FPGA-Based Graphics Acceleration

A Major Qualifying Project Report

December 20, 2010

Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
on this day of December 20, 2010
by

Eric Nadeau

Skyler Whorton

R. James Duckworth, **Major Advisor**

Emmanuel Agu, **Co-Advisor**

Abstract

The goal of this project was to develop an FPGA-based 3D graphics accelerator. Research of previous work in the field preceded the development of a full hardware implementation of a graphics platform capable of realizing 3D graphics with an FPGA. Furthermore, a graphics application programming interface (API) and various rendering algorithms were implemented in software. We tested these implementations and verified that our goal was achieved with reasonable performance. Methods explored by this project showed promise for future customizable and portable 3D graphics platforms that may be utilized within mobile and embedded applications.

Acknowledgments

We would like to sincerely thank the individuals who guided us through our Major Qualifying Project and made this experience a memorable one.

We would like to thank our project advisors, Professor R. James Duckworth and Professor Emmanuel Agu for their continuous help and support throughout the project.

Executive Summary

The growing trend in the popularity of mobile computing has led to an increase in the demand for mobile graphics platforms. While graphics accelerators using Application-Specific Integrated Circuit (ASIC) designs have historically constituted the industry standard, they have presented roadblocks to developers who have wished to reconfigure the hardware or develop the platform in an open environment. Additionally, the long development cycle for ASIC designs has implied that only the leading hardware developers were able to produce worthy products. Although some open graphics acceleration platforms existed, these implementations (such as Mesa 3D, or Vincent 3D for embedded systems) were defined primarily as software systems. One possible option for a graphics acceleration platform was a design using a Field-Programmable Gate Array (FPGA). The FPGA presented design advantages such as its suitability for parallel processing applications, for example certain graphics operations, and its accessibility to developers who wish to modify the hardware platform to fit their specific needs. For these reasons, we explored a design of a graphics accelerator using an FPGA.

Our goals for a fully realized FPGA implementation of a graphics accelerator included a Hardware Descriptor Language (HDL) graphics hardware core, an accessible graphics software API using the OpenGL 1.1 specification and a basic implementation of the 3D graphics pipeline in Register Transfer Level (RTL) hardware. Finally, to test and verify the capabilities of our custom graphics platform, we needed to benchmark the system's performance using OpenGL-based applications.

The hardware we chose to use for this project was the Spartan-6 LXT FPGA residing on the Xilinx SP605 Development Board. Our implementation included a full Graphics Processing Unit (GPU), rasterizer, and video interface on the FPGA. Furthermore, we utilized the board's 128MB of DDR3 component memory for storing two framebuffers. We used the FPGA's MicroBlaze Central Processing Unit (CPU) to execute the platform's software driver, API and user graphics applications. The driver and API, implemented in C, included all of the functions necessary for a programmer familiar with the OpenGL specification to easily execute graphics applications on our platform. The platform implemented point, line, and triangle drawing, triangle filling, and color interpolation. Furthermore, we used the software to realize affine transformations of vertices and realistic ambient and diffuse lighting effects according to the Phong Reflection Model.

Additionally, we used OpenGL graphics applications to benchmark the performance of our graphics accelerator platform. Furthermore, we tested each demo application on both the hardware implementation and a custom, fully-realized software emulator designed to use precisely the same driver and libraries as the hardware. We found that the limitations of the CPU at only 75 MHz, and being responsible for the preparation of each primitive rendering, was often responsible for a bottleneck in performance. We concluded that this performance limitation was indicative of a shortcoming in the processor speed rather than in the FPGA's theoretical capability for graphics processing. Quantitatively, each of our simple graphics application demos achieved reasonable performance marks over thirty (30) Frames-Per-Second (FPS).

Considering the final architecture of our platform and the positive results of our test applications, we concluded that a graphics accelerator may be effectively realized through an FPGA design. For this reason, we recommended that future research be considered in the area of graphics applications of FPGA systems.

Contents

List of Figures

List of Tables

1 Introduction

When considering the growing popularity of a wide variety of mobile and embedded devices capable of three-dimensional (3D) graphics, it's prudent to consider the direction of the evolution of such devices. Current technology trends have precipitated newer graphics processing units (GPUs) that can produce 3D visualizations on compact and portable devices. These may reside on a single chip and can typically perform advanced 3D rendering techniques, including programmable pixel and vertex processors. As computing technologies have advanced, companies including NVIDIA and Texas Instruments have offered high end graphics systems with increased portability and functionality (with the Tegra and OMAP system-on-chip families, respectively). However, such implementations are only realizable by the few major graphics hardware vendors due to the substantial costs of developing such devices. Furthermore, once such hardware is developed, it may only be further customized or improved with a subsequent generation of chips.

While many devices capable of realizing 3D graphics currently do so through the implementation of application-specific integrated circuit (ASIC) designs, it may be worth considering the implementation of field-programmable gate arrays (FPGA) for future designs of mobile 3D graphics devices. An FPGA design may implement a hardware descriptor language (HDL) and may be fully configurable, making this option appealing to developers. Because the FPGA is a relatively small and powerful type of chip, its applications are intrinsically scalable and portable. These characteristics make the FPGA ideal for mobile and embedded devices. Furthermore, the use of parallel processing on the FPGA is conducive to applications such as graphics rendering. For these reasons, implementing FPGA designs to realize the rendering of 3D graphics in mobile devices could potentially herald great advances over current ASIC technology.

Research into an FPGA-based 3D graphics platform has yielded some new promising undertakings, but none of these had been fully developed and distributed to consumers. One such of these projects, Vincent 3D, had attempted to develop an open source implementation of the OpenGL ES API family with some success. However, at this time, Vincent 3D had only implemented a few components of the 3D graphics pipeline in HDL, with the majority of the API implemented as much slower software rendering. Another such project, Manticore, attempted to implement a 3D graphics accelerator on an FPGA, with complete triangle rasterization and VGA output. However, development on Manticore had completely ceased, leaving the project lacking even the most basic but essential abilities of a 3D accelerator.

In this project, we described and tested an FPGA-based 3D graphics accelerator design. With this design, we implemented various stages of the graphics pipeline and its necessary subsystems on an FPGA using hardware descriptor languages. These stages included command and vertex processing, primitive setup, color shading, triangle rasterization, and video output. Taking advantage of the FPGA's parallel processing, various advanced 3D mathematical algorithms were executed in parallel to rendering operations on a separate Microblaze softcore microprocessor. We used a double-buffering based approach to implement smooth, animated visualizations to an output display device via the Digital Visual Interface (DVI). Finally, we adapted a graphics API from the OpenGL specification. This implementation allowed for communication with the 3D graphics hardware via a platform-independent device driver. Within this software layer, we also implemented the Phong lighting model using the hardware's color shading abilities.

Through the completion of this project, we made significant contributions to the development of a complete 3D graphics platform on an FPGA. We met our goal of realizing 3D graphics output using a single chip with a platform that is both configurable and portable. However, time and resource constraints on the project resulted in several performance flaws. These shortcomings primarily included the implementation of the graphics perspective transformations on a softcore microprocessor rather than using HDL. With additional development time, a complete hardware implementation of these transformations would have yielded a substantial performance increase. Future work on this project could also include implementing the lighting model and more complex rendering techniques in hardware for optimal performance.

The remainder of this report first provides the reader with background research (Chapter 2) in the relevant areas of graphics processing and algorithms, FPGA technology, and related work. Chapter 3 then describes an overview of the project including goals, objectives, and a high level design. Chapter 4, 5, and 6 discuss the details of the embedded platform, the hardware graphics core, and the software and driver components of the project, respectively. Chapter 7 describes system testing and results. Chapter 8 contains the conclusions we drew about the process we used as well as our recommendations for future work. Finally, Chapter 8 contains our analysis of the opportunities to apply the knowledge we gained through realizing this system. Following the body of the report, the appendices contain additional technical information such as selected source code that support the project, but were too large to include as inline elements in the main text.

2 Background Research

In order to formulate an appropriate design, it was vital to understand the current state of portable graphics hardware and technology. This chapter discusses the graphics pipeline both in concept and with specific details about implementing it with an FPGA.

2.1 Graphics Rendering Pipeline

The graphics rendering pipeline traditionally refers to an abstraction for the process which modern Graphics Processing Units (GPUs) use to produce visualizations. Computer graphics hardware typically implement six stages within the pipeline and often various substages within these. At a very basic level, these six stages in sequence are: (1) the front end, (2) vertex processing, (3) primitive assembly, (4) rasterization and interpolation, (5) fragment processing and (6) frame buffer output. These stages all run in parallel, however it is important to note that vertices pass through each of them sequentially. *OpenGL*, a standard specification defining a cross-platform computer graphics API, implements the graphics pipeline as seen in the figure below:

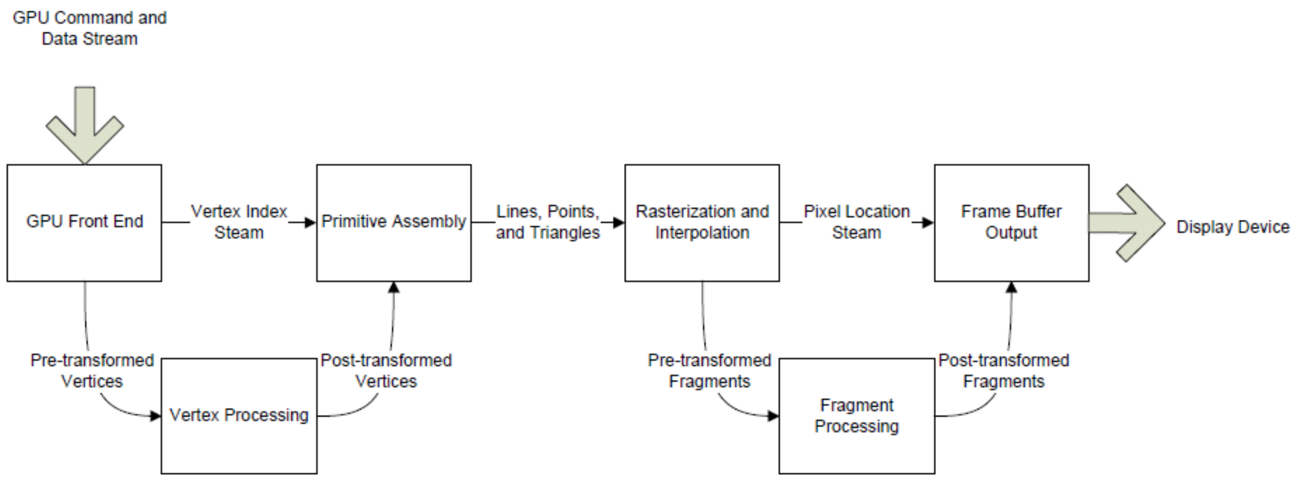

GPU Command and Data Stream
GPU Front End
Vertex Index Steam
Primitive Assembly
Lines, Points, and Triangles
Rasterization and Interpolation
Pixel Location Steam
Frame Buffer Output
Display Device
Pre-transformed Vertices
Vertex Processing
Post-transformed Vertices
Pre-transformed Fragments
Fragment Processing
Post-transformed Fragments

Figure 1: A high level representation of the graphics pipeline used in realtime rendering
(Source Adapted from: NVIDIA Cg User's manual, 2010)

This figure shows all of the major components of the graphics rendering pipeline for modern graphics hardware. This pipeline typically starts at the front end, where commands and vertices are sent from a host system. The graphics data will then filter through the pipeline, while the hardware performs necessary processing and transformation to produce raw displayable data. The graphics commands control how each stage conducts these processes and transformations. The output of this system is an image updating in real time that can be drawn by a display device. This section describes the high level functionality of these stages, and how they interact with each other.

2.1.1 Front End

The front end of graphics hardware provides the necessary interface for all commands and data sent to the GPU. This block typically consists of a number of basic layers: (1) a device driver on the host system, (2) a hardware interface physically connecting the GPU to the host system, and (3) a command processor that receives and decodes commands and data from the hardware interface. After this third layer, decoded commands and data are then used to define the operations of the GPU and render visualizations by distributing them to the other components within the graphics hardware. This may be implemented with only a write interface, but it is often advantageous to have certain read signals, for example, to initiate an interrupt request upon the completion of each frame drawing.

2.1.2 Vertex Processing

In 3D graphics processing, a vertex is typically specified in the form of an (x,y,z) triple as a discrete position within the 3D coordinate system. These are often accompanied by many other parameters and vectors, representing such data as shading color, texture coordinates, or normal direction. The vertex processing block is responsible for decoding this data received by the GPU and preparing them to be assembled as primitives and then rasterized. The vertex positions must be transformed from their 3D coordinates into 2D space as approximated by a display device for this to be done. This occurs in two distinct transformation phases. In the first phase, the *modelview* transformation, vertices are transformed within the 3D coordinate system, thus providing hardware accelerated rotation, translation, and scaling of objects.[11, 13] In the second phase, *perspective* transformation, vertices are transformed to be drawn as they appear on a 2D plane from the viewer's perspective.

2.1.3 Primitive Assembly

A primitive consists of one or more vertices to form a point, line, or closed polygon. This stage takes the perspective-transformed vertices from the vertex processing stage and groups them into primitives.[14] In the stage immediately following primitive assembly, primitives are clipped to fit just within the viewport or view volume and then prepared for rasterization to the display device, typically within some form of buffer.

2.1.4 Rasterization and Interpolation

In computer graphics, a raster image is a 2D array of discrete pixels that represent intensity samples.[13] As such, *rasterization* is the stage within the graphics rendering pipeline where a 2D image is generated from transformed primitive data. More formally, rasterization is defined as converting a line drawing, mathematical expression in space, or a 3D scene into intensity values of a group of pixels to be written to the frame buffer, which is then propagated to an output device. As part of rasterization, *interpolation* is the process of constructing intermediate data points in the form of color intensity values within the interval connecting two vertices.[5] Interpolation is necessary in computer graphics for generating vertex colors and fog, among other effects.

There are two main challenges that designers often face when creating a rasterizer: (1) determining the pixel(s) that accurately describe the current primitive being rendered, and (2) efficiency.[13] For this project, rasterization of primitives is limited to simply points, lines, and triangles. This section describes ideal algorithms for rasterizing these, as well as proper methods of clipping and Z-buffer generation.

Points In rasterization, point primitives represent a single pixel drawn to the framebuffer. Points are rasterized by writing the pixel's color to the framebuffer memory at the address specified by the pixels X- and Y-coordinates. This is expressed as the equation:

$$Address_{pixel} = BaseAddress_{framebuffer} + (y_{pos} * Width_{framebuffer}) + x_{pos}$$

Lines Line primitives are two points on the framebuffer connected by the pixel approximation of a straight line. One of the more common and simple methods for rasterizing a line is Bresenham's algorithm, which is able to correctly approximate a line primitive.[13][6] This algorithm computes a line starting at a pixel at location (x0, y0) and approximates subsequent pixels downward until reaching the ending location, (x1, y1). The algorithm is initially defined for the first *octant* (the first of eight divisions of a 2D coordinate system) such that the line extends downward and to the right. As such, seven separate versions of the algorithm must also be implemented to rasterize a line within the remaining seven corresponding octants. The illustration below shows a line approximation computed by Bresenham's algorithm.

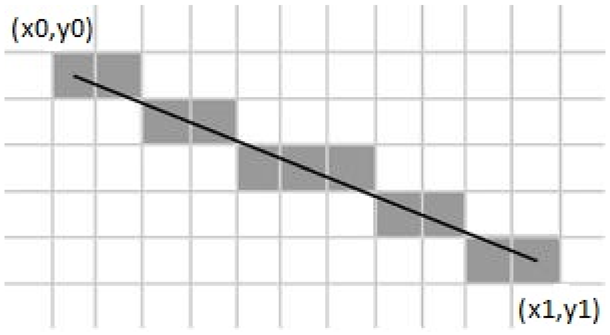


Figure 2: Illustration of Bresenham's line algorithm
(Source: http://www.jobscochin.com/introduction-computer-graphics-algorithms, 2010)

Bresenham's line algorithm selects the next pixel integer Y-coordinate that is closest to the fractional Y- for the same X-coordinate. Successively, Y- can as such remain the same or increase by one. Pixels chosen by Bresenham's algorithm are then rasterized to the framebuffer as point primitives.

Triangles Triangles are the base primitive for rendering complex 3D meshes. An example of how a complex mesh can be constructed from simple triangle primitives can be seen in the figure below.

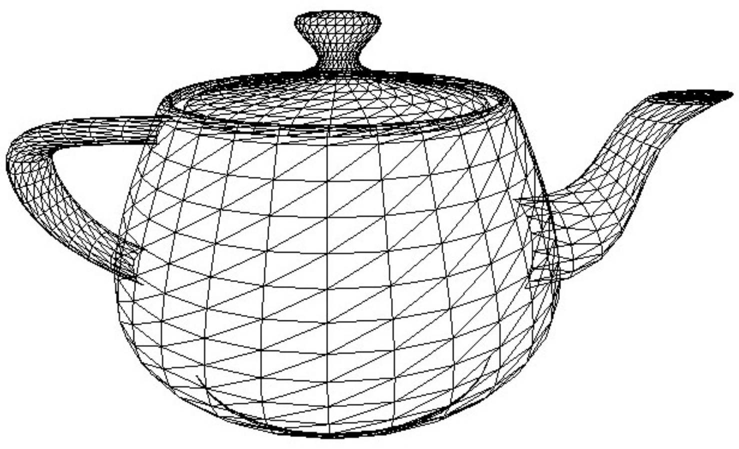


Figure 3: Wireframe teapot mesh

(Source Adapted from: http://caig.cs.nctu.edu.tw/course/CG2007/, 2007)

One very simple method of rasterizing a triangle can be performed by first computing the wireframe of the triangle (three connected lines). This can be done by connecting the three vertices of a triangle using Bresenham's Line Algorithm and saving the X-coordinate, Y-coordinate, and color value for each pixel in that wireframe. Next, the triangle wireframe can then be filled by using a scan line rasterization algorithm to draw horizontal lines across the triangle (with the saved pixel data) for each vertical pixel in the triangle wireframe. Figure 4 illustrates this concept below.

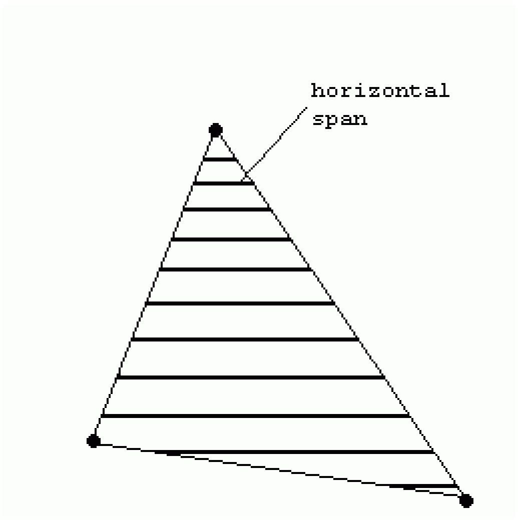


Figure 4: Illustration of the triangle filling algorithm

(Source: http://www.devmaster.net/articles/software-rendering/part3.php, 2004)

Consequently, all primitive rasterization stems from the ability to put a single pixel in the framebuffer - point rasterization. Line rasterization can then be performed by successive points, and triangle rasterization can be performed by successive lines. Complex scenes can then be constructed by rasterizing a series of triangles.

2.1.5 Frame Buffer

After the GPU has produced a synthesized image and performed on it the necessary pixel transformations, the pixels are stored in an image buffer - the *frame buffer*. As its name suggests, the frame buffer stores the current "frame" to be rendered, in an animation sequence.[13] This intermediate storage between the graphics pipeline and the output display is necessary because the two are not synchronized with each other. Displays are typically clocked, which depend on both the display resolution and refresh rate. Display updates are synchronous processes that perform continuously and sequentially by a constant clock signal. The GPU conversely produces data asynchronously in a manner depending on the CPU sending it data or commands.

The frame buffer for a graphics accelerator is generally either pre-allocated in a systems main memory, or in a dedicated memory device onboard the graphics accelerator. Specifically, with realtime graphics systems, the frame buffer is where all pixel color data from rasterization is stored before being drawn to the display. Furthermore, this is necessary because rasterization is performed on primitives and there is no guarantee that the rasterized primitives are actually drawn to the display.[13] For simplicity and efficiency, pixel color data within the framebuffer is typically encoded in a format most compatible with the input signals of the display device.

Many issues can arise due to the conflict between a GPU's sporadic access to the frame buffer and the display device's sequential access. First and foremost, it is likely that the display device will begin reading a *scanline*, the a horizontal pixel line traced by a display's rasterbeam, before the GPU has finished drawing it.[13] An ideal solution to this problem would be for the output circuit of the graphics accelerator to wait for the rendering of a frame to be completed before starting to read the frame buffer. However, this is not possible because the output image must be updated at a very specific rate that is independent of rasterization time.

The solution to this problem is to introduce *double buffering*, which is the use of two frame buffers by the GPU.[13, 6] The first frame buffer, the back buffer, is used for writing only from the rasterizer and therefore may accept data liberally. The second framebuffer, the front buffer, is used only for reading from the output circuit, and can therefore be read sequentially and independently from the time at which data is written. When the current frame is finished rendering to the back buffer, the two buffers then switch their positions and the next frame is drawn. This mechanism prevents the buffer currently being drawn from ever being overwritten.

Basic double buffering may fail to prevent all frame buffer-related artifacts. The most common of these, *tearing*, occurs when the buffers swap before the sequential reading of the front buffer has completed the frame.[13] This results in some lower portion of the display coming up a frame ahead of the upper portion. Tearing may be avoided by only swapping the buffers in the refresh interval, the vertical blank count (VBLANK), between the previous and next frames. Swapping during the VBLANK period, however, introduces potentially significant latency in this process. On many modern graphics accelerators, this is a setting that can be disabled in order to maximize speed.

2.2 3D Mathematics

Graphics accelerators exist to perform the mathematical calculations that render 3D visualizations more efficiently than computer CPUs. These mathematics are often complex as they provide all of the necessary routines to build a 3D scene and animate it as needed. This section describes the fundamental concepts of these operations, as well as the necessary considerations for implementing such routines on an FPGA.

2.2.1 Transformation

Modern computer graphics accelerators typically take advantage of highly optimized math co-processors to perform floating point transformations of objects in 3D space. Transformation is divided into two categories. The first of these is *modelview* transformation, which provides the necessary operations to translate, rotate, and scale objects.[5] The second of these is *perspective* transformation, which transforms objects in 3D space so that they will appear on a 2D display as if they were being viewed from the camera's perspective. All of these transformations are implemented as matrix multiplications with specially defined scale, rotate, translate, and perspective matrices.

In a modelview transformation for 3D objects, quaternions given as four-by-four matrix specify the movements of each vertex in a homogeneous coordinate space.[5] Although a vertex represents an imaginary point in space with no visible size, the notion of scaling its position refers to the modification of the distance from the origin to that point. For example, the distance of a point along a single axis at position 2 scaled by a factor of 2 would result in $2 * 2 = 4$. In 3D space, a point specified by an ordered triple $(x,\ y,\ z)$ may be scaled by the matrix:

$$T = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where each S value corresponds to the factor by which each distance value will be scaled. A given vertex will be scaled by the multiplication:

$$T = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} S_x x \\ S_y y \\ S_z z \\ 1 \end{bmatrix}$$

In a vertex rotation, the vertex is typically rotated about one of the axes of the coordinate space. The transformation matrix used to rotate each vertex is dependent upon the axis used for the rotation. For example, a rotation of n degrees counterclockwise about the x-axis, called an x-roll, is given by the transformation matrix:

$$T = \begin{bmatrix} \cos(n) & -\sin(n) & 0 & 0 \\ \sin(n) & \cos(an) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Finally, vertex translation is accomplished by specifying the vector by which the vertices are translated in the rightmost column in the transformation matrix. To translate a vertex by the vector $[T_x,\ T_y,\ T_z]$, one may use the following transformation matrix:

$$T = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Using this combination of *affine* transformations as they are called, one may describe any movement or reshaping of vertices in a 3D space.[5] In 3D graphics using 2D display devices, however, it becomes necessary to simulate the third dimension or axis through a series of *perspective transformations.* Using perspective projection, objects far away from the viewer will have a smaller projection and objects close to the viewer will have a larger projection. A perspective projection is defined by a series of parameters that define the dimensions of the view volume: l and r as the left and right boundaries, b and t as the bottom and top boundaries, and n and f as the near and far boundaries, respectively. The following transformation matrix is valid as long as $l \neq r$, $b \neq t$, and $n \neq f$:

$$T = \begin{bmatrix} \frac{2n}{r-1} & 0 & \frac{r+1}{r-1} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

By combining these affine and perspective transformations, graphics software and hardware may convert mathematical data in the form of 3D vertices into a 2D mapping of objects that faithfully represent a scene in space.

2.2.2 Fixed-Point Computations

Floating-point units (FPUs), the components of CPUs used to natively perform arithmetic operations on floating point numbers, are often unavailable on FPGAs and are few and far between on low-power micocontrollers and microprocessors. In these cases, floating point math may be implemented in software, or simply not used at all. On such devices, acceptable computational performance may be achieved using fixed-point arithmetic operations for 3D mathematics rather than using a software-defined FPU.[9][8]

With fixed-point arithmetic, the decimal point of a number is "fixed" before a specific digit. For example, during integer math, the decimal is placed after the least significant bit (LSB), meaning that there is no memory allocated for a possible fraction.[9] In this scenario, the fraction is discarded and the result of a computation is rounded to whole numbers. However, it is possible to fix the decimal point at a different position within the number. This allows for fractions to be represented, even if they are still just rounded approximations. With this form of number representation, math can be done by a traditional arithmetic logic unit (ALU) and 3D vectors may still be approximated enough to render realistic visualizations. This incorporates simple integer math, along with some additional steps to account for the decimal point position.

For the application of 3D mathematics, it is seldom ideal to represent fixed numbers with equal allocation for both whole numbers and fractions.[9]This is due to the frequent need for a signed most significant bit (MSB), which constrains the integer component of the number. Instead, a format such as 18.14 or 17.15 (in the format M.N, where M represents the integer component in bits and N represents the fractional component in bits) is often much more effective for 32-bit 3D fixed-point math.[15] For 16-bit math, the format 10.6 is ideal to ensure a substantial enough integer component.

2.3 Target Platforms

This project required two concentric platforms: the FPGA development board upon which the system was developed, and the embedded system that served as the host device, which utilized the graphics accelerator to render 3D visualizations. This section identifies the selected platforms and discusses how the selected devices met the needs of the project.

2.3.1 Spartan-6 FPGA and SP605 Evaluation Kit

The Spartan-6 is the latest iteration of Xilinx's Spartan family of low-cost, low-power FPGAs. Xilinx, the leading FPGA manufacturer, has designed the Spartan 6 with high performance and cost-sensitive applications in mind.[2] Using 45nm, 9-metal copper layer, dual-oxide process technology, the Spartan 6 includes advanced power management and memory support, among other features. The selected XC6SLX45T-FGG484-3C FPGA includes 150,000 logic cells and supports integrated hard memory, block RAM, high performance clocking and serial IO, and an integrated PCI-Express (PCI-E) endpoint block.

The Spartan-6 LTX is Xilinx's ideal offering for FPGA-based graphics acceleration as it balances both price-point and the resources necessary for GPU computations.

The SP605 Evaluation Kit enables developers to easily prototype designs with the XC6SLX45T FGG484-3C Spartan-6 FPGA. The kit includes all the basic components of the Xilinx Base Targeted Design Platform in one package. [2] With the SP605, developers can easily take advantage of the features of the Spartan 6. Additionally, the kit includes DVI video output, a 200MHz oscillator, 128MB of DDR3 memory, and various expansion connections. The figure below shows the SP605 board.

