this transformation.

II.E. Machine Problem 3 (MP-3)

The rasterization pipeline stage is split into two labs, MP-3 and MP-4. In the rasterization stage, predefined OpenGL primitive types are pixelized such that every pixel enclosed in the primitive type is identified. All OpenGL primitive types are defined such that they all can be composed of 3D triangles. For example, a quad can be composed of 2 triangles. By having the basic primitive building block be a triangle, a single GPU rasterization hardware primitive can be implemented. Since the OpenGL API supports multiple primitive types, the process of composing primitives using individual triangles must be implemented in the driver or in hardware. Also, due to the fact that different primitives can reduce the number of vertices required to be passed from the workstation to FPGA, we implement primitive composition in hardware. For example a quad requires only 4 vertices whereas implementing a quad using 2 triangles requires 6 points (2 points being replicated).

MP-3 requires students to form individual triangles for rasterization from the different primitive types supported by OpenGL. Since OpenGL uses primitive types that are easily composed of triangles, creating the individual triangles can be performed by pipelining vertices and using some logic to select vertices from different stages of this pipeline. The second requirement for MP-3 is to generate the pixels that fill the triangle. Calculating the color for each pixel requiring interpolation using the vertex data is done in MP-4.

Students were provided with an algorithm and HDL code that checks if a pixel defined in X,Y screen coordinates is inside of the triangle being rasterized. To simplify the hardware requirements the algorithm only worked for testing pixels sequentially and sequential pixels must be adjacent. The starting pixel is one of the vertices of the triangle that is by definition inside the triangle. For students to check if a pixel is inside of a triangle, students must send commands to

the provided HDL rasterization code. The commands are POP_MOVE_LEFT_CMD, POP_MOVE_RIGHT_CMD, MOVE_RIGHT_CMD, MOVE_LEFT_CMD and PUSH_MOVE_DOWN_CMD. Using the PUSH commands allows for students to save their current location and return to it using the POP command. The push buffer is only a single entry deep.

II.E.1. MP-3 Student Results

To rasterize a triangle, all students implemented a FSM that used a zig-zag rasterization pattern. A sample zig-zag pattern is shown in Figure 6. The logic for the FSM is as follows. First the current state is pushed. Then the Left direction is checked. The state stays in the left direction until the pixel is no longer in the triangle. The state is then popped, re-pushed and the right direction is checked. Once the pixel is out of the triangle the state is popped again and moved down. One exception to this is when the pixel is moved down and the pixel is not in the triangle. In this case both directions must be searched until the pixel is found to be inside of the triangle or the triangle bounding box is reached.

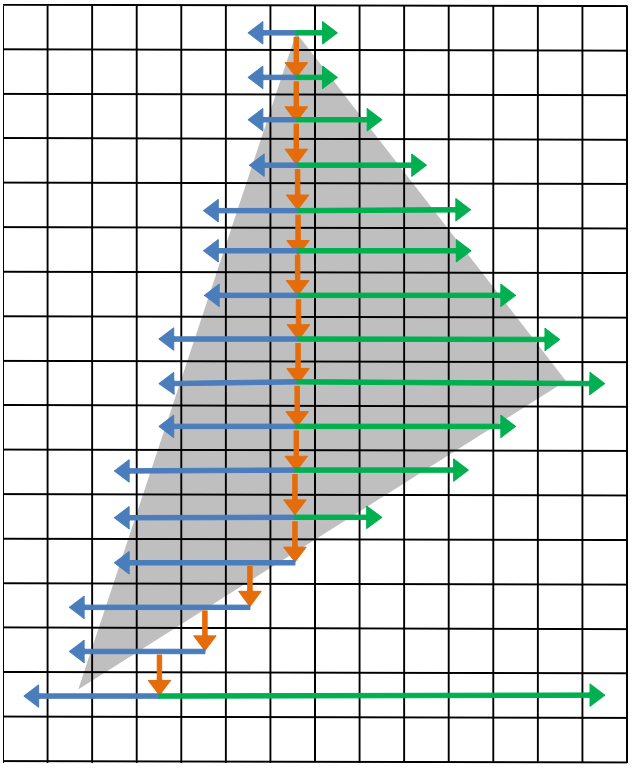

Figure 6: A rasterization zig-zag pattern implemented by students to identify all pixels inside of a triangle.