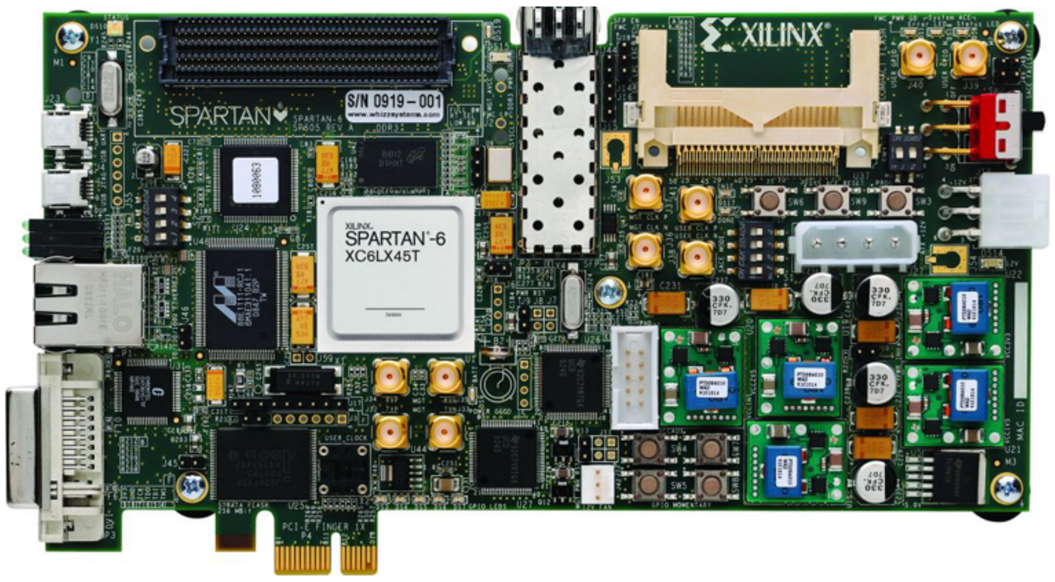


Figure 5: Spartan-6 SP605 Evaluation Kit
(Source: http://www.xilinx.com/products/devkits/EK-S6-SP605-G.htm, 2010)

The support for the Spartan-6 LXT FPGA, digital video output interface, and 200MHz clock all made the SP605 an appropriate choice for FPGA-based graphics acceleration. Additionally, its support for PCI-E created an additional advantage for potential applications in a desktop PC environment.

2.4 Graphics Application Programming

Developing graphics applications necessitates the consideration of portability. A graphics acceleration system must provide an accessible interface to allow its users to develop high-level programs that are easily maintained and may potentially be ported to and from other systems.

2.4.1 OpenGL

OpenGL is an open source graphics library originally developed specifically to enable device-independent graphics programming. Its free and open nature combined with its widespread use in industry and academia additionally have made it an ideal choice for many computer graphics systems. [6]

As an Applications Programming Interface (API), OpenGL specifies a library of functions and data types that interact together to enable a wide range of simple or complex computer graphics applications. Its definition as an interface provides developers with a powerful basis to structure a graphics system while providing users with flexibility and the assurance that their application may be compiled and run on any two OpenGL-compliant systems. In this way, the user has access only to the interface and need not consider the underlying graphics driver or hardware. The developers, in turn, need only focus on rigorous compliance to the standard rather than providing support for users of a graphics driver. In this way, such an interface is a valuable tool to both developers and users.

2.4.2 Lighting in OpenGL

OpenGL provides the tools to develop many visual effects to enhance the images attainable with basic modeling. Among these features is the capability for lighting in scenes rendered by OpenGL as a means of increasing scene realism and enriching the viewer experience. Lighting effects are accomplished in OpenGL through the use of shading models. The shading model or *shader* describes how light is scattered or reflected from a surface. [6][11]

Most shaders use two types of light sources to illuminate objects in a scene: *incident* and *ambient* light sources.[11] While incident light sources model the physical behavior of light in the real world, ambient light sources simulate the seemingly nondirectional background light that is difficult and expensive to model accurately. Ambient light may also be considered to be the background glow in the surrounding environment. [6]

In OpenGL implementations of 3D lighting, there are two categories of incident light:

- *Diffuse scattering*, which describes the effect of light slightly penetrating a surface and radiating uniformly in all directions. Since the light has such a strong interaction with the object's surface, the qualities of the surface material, such as color, are very important to this effect. Reflections from diffuse scattering are independent of the position of the viewer in relation to the surfaces or lights.[6]

- *Specular reflections*, which describe the mirror like reflections of light off of an object's outer surface. This effect produces bright highlights on a surface and suggests a shiny quality. In general, the color of the highlights observed on a surface are determined by the color of the light rather than the color of the surface. Reflections from specular highlights are inextricably related to the position of the viewer in relation to the surface and light sources.[6]

In general, most objects in a scene will include varying levels of both diffuse and specular reflections. The specific amounts of each reflection type depend upon the material properties defined as part of the scene. [6][11]

Ambient light is included in shading models address the inability of diffuse and specular reflection to accurately simulate background light in an environment. When considering the effects of only incident light on a scene, any unaffected surface will appear totally black. This creates deep shadows on objects that give the scene a harsh look. Ambient light is therefore used to provide an approximation of the light that is reflected from many directions from many different surfaces in the scene. This effectively provides a soft glow that acts on all objects in a scene and provides a more realistic effect.[6][11]

2.4.3 Mathematics of Lighting

To apply the shading model to a scene, a series of calculations are performed upon each user-defined vertex in the scene based on the lights' properties as objects in the scene, the vertices' positions, colors and orientations and the camera's position and orientation in the scene. The color of a vertex is a simple sum of the lighting contributions for all lights, i, acting on it:

$$Vertex\,color = \sum_i^{lights} (ambient_i + diffuse_i + specular_i)$$

Each lighting component, as seen in the figure below, has a unique effect on the scene.

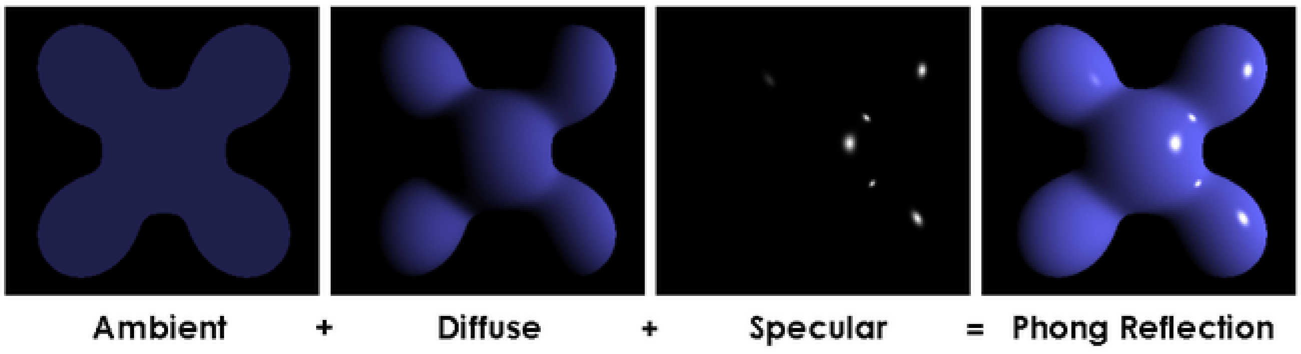


Figure 6: Observable Components of the Phong Reflection Model
(Source: http://en.wikipedia.org/wiki/File:Phong_components_version_4.png, 2010)

Ambient lighting is mathematically the simplest of these components since it acts as a linear scaling effect on the vertices' material properties.[6] The ambient light contribution is a simple product of the ambient material color and the ambient light value:

$$Ambient\,contribution = ambient_{light} * ambient_{material}$$

Diffuse lighting intensity depends on the dot product of the unit vector pointing from the vertex to the light position, **L**, and the unit normal vector of the vertex, **n**. If the dot product is negative, then the vertex is not acted upon by diffuse light and the diffuse component is 0 by default.[6] The diffuse lighting component for each light is thus defined as:

$$Diffuse\,contribution = diffuse_{light} * diffuse_{material} * max\left(\mathbf{L} \bullet \mathbf{n},\, 0\right)$$

Similarly to the diffuse component, the specular component only applies if the dot product of the light and normal vectors is greater than zero. The specular term is further dependent upon the dot product of normalized sum of the light and view vectors, **s**, and the unit normal vector, **n**. This sum is then raised to the power of a user-defined term, shininess, which is a number between 0 and 128 that controls the focus of the specular highlight.[6]

$$Specular\,contribution =$$
$$\begin{cases} specular_{light} * specular_{material} * \left(max\left(\mathbf{s} \bullet \mathbf{n},\, 0\right)\right)^{shininess} & \mathbf{L} \bullet \mathbf{n} > 0 \\ 0 & \mathbf{L} \bullet \mathbf{n} \leq 0 \end{cases}$$

These contributions are then added together to define the final vertex color. According to the Phong model, the colors of the intermediate pixels between these vertices may be determined by a variety of shading or interpolation methods. One method of coloring objects in the scene is on a *per-vertex* basis, that is, each pixel's color in the displayed scene is determined by lighting through linear interpolation of colors from vertex to vertex. Given the 2D coordinates and the color value of two vertices, the *color slope* between them is defined as:

$$m = \frac{\Delta C}{\Delta X}$$

where ΔC specifies the change in color values and ΔX specifies the change along the independent axis (i.e. X-axis for mostly horizontal lines or Y-axis for mostly vertical lines). Using this color slope, the formula of a line functionally determines each intermediate point's color value as follows:

$$y = mx + b$$

where y is the intermediate color value, x is the independent axis position and b is the color value of the first or "source" vertex taken in the interpolation.

2.5 Related Work

Several projects have attempted to implement the OpenGL specification and computer graphics pipeline as discussed in this chapter. However, the majority of these have consisted primarily of software implementations rather than FPGA-based implementations. This section discusses a few of these related works.

2.5.1 Mesa 3D

Mesa is an open-source implementation of the OpenGL specification, providing a system of rendering interactive 3D graphics[1]. Originally started in 1993 by Brian Paul, Mesa has evolved to a very comprehensive set of libraries used in a variety of device drivers, software emulators, and modern GPUs. In addition to being cross-platform, Mesa has been one of the most advanced and complete implementations of OpenGL. However, Mesa consists entirely of a software implementation, which would be much slower than a potential hardware implementation (on either an ASIC or an FPGA). At the time of this project, Mesa had implemented the complete OpenGL 2.1 specification, as well as various extensions from OpenGL 3 and 4. The figure below shows a scene rendered entirely by the Mesa 3D library.

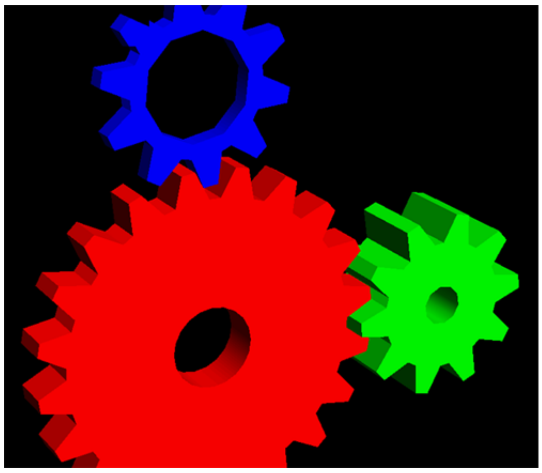


Figure 7: OpenGL gears rendered with Mesa 3D
(Source: http://www.icewalkers.com/Linux/Software/534890/Mesa3D-for-MiniGUI.html, 2010)

2.5.2 Vincent 3D

The Vincent 3D Rendering library, like Mesa, is an open source graphics library that implements the OpenGL specification. However, unlike Mesa, Vincent instead had implemented the OpenGL ES 1.1 API specification, published by the Khronos Group[3]. While Vincent had recently shifted its focus to various HDL implementations of the graphics pipeline, at the time of this project the current release of Vincent was entirely software-based. Like Mesa, this made the project much slower than a potential hardware implementation of a graphics accelerator. Figure 8 shows a visualization produced by the Vincent 3D renderer on a mobile device.

Figure 8: Vincent3D running on a mobile device

(Source: http://www.vincent3d.com/software/software.html, 2010)

2.5.3 Manticore

The Manticore project was an attempt at an open source hardware design for a 3D graphics accelerator written entirely in VHDL[4]. At the time the project was discontinued, Manticore was capable of triangle rasterization, framebuffer support, VGA support, and an SDRAM controller. This implementation was tested on a Altera APEX20K200E FPGA[4]. However, while Manticore was one of the earliest and only strictly HDL implementations of a 3D graphics core, it never implemented enough of the graphics pipeline to render a complete 3D scene. The project has not been updated since 2002.

2.6 Summary

The main purpose of this chapter was to present the background research of elementary 3D graphics processing. Furthermore, the necessary information to implement these graphics principles and algorithms on an FPGA were also discussed. The next chapter (Chapter 3: Project Overview and Design) discusses goals, objectives and a high-level design of the project.

3 Project Overview and Design

This chapter presents a general overview of the project, discussing its goal and the objectives that had to be met. Additionally, this chapter presents a high level design, describing how we implemented the project's various components and met the design requirements.

3.1 Goal

The main goal of this project was to implement a portable graphics accelerator on an FPGA. This device needed to provide single-chip 3D graphics acceleration for low-power and space-conscious environments, such as with mobile and embedded systems. Implemented in Hardware Description Language (HDL), this accelerator need to be entirely open and configurable, providing a graphics system that could be easily adapted or optimized for specific system requirements.

3.2 Objectives

To achieve the goal of this project, three primary objectives had to be met. The graphics accelerator needed to:

- Provide a fast and simple input interface
- Generate an image at reasonable speeds from graphics data
- Provide a display output interface

These objectives were completed through the design and implementation of a number of subsystems.

The first objective was to interface with other systems that needed to display graphics. The device needed to easily connect to other systems through a standard input/output (I/O) format, wherein it would use an easy and intuitive means of communication.

The second objective was to rasterize a two-dimensional (2D) image from the three dimensional (3D) data provided by another system. This needed to be accomplished efficiently because many images needed to be generated per second to produce smooth animated visualizations. Consequently, the device needed to use optimal algorithms to perform these operations.

The third and final objective was to provide a means to display the generated image. This needed to adhere to a standardized display format, allowing the graphics accelerator to be used with many different platforms. Also necessary were an output connection and the underlying hardware controller for the selected format.

3.3 Design Requirements

The design for this project included five subsystems. These systems and their responsibilities are described in the table below.

Input Interface	Provided a cross-platform interface for graphics commands and data.
Graphics Processor	Decoded, transformed, and rasterized 3D primitive data to a 2D pixel buffer.
Output Interface	Implemented a standardized display output.
Device Driver	Provided the interface between software and the graphics hardware.
Graphics API	Abstracted the driver interface to a cross-platform graphics programming specification.

Table 1: The major components of the design requirements

Each subsystem had its own specific requirements. The *input interface* provided a communication connection for graphics commands and data to the accelerator. This hardware implemented the link between a host system and the the graphics processing unit (GPU). This interface needed to be independent of the platform used with the accelerator, adhering to a known and standardized I/O format. This format needed to transfer graphics commands and data at a fairly high rate such that 3D visualizations could be realized at smooth frame rates.

The *graphics processor* needed to render 3D primitive data to 2D space and was fulfilled by the functional implementation of a GPU. The GPU implemented the 3D graphics pipeline and all of its components. The GPU received encoded commands and data from the input interface and decoded these and their required parameters. Next, it performed necessary modelview and perspective transformations on them and rasterized the result to a pixel buffer. These components needed to run efficiently and integrate seamlessly with the desired input and output devices. The GPU needed to also implement optimal rasterization and transformation algorithms that run in parallel to each other and achieve the highest possible speeds for the target platform.

The *output interface* needed to implement a standardized display output format, providing the interface between the GPU and a target output device. This interface implemented a digital video out standard ensuring that it could be used for multiple different displays. This required a hardware implementation of the display format controller to generate necessary output signals.

The *device driver* needed to provide the software interface between the graphics API and the hardware input interface. The device driver implemented the defined functionality of the API by sending commands and data to the input interface. This driver needed to be compatible with multiple hardware systems by being implemented within a standard driver model.

The *graphics API* (Application Programming Interface) needed to define an abstraction layer that allowed a user to easily send graphics commands and data to the accelerator from their applications. This API needed to be implemented in a standard programming language and follow a consistent and logical specification. The API implemented all configuration settings of all graphics features of the accelerator. The API needed to also provide an intuitive interface for drawing and manipulating graphical data using the GPU.

3.4 Design

This section describes the overall system design for the project, implementing the requirements previously discussed. The core of the system, the graphics processor and all of its necessary subsystems, was implemented within the Xilinx Spartan-6 LXT FPGA on the SP605 Development Board (described in Chapter 2: Background Research).

To render 3D visualizations, data flowed first from the application, then to the graphics driver, and then to the FPGA. The FPGA performed all necessary processing to rasterize the 3D visualizations to a framebuffer located in memory. Due to project time and resource constraints, certain stages of the transformation (including the lighting and normal transformations) were implemented in driver-space.

We selected the 128MB DDR3 component memory, located on the SP605, to hold all video memory. This memory synchronously sent framebuffer data to the display device via the SP605's Digital Visual Interface (DVI) control circuitry. Figure 9 provides a simple overall system diagram of the graphic hardware and its input and output interfaces.

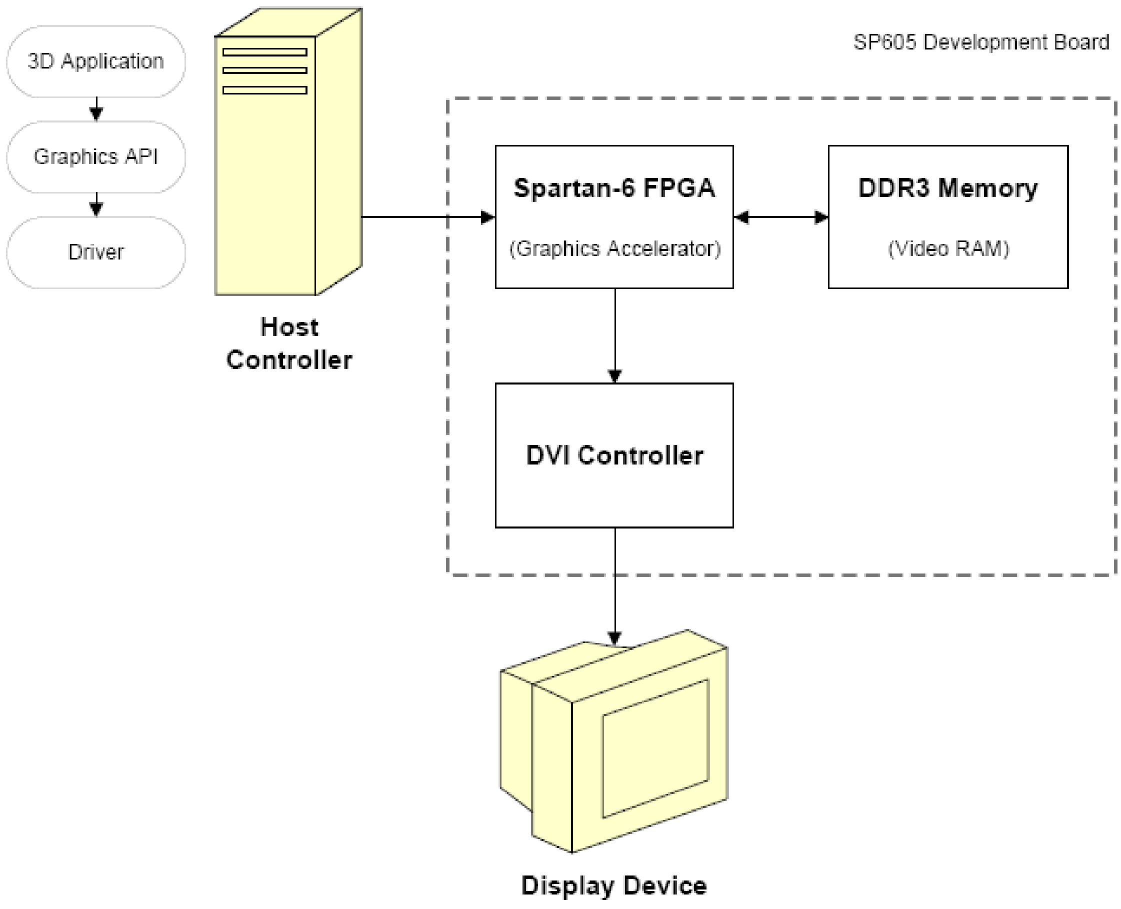


Figure 9: System overview

3.4.1 Software Design Overview

The software design for this project implemented all of the necessary abstractions for an application programmer to render 3D data with the graphics accelerator. This composite system consisted of: (1) the graphics API as a wrapper for (2) the softcore graphics driver commands, and (3) the stages of the graphics pipeline that could not be implemented in hardware. Additionally, the software designs for this project included various test modules whose purpose was to qualitatively and quantitatively benchmark the FPGA-based graphics accelerator, and an implementation of the Phong lighting model.

Graphics Driver The graphics driver served as the software interface to the softcore GPU, allowing 3D visualizations to be drawn for an application by invoking a logical and standardized set of functions. In the interest of hardware independence and streamlined use, the system partially implemented OpenGL 1.1 as the graphics API rather than designing a custom software interface. However, many functions of the OpenGL library had been left out due to missing hardware features. The API itself simply served as a high-level wrapper for users familiar with the widely used OpenGL specification to communicate in terms of the graphics core's instruction set. In this way, the API and the graphics driver comprised an inseparable link between the user's 3D graphics application and the data transfer to the graphics core. The graphics driver also implemented various essential API functions that do not exist within OpenGL, including frame swapping, initialization, and cleanup.

The scope of this project included the implementation of those functions that could be defined within the constraints of the graphics hardware features in addition to several advanced features that serve to extend the capabilities of the hardware. Vertex lighting, for example, would have required complex trigonometric calculations in a hardware implementation that were much more easily handled by the software. The remaining functions in the API were implemented as blank stubs to ensure compatibility with OpenGL applications and to increase the potential for future scalability.

Lastly, certain phases of the hardware modelview and perspective transformation were implemented on a softcore CPU within the driver-space due to RTL limitations on the Spartan 6 LXT. This specifically included the matrix/vector multiplications necessary to calculate 2D screen coordinates from 3D space coordinates, which required signed 48-bit multiplication.

Test Modules A series of test modules were written to qualitatively and quantitatively benchmark the graphics hardware. These were written as 3D demos, and performed the following:

- Identified the maximum performance limitations of the graphics hardware, and produced quantified data on these (e.g. triangles-per-second, etc.)
- Implemented test cases to demonstrate and validate each individual feature of the graphics hardware
- Created a final, presentable, 3D visualization that took full advantage of the graphics hardware to produce an animated scene, demonstrating all of the capabilities of the system

3.4.2 RTL Design Overview

The Register Transfer Level (RTL) design for the graphics accelerator implemented a number of subsystems to control the various stages of the graphics pipeline and its necessary input and output interfaces. The figure below provides a system diagram for the RTL hardware design of the project. This system performed the necessary steps to render 3D graphics to an output display device from an input command stream provided by a user application.

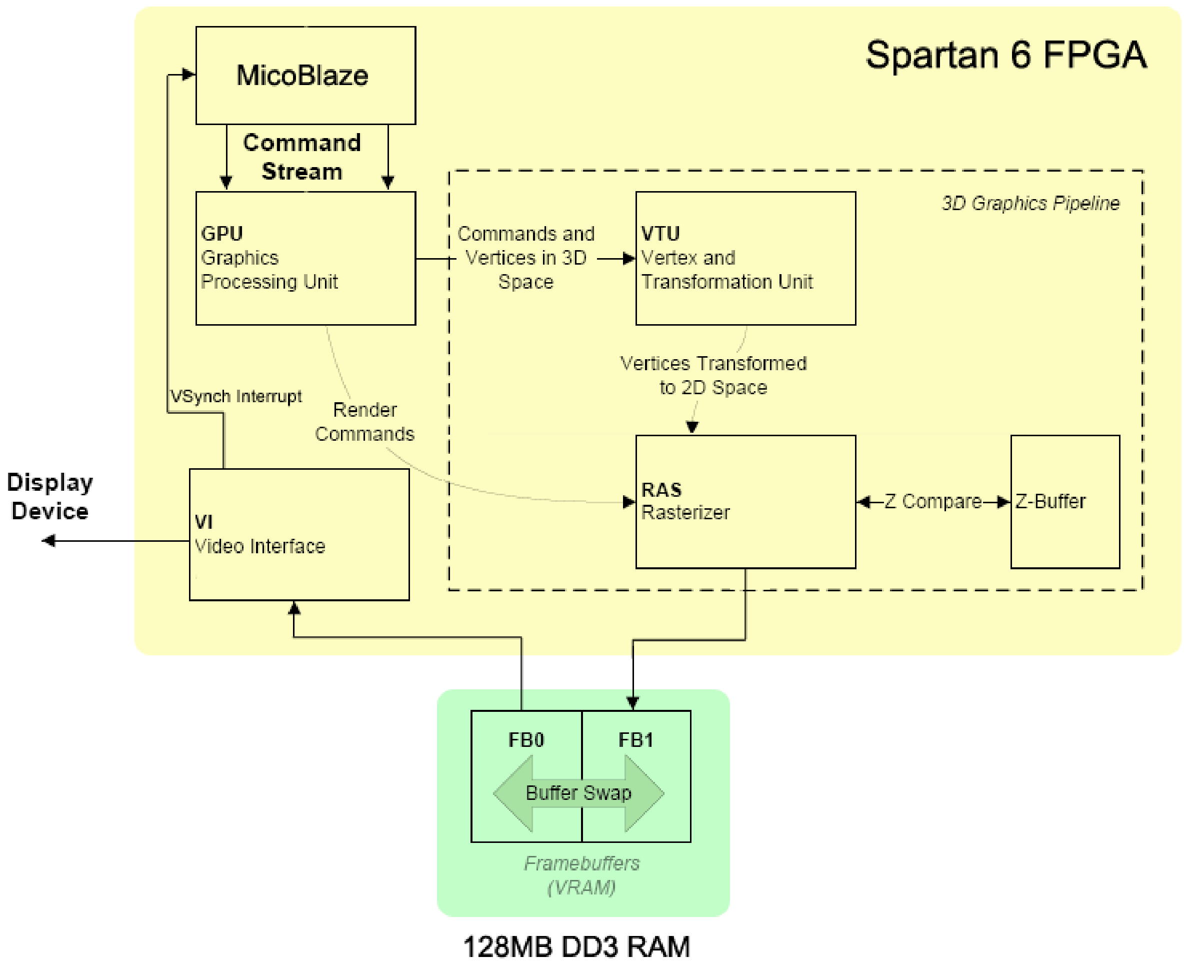


Figure 10: RTL design overview

Commands controlling the graphics accelerator were first pushed into the GPU by the graphics driver over the Processor Local Bus (PLB), followed directly by their relevant parameters or other data. The GPU then dispersed the commands and data to the necessary subsystems, specifically the *Vertex and Transformation Unit* (VTU) and the

Rasterizer (RAS). These two units ran in parallel, albeit sequentially on data. Transformed and rasterized data were then read sequentially by the *Video Interface* (VI), which ran independently of the other subsystems, constantly refreshing the output display device signals. An example execution for a program that draws simple 3D primitives with the proposed implementation is as follows:

1. Initialization routine
    (a) Set clear color and other state values
    (b) Set viewport
    (c) Set view transformation mode (perspective or orthogonal)
2. Draw primitives
    (a) Translate, scale, and rotate the modelview matrix
    (b) Send vertex position and color data
3. Wait for vertical synchronization (last frame to finish being displayed)
4. Repeat steps 2 and 3 indefinitely

Graphics Processing Unit The GPU processed the stream of data and commands (graphics instructions) sent to the graphics accelerator. Acting as the frontend for the entire system, the GPU implemented the various instructions needed to control the states of the remaining subsystems and draw primitive data. These instructions were queued into the GPU by a First-In First-Out (FIFO) buffer, and then decoded and executed sequentially by the GPU. Consequently, the FIFO stored the sequence of instructions to be executed, as well as the necessary parameters these commands provided to the graphics accelerator. The GPU instruction set that defined all control over the graphics accelerator is presented below.

Name	Description
NOP	No operation
DISPCNT	Sets GPU control register
DISPSTAT	Sets GPU status register
DRAWDONE	Denotes the end of a frame being drawn (swaps buffers)
VIEWPORT	Sets the viewport width, height, and X- and Y- locations
CLEAR_COLOR	Sets the clear color
PIXEL	Draws a pixel directly to the framebuffer
VTX_BEGIN	Begins a vertex list
VTX_END	Ends a vertex list
VTX_POS_X32	Set XYZ vertex coordinates, signed fixed-point 32-bit
VTX_COL_565	Set RGB vertex color, 565 mode 16-bit

For the graphics accelerator to process a command, the opcode first needed to be sent through the FIFO followed by its necessary parameters. The parameters for each command are discussed in Chapter 5: RTL Graphics Core Implementation.

Vertex and Transformation Unit The Vertex and Transformation Unit (VTU) defined two primary subsystems: (1) the Matrix Processor (MP) and (2) the Geometry Processor (GP). All vertex and matrix commands decoded by the GPU were processed by the VTU.

The Matrix Processor implemented all matrix operations and stored two matrices to define how primitives were transformed in 3D space and how 3D space was converted to the 2D viewing window (modelview and projection matrices, respectively). The MP implemented all of its matrix transformations by multiplying the current matrix by another matrix, which could be defined for translation, scaling, rotation, or orthogonal and perspective projection. All matrices were 4x4 32-bit fixed point, with the fractional part in the lower 16 bits (signed 16.16 format).

The Geometry Processor performed all of the necessary routines to prepare incoming vertex streams to be rasterized to the frame buffer. Upon receiving a VTX_BEGIN command, the GP began processing a vertex list of the specified primitive type (points, lines, or triangles). To do this, the VTU invoked the following sequence of events:

1. Upon receiving the VTX_BEGIN command, the VTU entered geometry processing mode, and could not receive any non-VTX commands until completion

2. The VTU then received a VTX_POS command for the specified X, Y, and Z

vertices of a primitive, and executed the following:

   (a) Transformed the X, Y, and Z vertices of the primitive by the modelview matrix
   (b) Transformed the X, Y, and Z vertices of the primitive by the projection matrix
   (c) Sent the resulting 2D primitive X and Y values to the rasterizer, as well as the last color data specified by the VTX_COL command

3. The VTU then repeated step 2 for the remaining vertices that had been sent to the GPU

4. The VTU then exited geometry processing mode upon receiving the VTX_END command

At the completion of all drawing, the VTU would then wait to be signaled that drawing of the current frame was completed with the DRAWDONE command.

Rasterizer The Rasterizer's (RAS) main purpose was to perform all of the drawing to the framebuffer. The RAS performed this in two steps: (1) primitive rasterization, the process of computing the necessary pixels to be written to approximate points, lines, and triangles, and (2) color rasterization, the process of determining the necessary pixel color via interpolation. The core functionality of the rasterizer was the implementation of a line rasterization algorithm, which was then used to draw the other primitives. For example, a single point could be rasterized with a line of length 1, and a triangle could be rasterized by drawing a series of horizontal lines (see Chapter 2: Background Research for discussion on these methods). The rasterizer received data from the VTU and rasterized all vertices to the framebuffer.

Upon completion of rasterization, the rasterizer then waited until the Video Interface entered the vertical blanking period, the time elapsed between two frames being drawn to the display device. During this window, the framebuffers swap addresses, and the new framebuffer is subsequently cleared. The Rasterizer would then proceed to process the next frame upon receiving new data from the VTU.

Video Interface The Video Interface provided all of the necessary interfacing to the output display device. This primarily included, (1) timing synchronization of the whole system with the display rate, and (2) sequential generation of the RGB signals to the output display device from framebuffer 1. The VI operated completely independent of the entire system on its own 25MHz clock, constantly outputting the data from framebuffer 1 synchronously with the vertical and horizontal timing signals. This resulted in smooth, 60 frames per second (FPS) visualizations. The VI fed back a single active-high control signal to the GPU, indicating when it is in the VBLANK phase, for synchronization purposes. This ultimately triggered the entire system's only interrupt to prevent the CPU from starting to send the next frame of data.

3.5 Summary

This chapter discussed the goals and objectives of this project. Furthermore, a high level design of this project was detailed. In the next chapter (Chapter 4: Embedded Platform Implementation) the hardware design and implementation of the embedded platform is described. It focuses on how the graphics core was implemented within an embedded system.

4 Embedded Platform Implementation

After developing an overall design of the system, we implemented a hardware platform that would drive the graphics acceleration core. For the hardware platform, we used the Xilinx Embedded Development Kit (EDK) to generate a single-core Microblaze system. Furthermore, this system utilized Xilinx's Multi-Port Memory Controller (MPMC) to arbitrate memory transactions and minimize time that we would have needed to spend on implementing a custom memory interface. This chapter describes the embedded platform.

4.1 Hardware Implementation

With the hardware implementation of the embedded platform, we made a few key design decisions to allow for all necessary functionality of the graphics core and to ensure speed and efficiency. Primarily, the system needed to utilize a bus for peripherals that supported FIFO-like write access and external interrupts. Based on these needs, we chose the CoreConnect Architecture Processor Local Bus (PLB) v4.6 as the interface to our graphics hardware core.

The Processor Local Bus not only allowed for a configurable FIFO interface and an external interrupt signal, but it also provided register space that could be mapped to main memory. As such, this allowed for a simplified graphics command set, as certain graphics commands could be designated a register rather than being queued into the FIFO. This additionally provided for near-immediate hardware response, as there is a several-cycle delay before command data is popped off the FIFO, decoded, and executed.

Furthermore, we also generated a few key peripherals that ensured complete functionality of the system. In addition to the MPMC, we added a RS232 UART core for debug communication, as well as the Xilinx Clock Generator core to generate the bus and pixel clocks. For the development platform, we chose a clock frequency of 75MHz to drive the peripheral bus and Microblaze CPU. However, this frequency could very easily be increased or decreased, depending on the performance and power consumption requirements of the system. We chose a constant pixel clock of 25MHz, as the graphics hardware core was designed only to use a single resolution.

With the essential hardware components included in the embedded platform, we added a few other Xilinx cores for extra miscellaneous features. These included the Xilinx Platform Studio General Purpose Input/Output (GPIO) for the development board's (SP605) LEDs, push buttons, and DIP switches. This was so that they could potentially be used as debug tools within the software stack, or for more complex user-controlled demos. Lastly, we added the Compact Flash core as a potential storage medium for application data that would use the graphics core.

The figure below shows the EDK generated block diagram of the embedded platform. This entire system was synthesized and loaded on to the Spartan 6 FPGA. With the exception of the OGC_IP_0 (Open Graphics Core) peripheral instance, we generated all components of the system with the Xilinx toolset. OGC implemented the custom VHDL that describes the implementation of the RTL hardware graphics core (See Chapter 5 - RTL Graphics Core Implementation).

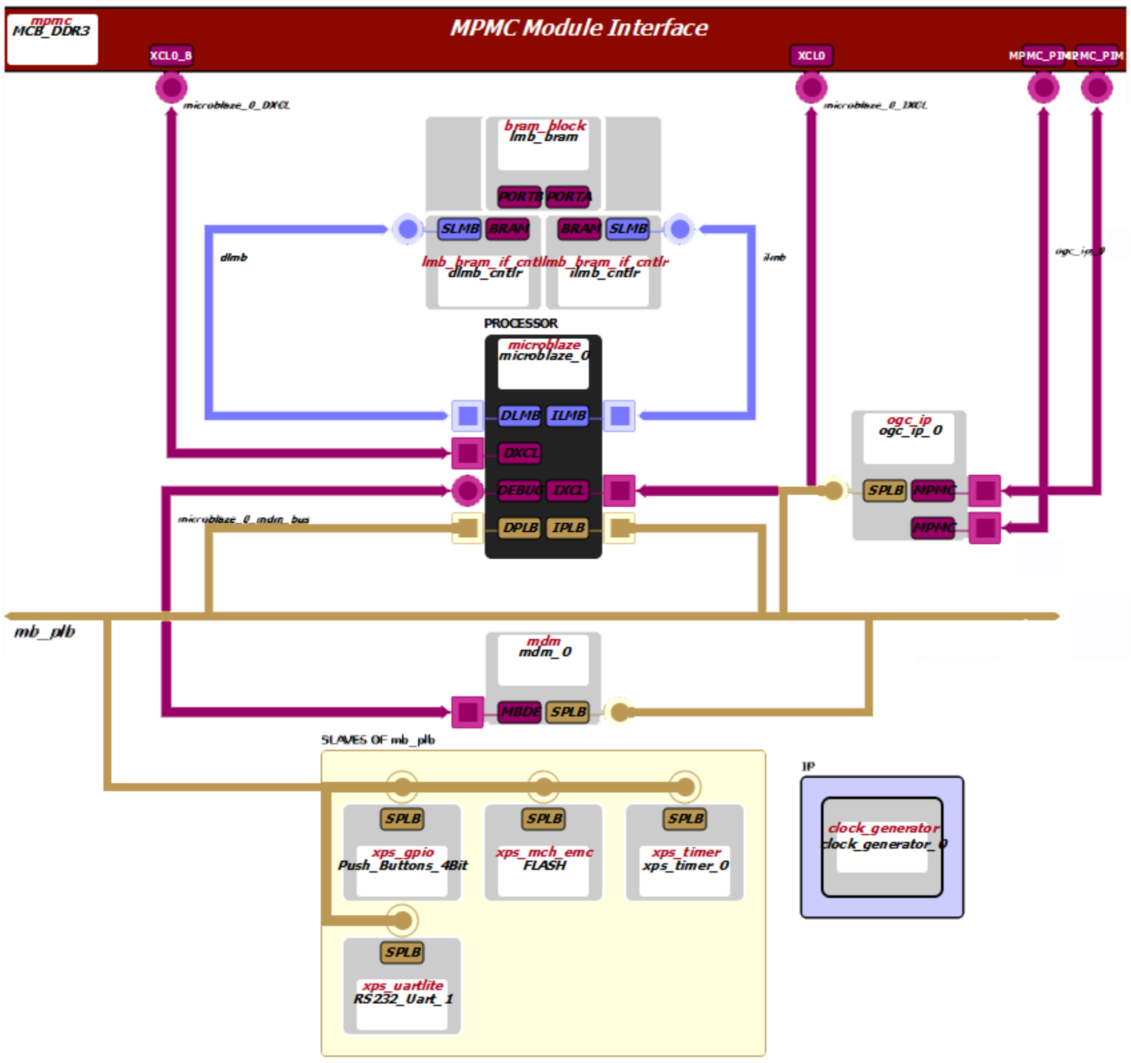


Figure 11: Embedded platform block diagram

4.2 Software Implementation

As part of the software implementation of the embedded platform, we used a script-based scheme to automatically generate the hardware's Board Support Package (BSP). This script, invoked by the Eclipse IDE, would automatically fetch all of the latest driver sources (based on the hardware platform specification) for all peripherals used (including both the Xilinx cores and our custom graphics core), and then rebuild it for use with the current application being loaded on the platform. Such a scheme allowed us to easily make driver modifications to the graphics core without having to manually reconstruct the BSP each time. Furthermore, this allowed for a more modular graphics source base, which opened the possibility for the core to be shared among multiple embedded hardware platforms.

Each individual application written to run on both the hardware and graphics platform was created as a C stand-alone application. The stand-alone option was chosen for simplicity, as none of the implemented demos required any OS-centric features (e.g. no threading or extensive memory management requirements). Alternatively, for more complex applications, the Linux operating system or Xilkernel (a lightweight Microblaze kernel and software stack) could possibly have been used with the platform. Each stand-alone demo statically linked the BSP libraries and could be independently loaded and executed on the hardware platform. The BSP included our custom graphics hardware driver.

4.3 Summary

This chapter discussed the hardware design and implementation of the embedded platform. It focused on the design decisions we made to construct the environment to implement the hardware graphics core. The next chapter (Chapter 5: RTL Graphics Core Implementation) discusses the hardware architecture and implementation of the graphics core peripheral.

5 RTL Graphics Core Implementation

We implemented the RTL graphics core, the center of the FPGA-based graphics acceleration platform, as a custom intellectual property (IP) core within the Spartan 6 FPGA. The core adhered to the Xilinx specification for PLB v4.6 cores, making it entirely portable to other Xilinx platforms. We implemented this core primarily in VHDL. Additionally, we used Verilog for several digital video control modules. This chapter discusses the hardware implementation of the RTL graphics core.

5.1 Architecture and Features

The implementation of the graphics core met and exceeded the initial design criteria. The following features were implemented in hardware:

1. Graphics Processing Unit
   (a) Command Processor running at 75MHz
   (b) 32-bit FIFO command and data interface
2. Video Interface
   (a) Digital Video Out (DVI) and VGA support (25MHz, 640x480 resolution)
   (b) Framebuffer (16-bit color depth, format RGB565)
   (c) Double buffering with a video-synchronization interrupt
3. Rasterizer
   (a) Point, line, and triangle primitive rasterization
   (b) Color interpolation (16-bit)

We implemented these features within a hierarchical architecture. As a Xilinx PLB peripheral, a top level VHDL module was generated that interfaced with the Microblaze system and processor bus. This interface was constant and as such not at all modified during the development of the peripheral. It did, however, instantiate a user logic module that served as the top level for all of the custom user logic that described the functionality of the graphics hardware core. The four main modules that it instantiated were: (1) the Graphics Processing Unit (GPU), (2) the Memory Interface (MI), (3) the Video Interface (VI) and (4) the Rasterizer (RAS). The hierarchical structure of this implementation can be seen in Figure 12.

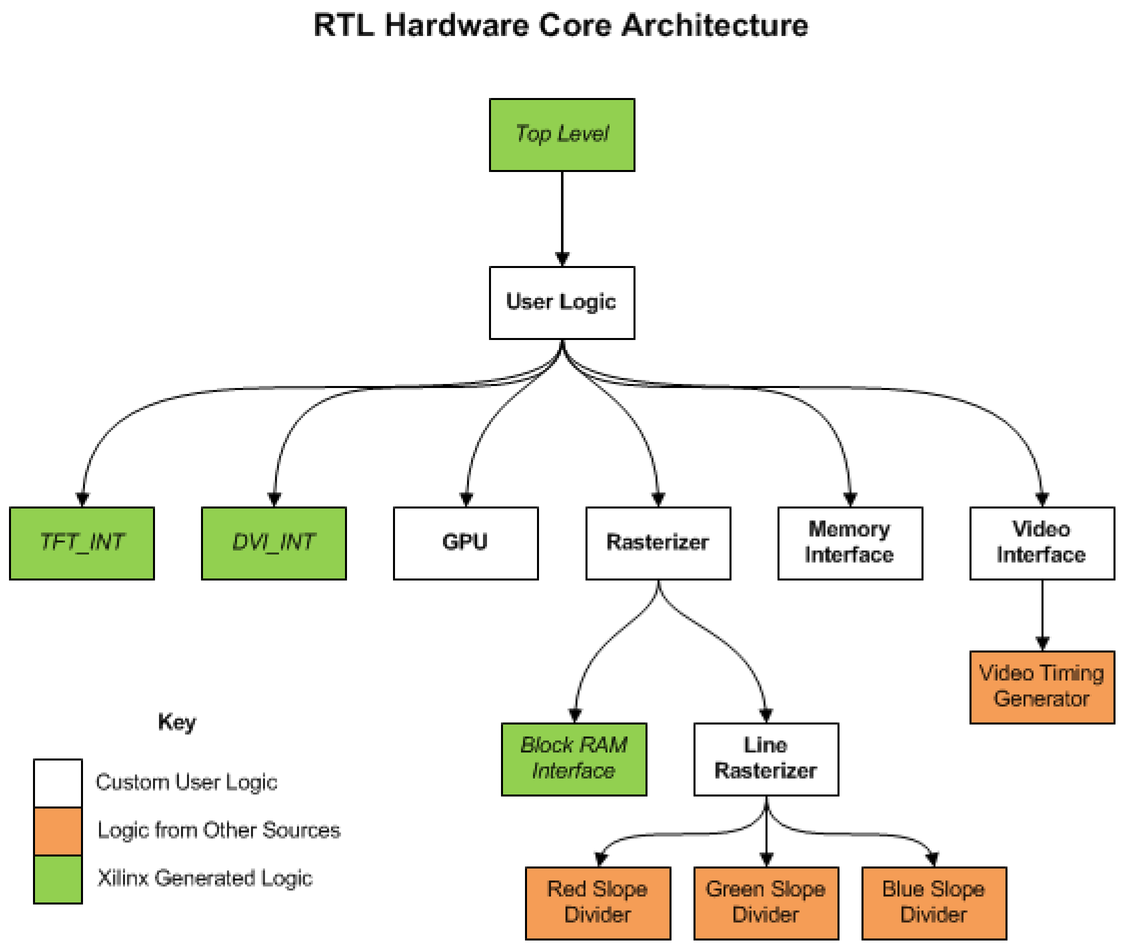


Figure 12: Hierarchical look at the RTL hardware core architecture

Additionally, the system instantiated several other logic blocks that were not custom user logic. These were: (1) a Block RAM interface, (2) a TFT interface and (3) a DVI interface. These modules were all Xilinx generated interfaces. Additionally, there were two other modules that were utilized within the project: the serial division module by John Clayton (licensed under the GNU Lesser General Public License) and the 640x480 video timing module by Ulrich Zoltán (licensed by Diligent, Inc). These modules were used to implement the red, green, and blue slope dividers (of the rasterizer) and the video timing generator (of the video interface), respectively.

5.2 Implementation

We divided the implementation phase of the RTL graphics core into the five main custom user modules. This section discusses their specific implementations.

5.2.1 Graphics Processing Unit

The Graphics Processing Unit (GPU) interfaces all input data streams to the graphics hardware, decodes these data, and subsequently sends these data to the other modules within the core. Consequently, we implemented the GPU in two tiers. With the first tier, the GPU received the command and data stream from the PLB FIFO. With the second tier, it decoded and dispatched the commands and data received to the various other subsystems. This two-tier architecture is illustrated by the diagram in Figure 13.

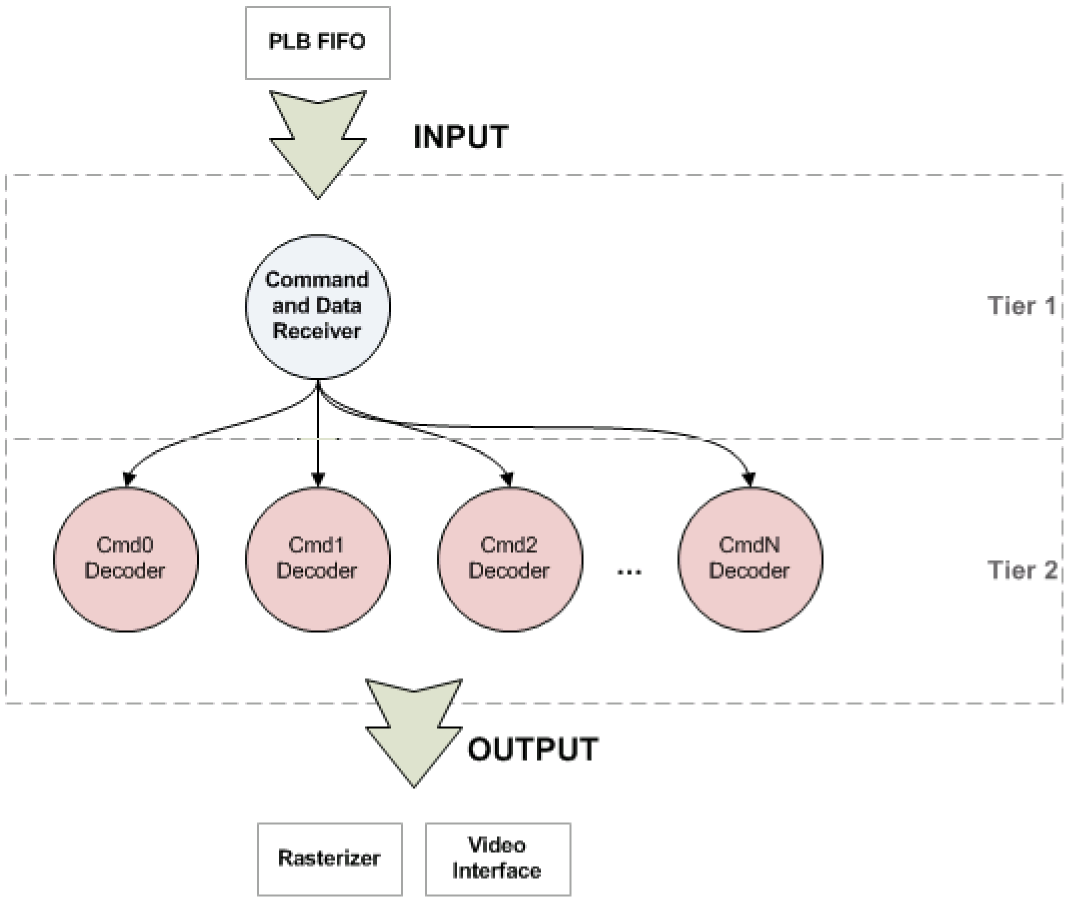


Figure 13: GPU tiered architecture

In Figure 13 above, each circle represents a VHDL process running concurrently. As such, there is one process that implemented the command and data receiver, and another process for each command decoder.

We implemented the first tier of the GPU, the command and data receiver, using a simple VHDL process that synchronously checked whether a new data word had been enqueued in the FIFO (the `WFIFO2IP_RdAck` signal of the FIFO interface will go high). If this signal went high, the GPU would then assume that the first word is the command data, and that every subsequent word received would be parameter data for that command. Each GPU command (see Chapter 3: Project Overview and Design) has a fixed size, so subsequently the GPU would count received words until it has received all necessary data. At this point, it would then assume that the next word received is a command, and the process would continue.

With this design, it was imperative that data be sent to the FIFO correctly aligned with the proper command size, otherwise the FIFO may corrupt with invalid fields. Consequently, if fields need not be specified for a particular command, the command must still be padded with zeros such that it is still aligned within the FIFO.

With the second tier of the GPU, the command decoders were each implemented in the same fashion - as state machines that wait for each parameter of that command to be received. A generic command decoder state machine diagram can be seen in the figure below. This diagram accurately describes the behavior of all of the command decoders.

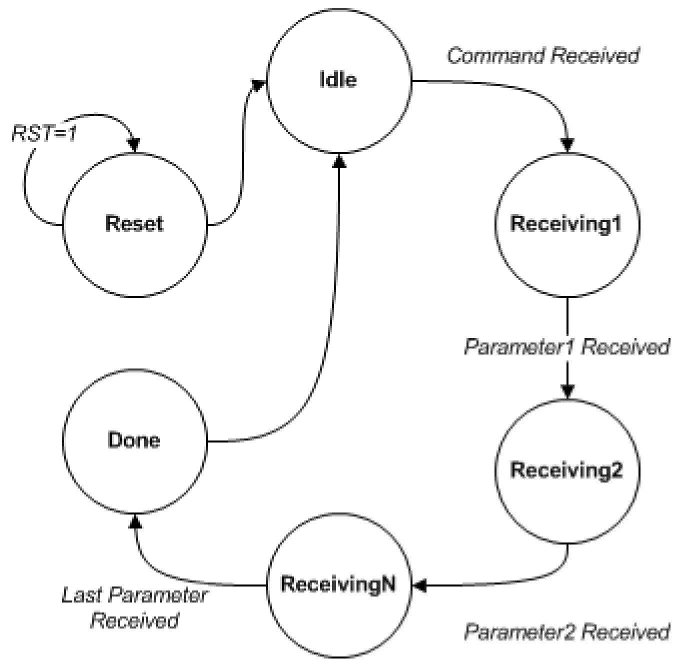


Figure 14: GPU command decoder state machine

When the hardware was powered on, all of the command decoders started in the *reset* state, which reset all states, variables, and signals. The command coders then went into the *idle* state, where they waited until their specific command was received. Each command decoder checked within its idle state if the current command was its command. If it was, it proceeded to the next state - if not, it simply waited until it did receive the correct command.

Next state, *receiving1*, simply waited until the next data word was received from the FIFO. Once it was received, it then decoded the parameters of it for that given command, and then continued to the *receiving2* state. This continued until all parameters were received, and then it entered a done state. The done state returned back to the idle state, where the process continued when the same command was send through the FIFO again.

The table below shows the commands that were implemented in the final RTL design, using the command decoder scheme described above. For a description of each command, see Chapter 3: Project Overview and Design.

	Command	**Parameter1**	**Parameter2**	**Parameter3**
`NOP`	`0x45000000`	N/A	N/A	N/A
`DISPCNT`	`0x45000001`	Unused[31:1] WireframeOnOff[0]	N/A	N/A
`DRAWDONE`	`0x45000003`	N/A	N/A	N/A
`PIXEL`	`0x45000007`	Unused[31:16] Color[15:0]	X-Pos[31:16] Y-Pos[15:0]	N/A
`VTX_BEGIN`	`0x45000020`	Unused[31:4] PrimitiveType[3:0]	N/A	N/A
`VTX_END`	`0x45000021`	N/A	N/A	N/A
`VTX_POS_X32`	`0x45000025`	X-Pos[31:0]	Y-Pos[31:0]	Z-Pos[31:0]
`VTX_COL_565`	`0x45000027`	Unused[31:16] Color[15:0]	N/A	N/A

Table 2: Command parameter decodings

All commands implemented in hardware, with the exception of `VTX_POS_X32`, simply set states within the graphics core. These states were used to describe how primitives were to be rendered. The `VTX_POS_X32` command, on the contrary, was different in that vertex positions were not controlled as states, but rather as continuous streams of data. Furthermore, there was an additional process running in the GPU hardware that stored each vertex position received and waited until enough had been received to rasterize the next primitive. Upon having received all necessary information (e.g., 3 vertices to draw a triangle, 2 for a line) it sent the information to the rasterizer core and triggered the rasterization to the backbuffer. The other states set by the other commands (e.g. color, primitive type, wireframe flag) were also passed along to the rasterizer.

5.2.2 Memory Interface

The memory interface (MI) provided an interface to an external memory device (the 128MB DDR3 RAM on the SP605 Development Board) for the purpose of reading the frontbuffer to the Video Interface. However, it was not the only interface to memory since the rasterizer implemented its own separate write interface to the backbuffer. The HDL implementation simply wrapped Xilinx's Native Port Interface (NPI) with a much simpler read interface. NPI is a Personality Interface Module (PIM) type for Xilinx's Multi-Port Memory Controller (MPMC), the controller for the DDR3 RAM, among many other memory devices. This read interface performed 8-word burst reads, and sent those 8 words back to the Video Interface. This was ideal for a framebuffer read, as the entire framebuffer must be synchronously read and displayed on the output video device. The read interface was configured for 32-bit address and data buses.

The MI was implemented as an HDL state machine. This state machine is described in Figure 15.

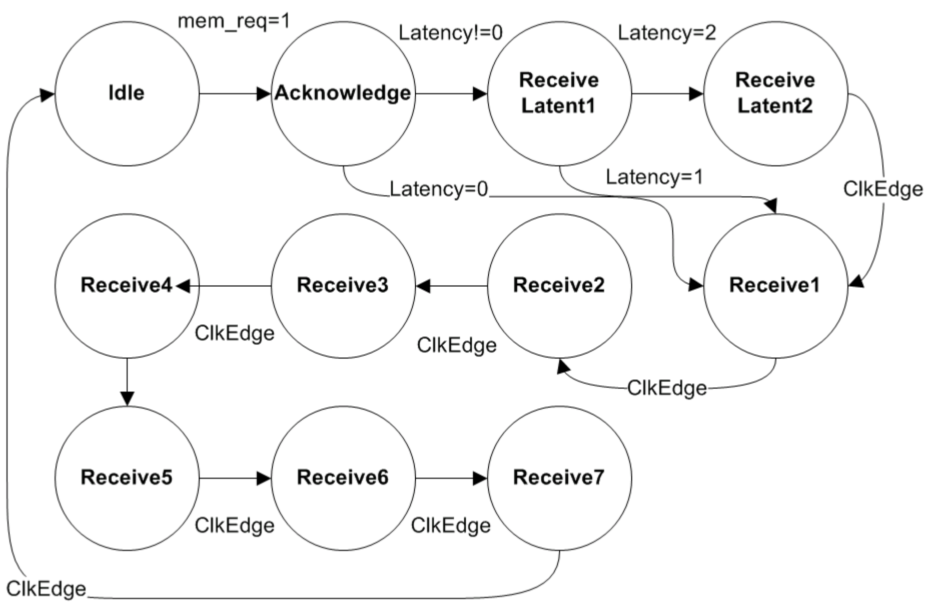


Figure 15: Memory Interface state machine

As can be seen in this diagram, the state machine started in the *idle* state, and waited until the `mem_req` signal went high. This signal was part of the external interface to the memory interface (along with an acknowledge signal, a start address, and 8 data out signals), and was used to start a burst read. Once it had been received, the state machine then went to the *acknowledge* state, where it waited for the NPI to acknowledge the read transaction. In this state, the `mem_ack` signal went high, signaling to the external interface that the memory interface was busy. Once the transaction was acknowledged by the MPMC, the state machine then proceeds based on the latency of the read transaction. This latency could be one or two clock cycles, depending on the state of the MPMC.