II.F. Machine Problem 4 (MP-4)

The second stage of rasterization is to interpolate each pixel color from the three triangles vertex colors. Pixel color can be calculated in parallel with the rasterization check from MP-3. Students were required to implement the hardware unit that takes the same commands from MP-3 zig-zag rasterization commands and generate the pixel colors. The provided HDL code provides students

with each color channel DX and DY values for the current triangle along with the vertex color for the starting vertex in the rasterization processes. Since the rasterization commands are sequential pixel offsets, students were required to subtract or add to the vertex colors based on the commands being inputted to the unit. Students were also required to implement their own OpenGL application that uses multiple primitive types to draw 3D objects that will also be used to test their color interpolation.

II.G. Machine Problem 5 (MP-5)

An important operation for drawing 3D objects is depth testing, which determines if a pixel for a triangle should be written to the framebuffer. While there are multiple depth testing functions, the most commonly used one is to test if a pixel for a triangle is in front of the current pixel in the framebuffer. If the recently generated pixel is in front of the one currently in the framebuffer the pixel should be written to the framebuffer. If the current pixel in the framebuffer is in front of the new pixel, the new pixel is discarded. To keep track of the depth for each pixel in the framebuffer a depth buffer is allocated in DDR2 memory, each pixel in the framebuffer has a corresponding address in the depth buffer. When depth testing is enabled, each pixel generated from rasterization must read the depth value from the depth buffer. Then, test the depth buffer value against the depth of the generated triangle using a specified function. The function used is set through the OpenGL API. If the new pixel passes the depth test, both buffers are written to with the new pixel values (color and depth).

MP-5 required students to implement the HDL code for reading depth values from the depth buffer and testing them against the depth function. All eight depth functions were required to be implemented and students were also required to write OpenGL applications to stress test their design as well as prove correct functionality. It is important to note that the XUPV5-LX110T FPGA board we use only allows for a single DDR2 memory controller. As depth checking requires the hardware to read from the depth buffer, perform the test, write to the depth buffer, and write to the framebuffer before subsequent pixels can be processed, students were not able to fully pipeline this stage. Because of this limitation students are also required to implement methods for measuring the performance overhead of using the depth test.

II.G.1. MP-5 Student Results

Due to the long latencies of reading from the DDR2 memory with only a single memory controller, most students implemented another FSM for this MP. The first step is to read from the DDR2 memory to get the depth value. Once the depth buffer is read, they evaluate the depth function. The eight depth function are implemented in a similar method as a conventional ALU. All eight functions evaluate the results and a mux is used to select the results for the depth functions specified by the OpenGL API. If the depth function passes, the depth value and color value are written to memory.

The second part of this MP was for students to implement OpenGL code to test their depth function. One innovative application is shown in Figure 7. For each horizontal bar, two primitives are drawn using different colors. The first primitive is drawn at a constant depth value of zero using the color black. The second primitive uses a sin wave for the depth and sets the color based