If there was no latency on the transaction, the state machine received the first data word on the *acknowledge* state, and then proceeded to receive one word each clock cycle until all 8 words were received. If there was a latency of one clock cycle, the state machine proceeded to the *receive_latent1* state, received the first data word, and then continued to receive the remaining 7 words. If there was a latency of two clock cycles, the state machine proceeded to first the *receive_latent1* state, then the *receive_latent2* state, and then received the first data word. Again, it finished receiving the other 7 data words after this, as the prior two examples did. The state machine changed states on a rising clock edge, and latched data words on a falling clock edge.

Once all 8 data words had been received by the burst read, the state machine then returned to the *idle* state and the `mem_ack` signal went low. This signified that the transaction completed and that the data words were now available to be used by the Video Interface.

5.2.3 Video Interface

We implemented the Video Interface (VI), the interface to the DVI output display device, with two objectives: (1) generating the video output color and timing signals and (2) retrieving the color information from the framebuffer . Consequently, we implemented the VI as two concurrent modules.

The first of these two modules generated the video output color and timing signals, primarily using the video timing module by Ulrich Zoltán (see section 5.1: Architecture and Features). This module was developed to generate VGA timing signals for the Nexys2 Development Board by Diligent, Inc. However, because DVI uses the same timing as VGA, we chose to use this module for the sake of convenience. This module generated the timing signals, described in Table 3, for the 640x480 resolution.

Signal	**Description**
`HorizontalSync`	Active low, goes active at the end of a row being drawn.
`VerticalSync`	Active low, goes active at the end of a frame being drawn.
`VerticalBlank`	Active low, goes active during the period between two frames being drawn.
`HorizontalCount`	Horizontal pixel that is currently being drawn (current pixel X-position).
`VerticalCount`	Vertical pixel that is currently being drawn (current pixel Y-position)

Table 3: VGA timing signals

This module was also driven by a 25MHz pixel clock that was generated by the Xilinx Clock Generator (see Chapter 4: Embedded Platform Implementation). We then used the five output VGA signals to correctly time the DVI signals.

The DVI output required 6 main control signals - a HorizontalSync and VerticalSync (which could be directly used from the VGA timing module), a 5-bit red signal, a 6-bit green Signal, a 5-bit blue signal (RGB565), and a DE (deinterlacing) signal. To implement these, a VHDL process was used that counted on the bus clock and checked if the system was in a blanking period (between frames) or not. If it was, it would deassert DE, and output no color on the DVI color signals (black). If it was not in a blank, it asserts DE, and fetches pixels from the framebuffer as needed to draw them to the output DVI red, green, and blue color signals.

We implemented the second main module, which retrieved color information from the framebuffer, primarily by instantiating the MI (described in section 5.2.3: Memory Interface). As previously noted, the MI reads 8-word bursts from RAM which allows it to read a maximum of 16 pixels at a time. As such, we implemented the module by creating a VHDL process that used the VGA timing signals to trigger the MI to burst read on a 16 pixel interval. This had to occur while the system was not in a blanking period, and the next 16 pixels always had to be read while the current 16 pixels were being drawn. As such, the starting address of the burst read was calculated by:

$$Pixel_{address} = Frontbuffer_{BaseAddress} + (VerticalCount * Framebuffer_{PixelWidth} * Color_{ByteWidth}) + (Burst_{Count} * Color_{ByteWidth} * 16)$$

In the VI, the color byte width was of course 2 (16-bit color, RGB565) and the frontbuffer base address was passed to the VI from the software driver (as the framebuffers are allocated in RAM by the MicroBlaze CPU). To ensure that the next 16 pixels were always fetched, and not the current ones, burst count was initialized as 1, not 0.

At the end of each 16 pixel intervals, the color data from the burst read data words are stored, and then the next fetch is triggered. The stored words were then used by the DVI timing module to output the correct color data at the correct pixel locations.

5.2.4 Rasterizer

The rasterizer (RAS) was the module responsible for fabricating the 3D scene from vertex data. We implemented this module hierarchically, by first implementing pixel rasterization. We then used the pixel rasterization to implement line rasterization. Lastly, line rasterization was used to implement triangle rasterization. This section describes the specific implementations of these.

Pixels We implemented pixel rasterization by performing the inverse of the VI (which read from the frontbuffer) by writing to the backbuffer. That being said, the hardware required a second interface to external memory that could write concurrently to the VI's reads. This was also done by using the NPI PIM, as the MPMC arbitrated these memory transactions. Like in the MI, memory write transactions were implemented using a state machine similar to the one discussed in Section 5.2.2.

Due to the fact that pixels are not often written sequentially in the framebuffer, this NPI interface did not use burst writes. Instead, it was implemented to write only one 32-bit data word at a time. Furthermore, either the two high or two low bytes were disabled, as pixels are only 16 bits wide (and NPI does not support a 16-bit interface). This simple state machine is described in Figure 16.

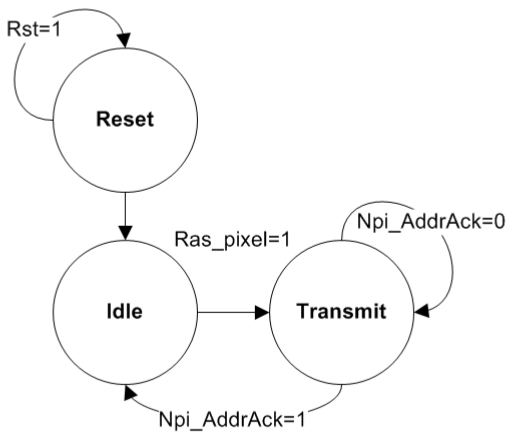


Figure 16: Rasterizer pixel write state machine

As can be seen in this diagram, the RAS pixel write state machine started in a *reset* state upon powerup of the graphics hardware. Immediately after this, it went into the *idle* state, where is waited until the system requested a pixel to be rasterized. When this request was received, the state machine pushed the pixel address within the backbuffer into the NPI's address FIFO and then changes to the *transmit* state.

While in the *transmit* state, the state machine holds the address push until the transaction was accepted, which is delineated by the NPI address acknowledge signal going high. Upon receiving this, the state machine then pushed the pixel color into the NPI data FIFO, and returns to the *idle* state. The state machine is then ready to rasterize another pixel, as needed. With the last pixel's address and color value in the respective NPI FIFOs, the MPMC will complete the write transaction to external memory automatically.

Lines Line rasterization was performed by developing an RTL implementation of Bresenham's Line Algorithm (see Chapter 2: Background Research for more information). Furthermore, this implementation wrapped the pixel rasterizer, due to the limited number of NPI interfaces available (as such, to rasterize a pixel, the hardware draws a line starting and ending at the same point). Bresenham's Line Algorithm (BLA) was implemented using 3 separate concurrent processes: (1) the setup process, (2) the pixel position approximation process and (3) the pixel color approximation process.

We implemented the setup process to complete any calculations necessary before line rasterization. These included calculating the change in red, green and blue values from the beginning of the line to the end of the line and triggering the division to calculate the color slope for these components. The system performed division in hardware serially using the Serial Division Module by John Clayton (licensed under the GNU Lesser General Public License from OpenCores.org). As can be seen in Section 5.1, an instance of this module was created for each color component. The dividend of the color division, the change in color from endpoint to endpoint, was shifted left 8 bits (and padded with zeros) in order to maintain fractional precision. The divisor of this division was the change in y-value of the line from endpoint to endpoint (or x-value, if the x-axis is the major axis). As this requires at least a 14-bit division (for the largest shifted color value, the 6-bit green field), 14 clock cycles are necessary to compute the division serially. For the sake of simplicity, all fields were defined as 16-bits, so there was a 16-bit latency in calculating the color slopes.

With the pixel approximation process, the line rasterizer calculated the X- and Y-positions of each subsequent pixel from endpoint to endpoint to closely approximate the line. This specifically was the core of the RTL implementation of Bresenham's Algorithm. To implement the algorithm, the major octant of the line was first determined (by comparing the change in X- and change in Y- of the line), and then the line was drawn pixel by pixel incrementally from the endpoint closest to the origin to the endpoint furthest away. Each pixel iteration increased by only one pixel, either in the X- or Y- directions, or both. This continued until a close approximation of the line was completed.

Lastly, the pixel color approximation process used the color slopes calculated in the setup process to interpolate color data points for each pixel between the color defined at one endpoint of the line and the color defined at the other. This was calculated at the beginning of each new pixel for the next pixel in sequence. The following example describes the necessary calculation:

$$Color_{Red} = (ColorSlope_{Red} * (Endpoint0_{Y-pos} - Pixel_{Y-pos})) + Endpoint0_{RedComponent}$$

With this equation, the color component (in this example, red) was calculated by multiplying the color slope of that component by the change in pixel position and adding the original color component at the first endpoint to it. This was a simple linear interpolation (i.e. Y=m*X + b). Furthermore, with the actual RTL implementation, the color slope was left shifted by 8-bits to maintain a fractional component through the division. Consequently, the end result needed to be shifted back 8-bits to the right.

Triangles Triangle rasterization was implemented by invoking the line rasterizer in two phases, (1) drawing the wireframe of the triangle and (2) filling the wireframe of the triangle using horizontal lines (see Chapter 2: Background Research for more information on this process). These two phases were implemented using a VHDL state machine. This state machine is described in Figure 17.

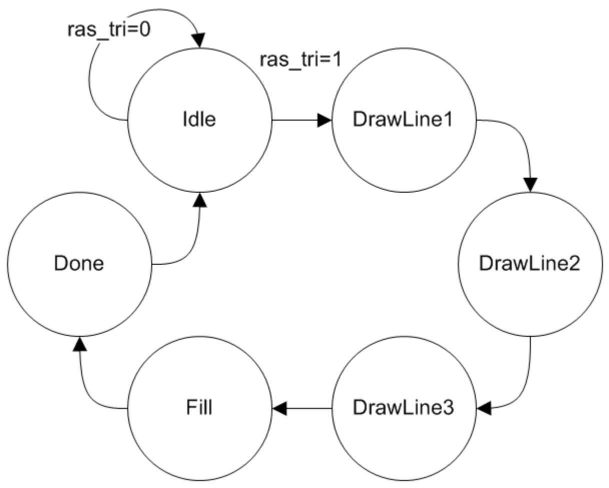


Figure 17: Triangle rasterization state machine

In this state machine, the first phase of the triangle rasterization was kicked off from the *idle* state when the `ras_tri` signal went high. This resulted in the first line of the triangle being rasterized by invoking the line rasterizer (which was previously instantiated within the triangle rasterizer) using two of the endpoints of the specified triangle. Subsequently, the next two lines of the triangle were rasterized within the next two states using the remaining endpoint of the triangle.

With the rasterization of the triangle wireframe, each pixel X-position and color value must be stored to perform the filling algorithm. Due to the fact that this could be a relatively sizable amount of data points dependent on the size of the triangle, these data needed to be stored in the Block RAM located within Spartan 6 FPGA. To do this, a Block RAM (BRAM) core was created via the Xilinx CoreGen, specified to have a 32-bit data width and a depth of 1440 words (to store three buffers of a depth of 480 words for each line of the triangle). As such, this contained enough storage for all pixel information of any possible triangle wireframe, as the X-position and color of each pixel of each line are stored at their respective Y-position address (each being 16-bits wide, both components were encoded in single 32-bit words).

In the next phase of triangle rasterization, the triangle was filled by first calculating which line (the major line) of the triangle had the greatest Y-distance between endpoints. Then, for each Y-position of the framebuffer, the state machine checked if there existed pixel information stored in the BRAM for both the major line and either of the two minor lines. If both of these conditions were satisfied, then a horizontal line was drawn via the line rasterizer between these two points, using the endpoint data previously stored in BRAM. This process was completed for every vertical pixel in the framebuffer (480 pixels total), which resulted in the triangle primitive being filled.

5.3 Summary

This chapter discussed the RTL implementation of the graphics core. It focused on how all of the main subsystems of the graphics core were implemented in hardware and the communication interfaces between them. In the next chapter (Chapter 6: Driver and API Implementation) the software architecture and implementations are discussed.

6 Driver and API Implementation

The driver for the 3D graphics hardware platform served as the interface between a user's OpenGL application and the platform's hardware. Running as a C module on the MicroBlaze softcore processor, this software library provided the framework to allow any user familiar with the OpenGL standard to run graphics applications on the hardware. This chapter describes the software implementation of the driver and application programming interface.

6.1 Architecture and Features

The implementation of the software driver and graphics API were sufficient for the proof of the project's concept, which was the realization of hardware-based graphics acceleration. The driver consisted of the following software components and capabilities:

1. Graphics Hardware Memory Interface
    (a) FIFO command and data queue
    (b) Hardware register read/write
2. Graphics Hardware Driver
    (a) Pixel and buffer data management
    (b) Hardware interrupt capability
3. OpenGL Implementation
    (a) Point, line and triangle primitive drawing

(b) Full color and material support
(c) Affine vertex transformations
(d) Parallel and perspective projection modes
(e) Ambient and diffuse lighting according to the Phong reflection model

The software necessary to implement graphics capabilities in the system was fully contained in items 1 and 2, the graphics hardware memory interface and the driver. Considering only these components, the user could hypothetically render a scene by sequentially drawing individual pixels or by sending vertex data. To make the graphics capabilities of the platform more accessible, users could also work in the system's custom implementation of OpenGL. However, many features provided in the vast OpenGL specification remain unimplemented and represent a significant area of potential future expansion. These areas are discussed in section 1.3. A full diagram of data flow in the driver is given in Figure 18.

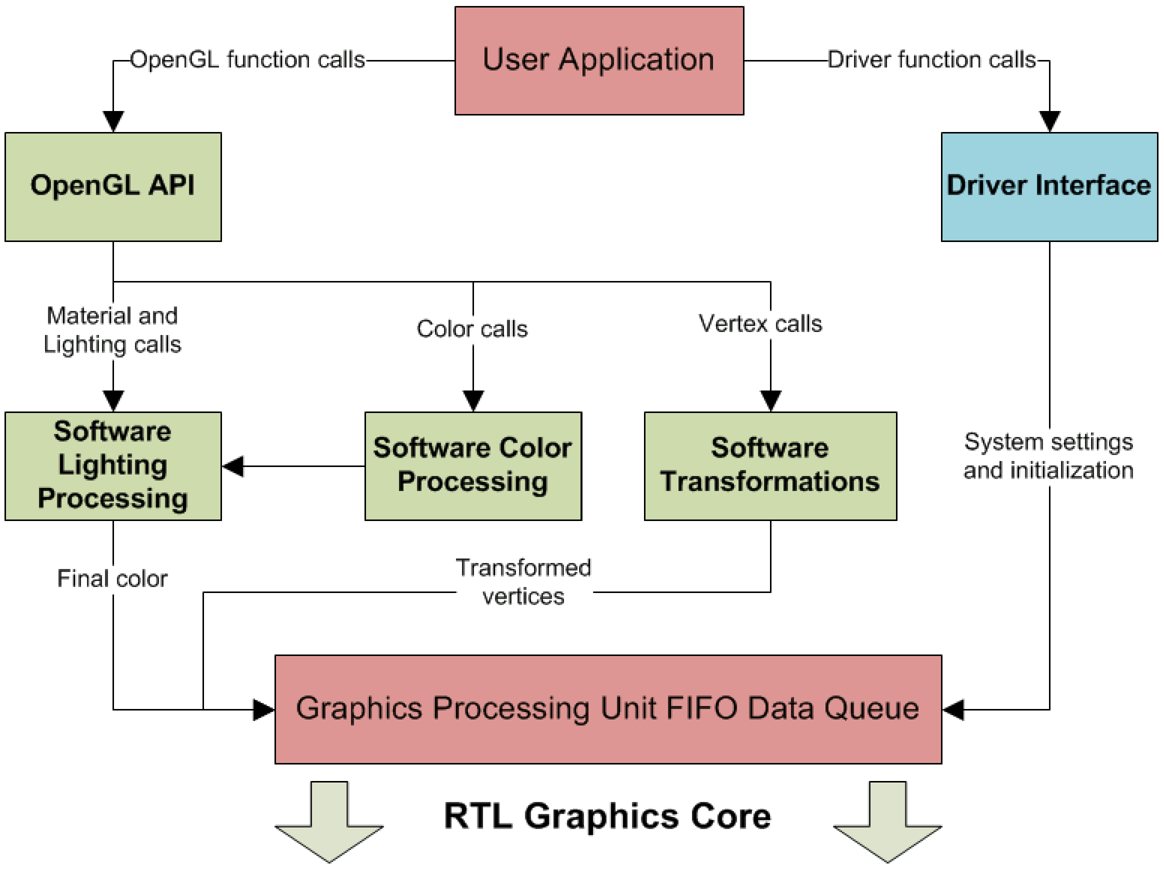


Figure 18: Driver and API data flow

6.2 Hardware Command Interface

The hardware interface is conceptually divided into a small subset of memory management routines and a larger subset of graphics capabilities routines that use them as part of the platform's API.

6.2.1 Memory Management

At the lowest level of the software interface, the platform provided several basic commands from which all of the graphics programming capabilities were devised. The driver's most basic function was to facilitate data to the graphics hardware FIFO queue and registers.

Driver Method Name	Purpose
GPU_FIFO_PUT32	Pushes a 32-bit word onto the FIFO queue
OGC_WriteReg	Writes a 32-bit word to a given hardware register
OGC_ReadReg	Reads the contents of a given hardware register

Table 4: Graphics hardware memory management

As seen in the table above, the three basic functions served to push data onto the FIFO queue (which were popped by the hardware programming in the GPU) and to read and write data to and from the hardware registers, respectively. Using these commands, the user may send any 32-bit word of data to the hardware and read the contents of any register from the hardware. When it was necessary in the driver to send less than 32 bits of data, we used a mask or offset to pad the data sent. In general, we used the FIFO queue to send data and commands related to pixels and vertices. We used the hardware register methods in the API to manage the display synchronization and interrupts and double buffering configuration. The following table presents the hardware register memory map.

Address	Register
`BASEADDR+0x00`	Front buffer address (R/W), 32-bit
`BASEADDR+0x04`	Back buffer address (R/W), 32-bit
`BASEADDR+0x08`	Initialization complete on write, (W), 32-bit
`BASEADDR+0x0C`	Clear Color (W), least significant 16-bits only
`BASEADDR+0x10`	Video interface status/fault register (R), 32-bit

Table 5: Hardware register memory map

6.2.2 Graphics Driver

We utilized the basic functionality of memory management in software by defining a set of command words in the driver that were read as operation codes by the hardware. These codes corresponded precisely to the RTL commands previously described in Section 5.2.1. The data prefaced by these codes were then interpreted by specific functions within the hardware program.

Data/Command Code	**Purpose**
OGC_CMD_NOP	No-Operation
OGC_CMD_DISPCNT	Display control
OGC_CMD_DRAWDONE	Informs the hardware that the driver has finished drawing to the backbuffer frame
OGC_CMD_PIXEL	Defines the next pixel in terms of screen coordinates and color value
OGC_CMD_VTX_BEGIN	Set the hardware to begin acceptance of vertex data
OGC_CMD_VTX_END	Set the hardware to terminate acceptance of vertex data
OGC_CMD_VTX_POS_X32	Defines current pixel 3D position in the X32 numerical data union format
OGC_CMD_VTX_COL_565	Define current pixel color in the RGB565 color format

Table 6: Graphics hardware FIFO commands

Using the operation codes in Table 6, the user could push many types of data and commands through the GPU FIFO queue for the hardware to interpret. These commands combined with the ability to read and write to various hardware registers (e.g. frame buffer and display control registers) provided us with the tools necessary to develop more advanced graphics capabilities. The complete list of graphics driver routines is described in Table 7.

Driver Method Name	Purpose
OGC_VI_DrawPixel	Sends single pixel data (X-position, Y-position, and 16-bit color value) through the FIFO queue
OGC_VI_SwapBuffers	Exchanges the registers containing front and back buffers, effectively drawing the next frame in an animation
OGC_VI_WaitForVSync	Executes a wait loop which allows the front buffer to finish drawing before swapping with the back buffer
OGC_EnableInterrupt	Sets the Interrupt Enabled Registers in hardware to allow the software to send interrupt messages
OGC_VI_VSyncIntrHandler	Interrupt handler for the Video Interface vertical synchronization interrupt
OGC_Init	Initializes the hardware components for software interface
OGC_Shutdown	Shuts down the hardware components' software interface

Table 7: Graphics hardware driver functions

Used together, these driver routines formed the basis of the graphics engine running on the hardware platform. These methods functioned together to allow the platform to draw and swap entire frame buffers filled with pixel data. To make use of this hardware capability to render graphics, we then implemented our own custom version of the OpenGL specification to work within the constraints of the platform's graphics driver.

6.3 OpenGL Implementation

For users to develop 3D graphics applications, we implemented the OpenGL specification for the graphics driver according to the provisions and limitations of our hardware platform. The driver defined many basic functions declared in the OpenGL standard libraries and used many data types, enumerated types, and constants typical of OpenGL implementations. Although several of the OpenGL functions included in our custom API served simply as wrappers for the most basic driver functions, others required extensive data and state management within the software system. Although the API we developed includes only a small fraction of the large quantity of OpenGL methods, it included enough functionality to fully realize all the steps of the basic graphics pipeline in conjunction with the hardware implementations.

6.3.1 State Management

In a typical implementation of the OpenGL specification, many of the data used in graphics calculations were set in the API as states. States in OpenGL were global in that any part of the application and API may access the data contained there. Once a state variable had been set by an API call, any future reference to that data invoked the data stored in that state. Only once the state had been modified by an API call may references to the data reflect any changes. Some examples of OpenGL states implemented in our API are given in Table 8.

OpenGL State Variable	Diver Struct	Purpose
GL_CURRENT_COLOR, GL_COLOR_MATERIAL	API_Color	Stored current color data used by vertex processing and whether color material mode was enabled
GL_CURRENT_NORMAL	API_Normal	Stored vertex normal vector coordinates used by lighting processing
GL_COLOR_MATERIAL_ PARAMETER	API_Material	Controlled the color material mode used by color processing

Table 8: Implemented OpenGL states

In our implementation, we defined some of these states in a combined fashion using data structures to store many thematically related state variables. In this way, states were conveniently grouped and tracked according to their effects and instantiated together as global structures. To modify states through the API, the application developer would need only use the standard OpenGL functions according to the specification, such as glEnable.

Supporting OpenGL states in this way would allow users familiar with OpenGL to develop their graphics applications in the usual way even though the driver interfacing with the hardware platform transmits data according to a much different protocol. In our implementation, states like the color and material attributes were finally used in color and lighting calculations in our glVertex function, at the end of which a simple burst of FIFO push commands sent the vertex positions and final color values. Consequently, the data stored in OpenGL states were used only by the API routines and never explicitly passed to hardware calculations. This protocol saved complexity and resource requirements on the FPGA and worked conveniently within the scope of the API.

6.3.2 OpenGL Features

We used open-source Mesa 3D OpenGL 1.1 header files, namely gl.h and glu.h, as the framework for our API implementation. These files provided the specification necessary for all graphics commands that might be invoked by the user in this environment. In a commercial implementation of OpenGL, there are extensive graphics features supported by a wide variety of functions, types and data structures. For the purposes of this project we defined twenty (20) OpenGL functions in our API. These functions are enumerated below in Table 9.

OpenGL Function	**Purpose**
glClear	Clears the display
glClearColor	Resets color values
glViewport	Defines screen dimensions
glEnable	Toggles OpenGL states
glDisable	Toggles OpenGL states
glBegin	Prepares driver for primitive data input
glEnd	Indicates conclusion of primitive data input
glColor3f	Sets OpenGL color state data
glMaterialf	Sets certain material parameters
glMaterialfv	Sets OpenGL material state data
glVertex3f	Defines a vertex in the API
glNormal3f	Sets OpenGL normal state data
glLightfv	Defines lights in the scene
glFrustum	Multiplies the current matrix by a perspective matrix
gluPerspective	Sets up a perspective matrix
glMatrixMode	Defines the current working matrix
glLoadIdentity	Loads an identity matrix onto matrix stack
glTranslatef	Performs affine translation
glScalef	Performs affine scaling
glRotatef	Performs affine rotation

Table 9: OpenGL Function Implementations

Preparing the Display Before any visualizations could be rendered in an OpenGL application, the screen needed to be initialized and prepared for input. Our API defined the OGC_Init() function as the method by which the driver initialized the graphics hardware and API global state variables. This method was not a feature of OpenGL, but was necessary in graphics applications running on our platform before any OpenGL functions could be used. After this step, the application programmer would proceed by invoking the typical sequence of OpenGL routines such as glClearColor() as necessary. Finally, the applications programmer could use the view and perspective functions to define the screen coordinate system and viewing volume.

Managing Matrices We fully implemented the modelview and perspective matrix transformations in our API in such a way that the user could rely on glMatrixMode() and glLoadIdentity() to function as expected to manage the perspective and modelview matrices. These matrices were allocated by the driver. In the current implementation, transformations and other mathematical operations on these were calculated in the driver due to resource limitations on the FPGA. Furthermore, our current implementation did not include a matrix stack.

Drawing Primitives In an OpenGL application, the standard method of drawing primitive shapes involves first preparing the driver to accept vertex data. Our API used the glBegin() and glEnd() functions according to specification including the enumerated values GL_POINTS, GL_LINES and GL_TRIANGLES to specify the type of primitive to draw. Other more complex primitive types were not supported by our implementation.

Defining Lights and Colors An application programmer using the graphics platform would have many options regarding lighting and color control within their graphics application. Our platform featured full support of the color data states and partial implementations of the material data states. Furthermore, we implemented many of the basic lighting states, available in the API.

To set the current vertex color, the glColor() function was used to define a 16-bit RGB565 color value in a global data structure that was accessible as a state anywhere in the application and the API. Additionally, the user could enable color material mode to define vertex material properties for applications that utilized lighting features. Since this implementation didn't distinguish between front- and back-faces, it was not possible to specify this option in the material parameters.

In applications where lighting may be used, the user had the option to either specify each change in material color state using glMaterialfv(), or to save time by enabling color material mode. When this mode was enabled, invoking the glColor() function served as a wrapper for glMaterial() and could be more convenient. To define lighting effects in this environment, the user could enable and disable lights according to the OpenGL specification lighting values. Our implementation included ambient and diffuse lighting modes, but did not include specular lighting effects. Directional lights could be specified through the glLightfv() function and need not be normalized by the user as that step was performed internally by the API before lighting calculations were made.

Ending the Application Our platform currently required the user to end each display loop with a custom function we developed, OGC_VI_WaitForVSync(). This is the method that enables double buffering by waiting for the front and back buffers to be swapped. Additionally, the user should terminate graphics applications on the platform with the custom method OGC_Shutdown() that performs all cleanup routines.

6.4 Summary

This chapter discussed the Driver and API implementation of the software components. It focused on the methods we used to tailor our graphics API to the hardware driver we developed while still faithfully implementing various core components of the OpenGL 1.1 specification. Examples of graphics applications developed using our API are available in Appendix E. In the next chapter (Chapter 7: Testing and Results) the entire graphics hardware and software system was tested to the extent of its capability and evaluated according to quantitative and qualitative criteria through a series of graphics applications we developed.

7 Testing and Results

To verify the functionality of the systems described in chapters 4, 5, 6, and 7, we carried out a number of tests. We performed qualitative tests to visually verify the implemented features and quantitative tests to benchmark the platform's performance. These tests were implemented as OpenGL applications that implemented these features and measured the performance in frames-per-second (FPS). Sample source code for many of these tests can be found in Appendix E: Simple OpenGL Demo Source.

For this project, we developed a software emulator that implemented all of the features of the intended hardware (See Appendix B: Graphics Accelerator Emulator for more information). We performed two phases of testing: the first was running the given demos in the emulator, and the second was running the given demos on the actual graphics hardware. The emulator served as a qualitative and quantitative benchmark that we attempted to match with the hardware implementation. This chapter compares these benchmarks for the two implementations for each demo.

All demos tested with the emulator were launched on a Core 2 Duo T9600 2.8GHz computer, with 4GB of RAM and Windows 7 Ultimate 64-bit. Due to the huge speed difference between the test computer and the embedded platform (which ran the demos at 75MHz), the embedded platform was expected to perform much slower than the emulator. With the hardware tests, all demos were ran on the XC6SLX45T FGG484-3C Spartan-6 FPGA and SP605 development board using a DVI-HDMI cable to display the results on a 32" SHARP Aquos LCD television. While the output display specifics are negligible, this television was used over a standard DVI-compatible monitor as it provided better color contrast for photography.

It also should be noted that all demos executed on the FPGA hardware had yellow text overlayed on the framebuffer to display frame rate data. Demos launched within the emulator instead displayed a frame rate on the title bar.

7.1 Framebuffer

The first demo that we implemented to test the functionality of the graphics core was a framebuffer demo. This framebuffer demo tested the 2D capabilities of the graphics core, which were essential as all 3D renderings are rasterized to a 2D plane. This demo generated a random fire pattern in C, and then wrote that pattern to the backbuffer. The graphics core then swapped the frame buffers, and this process was repeated. An 'X' pattern was also overlayed on top of the fire to verify that pixels are drawn exactly where they are written. The result of this demo was a smooth flame animation that can been seen running in the emulator in Figure 19, below.

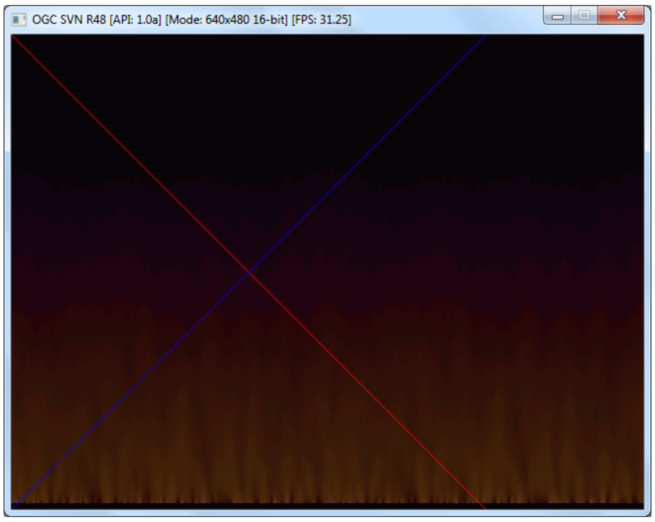


Figure 19: Framebuffer demo running in the emulator

With the emulator, this demo ran a consistent 30-35 FPS. However, as expected, the demo performed more slowly on the actual graphics hardware. The fire demo can be seen below running in Figure 20, below, on the FPGA.

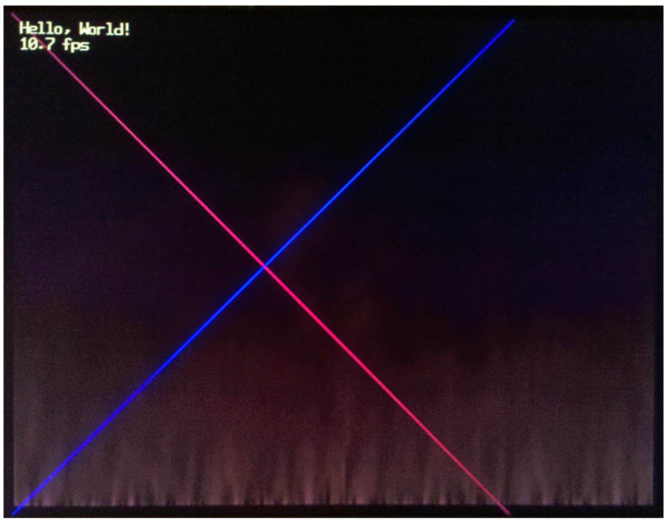


Figure 20: Framebuffer demo running on the FPGA

Qualitatively, the results were identical to the emulator, verifying that double buffering worked as expected, and that pixels appear as they were written to the framebuffer. Quantitatively, this demo averaged at approximately 10 FPS, which was quite slow. It must be noted that this was not due to a limitation in the RTL implementation, but rather the speed that the CPU generates the patterns. As will be seen in subsequent tests, demos that used just hardware for drawing performed substantially better. It also must be noted that for this demo, and the others presented in this chapter, colors values were significantly under saturated due to the photography quality. In the actual demos, the color values displayed from the FPGA implementation were essentially identical to their emulator counterparts.

7.2 Simple Triangle Drawing

The second test, the triangle demo, was implemented as a qualitative test to ensure the correct drawing of a single-color (no interpolation) triangle to a 2D plane. This test verified the rasterizer's ability to correctly approximate a triangle primitive, the building block of complex 3D scenes. This was a very simple OpenGL demo that simply sent three vertices to the graphics hardware. This demo can be seen running on the emulator in the figure below.

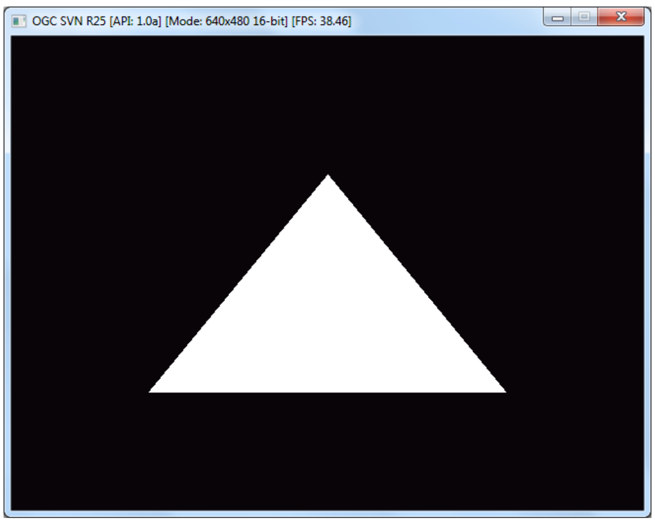


Figure 21: Triangle demo running on the emulator

Qualitatively, this was an ordinary rendering of a simple white triangle. Because this is only one triangle, the performance of the demo (in either emulator or hardware) was not relevant, as the speed limitation was within the overhead to setup a frame, not the actual drawing of it. This demo running on the graphics hardware implemented on the FPGA can be seen in Figure 22, below.

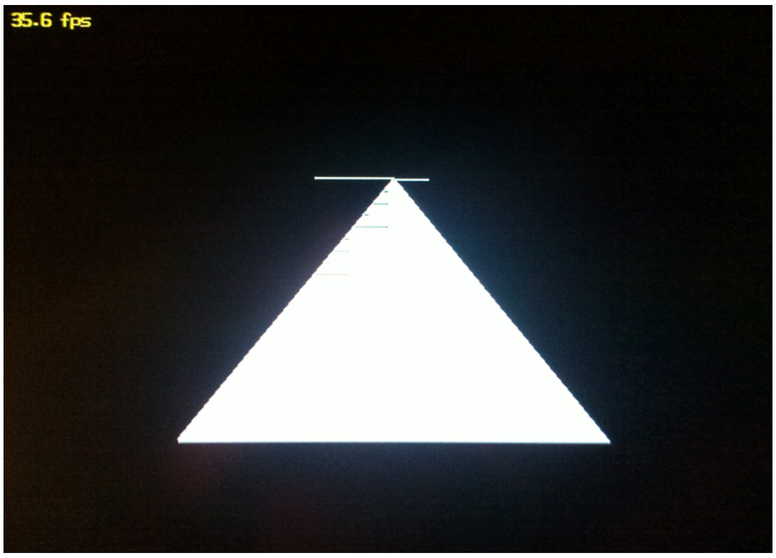


Figure 22: Triangle demo running on the FPGA

The most noticeable glitch in the hardware implementation of the project is very obvious in this photograph. That is, the white line appearing horizontally on the top of the triangle, which is an artifact of the filling algorithm used. However, it still may be visually observed that the filling works nearly as well as expected. Due to time constraints, we were unable to further develop the triangle rasterizer to the point where this bug was no longer present. However, the objective of triangle rasterization was still nonetheless achieved.

It may also be noted that this demo ran near the speed of the emulated test, which was roughly three orders of magnitude faster than the frame buffer demo running on hardware. This was, of course, due to the fact that the graphics core, not the MicroBlaze, was performing the drawing in this demo.

7.3 Transformation

In the next test, we used a demo to test vertex transformation. This demo used our OpenGL implementation to draw an octahedron (8 sided shape) of solidly filled triangles. Furthermore, this demo used OpenGL to scale, rotate, and translate the vertices of this object in 3D space. The result was a 3D octahedron spinning on the Y-axis. A screen capture of this can be seen in Figure 23, below, running on the emulator.

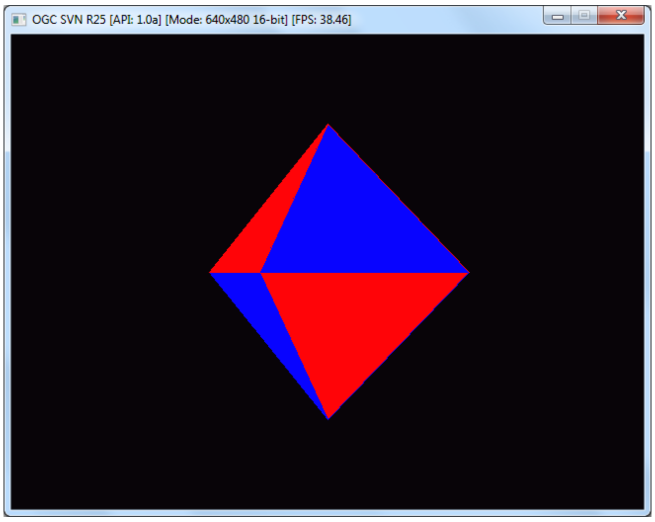


Figure 23: Transformation demo running on the emulator

As can be seen above, this demo still averaged at roughly 40 FPS. This demonstrated, again, that the bottleneck was not the rendering, but rather the frame setup. Next, this demo was launched on the actual FPGA graphics hardware, seen in Figure 24, below.

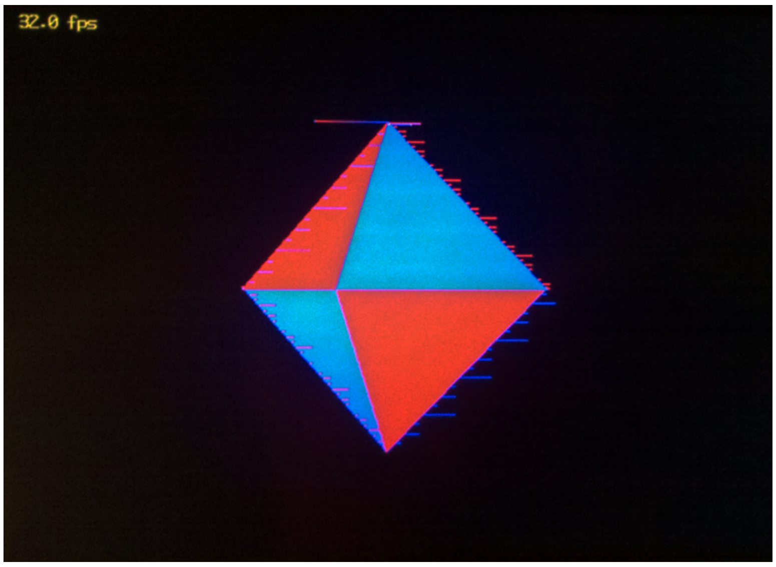


Figure 24: Transformation demo running on the FPGA

With this demo, as more triangles are drawn, the glitching due to the triangle filling has become much more apparent. However, the triangle primitives were still drawn nearly correctly, demonstrating that qualitatively the objective of triangle rasterization was essentially met. Furthermore, no new qualitative issues were presented by this test demo. All transformation, including scaling, rotating, translation, and perspective, were verified to function as expected. The frame rate of this demo, again, peaked at 32 FPS - the same as the single triangle demo - showing that the performance bottleneck, even on hardware, was still not in the number of vertices processed.

7.4 Color Interpolation

In the next test, the previous demo was modified to demonstrate color interpolation. This demo also rendered an octahedron, but this time rotating around the X-axis. Additionally, each vertex of each triangle was specified a different color (red, green, or blue) to give a smooth shaded effect. This smooth shading was generated by the ability of the graphics hardware to approximate color values at each pixel of the triangle. This can be seen as executed by the emulator in Figure 25, below.

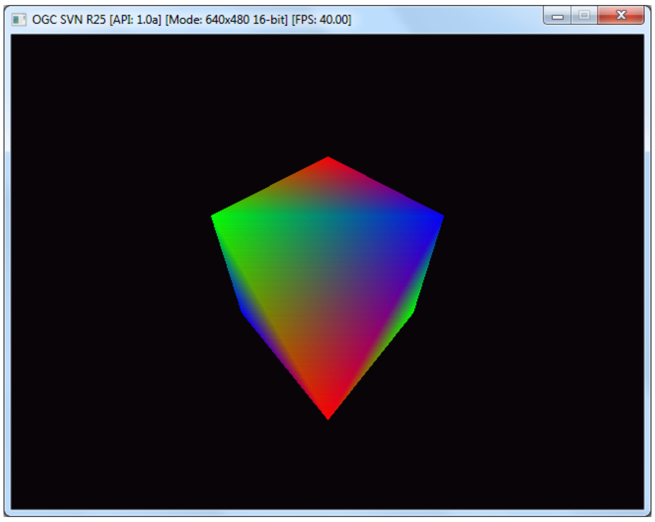


Figure 25: Transformation demo running on the emulator

Again, this demo was also tested in hardware. A photograph of the color interpolation test demo running on the FPGA can be seen in Figure 26, below.

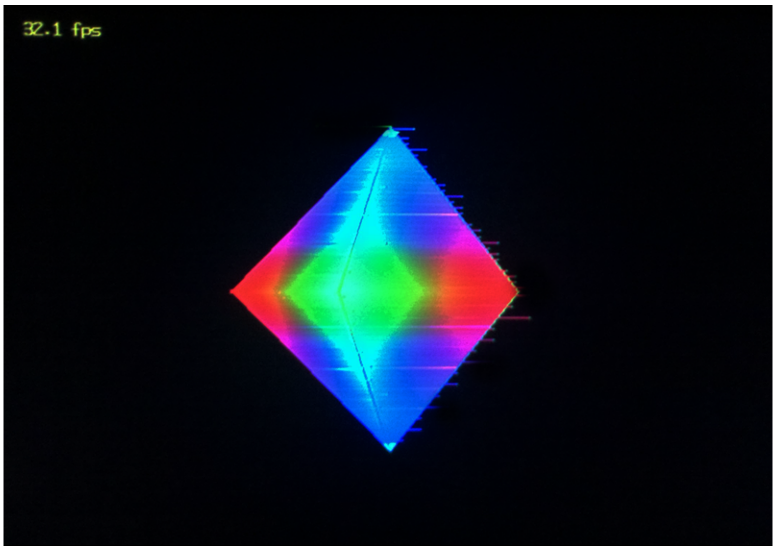


Figure 26: Color interpolation demo running on the FPGA

In this test, the triangle filling glitch is still apparent. However, as in the last demo, no new qualitative errors were introduced by this test. The demo was visually verified to interpolate the colors as expected, resulting in smoothly shaded triangles. Furthermore, as the same number of triangles were drawn, performance was near identical to the previous octahedron demo. This is due to the fact that the same line algorithm is utilized in RTL regardless of whether the two endpoints of a line have different color values. Discussion on this implementation may be found in chapter 5.

7.5 Lighting

The purpose of the lighting demo was to test the functionality of the software implementation of directional lighting. With this demo, a single directional light source was applied to the OpenGL color interpolation demo discussed in section 7.4. This demo used the OpenGL implementation of diffuse and ambient lighting. This can be seen running in the emulator with an average framerate between 38-42 FPS in the figure below.

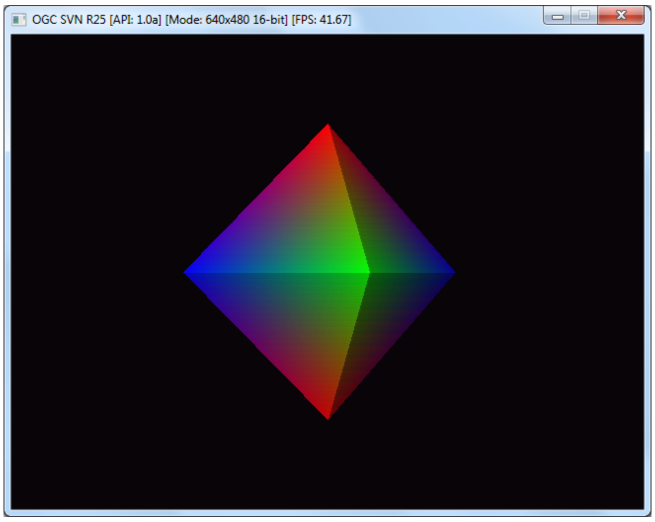


Figure 27: Lighting demo running in the emulator

A photograph of the same demo can be seen running on the actual graphics hardware FPGA implementation in the figure below.

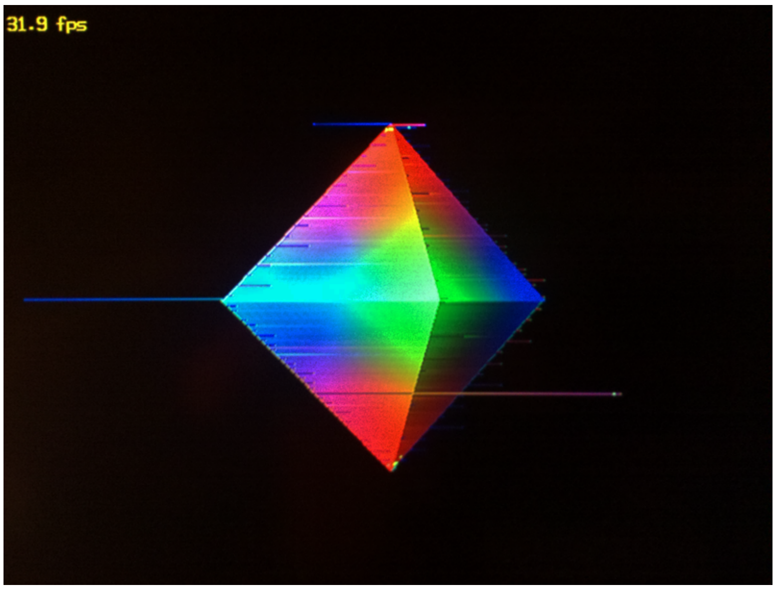


Figure 28: Lighting demo running on the FPGA

With this demo, the frame rate ran at a near constant 32 FPS, demonstrating the capabilities of the FPGA implementation of the graphics hardware. Even though the FPGA was clocked at only 75MHz (and the test computer at 2.8GHz), it was still able to run the demo at nearly 80% of the test computer's 40 FPS average. Qualitatively, as with the previous demos, some triangle filling glitching were evident by a few of the fill lines being drawn at seemingly random locations. Nonetheless, this demo demonstrated that the software lighting implementation worked as expected, as the 3D octahedron was still directionally lit correctly.

7.6 Complex Mesh

With the last demo, the purpose of the complex mesh test was to measure the limitations of the graphics hardware implementation. Consequently, the goal of this test was to approximate the number of triangles that could be drawn in a second. To do this, a humanoid 3D model that had 2012 triangle faces was drawn in OpenGL. This model was released royalty-free from TurboSquid, a website that distributes 3D art content. This mesh can be seen in Figure 29, below, rendered on a PC using a commercial graphics accelerator.

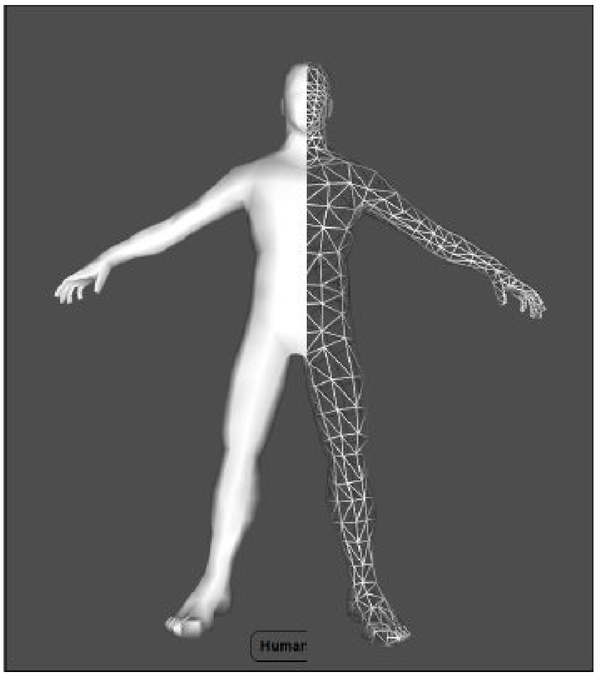


Figure 29: Humanoid model rendering on a PC
(Source: http://www.turbosquid.com/FullPreview/Index.cfm/ID/351964, 2010)

Next, the mesh was converted to OpenGL vertex calls using a Python script, and executed as an OpenGL demo on the FPGA-based graphics hardware. This can be seen in the figure below. No normals or color data were converted, the demo was purely written to test the number of triangles that the graphics hardware was capable of rendering.

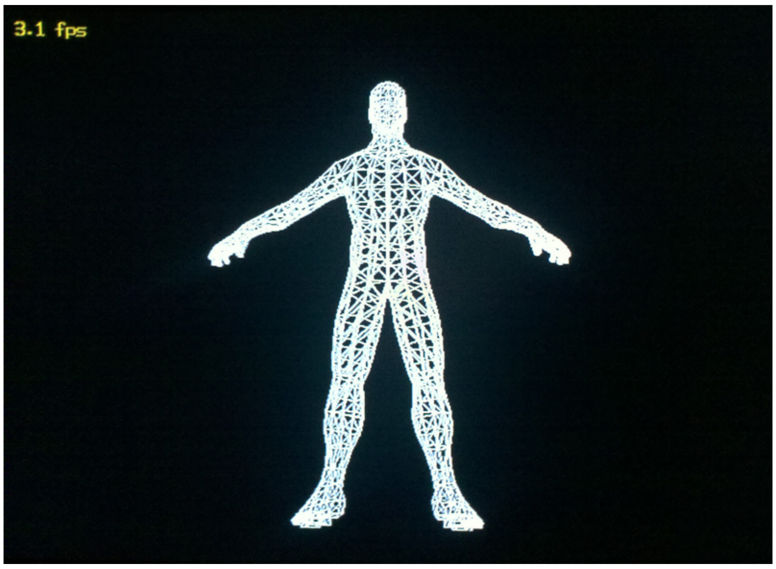


Figure 30: Humanoid mesh demo running on the FPGA

This demo was executed in wireframe mode so as to visually demonstrate the shear magnitude of triangles being drawn. As can be seen above, the demo qualitatively drew the humanoid mesh perfectly, albeit in wireframe. It must be noted that wireframe rasterization was not any faster than full triangle primitive rasterization. This was due to the fact that the bottleneck in speed in both situations was with the software perspective transformation, not the hardware rasterizer.

Quantitatively, this demo really pushed the limits of the graphics hardware, bringing it to an unbearably slow 3.1 FPS. Being that this mesh was 2012 triangles, this quantified the **graphics accelerators performance to be approximately 6240 triangles-per-second** (frame rate multiplied by the number of triangles being drawn). Unfortunately, this was only a fraction of the speed of modern 3D graphics accelerators, which are able to render millions of triangles per second. Regardless, the performance still demonstrated that as a proof of concept, the results for the project were very promising.

The main bottlenecks in this final performance test were determined to be the lack of optimizations within the RTL and the perspective transformation stages performed in software. Modern graphics accelerators utilize significantly more optimized and complex algorithms than could be implemented within the constraints of the project. Furthermore, the perspective transformation stages were originally intended to be implemented in RTL, but were instead implemented on a MicroBlaze CPU due to the time limitations that we experienced. Lastly, the demos were only running on a MicroBlaze clocked at 75MHz, which severely limited the speed that data could be sent to the RTL graphics hardware.

7.7 Summary

This chapter discussed the system testing that was performed to qualitatively and quantitatively verify the final implementation of the project. We found that most qualitative goals were achieved, except for a minor bug that still persisted within the triangle filling algorithm. Furthermore, we found that quantitatively, the implementation could achieve a maximum performance of approximately 6240 triangles-per-second. The next chapter (Chapter 8: Conclusions and Recommendations) contains conclusions drawn about the process we used as well as our recommendations for future work.

8 Conclusions and Recommendations

We identified four objectives in this project that would achieve our original goal of developing a 3D graphics accelerator on an FPGA. These were: (1) to implement the core entirely in a hardware description language, (2) to develop an API for this core by implementing various components of the OpenGL 1.1 specification, (3) to implement the core components of the 3D graphics pipeline and (4) to provide a series of technical demos that qualitatively and quantitatively tested and benchmarked the capabilities of our implementations. We have successfully realized these objectives and met our goal in the following ways.

We divided the fulfillment of these goals into four distinguishable phases consisting of background research, design, implementation and testing. In our background research, we identified previous work that attempted to solve the problem presented by our project. We furthermore explored background information relating to 3D graphics, as well as the concepts and algorithms necessary to implement our designs. In the design phase, we devised solutions to meet our goal and objectives. In the implementation phase, we developed all subsystems of the project, specifically implementing the various hardware and software stages of the 3D graphics pipeline per our designs. In order to verify that the preceding phases of the project met our objectives, we carried out tests in the benchmarking phase of our project to qualitatively and quantitatively measure the results of our implementations. Additionally, in the testing and benchmarking stages, we identified potential areas of improvement and suggestions for future work.

In response to the first objective, we determined that there were no true, complete, open implementations of a 3D graphics accelerator implemented in HDL. Consequently, we decided to develop an HDL implementation of a 3D graphics accelerator that would be easily customizable, portable, and low in cost. We did this by using the Spartan-6 FPGA and Xilinx SP605 Development board as our primary platform. We started by instantiating a MicroBlaze softcore CPU on the Spartan-6 to drive graphics applications and developing a hardware implementation of a 3D graphics core. This graphics core implemented various stages of the 3D graphics pipeline, providing a graphics processing unit command decoding interface, vertex setup, and primitive rasterization to a framebuffer located within external component memory. We implemented the basic necessities of the graphics pipeline in RTL logic to realize satisfactory hardware-accelerated graphics performance using a system-on-chip architecture.

As part of the hardware architecture, we implemented an instruction-based protocol that worked seamlessly with the OpenGL specification. As such, we were able to meet the second objective wherein the hardware fostered an environment that could easily adapt to the OpenGL structure. We implemented many OpenGL methods using several simple graphics core instructions. However, in the driver development, we attempted to implement more complex OpenGL features in software that could not be developed in hardware within our time and resource constraints. These included a partial implementation of the Phong lighting model, as well as affine rotation, translation, and scaling transformation. This resulted in enough of the core components of the OpenGL 1.1 specification to be implemented for an application programmer to be able to develop simple 3D applications, thus meeting our second objective.

The third objective, an implementation of the 3D graphics pipeline, was realized throughout the completion of all components of the project. For ideal performance speed, the platform would have implemented as much of the pipeline as possible in hardware. However, through the completion of the project, we were only able to develop a GPU command processor, primitive rasterizer, and output video signal interface. Consequently, in order to complete our implementation, we fulfilled the remaining core components of the pipeline in software. This included vertex setup, transformation and color calculation. However, as all components (hardware and software) were implemented through one means or another, we succeeded in addressing the third objective of our project.

Finally, we successfully met the fourth objective of our project through developing various technical demonstrations to test and benchmark the capabilities of our hardware and software implementations. We accomplished this aspect of our project through a series of OpenGL applications that utilized the primary features of our graphics core and demonstrated its capabilities. The demos rendered various images so that we could determine on a qualitative level whether we had correctly implemented each feature. Additionally, we also used the performance results of these applications to determine the speed capabilities of our graphics platform, in frames per second.

Through the development of this graphics accelerator, we found that our design process was effective. The approach that we took for implementing this, in first developing an embedded platform, then an RTL graphics hardware core, and lastly the software and driver components, proved remarkably effective. We mitigated several potential hardware and software integration setbacks by developing for our platform a software emulator that allowed us to develop the API and driver without requiring the hardware implementation to be complete.

We left some of the features of modern graphics hardware unimplemented in our project. Most importantly, various basic components of a 3D graphics accelerator were implemented in software rather than in the hardware. In order to attain optimal performance, we recommend that essentially all of these software components be implemented in hardware on the FPGA. Furthermore, we recommend pursing more complete implementations of the OpenGL specification, including the complete Phong reflection model. With future iterations of this project, we would recommend developing within hardware the support for clipping, depth sorting, and texturing. These features comprise many of the basic capabilities of OpenGL 1.1 that were beyond the scope of our work. Additional technologies beyond those already mentioned include programmable pixel and vertex processors that can be utilized for such advanced features as multitexturing, reflecting and refraction, per-pixel lighting, and shadow projection.