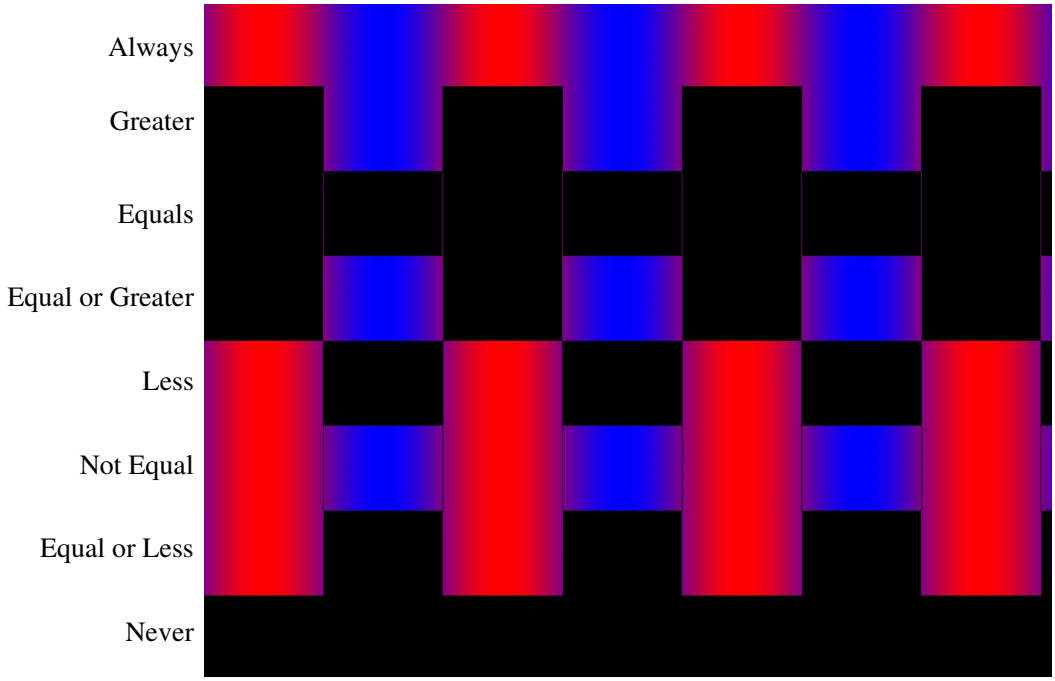

Figure 7: OpenGL application used to test all 8 depth functions. Each horizontal bar uses a different depth function test. The black color is a plane at depth 0 and the red and blue colors are a sin wave with depths from -1 to 1.

on the depth (a depth of 1 is blue and -1 is red). Each horizontal bar uses a different depth function. This team then examined the pattern of the two colors to determine if the depth function is working correctly. For example, the second horizontal bar uses the depth function of greater than. In a correct implementation, only pixels from the sine wave that are greater then zero (colored in blue) should be drawn.

II.H. Machine Problem 6 (MP-6)

The final MP completes the basic OpenGL pipeline functionality by allowing images to be mapped onto triangles. This process is commonly referred to as texturing, or texture mapping. Each pixel generated from rasterization has both a color value (interpolated from the triangle's vertex color values) and also a texture coordinate (also interpolated). Texture coordinates typically range from 0 to 1 as shown in Figure 8. Each pixel has two texture coordinates that map the image's X and Y axis. The texture coordinates are then used to look up the pixel color from the texture image. While there are multiple filtering algorithms used to determine the pixel color from a texture image, only the nearest pixel value is implemented requiring no image processing algorithms. In addition, texture coordinate interpolation from rasterization currently does not perform projection transformation resulting in a slight errors when rendering images using a perspective viewing transformation.

MP-6 required students to implement the hardware architecture to compute the memory address of the texture image stored in memory from the pixel texture coordinates. When a texture is to be applied to a triangle, the image is copied into a texture cache allowing the starting memory

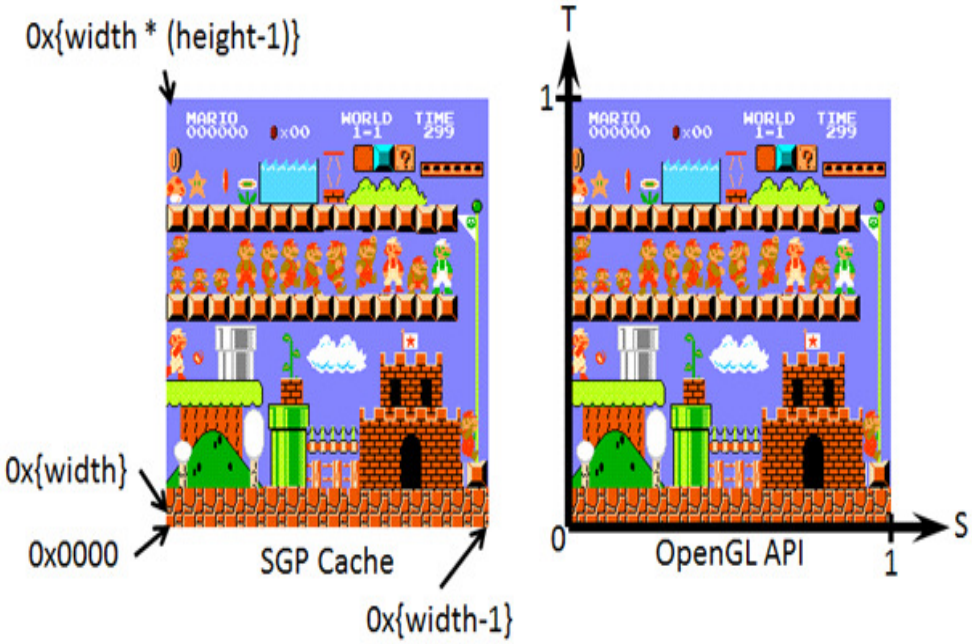

Figure 8: Texture coordinates and how they map to the texture image memory address.