In conclusion, we were able to meet our objectives, resulting in a graphics accelerator platform that achieved reasonable performance within the scope of our project. We implemented all of the features that we planned for our 3D graphics accelerator and we determined that our graphics core was capable of rendering roughly 6240 triangles per second. These features, however, represent only some of the most basic features possible in a graphics accelerator. In comparison to modern commercial graphics platforms, the performance of our system in frames per second was very poor. While we were able to render complex 3D meshes, the speed at which the platform did so proved impractical for most animated applications. However, the results of our project still provided proof of concept for our goal of implementing a 3D graphics accelerator on an FPGA. Based on the results of our project, we believe that a full implementation of the 3D graphics pipeline with practical performance is attainable.

References

[1] Mesa 3d graphics library. http://www.mesa3d.org, accessed on December 9, 2010.

[2] Spartan-6 fpga. http://www.xilinx.com/products/spartan6/index.htm, accessed on December 9, 2010.

[3] Vincent 3d rendering library. http://www.vincent3d.com, accessed on December 9, 2010.

[4] Benj Carson and Jeff Mrochuk. Manticore.

[5] D. H. Eberly. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann, San Fransisco, CA, 1st edition, 2001.

[6] Kelley Stephen Hill, F.S. *Computer Graphics Using OpenGL*. Pearson Prentice Hall, Upper Saddle River, NJ, 3rd edition, 2007.

[7] J. Khatib. Fifo, first-in first-out memory. *Computer Based Learning Unit*, 1999.

[8] Alessio Malizia. *Mobile 3D Graphics*. Springer-Verlag, London, UK, 1st edition, 2006.

[9] R. C. Pendleton. Doing it fast: Fixed point arithmatic and transformations. 1997.

[10] T. Aarnio V. Miettinen K. Roimela Pulli, K. and J. Vaarala. *Mobile 3D Graphics with OpenGL ES and M3G*. Morgan Kaufmann, Burlington, MA, 1st edition, 2008.

[11] M. Woo J. Neider Shreiner, D. and T. Davis. *OpenGL Programming Guide*. Addison-Wesley, Boston, MA, 4th edition, 2003.

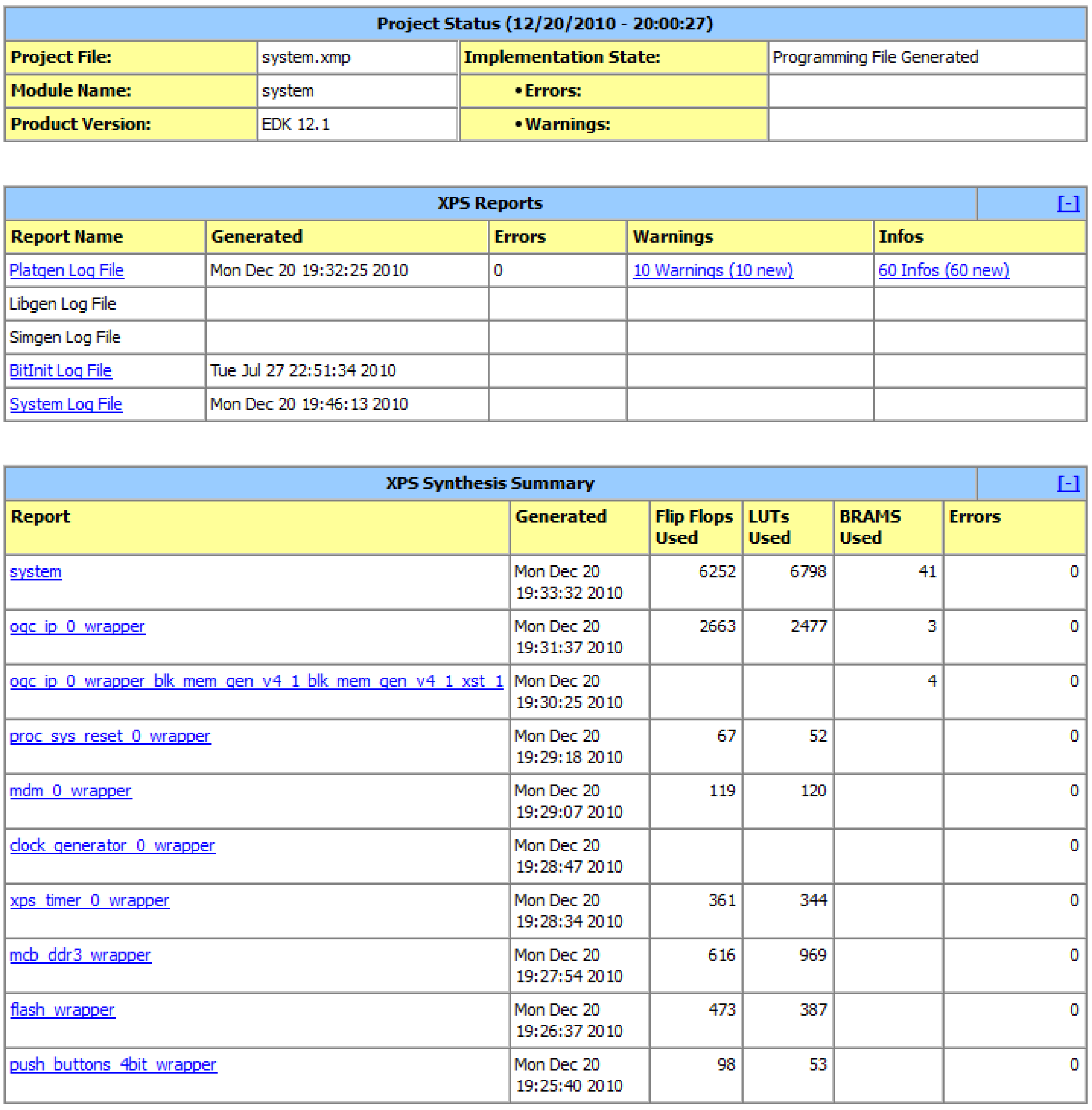
[12] T. Tessier. Using vhdl abstract data types to design a high performance 3-d graphics pipeline. *Design SuperCon 97*, 1996.

[13] G. Papaioannou N. Platis Theoharis, T. and N. M. Patrikalakis. *Graphics and Visualization: Principles and Algorithms*. A K Peters, Ltd., Wellesly, MA, 1st edition, 2008.

[14] J. M. Van Verth and L. M. Bishop. *Essential Mathematics for Games and Interactive Applications: A Programmer's Guide.* Morgan Kaufmann, San Fransisco, CA, 1st edition, 2004.

[15] R. Yates. Fixed point arithmetic: An introduction. 2001.

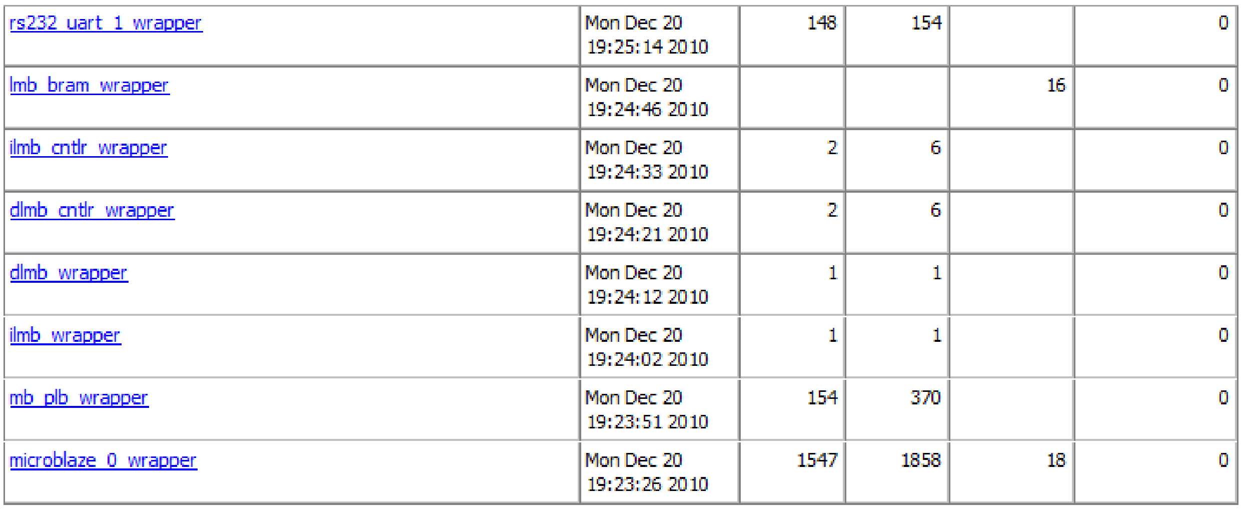
Appendix A: Hardware Synthesis Reports



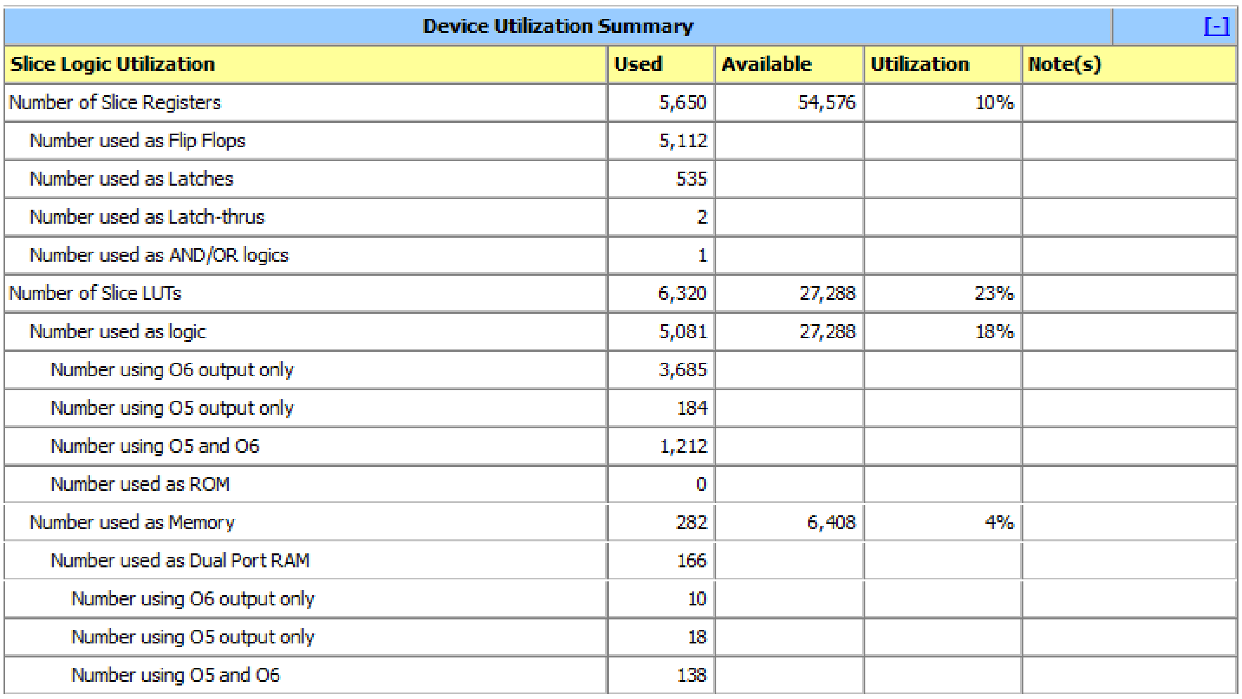
Project Status (12/20/2010 - 20:00:27)			
Project File:	system.xmp	**Implementation State:**	Programming File Generated
Module Name:	system	• **Errors:**	
Product Version:	EDK 12.1	• **Warnings:**	

XPS Reports				[-]
Report Name	**Generated**	**Errors**	**Warnings**	**Infos**
Platgen Log File	Mon Dec 20 19:32:25 2010	0	10 Warnings (10 new)	60 Infos (60 new)
Libgen Log File				
Simgen Log File				
BitInit Log File	Tue Jul 27 22:51:34 2010			
System Log File	Mon Dec 20 19:46:13 2010			

XPS Synthesis Summary					[-]
Report	**Generated**	**Flip Flops Used**	**LUTs Used**	**BRAMS Used**	**Errors**
system	Mon Dec 20 19:33:32 2010	6252	6798	41	0
ogc_ip_0_wrapper	Mon Dec 20 19:31:37 2010	2663	2477	3	0
ogc_ip_0_wrapper_blk_mem_gen_v4_1_blk_mem_gen_v4_1_xst_1	Mon Dec 20 19:30:25 2010			4	0
proc_sys_reset_0_wrapper	Mon Dec 20 19:29:18 2010	67	52		0
mdm_0_wrapper	Mon Dec 20 19:29:07 2010	119	120		0
clock_generator_0_wrapper	Mon Dec 20 19:28:47 2010				0
xps_timer_0_wrapper	Mon Dec 20 19:28:34 2010	361	344		0
mcb_ddr3_wrapper	Mon Dec 20 19:27:54 2010	616	969		0
flash_wrapper	Mon Dec 20 19:26:37 2010	473	387		0
push_buttons_4bit_wrapper	Mon Dec 20 19:25:40 2010	98	53		0

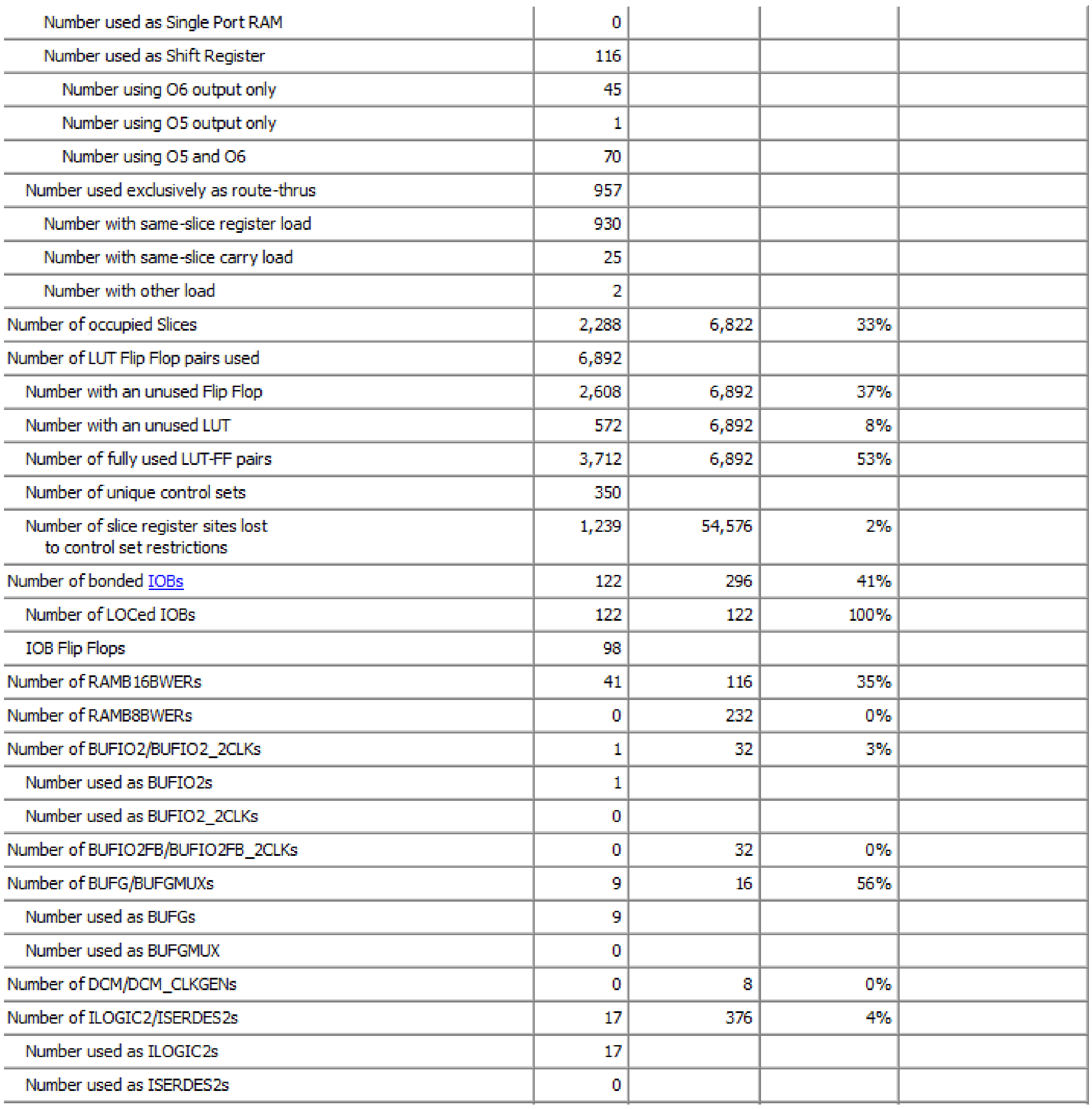


rs232_uart_1_wrapper	Mon Dec 20 19:25:14 2010	148	154		0
lmb_bram_wrapper	Mon Dec 20 19:24:46 2010			16	0
ilmb_cntlr_wrapper	Mon Dec 20 19:24:33 2010	2	6		0
dlmb_cntlr_wrapper	Mon Dec 20 19:24:21 2010	2	6		0
dlmb_wrapper	Mon Dec 20 19:24:12 2010	1	1		0
ilmb_wrapper	Mon Dec 20 19:24:02 2010	1	1		0
mb_plb_wrapper	Mon Dec 20 19:23:51 2010	154	370		0
microblaze_0_wrapper	Mon Dec 20 19:23:26 2010	1547	1858	18	0

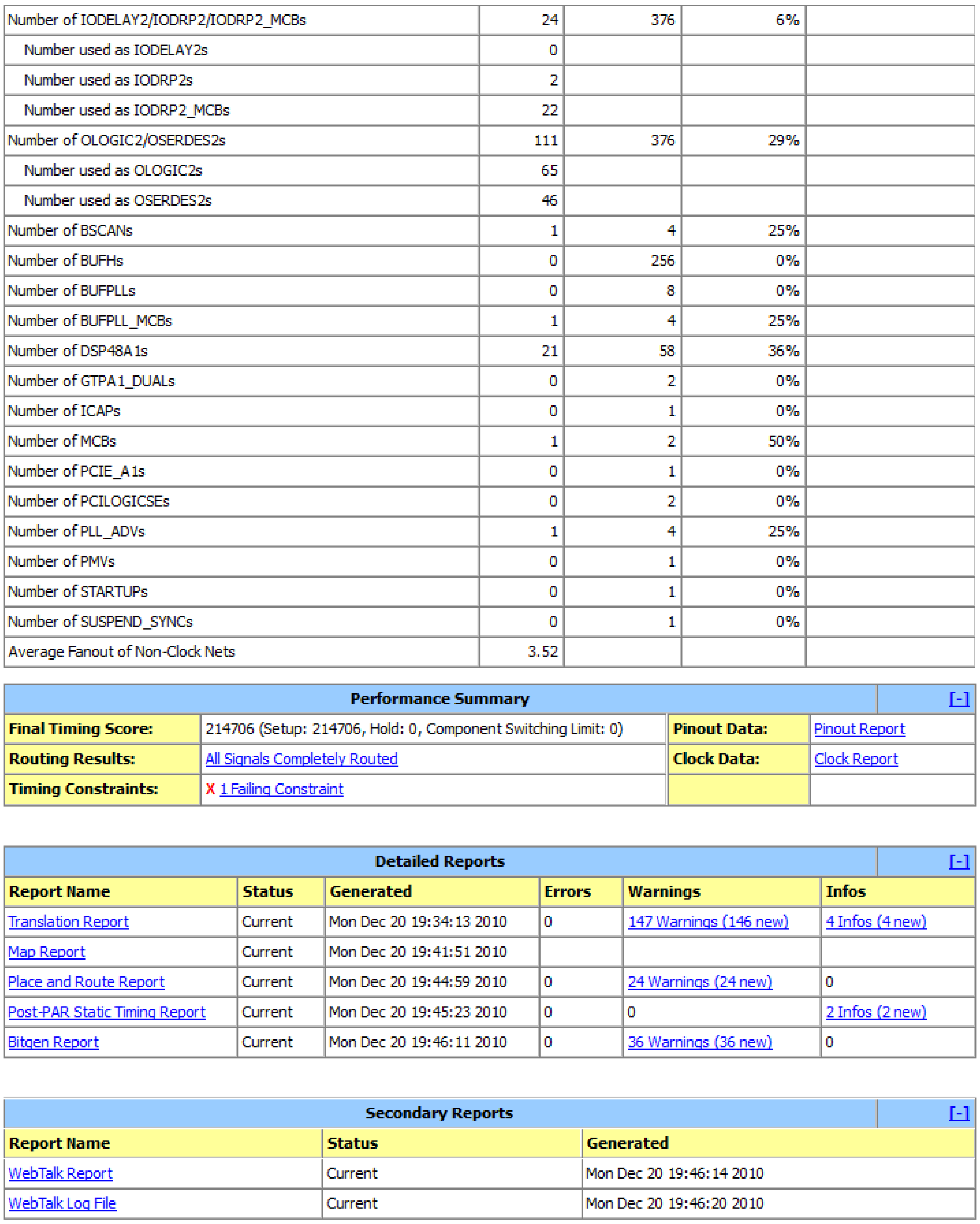


Device Utilization Summary [-]

Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	5,650	54,576	10%	
Number used as Flip Flops	5,112			
Number used as Latches	535			
Number used as Latch-thrus	2			
Number used as AND/OR logics	1			
Number of Slice LUTs	6,320	27,288	23%	
Number used as logic	5,081	27,288	18%	
Number using O6 output only	3,685			
Number using O5 output only	184			
Number using O5 and O6	1,212			
Number used as ROM	0			
Number used as Memory	282	6,408	4%	
Number used as Dual Port RAM	166			
Number using O6 output only	10			
Number using O5 output only	18			
Number using O5 and O6	138			



Number used as Single Port RAM	0			
Number used as Shift Register	116			
Number using O6 output only	45			
Number using O5 output only	1			
Number using O5 and O6	70			
Number used exclusively as route-thrus	957			
Number with same-slice register load	930			
Number with same-slice carry load	25			
Number with other load	2			
Number of occupied Slices	2,288	6,822	33%	
Number of LUT Flip Flop pairs used	6,892			
Number with an unused Flip Flop	2,608	6,892	37%	
Number with an unused LUT	572	6,892	8%	
Number of fully used LUT-FF pairs	3,712	6,892	53%	
Number of unique control sets	350			
Number of slice register sites lost to control set restrictions	1,239	54,576	2%	
Number of bonded IOBs	122	296	41%	
Number of LOCed IOBs	122	122	100%	
IOB Flip Flops	98			
Number of RAMB16BWERs	41	116	35%	
Number of RAMB8BWERs	0	232	0%	
Number of BUFIO2/BUFIO2_2CLKs	1	32	3%	
Number used as BUFIO2s	1			
Number used as BUFIO2_2CLKs	0			
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	9	16	56%	
Number used as BUFGs	9			
Number used as BUFGMUX	0			
Number of DCM/DCM_CLKGENs	0	8	0%	
Number of ILOGIC2/ISERDES2s	17	376	4%	
Number used as ILOGIC2s	17			
Number used as ISERDES2s	0			

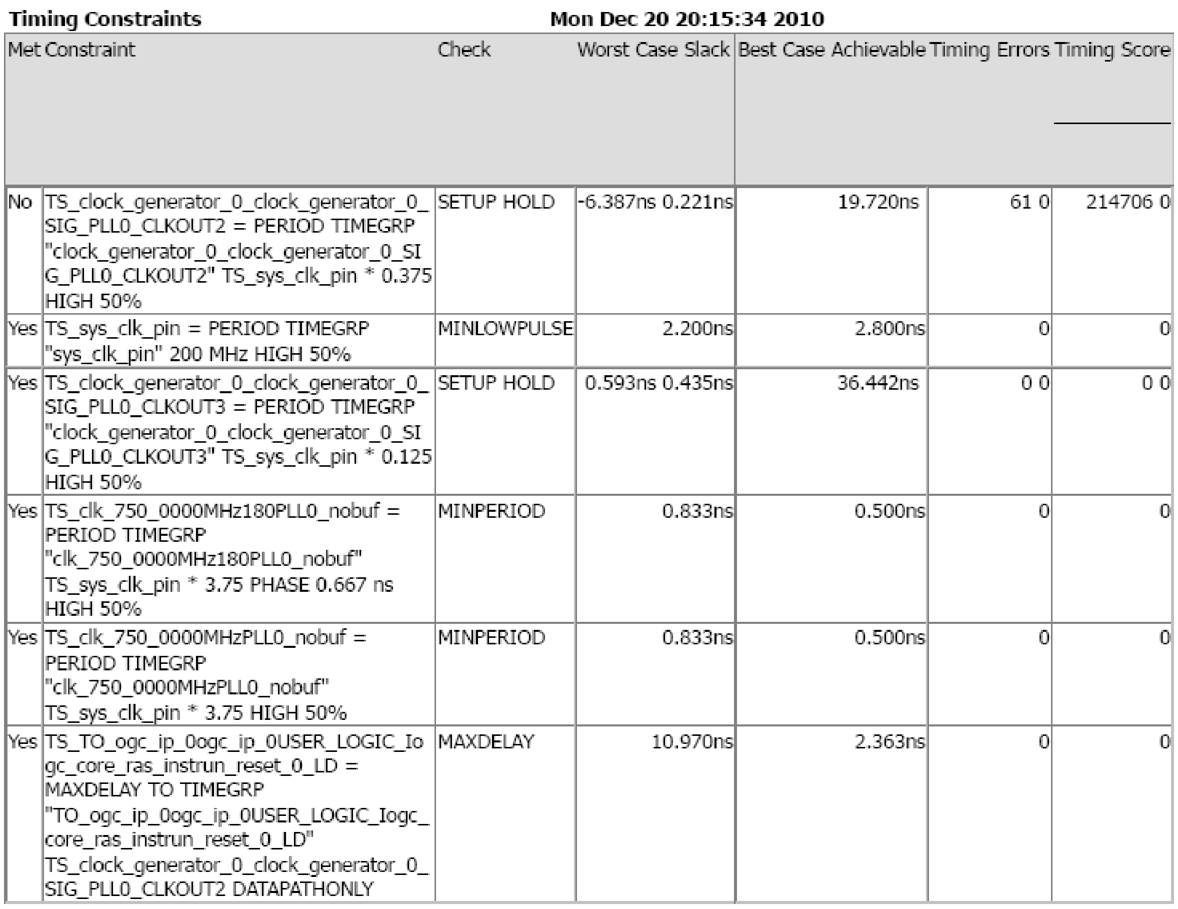


Number of IODELAY2/IODRP2/IODRP2_MCBs	24	376	6%	
Number used as IODELAY2s	0			
Number used as IODRP2s	2			
Number used as IODRP2_MCBs	22			
Number of OLOGIC2/OSERDES2s	111	376	29%	
Number used as OLOGIC2s	65			
Number used as OSERDES2s	46			
Number of BSCANs	1	4	25%	
Number of BUFHs	0	256	0%	
Number of BUFPLLs	0	8	0%	
Number of BUFPLL_MCBs	1	4	25%	
Number of DSP48A1s	21	58	36%	
Number of GTPA1_DUALs	0	2	0%	
Number of ICAPs	0	1	0%	
Number of MCBs	1	2	50%	
Number of PCIE_A1s	0	1	0%	
Number of PCILOGICSEs	0	2	0%	
Number of PLL_ADVs	1	4	25%	
Number of PMVs	0	1	0%	
Number of STARTUPs	0	1	0%	
Number of SUSPEND_SYNCs	0	1	0%	
Average Fanout of Non-Clock Nets	3.52			

Performance Summary			[-]
Final Timing Score:	214706 (Setup: 214706, Hold: 0, Component Switching Limit: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	X 1 Failing Constraint		

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Translation Report	Current	Mon Dec 20 19:34:13 2010	0	147 Warnings (146 new)	4 Infos (4 new)
Map Report	Current	Mon Dec 20 19:41:51 2010			
Place and Route Report	Current	Mon Dec 20 19:44:59 2010	0	24 Warnings (24 new)	0
Post-PAR Static Timing Report	Current	Mon Dec 20 19:45:23 2010	0	0	2 Infos (2 new)
Bitgen Report	Current	Mon Dec 20 19:46:11 2010	0	36 Warnings (36 new)	0

Secondary Reports		[-]
Report Name	**Status**	**Generated**
WebTalk Report	Current	Mon Dec 20 19:46:14 2010
WebTalk Log File	Current	Mon Dec 20 19:46:20 2010



Timing Constraints **Mon Dec 20 20:15:34 2010**

Met	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
No	TS_clock_generator_0_clock_generator_0_SIG_PLL0_CLKOUT2 = PERIOD TIMEGRP "clock_generator_0_clock_generator_0_SIG_PLL0_CLKOUT2" TS_sys_clk_pin * 0.375 HIGH 50%	SETUP HOLD	-6.387ns 0.221ns	19.720ns	61 0	214706 0
Yes	TS_sys_clk_pin = PERIOD TIMEGRP "sys_clk_pin" 200 MHz HIGH 50%	MINLOWPULSE	2.200ns	2.800ns	0	0
Yes	TS_clock_generator_0_clock_generator_0_SIG_PLL0_CLKOUT3 = PERIOD TIMEGRP "clock_generator_0_clock_generator_0_SIG_PLL0_CLKOUT3" TS_sys_clk_pin * 0.125 HIGH 50%	SETUP HOLD	0.593ns 0.435ns	36.442ns	0 0	0 0
Yes	TS_clk_750_0000MHz180PLL0_nobuf = PERIOD TIMEGRP "clk_750_0000MHz180PLL0_nobuf" TS_sys_clk_pin * 3.75 PHASE 0.667 ns HIGH 50%	MINPERIOD	0.833ns	0.500ns	0	0
Yes	TS_clk_750_0000MHzPLL0_nobuf = PERIOD TIMEGRP "clk_750_0000MHzPLL0_nobuf" TS_sys_clk_pin * 3.75 HIGH 50%	MINPERIOD	0.833ns	0.500ns	0	0
Yes	TS_TO_ogc_ip_0ogc_ip_0USER_LOGIC_Iogc_core_ras_instrun_reset_0_LD = MAXDELAY TO TIMEGRP "TO_ogc_ip_0ogc_ip_0USER_LOGIC_Iogc_core_ras_instrun_reset_0_LD" TS_clock_generator_0_clock_generator_0_SIG_PLL0_CLKOUT2 DATAPATHONLY	MAXDELAY	10.970ns	2.363ns	0	0

Appendix B: Graphics Accelerator Emulator

The graphics accelerator emulator was a software implementation of the proposed hardware design that allowed the software components of the project to be developed independently of the actual hardware implementation. The emulator accomplished this by implementing a software renderer that had all of the proposed functionality of the hardware and used the same command write interface for render control. The emulator fully implemented and tested the project's graphics API and test module components without ever needing the actual FPGA interfaces. Additionally, the emulator provided a test bed for implementing and debugging graphics algorithms and other design decisions. Furthermore, it provided a benchmark to compare with the hardware implementation, as seen in Chapter 7: Testing and Results.

Current Implementation

By the start of the hardware implementations of the project, all components of the graphics accelerator had been completed in the emulator. We planned its development period to fall between July and August, but we completed it well in advance of this time. We implemented the following:

- The **Video Interface**, with a 16-bit framebuffer and double-buffering and a simulated vertical blanking period, which uses Simple DirectMedia Layer (SDL) for windowing and graphics output, allowing for cross-compatibility.
- The **Rasterizer**, which incorporates fixed-point rasterization algorithms for lines, triangles, and points, as well as 16-bit color interpolation.
- The **Graphics Processing Unit** command interface, with a fully simulated FIFO and data write, and command decoding for all proposed commands.
- The **Vertex and Transformation Unit** for vertex decoding and computing affine transformations in 3D space.

Again, the design for this emulator was nearly identical to the actual hardware implementation project, except implemented entirely in software. Consequently, Chapter 3:

Project Overview and Design may be consulted for indications of how the emulator was realized.

Example Use

Using the emulator was very simple. To alternate between compiling demo applications within the emulator the and the actual hardware libraries, the C/C++ test module or application needed to simply define or undefine the identifier `USE_GRAPHICS_ACCELERATOR_EMU`. Both the emulator and the hardware implemented the same write interface, and as such no further code changes needed to be made. Below shows a simple code example that will draw a pixel to the framebuffer using the PIXEL command with the emulator or hardware interface:

```
// Initialize subsystems
GPU_OPEN();

Vertex myPixel;
myPixel.x._u16 = myPixel.y._u16 = 10;
myPixel.col.rgba = 0x001f;

// Main demo loop
for(;;) {
        // Push the PIXEL command into the FIFO
        GPU_FIFO_PUT8(PIXEL);

        // Push the PIXEL command parameters into the FIFO
        GPU_FIFO_PUT16(myPixel.x._u16); // Pixel x-coord
        GPU_FIFO_PUT16(myPixel.y._u16); // Pixel y-coord
        GPU_FIFO_PUT16(myPixel.col.rgba); // Pixel 16-bit color

        // Push the DRAWDONE command into the FIFO (Swap buffers)
        GPU_FIFO_PUT8(DRAWDONE);
}
```

This simple code excerpt demonstrates the basic nature of communication with the graphics accelerator, regardless of hardware or simulation. Both the emulator and the hardware implemented a frontend for `GPU_OPEN` to initialize the hardware, as well as `GPU_FIFO_PUT` to push commands and data into the FIFO queue to be processed sequentially by the GPU. Data can be pushed in as a byte, halfword, or word, however the hardware simply implements these as single padded 32-bit FIFO pushes. As described in the Chapter 3, these low level command calls were abstracted by the graphics API implementation, such that a programmer using the graphics accelerator need not access them directly. Figure 31, below, shows a more complex demo OpenGL lighting demo running within the graphics accelerator emulator.

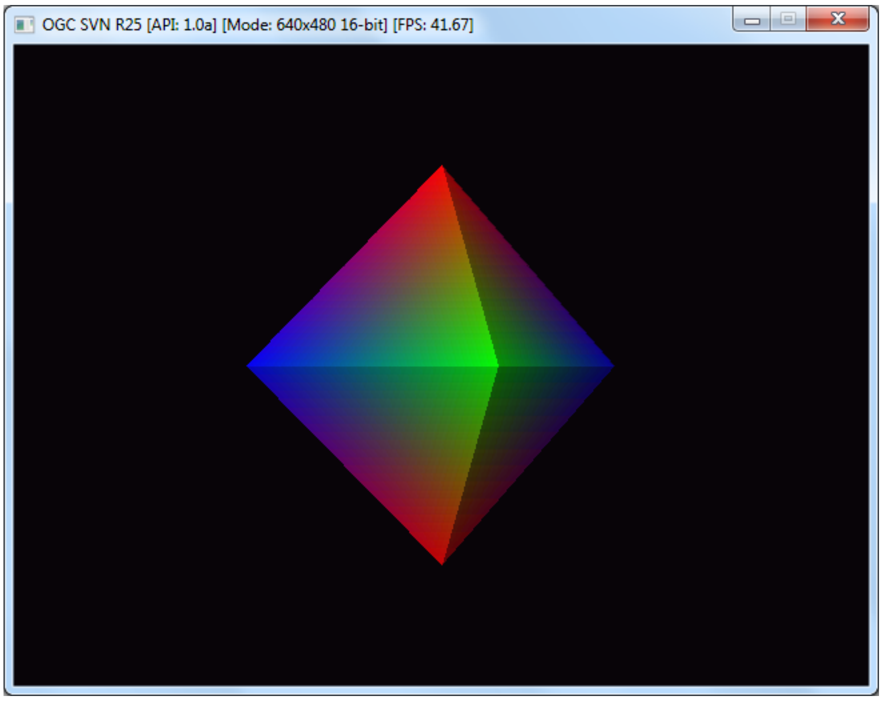


Figure 31: Emulator running an OpenGL demo

Furthermore, it is important to note that the emulator was developed to use the same exact driver sources as the hardware core, and that these were just simply recompiled depending on the build environment. This allowed us to test all of our software implementations locally before attempting to run them on our FPGA hardware.

Appendix C: Graphics Formats

C.1 RGB565

The RGB565 format is a 16-bit representation of an RGB (Red, Green, Blue) color. This was the only color format used throughout our project for a few primary reasons. Foremost, this was because our video output hardware only supported 16-bit color formats. Secondly, this format was chosen as it more accurately represents the visible color spectrum than High Color - which has 5 bits encoded for each color component. With RGB565, there is a 6th bit for the green component, as the green contributes most to the brightness of a color in the human eye. The figure below shows the color encoding of an RGB565 value:

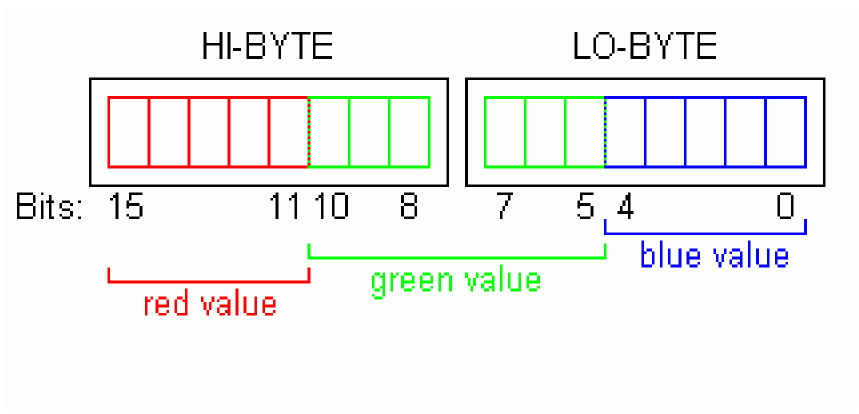


Figure 32: RGB565 color encoding
(Source: http://www.imagingcontrol.com/en_US/support/documentation/class/PixelformatRGB565.htm, 2010)

C.2 X32

To account for fractional precision on our FPGA hardware, we developed a signed 32-bit fixed-point number format, X32 (See Chapter 2: Background Research for more information on fixed-point mathematics). This format encoded a 16-bit signed integer number in the high 2 bytes of the data word, and the fractional component in the low 2 bytes. As such, the decimal point was "fixed" between bits 15 and 16 of the number. The figure below shows the encoding of this.

Integer component	**Fractional Component**
Bits 31-16	**Bits 15-0**

Figure 33: X32 number encoding

With each Nth-bit of the fractional component, an additional 1/Nth could be used to approximate decimal numbers. With 16-bits, this was enough precision to meet all of the needs of our applications. This furthermore could be easily and quickly processed on the FPGA and MicroBlaze CPU.

Appendix D: OpenGL Source

D.1 OGC_OPENGL.C

```
/*!  * Copyright (C) 2010 Eric M. Nadeau / Skyler B. Whorton
 *
 * \file    ogc_opengl.c
 * \author  Skyler B. Whorton <swhorton@wpi.edu>
 * \date    Created 22 Sept 2010
 * \brief   Implements OpenGL 1.0
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ogc.h>
#include <ogc_opengl.h>
API_GL_Vertex_Mode  API_Vertex_Mode;
API_GL_Lighting     API_Lighting;
API_GL_Color        API_Color;
API_GL_Material     API_Material;
API_GL_Normal       API_Normal;
u16                 g_reg_viewport_x        = 0;   ///< Viewport lower left x-value
u16                 g_reg_viewport_y        = 0;    ///< Viewport lower left y-value
u16                 g_reg_viewport_width    = 640;  ///< Viewport width (def: 640)
u16                 g_reg_viewport_height   = 480;  ///< Viewport height (def: 480)
extern u32*         vi_bfb_addr;
void apiInitGlobals() {

    API_Vertex_Mode.mode = GL_LINES;

    API_GL_Light API_Light0 = {0, {0.0, 0.0, 0.0, 1.0}, {1.0, 1.0, 1.0, 1.0},

        {1.0, 1.0, 1.0, 1.0} };

    API_GL_Light API_Light1 = {0, {0.0, 0.0, 0.0, 1.0}, {0.0, 0.0, 0.0, 1.0},

        {1.0, 1.0, 1.0, 1.0} };

    API_GL_Light API_Light2 = {0, {0.0, 0.0, 0.0, 1.0}, {0.0, 0.0, 0.0, 1.0},

        {1.0, 1.0, 1.0, 1.0} };

    API_GL_Light API_Light3 = {0, {0.0, 0.0, 0.0, 1.0}, {0.0, 0.0, 0.0, 1.0},

        {1.0, 1.0, 1.0, 1.0} };

    API_GL_Light API_Light4 = {0, {0.0, 0.0, 0.0, 1.0}, {0.0, 0.0, 0.0, 1.0},

        {1.0, 1.0, 1.0, 1.0} };

    API_GL_Light API_Light5 = {0, {0.0, 0.0, 0.0, 1.0}, {0.0, 0.0, 0.0, 1.0},

        {1.0, 1.0, 1.0, 1.0} };

    API_GL_Light API_Light6 = {0, {0.0, 0.0, 0.0, 1.0}, {0.0, 0.0, 0.0, 1.0},

        {1.0, 1.0, 1.0, 1.0} };

    API_GL_Light API_Light7 = {0, {0.0, 0.0, 0.0, 1.0}, {0.0, 0.0, 0.0, 1.0},

        {1.0, 1.0, 1.0, 1.0} };
    API_Lighting.isEnabled = (GLboolean) 0;
    API_Lighting.lights[0] = API_Light0;
    API_Lighting.lights[1] = API_Light1;
    API_Lighting.lights[2] = API_Light2;
    API_Lighting.lights[3] = API_Light3;
    API_Lighting.lights[4] = API_Light4;
    API_Lighting.lights[5] = API_Light5;
    API_Lighting.lights[6] = API_Light6;
    API_Lighting.lights[7] = API_Light7;
    API_Normal.currentNormal.x = 0.0;
    API_Normal.currentNormal.y = 0.0;
    API_Normal.currentNormal.z = 0.0;
}
void glClear(GLbitfield mask) {
    if (mask & GL_COLOR_BUFFER_BIT) {
#ifdef USE_OGC_EMU
        GPU_FIFO_PUT32(OGC_CMD_CLEAR);
        GPU_FIFO_PUT32(OGC_CNT_COLOR_BUFFER);
#else
        memset(vi_bfb_addr, 0, 640*480*2);
#endif
    }
}
void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha) {
    u8 r = (u8)(red * (GLclampf)OGC_FB_RED_MASK);
    u8 g = (u8)(green * (GLclampf)OGC_FB_GREEN_MASK);
    u8 b = (u8)(blue * (GLclampf)OGC_FB_BLUE_MASK);
#ifdef USE_OGC_EMU
    GPU_FIFO_PUT32(OGC_CMD_CLEAR_COLOR);
    GPU_FIFO_PUT32((r << 11) | (g << 5) | b);
#endif
}
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height) {
    g_reg_viewport_x        = x & 0xffff;
    g_reg_viewport_y        = y & 0xffff;
    g_reg_viewport_width    = width & 0xffff;
    g_reg_viewport_height   = height & 0xffff;
#ifdef USE_OGC_EMU
    GPU_FIFO_PUT32(OGC_CMD_VIEWPORT);
    GPU_FIFO_PUT32((g_reg_viewport_x << 16) | g_reg_viewport_y);
    GPU_FIFO_PUT32((g_reg_viewport_width << 16) | g_reg_viewport_height);
#endif
}
void glEnable(GLenum cap) {
    if (!apiSetEnabled(cap, (GLboolean) 1)) {
        LOG_WARNING(TAPI, "glEnable: Unsupported capacity 0x%08x!\n", cap);
    }
}
void glDisable(GLenum cap) {
    if (!apiSetEnabled(cap, (GLboolean) 0)) {
        LOG_WARNING(TAPI, "glDisable: Unsupported capacity 0x%08x!\n", cap);
    }
}
GLboolean apiSetEnabled(GLenum cap, GLboolean setting) {
    switch (cap) {
        case GL_LIGHTING:
            API_Lighting.isEnabled = setting;
            return 1;
        case GL_LIGHT0:
            API_Lighting.lights[0].isEnabled = setting;
            return 1;
        case GL_LIGHT1:
            API_Lighting.lights[1].isEnabled = setting;
            return 1;
        case GL_LIGHT2:
            API_Lighting.lights[2].isEnabled = setting;
            return 1;
        case GL_LIGHT3:
            API_Lighting.lights[3].isEnabled = setting;
            return 1;
        case GL_LIGHT4:
            API_Lighting.lights[4].isEnabled = setting;
            return 1;
        case GL_LIGHT5:
            API_Lighting.lights[5].isEnabled = setting;
            return 1;
        case GL_LIGHT6:
            API_Lighting.lights[6].isEnabled = setting;
            return 1;
        case GL_LIGHT7:
            API_Lighting.lights[7].isEnabled = setting;
            return 1;
        case GL_COLOR_MATERIAL:
            API_Color.colorMaterialEnabled = setting;
            return 1;
        default:
            return 0;
    }
}
void glLightfv(GLenum light, GLenum pname, const GLfloat * params) {
    int lightNumber;
    // Translate GL light enum to an integer to serve as API array index value
    switch (light) {
        case GL_LIGHT0:
            lightNumber = 0;
            break;
        case GL_LIGHT1:
            lightNumber = 1;
            break;
        case GL_LIGHT2:
            lightNumber = 2;
            break;
        case GL_LIGHT3:
            lightNumber = 3;
            break;
        case GL_LIGHT4:
            lightNumber = 4;
            break;
        case GL_LIGHT5:
            lightNumber = 5;
            break;
        case GL_LIGHT6:
            lightNumber = 6;
            break;
        case GL_LIGHT7:
            lightNumber = 7;
            break;
        default:
            lightNumber = -1;
            break;
    }
    // Validate light number 0-7. Call parameter "set" subroutine
    if (lightNumber >= 0 && lightNumber <= 7) {
        apiSetLightParam(lightNumber, pname, params);
    } else {
        LOG_WARNING(TAPI, "apiSetLightParam: Unsupported light number %d!\n", lightNumber);
        return;
    }
}
void apiSetLightParam(int lightNumber, GLenum pname, const GLfloat * params) {
    f32 apiParams[4];
    f32 mag;
    int i;
    // Copy parameter information into f32 formatted array
    for (i = 0; i <= 3; i++) {
        apiParams[i] = (f32) params[i];
    }
    // Set parameter "pname" in the API data structure to apiParams values
    switch (pname) {
        case GL_POSITION:
            // If supplying a direction vector (i.e. w == 1), normalize these values first
            if (apiParams[3] == 1.0) {
                mag = sqrt(pow(apiParams[0],2) + pow(apiParams[1],2) + pow(apiParams[2],2));
                apiParams[0] /= mag;
                apiParams[1] /= mag;
                apiParams[2] /= mag;
            }
            for (i = 0; i <= 3; i++) {
                API_Lighting.lights[lightNumber].position[i] = apiParams[i];
            }
            break;
        case GL_AMBIENT:
            for (i = 0; i <= 3; i++) {
                API_Lighting.lights[lightNumber].ambient[i] = apiParams[i];
            }
            break;
        case GL_DIFFUSE:
            for (i = 0; i <= 3; i++) {
                API_Lighting.lights[lightNumber].diffuse[i] = apiParams[i];
            }
            break;
        case GL_SPECULAR:
            for (i = 0; i <= 3; i++) {
                API_Lighting.lights[lightNumber].specular[i] = apiParams[i];
            }
            break;
        default:
            LOG_WARNING(TAPI, "apiSetLightParam: Unsupported parameter 0x%08x!\n", pname);
            return;
    }
}
```

D.2 OGC_OPENGL_MATRIX.C

```
/*!
 * Copyright (C) 2010 Eric M. Nadeau / Skyler B. Whorton
 *
 * \file    ogc_opengl_matrix.c
 * \author  Skyler B. Whorton <swhorton@wpi.edu>
 * \date    Created 22 Sept 2010
 * \brief   Implements all OGL 1.0 matrix related functions.
 *
 */
#include <math.h>
#include <ogc.h>
#include <ogc_opengl.h>
extern u16  g_reg_viewport_x;       ///< Viewport lower left x-value
extern u16  g_reg_viewport_y;       ///< Viewport lower left y-value
extern u16  g_reg_viewport_width;   ///< Viewport width (default: 640)
extern u16  g_reg_viewport_height;  ///< Viewport height (default: 480)
u32             g_mtx_mode = OGC_MODELVIEW;         ///< Current matrix mode
Matrix44        g_mtx_current[OGC_NUM_MATRICES];    ///< Current matrices
void MatrixMultiply44(Matrix44* _dst, Matrix44* _src0, Matrix44* _src1) {
    int i, j, k;
    x32 val;
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            val = 0;
            for (k = 0; k < 4; k++) {
                val += x32_mult(_src0->s[(i * 4) + k], _src1->s[(k * 4) + j]);
            }
            _dst->s[(i * 4) + j] = val;
        }
    }
}
/// Transforms a vertex in 3D space to 2D screen coordinates
void apiTransformVertex(Vertex* _p_vtx) {
#ifdef USE_OGC_EMU
#else
    x32 xf = _p_vtx->x._x32;
    x32 yf = _p_vtx->y._x32;
    x32 zf = _p_vtx->z._x32;
    Matrix44* pmtx = &g_mtx_current[OGC_MODELVIEW & 0xf];
    Matrix44* vmtx = &g_mtx_current[OGC_PROJECTION & 0xf];
    // Modelview transformation
    // ------------------------
    x32 x = x32_mult(pmtx->s[0], xf) + x32_mult(pmtx->s[4], yf) +
            x32_mult(pmtx->s[8], zf) + pmtx->s[12];
x32 y = x32_mult(pmtx->s[1], xf) + x32_mult(pmtx->s[5], yf) +
            x32_mult(pmtx->s[9], zf) + pmtx->s[13];
x32 z = x32_mult(pmtx->s[2], xf) + x32_mult(pmtx->s[6], yf) +
            x32_mult(pmtx->s[10], zf) + pmtx->s[14];
    x32 w = kOne;
    xf = x;
    yf = y;
    zf = z;
    // Perspective transformation
    // --------------------------
    x = x32_mult(vmtx->s[0], xf) + x32_mult(vmtx->s[4], yf) +
        x32_mult(vmtx->s[8], zf) + vmtx->s[12];
```

```
y = x32_mult(vmtx->s[1], xf) + x32_mult(vmtx->s[5], yf) +
        x32_mult(vmtx->s[9], zf) + vmtx->s[13];
z = x32_mult(vmtx->s[2], xf) + x32_mult(vmtx->s[6], yf) +
        x32_mult(vmtx->s[10], zf) + vmtx->s[14];
    w = x32_mult(vmtx->s[3], xf) + x32_mult(vmtx->s[7], yf) +
        x32_mult(vmtx->s[11], zf) + vmtx->s[15];
    x = x32_div(x, w);
    y = x32_div(y, w);
    z = x32_div(z, w);
    x32 width = x32_encode_s32(g_reg_viewport_width >> 1);
    x32 height = x32_encode_s32(g_reg_viewport_height >> 1);
    _p_vtx->x._u32 = (u32) x32_decode_s32(x32_mult(width, x)) +
         (g_reg_viewport_width >> 1) + g_reg_viewport_x;
    _p_vtx->y._u32 = (u32) x32_decode_s32(x32_mult(height, y)) +
         (g_reg_viewport_height >> 1) + g_reg_viewport_y;
     _p_vtx->z._u32 = z;
#endif
}
void glFrustum(GLdouble left, GLdouble right,
               GLdouble bottom, GLdouble top,
               GLdouble near, GLdouble far) {
#ifdef USE_OGC_EMU
    GPU_FIFO_PUT32(OGC_CMD_MTX_MULT_44);
GPU_FIFO_PUT32(x32_encode_f32((2*(f32)near) / ((f32)right - (f32)left)));
GPU_FIFO_PUT32(0);
GPU_FIFO_PUT32(0);
GPU_FIFO_PUT32(0);
GPU_FIFO_PUT32(0);
GPU_FIFO_PUT32(x32_encode_f32((2*(f32)near) / ((f32)top - (f32)bottom)));
GPU_FIFO_PUT32(0);
GPU_FIFO_PUT32(0);
GPU_FIFO_PUT32(x32_encode_f32(((f32)right + (f32)left) / ((f32)right - (f32)left)));
GPU_FIFO_PUT32(x32_encode_f32(((f32)top + (f32)bottom) / ((f32)top - (f32)bottom)));
GPU_FIFO_PUT32(x32_encode_f32(-((f32)far + (f32)near) / ((f32)far - (f32)near)));
GPU_FIFO_PUT32(-kOne);
GPU_FIFO_PUT32(0);
GPU_FIFO_PUT32(0);
GPU_FIFO_PUT32(x32_encode_f32(-(2 * ((f32)far*(f32)near)) / ((f32)far - (f32)near)));
GPU_FIFO_PUT32(0);
#else
    Matrix44 temp0 = {{
        x32_encode_f32((2*(f32)near) / ((f32)right - (f32)left)),
        0,
        0,
        0,
        0,
        x32_encode_f32((2*(f32)near) / ((f32)top - (f32)bottom)),
        0,
        0,
        x32_encode_f32(((f32)right + (f32)left) / ((f32)right - (f32)left)),
        x32_encode_f32(((f32)top + (f32)bottom) / ((f32)top - (f32)bottom)),
        x32_encode_f32(-((f32)far + (f32)near) / ((f32)far - (f32)near)),
        -kOne,
        0,
        0,
        x32_encode_f32(-(2 * ((f32)far*(f32)near)) / ((f32)far - (f32)near)),
        0
        }};
    Matrix44 temp1 = g_mtx_current[g_mtx_mode & 0xf];
    MatrixMultiply44(&g_mtx_current[g_mtx_mode & 0xf],
                     &temp0,
                     &temp1);
#endif
}
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar) {
   f32 xmin, xmax, ymin, ymax;
   ymax = (f32)zNear * tan((f32)fovy * 3.1415972 / 360.0);
   ymin = -ymax;
   xmin = ymin * (f32)aspect;
   xmax = ymax * (f32)aspect;
   glFrustum(xmin, xmax, ymin, ymax, zNear, zFar);
}
void glMatrixMode(GLenum mode) {
    u32 ogc_mode;
    switch(mode) {
        case GL_MODELVIEW:
            ogc_mode = OGC_MODELVIEW;
            break;
        case GL_PROJECTION:
            ogc_mode = OGC_PROJECTION;
            break;
        case GL_TEXTURE:
        case GL_COLOR:
        default:
            LOG_WARNING(TRAS, "glMatrixMode: Unsupported matrix type 0x%08x!\n", mode);
            return;
    }
#ifdef USE_OGC_EMU
    GPU_FIFO_PUT32(OGC_CMD_MTX_MODE);
    GPU_FIFO_PUT32(ogc_mode);
#else
    g_mtx_mode = ogc_mode;
#endif
}
void glLoadIdentity(void) {
#ifdef USE_OGC_EMU
    GPU_FIFO_PUT32(OGC_CMD_MTX_LOAD_I);
#else
    g_mtx_current[g_mtx_mode & 0xf].s[0] = kOne;
    g_mtx_current[g_mtx_mode & 0xf].s[1] = 0;
    g_mtx_current[g_mtx_mode & 0xf].s[2] = 0;
    g_mtx_current[g_mtx_mode & 0xf].s[3] = 0;
    g_mtx_current[g_mtx_mode & 0xf].s[4] = 0;
    g_mtx_current[g_mtx_mode & 0xf].s[5] = kOne;
    g_mtx_current[g_mtx_mode & 0xf].s[6] = 0;
    g_mtx_current[g_mtx_mode & 0xf].s[7] = 0;
    g_mtx_current[g_mtx_mode & 0xf].s[8] = 0;
    g_mtx_current[g_mtx_mode & 0xf].s[9] = 0;
    g_mtx_current[g_mtx_mode & 0xf].s[10] = kOne;
    g_mtx_current[g_mtx_mode & 0xf].s[11] = 0;
    g_mtx_current[g_mtx_mode & 0xf].s[12] = 0;
    g_mtx_current[g_mtx_mode & 0xf].s[13] = 0;
    g_mtx_current[g_mtx_mode & 0xf].s[14] = 0;
    g_mtx_current[g_mtx_mode & 0xf].s[15] = kOne;
#endif;
}
```

```
void glTranslatef(GLfloat x, GLfloat y, GLfloat z) {
#ifdef USE_OGC_EMU
    GPU_FIFO_PUT32(OGC_CMD_MTX_TRANS);
    GPU_FIFO_PUT32((u32)x32_encode_f32(x));  // Translate X-component
    GPU_FIFO_PUT32((u32)x32_encode_f32(y));  // Translate Y-component
    GPU_FIFO_PUT32((u32)x32_encode_f32(z));  // Translate Z-component
#else
    Matrix44 temp0 = {{
        kOne,
        0,
        0,
        0,
        0,
        kOne,
        0,
        0,
        0,
        0,
        kOne,
        0,
        x32_encode_f32(x),
        x32_encode_f32(y),
        x32_encode_f32(z),
        kOne
        }};
    Matrix44 temp1 = g_mtx_current[g_mtx_mode & 0xf];
    MatrixMultiply44(&g_mtx_current[g_mtx_mode & 0xf],
                     &temp0,
                     &temp1);
#endif
}
void glScalef(GLfloat x, GLfloat y, GLfloat z) {
#ifdef USE_OGC_EMU
    GPU_FIFO_PUT32(OGC_CMD_MTX_SCALE);
    GPU_FIFO_PUT32((u32)x32_encode_f32(x));  // Translate X-component
    GPU_FIFO_PUT32((u32)x32_encode_f32(y));  // Translate Y-component
    GPU_FIFO_PUT32((u32)x32_encode_f32(z));  // Translate Z-component
#else
    Matrix44 temp0 = {{
        x32_encode_f32(x),
        0,
        0,
        0,
        0,
        x32_encode_f32(y),
        0,
        0,
        0,
        0,
        x32_encode_f32(z),
        0,
        0,
        0,
        0,
        kOne
        }};
    Matrix44 temp1 = g_mtx_current[g_mtx_mode & 0xf];
    MatrixMultiply44(&g_mtx_current[g_mtx_mode & 0xf],
                      &temp0,
                      &temp1);
#endif
}
void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z) {
f32 axis[3];
f32 sine = sin(angle);
f32 cosine = cos(angle);
    f32 one_minus_cosine = 1.0f - cosine;
axis[0]=x;
axis[1]=y;
axis[2]=z;
#ifdef USE_OGC_EMU
    GPU_FIFO_PUT32(OGC_CMD_MTX_MULT_44);
GPU_FIFO_PUT32(

        (u32)x32_encode_f32(cosine + (one_minus_cosine * (axis[0] * axis[0]))));

GPU_FIFO_PUT32(

        (u32)x32_encode_f32((one_minus_cosine * (axis[0] * axis[1])) + (axis[2] * sine)));

GPU_FIFO_PUT32(

        (u32)x32_encode_f32(((one_minus_cosine * axis[0]) * axis[2]) - (axis[1] * sine)));

    GPU_FIFO_PUT32(0);
GPU_FIFO_PUT32(

         (u32)x32_encode_f32(((one_minus_cosine * axis[0]) * axis[1]) - (axis[2] * sine)));

GPU_FIFO_PUT32(

        (u32)x32_encode_f32(cosine + ((one_minus_cosine * axis[1]) * axis[1])));

    GPU_FIFO_PUT32(

        (u32)x32_encode_f32(((one_minus_cosine * axis[1]) * axis[2]) + (axis[0] * sine)));

    GPU_FIFO_PUT32(0);
GPU_FIFO_PUT32(

        (u32)x32_encode_f32(((one_minus_cosine * axis[0]) * axis[2]) + (axis[1] * sine)));

GPU_FIFO_PUT32(

        (u32)x32_encode_f32(((one_minus_cosine * axis[1]) * axis[2]) - (axis[0] * sine)));

GPU_FIFO_PUT32(

        (u32)x32_encode_f32(cosine + ((one_minus_cosine * axis[2]) * axis[2])));

    GPU_FIFO_PUT32(0);
    GPU_FIFO_PUT32(0);
    GPU_FIFO_PUT32(0);
    GPU_FIFO_PUT32(0);
    GPU_FIFO_PUT32((u32)kOne);
#else
    Matrix44 temp0 = {{
        x32_encode_f32(cosine + (one_minus_cosine * (axis[0] * axis[0]))),
        x32_encode_f32((one_minus_cosine * (axis[0] * axis[1])) + (axis[2] * sine)),
        x32_encode_f32(((one_minus_cosine * axis[0]) * axis[2]) - (axis[1] * sine)),
        0,
        x32_encode_f32(((one_minus_cosine * axis[0]) * axis[1]) - (axis[2] * sine)),
        x32_encode_f32(cosine + ((one_minus_cosine * axis[1]) * axis[1])),
        x32_encode_f32(((one_minus_cosine * axis[1]) * axis[2]) + (axis[0] * sine)),
        0,
        x32_encode_f32(((one_minus_cosine * axis[0]) * axis[2]) + (axis[1] * sine)),
        x32_encode_f32(((one_minus_cosine * axis[1]) * axis[2]) - (axis[0] * sine)),
        x32_encode_f32(cosine + ((one_minus_cosine * axis[2]) * axis[2])),
        0,
        0,
        0,
        0,
        kOne
        }};
    Matrix44 temp1 = g_mtx_current[g_mtx_mode & 0xf];
    MatrixMultiply44(&g_mtx_current[g_mtx_mode & 0xf],
                     &temp0,
                     &temp1);
#endif
}
```