address of the image to be zero. To complicate the computation, OpenGL only restricts the texture image to be a power of two (they can otherwise be any size). The size of the texture image are stored in two registers that students use to compute the memory address.

In addition, to the texture mapping students were required to analyze the entire graphics pipeline implementation to identify performance bottlenecks using some non-trivial applications, such as ID Software's Quake[2] video game.

II.H.1. MP-6 Student Results

In MP-6 students had difficulty implementing the calculation to compute the texture image memory address from texture coordinates and only a few groups were able to get textures working. In addition, while many groups were able to perform a detailed analysis of the pipeline performance, none were able to significantly alleviate any of the bottlenecks. Students struggled with understanding the fixed-point notation requirements needed since texture coordinates are implemented as fixed-point values. To add to the difficulty, both simulation and FPGA synthesis increased in time due to the increasing complexity from all the previous MPs.

III. Conclusions

We have created a senior elective computer architecture class that focuses on graphics processing and architecture. By the end of the semester, students gain an understanding of key concepts in computer graphics along with a complete system-wide perspective for rendering 3D images. Students also obtain valuable hands-on experience in implementing the 3D graphics pipeline on

an FPGA. Laboratory assignments allowed students to implement a 3D OpenGL pipeline from the workstation API all the way down to the GPU architecture running on a physical FPGA board.

Students filled out conventional evaluations for this course at the end of Spring 2011 semester. The feedback from students was generally positive - they enjoyed the opportunity of being able to work on the entire system rather than being limited to specific problems. Out of the 22 respondents, on average the students rated the course as a 4.91 out of a 5 point scale for overall effectiveness. In terms of whether or not the course inspired students to learn more about computer graphics and architecture, on average the rating was a 4.50 out of a 5 point scale. Specific feedback comments pointed to both positive and negative aspects of the MP structure. While the framework prevented students from falling more than 2 weeks behind the rest of the class, some concerns were raised that this also limited the potential for creative solutions to those same 2 week intervals.

The Spring 2012 offering was modified to replace MP6 with a multi-week project. Students can pick from a provided list of projects or implement their own idea. The project is designed to allow students the flexibility of generating their own hardware and software implementation strategies (either new functionality or improving current functionality) without any limitations on using the provided framework. Future modifications will look into improving the graphics pipe unit interfaces to reduce pipeline stall logic the students were required to design. While not all of the pipeline stall logic can be removed entirely since we want to support a wide design space, improvements can be made to reduce the complexity to allow students to focus more on the main MP learning objectives.

All provided student material (hardware designs, software infrastructure, and laboratory manuals) as well as solutions for the individual Machine Problems are available to instructors by request through our research group's website.[7]

References

[1] Mark Segal and Kurt Akeley, *The OpenGL Graphics System: A Specification (Version 1.2.1)*. http://www.opengl.org/documentation/specs/version1.2/opengl1.2.1.pdf

[2] ID Software, *Quake*. http://idsoftware.com/games/quake/quake

[3] NVIDIA Corporation, *GeForce GTX 480*, Available: http://www.nvidia.com/object/product_geforce_gtx_480_us.html

[4] NVIDIA Corporation, *DirectCompute PROGRAMMING GUIDE v3.2*, http://developer.download.nvidia.com/compute/DevZone/docs /html/DirectCompute/doc/DirectCompute_Programming_Guide.pdf

[5] Xilinx Inc, *XUPV5-LX110T Development Board*, Available: http://xilinx.com/univ/xupv5-lx110t.html

[6] Phil Frisbie Jr, *GLTrace*, Available: http://hawksoft.com/gltrace

[7] Reconfigurable Computing Laboratory, Iowa State University, http://rcl.ece.iastate.edu/projects/sgp