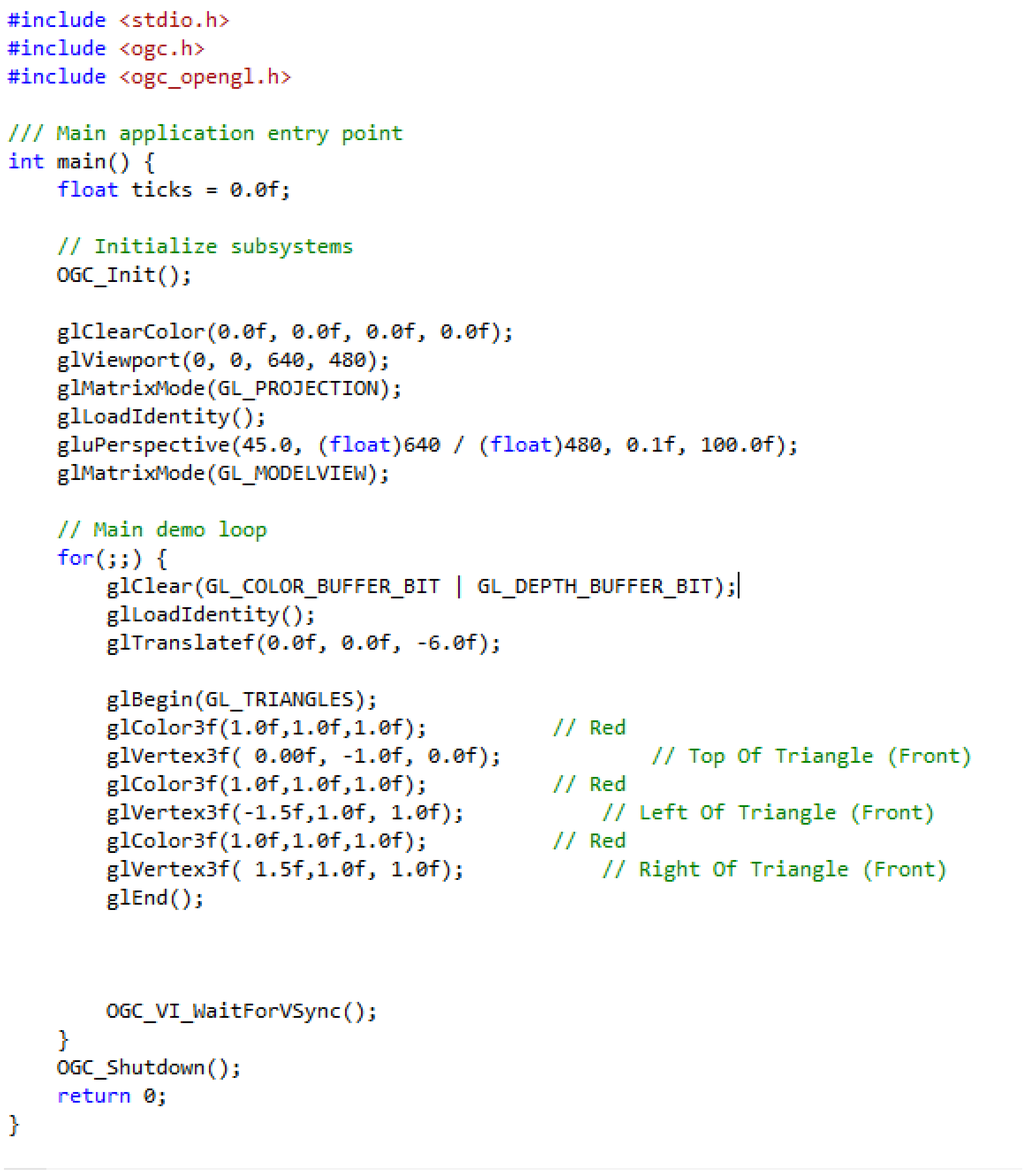
D.3 OGC_OPENGL_VERTEX.C

```
/*!
 * Copyright (C) 2010 Eric M. Nadeau / Skyler B. Whorton
 *
 * \file    ogc_opengl_vertex.c
 * \author  Skyler B. Whorton <swhorton@wpi.edu>
 * \date    Created 12 Sept 2010
 * \brief   Implements all OGL 1.0 vertex related functions
 *
 */
#include <ogc.h>
#include <ogc_opengl.h>
#include <math.h>
void glBegin(GLenum mode) {
    u32 ogc_mode;
    switch (mode) {
        case GL_POINTS:
            ogc_mode = OGC_POINTS;
            break;
        case GL_LINES:
            ogc_mode = OGC_LINES;
            break;
        case GL_TRIANGLES:
            ogc_mode = OGC_TRIANGLES;
            break;
        case GL_LINE_STRIP:
        case GL_LINE_LOOP:
        case GL_TRIANGLE_STRIP:
        case GL_TRIANGLE_FAN:
        case GL_QUADS:
        case GL_QUAD_STRIP:
        case GL_POLYGON:
        default:
            LOG_WARNING(TRAS, "glBegin: Unsupported primitive type 0x%08x!\n", mode);
            return;
    }
    API_Vertex_Mode.mode = mode;
    GPU_FIFO_PUT32(OGC_CMD_VTX_BEGIN);
    GPU_FIFO_PUT32(ogc_mode);
}
void glEnd(void) {
    GPU_FIFO_PUT32(OGC_CMD_VTX_END);
}
void glColor3f(f32 r, f32 g, f32 b) {
    f32 * mparam = (f32 *) malloc(4 * sizeof(f32));
    mparam[0] = r;
    mparam[1] = g;
    mparam[2] = b;
    mparam[3] = 1.0;
    int i;
    if (API_Color.colorMaterialEnabled && API_Lighting.isEnabled) {
        glMaterialfv(API_Color.colorMaterialFace, API_Color.colorMaterialMode, mparam);
    } else {
        for (i = 0; i <= 3; i++) {
            API_Color.rgba[i] = mparam[i];
        }
    }
    free(mparam);
}
void glMaterialf(GLenum face, GLenum pname, const GLfloat param) {
    switch (pname) {
        case GL_SHININESS:
            API_Material.shininess = (f32) param;
            break;
        default:
            LOG_WARNING(TAPI, "glMaterialf: Unsupported pname 0x%08x!\n", pname);
            break;
    }
}
void glMaterialfv(GLenum face, GLenum pname, const GLfloat *params) {
    int i;
    // Ignore face parameter; by default, apply material only to front face
    switch (pname) {
        case GL_AMBIENT:
            for (i = 0; i <=3 ; i++) {
                API_Material.ambient[i] = params[i];
            }
            break;
        case GL_DIFFUSE:
            for (i = 0; i <=3 ; i++) {
                API_Material.diffuse[i] = params[i];
            }
            break;
        case GL_AMBIENT_AND_DIFFUSE:
            for (i = 0; i <=3 ; i++) {
                API_Material.ambient[i] = params[i];
                API_Material.diffuse[i] = params[i];
            }
            break;
        case GL_SPECULAR:
            for (i = 0; i <=3 ; i++) {
                API_Material.specular[i] = params[i];
            }
            break;
        default:
            LOG_WARNING(TAPI, "glMaterialfv: Unsupported pname 0x%08x!\n", pname);
            break;
    }
}
void glColorMaterial(GLenum face, GLenum mode) {
    API_Color.colorMaterialFace = face;
    API_Color.colorMaterialMode = mode;
}
void glVertex3f(f32 x, f32 y, f32 z) {
    Vertex vtx;
    Vertex newNorm;
    Color16 col;
    int i, j;
    f32 dot;
    f32 * lightVector = (f32 *) malloc(3 * sizeof(f32));
    f32 * finalColor = (f32 *) malloc(4 * sizeof(f32));
    f32 * vertex = (f32 *) malloc(3 * sizeof(f32));
    f32 * lightPos = (f32 *) malloc(3 * sizeof(f32));
    // Set normal data
    newNorm.x._x32 = x32_encode_f32(API_Normal.currentNormal.x);
    newNorm.y._x32 = x32_encode_f32(API_Normal.currentNormal.y);
    newNorm.z._x32 = x32_encode_f32(API_Normal.currentNormal.z);
    // Set vertex data
    vtx.x._x32 = x32_encode_f32(x);
    vtx.y._x32 = x32_encode_f32(y);
    vtx.z._x32 = x32_encode_f32(z);
    apiTransformVertex(&vtx);
    vertex[0] = x32_decode_f32(vtx.x._x32);
    vertex[1] = x32_decode_f32(vtx.y._x32);
    vertex[2] = x32_decode_f32(vtx.z._x32);
    finalColor[0] = 0.0;
    finalColor[1] = 0.0;
    finalColor[2] = 0.0;
    finalColor[3] = 1.0;
    // Modelview transformation
    // -----------------------
    Matrix44 pmtx = {kOne, 0, 0, 0,
                     0, kOne, 0, 0,
                     0, 0, kOne, 0,
                     0, 0, 0, kOne};
    f32 xn = (x32_decode_f32(pmtx.s[0]) * API_Normal.currentNormal.x)
           + (x32_decode_f32(pmtx.s[4]) * API_Normal.currentNormal.y)
            + (x32_decode_f32(pmtx.s[8]) * API_Normal.currentNormal.z);
f32 yn = (x32_decode_f32(pmtx.s[1]) * API_Normal.currentNormal.x)
           + (x32_decode_f32(pmtx.s[5]) * API_Normal.currentNormal.y)
           + (x32_decode_f32(pmtx.s[9]) * API_Normal.currentNormal.z);
f32 zn = (x32_decode_f32(pmtx.s[2]) * API_Normal.currentNormal.x)
           + (x32_decode_f32(pmtx.s[6]) * API_Normal.currentNormal.y)
           + (x32_decode_f32(pmtx.s[10]) * API_Normal.currentNormal.z);
    f32 mag = sqrt(pow(xn,2) + pow(yn,2) + pow(zn,2));
    xn /= mag;
    yn /= mag;
    zn /= mag;
    // If there are lighting effects, apply them to the color applied to this
    // vertex. Otherwise, use the pre-existing color data sent explicitly by
    // the OpenGL program.
    if (API_Lighting.isEnabled) {
        col.rgb565.rgb = (u16) 0;
        // Calculate OpenGL lighting contributions to vertex color for each
        // light source
        for (i = 0; i <= 7; i++) {
            if (API_Lighting.lights[i].isEnabled) {
                // Calculate light vector
                if (API_Lighting.lights[i].position[3] == 1.0) {
                    for (j = 0; j <= 2; j++) {
                        lightPos[j] = API_Lighting.lights[i].position[j];
                    }
                } else {
                    for (j = 0; j <= 2; j++) {
                        lightPos[j] = vertex[j] + API_Lighting.lights[i].position[j];
                    }
                }
                apiGetUnitVector(lightPos, vertex, lightVector);
                // Calculate Ln dot product
                dot = (lightVector[0] * xn) +
                      (lightVector[1] * yn) +
                      (lightVector[2] * zn);
                if (dot < 0.0) {
                    dot = 0.0;
                }
                for (j = 0; j <= 2; j++) {
                    // Calculate ambient contribution
                    finalColor[j] += API_Material.ambient[j] *

                        API_Lighting.lights[i].ambient[j];

                    // Calculate diffuse contribution
                    finalColor[j] += dot * API_Material.diffuse[j] *

                        API_Lighting.lights[i].diffuse[j];

                    // Clamp the color at 1.0
                    if (finalColor[j] > 1.0) {
                        finalColor[j] = 1.0;
                    }
                }
            }
        }
    } else {
        for (i = 0; i <= 3; i++) {
            finalColor[i] = API_Color.rgba[i];
        }
    }
    // Encode final color data
    col.rgb565.r = (u8) (OGC_FB_RED_MASK * finalColor[0]);
    col.rgb565.g = (u8) (OGC_FB_GREEN_MASK * finalColor[1]);
    col.rgb565.b = (u8) (OGC_FB_BLUE_MASK * finalColor[2]);
    // Send final color data
    GPU_FIFO_PUT32(OGC_CMD_VTX_COL_565);
    GPU_FIFO_PUT32(col.rgb565.rgb);
    // Send vertex data
    GPU_FIFO_PUT32(OGC_CMD_VTX_POS_X32);
    GPU_FIFO_PUT32(vtx.x._u32);
    GPU_FIFO_PUT32(vtx.y._u32);
    GPU_FIFO_PUT32(vtx.z._u32);
    free(lightVector);
    free(finalColor);
    free(vertex);
}
```

```
void glVertex3x(x32 x, x32 y, x32 z) {
    Color16 col;
    col.rgb565.r = x32_encode_f32(API_Color.rgba[0]);
    col.rgb565.g = x32_encode_f32(API_Color.rgba[1]);
    col.rgb565.b = x32_encode_f32(API_Color.rgba[2]);
    GPU_FIFO_PUT32(OGC_CMD_VTX_COL_565);
    GPU_FIFO_PUT32(col.rgb565.rgb);
    GPU_FIFO_PUT32(OGC_CMD_VTX_POS_X32);
    GPU_FIFO_PUT32(x);     GPU_FIFO_PUT32(y);
    GPU_FIFO_PUT32(z);
}
void apiGetUnitVector(f32 * tip, f32 * tail, f32 * output) {
    int i;
    float length;
    for (i = 0; i <=2; i++) {
        output[i] = tip[i] - tail[i];
    }
    length = sqrt(pow(output[0], 2) + pow(output[1], 2) + pow(output[2], 2));
    for (i = 0; i <=2; i++) {
        output[i] /= length;
    }
}
void glNormal3f(f32 nx, f32 ny, f32 nz) {
    API_Normal.currentNormal.x = nx;
    API_Normal.currentNormal.y = ny;
    API_Normal.currentNormal.z = nz;
}
```

Appendix E: Simple OpenGL Demo Source

E.1 Triangle Demo



```c
#include <stdio.h>
#include <ogc.h>
#include <ogc_opengl.h>

/// Main application entry point
int main() {
    float ticks = 0.0f;

    // Initialize subsystems
    OGC_Init();

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glViewport(0, 0, 640, 480);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)640 / (float)480, 0.1f, 100.0f);
    glMatrixMode(GL_MODELVIEW);

    // Main demo loop
    for(;;) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        glTranslatef(0.0f, 0.0f, -6.0f);

        glBegin(GL_TRIANGLES);
        glColor3f(1.0f,1.0f,1.0f);          // Red
        glVertex3f( 0.00f, -1.0f, 0.0f);            // Top Of Triangle (Front)
        glColor3f(1.0f,1.0f,1.0f);          // Red
        glVertex3f(-1.5f,1.0f, 1.0f);           // Left Of Triangle (Front)
        glColor3f(1.0f,1.0f,1.0f);          // Red
        glVertex3f( 1.5f,1.0f, 1.0f);           // Right Of Triangle (Front)
        glEnd();



        OGC_VI_WaitForVSync();
    }
    OGC_Shutdown();
    return 0;
}
```

E.2 Transformation Demo

```
#include <stdio.h>
#include <ogc.h>
#include <ogc_opengl.h>

/// Main application entry point
int main() {
    float ticks = 0.0f;

    // Initialize subsystems
    OGC_Init();

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glViewport(0, 0, 640, 480);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)640 / (float)480, 0.1f, 100.0f);
    glMatrixMode(GL_MODELVIEW);

    // Main demo loop
    for(;;) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glLoadIdentity();
        glTranslatef(0.0f, 0.0f, -6.0f);
        glRotatef((ticks += 0.1), 0.0f, 1.0f, 0.0f);
        glScalef(1.0f, 1.0f, 1.0f);

        glBegin(GL_TRIANGLES);

        glColor3f(1.0f,0.0f,0.0f);          // Red
        glVertex3f( 0.00f, 1.5f, 0.0f);         // Top Of Triangle (Front)
        glColor3f(1.0f,0.0f,0.0f);          // Red
        glVertex3f(-1.0f,-0.0f, 1.0f);          // Left Of Triangle (Front)
        glColor3f(1.0f,0.0f,0.0f);          // Red
        glVertex3f( 1.0f,-0.0f, 1.0f);          // Right Of Triangle (Front)

        glColor3f(0.0f,0.0f,1.0f);          // Blue
        glVertex3f( 0.0f, 1.5f, 0.0f);          // Top Of Triangle (Right)
        glColor3f(0.0f,0.0f,1.0f);          // Blue
        glVertex3f( 1.0f,-0.0f, 1.0f);          // Left Of Triangle (Right)
        glColor3f(0.0f,0.0f,1.0f);          // Blue
        glVertex3f( 1.0f,-0.0f, -1.0f);         // Right Of
```

```
glColor3f(1.0f,0.0f,0.0f);          // Red
glVertex3f( 0.0f, 1.5f, 0.0f);          // Top Of Triangle (Back)
glColor3f(1.0f,0.0f,0.0f);          // Red
glVertex3f( 1.0f,-0.0f, -1.0f);         // Left Of Triangle (Back)
glColor3f(1.0f,0.0f,0.0f);          // Red
glVertex3f(-1.0f,-0.0f, -1.0f);         // Right Of Triangle (Back)

glColor3f(0.0f,0.0f,1.0f);          // Blue
glVertex3f( 0.0f, 1.5f, 0.0f);          // Top Of Triangle (Left)
glColor3f(0.0f,0.0f,1.0f);          // Blue
glVertex3f(-1.0f,-0.0f,-1.0f);          // Left Of Triangle (Left)
glColor3f(0.0f,0.0f,1.0f);          // Blue
glVertex3f(-1.0f,-0.0f, 1.0f);          // Right Of Triangle (Left)

glColor3f(0.0f,0.0f,1.0f);          // Blue
glVertex3f( 0.00f, -1.5f, 0.0f);            // Top Of Triangle (Front)
glColor3f(0.0f,0.0f,1.0f);          // Blue
glVertex3f(-1.0f,-0.0f, 1.0f);          // Left Of Triangle (Front)
glColor3f(0.0f,0.0f,1.0f);          // Blue
glVertex3f( 1.0f,-0.0f, 1.0f);          // Right Of Triangle (Front)

glColor3f(1.0f,0.0f,0.0f);          // Red
glVertex3f( 0.0f, -1.5f, 0.0f);         // Top Of Triangle (Right)
glColor3f(1.0f,0.0f,0.0f);          // Red
glVertex3f( 1.0f,-0.0f, 1.0f);          // Left Of Triangle (Right)
glColor3f(1.0f,0.0f,0.0f);          // Red
glVertex3f( 1.0f,-0.0f, -1.0f);         // Right Of

glColor3f(0.0f,0.0f,1.0f);          // Blue
glVertex3f( 0.0f, -1.5f, 0.0f);         // Top Of Triangle (Back)
glColor3f(0.0f,0.0f,1.0f);          // Blue
glVertex3f( 1.0f,-0.0f, -1.0f);         // Left Of Triangle (Back)
glColor3f(0.0f,0.0f,1.0f);          // Blue
glVertex3f(-1.0f,-0.0f, -1.0f);         // Right Of Triangle (Back)

glColor3f(1.0f,0.0f,0.0f);          // Red
glVertex3f( 0.0f, -1.5f, 0.0f);         // Top Of Triangle (Left)
glColor3f(1.0f,0.0f,0.0f);          // Red
glVertex3f(-1.0f,-0.0f,-1.0f);          // Left Of Triangle (Left)
glColor3f(1.0f,0.0f,0.0f);          // Red
glVertex3f(-1.0f,-0.0f, 1.0f);          // Right Of Triangle (Left)
```

```
        glEnd();



        OGC_VI_WaitForVSync();
    }
    OGC_Shutdown();
    return 0;
}
```

E.3 Interpolation and Lighting Demo

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <ogc.h>
#include <ogc_opengl.h>

/// Main application entry point
int main() {
    float z_offs = -6.0f, x_offs = 0.0f;
    int i;
    int push_check1, push_check=0;

    XGpio Push;

    XGpio_Initialize(&Push,XPAR_PUSH_BUTTONS_4BIT_DEVICE_ID);
    XGpio_SetDataDirection(&Push,1,0xffffffff);

    // Initialize subsystems
    OGC_Init();

    printf("****************************************************\n");
    printf("simple-oct-interp - demonstrates basic interpolation\n");
    printf("****************************************************\n");

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glViewport(0, 0, 640, 480);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)640 / (float)480, 0.1f, 100.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    GLfloat light0_ambient[] = { 0.4, 0.4, 0.4, 1.0 };
    GLfloat light0_diffuse[] = { 0.9, 0.9, 0.9, 1.0 };
    GLfloat light0_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light0_position[] = { -4.0, -2.0, 0.0, 0.0 };
    glLightfv(GL_LIGHT0, GL_AMBIENT, light0_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light0_specular);
    glLightfv(GL_LIGHT0, GL_POSITION, light0_position);
```

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

// Main demo loop
for(;;) {

    push_check1 = XGpio_DiscreteRead(&Push,1);
    if (push_check1) {
        if(push_check1 == 0x1)
            x_offs+=0.1;
        else if(push_check1 == 0x2)
            z_offs+=0.1;
        else if(push_check1 == 0x4)
            x_offs-=0.1;
        else if (push_check1 == 0x8)
            z_offs-=0.1;
    }

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glTranslatef(0.0f-x_offs, 0.0, z_offs);
    glRotatef(45.0, 0.0f, 1.0f, 0.0f);
    glScalef(1.0f, 1.0f, 1.0);

    glBegin(GL_TRIANGLES);

    glNormal3f(0.0, 0.5547, 0.8321);
    glColor3f(0.0f,0.0f,1.0f);          // Red
    glVertex3f( 0.00f, 1.5f, 0.0f);         // Top Of Triangle (Front)
    glColor3f(0.0f,1.0f,0.0f);          // Green
    glVertex3f(-1.0f,-0.0f, 1.0f);          // Left Of Triangle (Front)
    glColor3f(1.0f,0.0f,0.0f);          // Blue
    glVertex3f( 1.0f,-0.0f, 1.0f);          // Right Of Triangle (Front)

    glNormal3f(-0.8321, 0.5547, 0.0);
    glColor3f(0.0f,0.0f,1.0f);          // Red
    glVertex3f( 0.0f, 1.5f, 0.0f);          // Top Of Triangle (Left)
    glColor3f(1.0f,0.0f,0.0f);          // Blue
    glVertex3f(-1.0f,-0.0f,-1.0f);          // Left Of Triangle (Left)
```

```
        glNormal3f(0.0, -0.5547, 0.8321);
        glColor3f(0.0f,0.0f,1.0f);          // Red
        glVertex3f( 0.00f, -1.5f, 0.0f);            // Top Of Triangle (Front)
        glColor3f(0.0f,1.0f,0.0f);          // Green
        glVertex3f(-1.0f,-0.0f, 1.0f);          // Left Of Triangle (Front)
        glColor3f(1.0f,0.0f,0.0f);          // Blue
        glVertex3f( 1.0f,-0.0f, 1.0f);          // Right Of Triangle (Front)

        glNormal3f(-0.8321, -0.5547, 0.0);
        glColor3f(0.0f,0.0f,1.0f);          // Red
        glVertex3f( 0.0f, -1.5f, 0.0f);         // Top Of Triangle (Left)
        glColor3f(1.0f,0.0f,0.0f);          // Blue
        glVertex3f(-1.0f,-0.0f,-1.0f);          // Left Of Triangle (Left)
        glColor3f(0.0f,1.0f,0.0f);          // Green
        glVertex3f(-1.0f,-0.0f, 1.0f);          // Right Of Triangle (Left)

        glEnd();

        ogc_printf(10,230, "light");
        ogc_printf(10,245, "source");

        OGC_VI_WaitForVSync();
    }
    OGC_Shutdown();
    return 0;
}
```