

**Power Consumption Characterization  
of a Graphics Processing Unit**

A Thesis  
in STS 402

Presented to

The Faculty of the  
School of Engineering and Applied Science  
University of Virginia

In Partial Fulfillment  
of the Requirements for the Degrees

Bachelor of Science in Computer Engineering  
and  
Bachelor of Science in Computer Science

by

Richard David McWhorter, III

March 29, 2005

On my honor as a University student, on this assignment I have  
neither given nor received unauthorized aid as defined by the Honor  
Guidelines for Papers in Science, Technology, and Society Courses.

\_\_\_\_\_  
(Full Signature)

Approved: \_\_\_\_\_ (Technical Advisor)  
Kevin Skadron (Signature)

Approved: \_\_\_\_\_ (Science, Technology,  
and Society Advisor)  
Catherine D. Baritaud (Signature)

**TABLE OF CONTENTS**

**LIST OF FIGURES ..... iv**

**GLOSSARY OF TERMS..... V**

**ABSTRACT..... VI**

**CHAPTER ONE: THE NEED FOR A POWER CHARACTERIZATION..... 1**

    HISTORY OF GRAPHICS PROCESSORS ..... 1

    ENERGY AMBIGUITY OF MODERN GRAPHICS PROCESSORS ..... 2

    POWER CONSUMPTION OF PROGRAMMABLE GRAPHICS PROCESSORS  
    ..... 3

    OVERVIEW OF PROJECT PROCEDURES ..... 4

    OVERVIEW OF THE REPORT ..... 5

**CHAPTER TWO: THE EFFECTS OF A POWER CHARACTERIZATION..... 6**

    COMPUTER GRAPHICS COMMUNITY ..... 6

    GRAPHICS CARD MANUFACTURERS ..... 7

    GRAPHICS CARD CONSUMERS ..... 8

**CHAPTER THREE: THE BASIS FOR A POWER CHARACTERIZATION ..... 10**

    ARCHITECTURE-LEVEL CPU MODEL ..... 10

    POWER CONSUMPTION MINIMIZATION EFFORTS..... 11

    COMPUTER GRAPHICS POWER RESEARCH ..... 11

**CHAPTER FOUR: THE METHOD FOR THE POWER CHARACTERIZATION  
..... 15**

MATERIALS.....	15
<b>Provided by the UVA Computer Science Department:.....</b>	<b>15</b>
<b>Provided by Researcher: .....</b>	<b>15</b>
METHODS .....	16
<b>Preliminary Test.....</b>	<b>16</b>
<b>Vertex and Fragment Program Tests .....</b>	<b>17</b>
<b>CHAPTER FIVE: THE RESULTS OF THE POWER CHARACTERIZATION ..</b>	<b>21</b>
PRELIMINARY POWER CONSUMPTION TEST .....	21
VERTEX PROGRAM OPERATIONS TESTS .....	22
FRAGMENT PROGRAM OPERATIONS TESTS .....	23
<b>CHAPTER SIX: THE MEANING OF THE POWER CHARACTERIZATION....</b>	<b>24</b>
PRELIMINARY POWER CONSUMPTION TEST .....	24
VERTEX AND FRAGMENT PROGRAM OPERATIONS POWER TEST .....	25
<b>Explanation for the Lack of Power Correspondence: Clock Gating .</b>	<b>25</b>
<b>Implications of the Lack of Power Correspondence.....</b>	<b>26</b>
<b>CHAPTER SEVEN: THE FUTURE OF THE POWER CHARACTERIZATION</b>	<b>27</b>
<b>WORKS CITED.....</b>	<b>29</b>
<b>APPENDICES.....</b>	<b>33</b>
APPENDIX A – SOURCE OF THE PRELIMINARY POWER CONSUMPTION TEST .....	33
APPENDIX B – SOURCE OF THE VERTEX PROGRAM TESTER .....	38
APPENDIX C – SOURCE OF THE FRAGMENT PROGRAM TESTER .....	44

APPENDIX D – EXAMPLE TEST FRAGMENT PROGRAM (ADDITION TEST).....	50
APPENDIX E – LISTING OF OPENGL FRAGMENT PROGRAM OPERATIONS.....	52
APPENDIX F – LISTING OF OPENGL VERTEX PROGRAM OPERATIONS .....	53
APPENDIX G – EXAMPLE TEST VERTEX PROGRAM (ADDITION TEST) .....	54
APPENDIX H – RESULTS OF PRELIMINARY POWER CONSUMPTION TEST .....	56
APPENDIX I – VERTEX PROGRAM OPERATION TEST RESULTS .....	57
APPENDIX J – FRAGMENT PROGRAM OPERATION TEST RESULTS .....	59

## LIST OF FIGURES

- Figure 1 – “A History of Graphics Processors” – Page 1
- Figure 2 – “NVIDIA GeForce FX 5900” – Page 15
- Figure 3 – “Fluke 189 Digital Multimeter” – Page 15
- Figure 4 – “RadioShack 22-812 Digital Multimeter” – Page 16
- Figure 5 – “Molex PC Power Extension Cable” – Page 16
- Figure 6 – “Power consumption test setup for 5V line” – Page 20
- Figure 7 – “Power consumption test setup for 12V line” – Page 20
- Figure 8 – “Results of the preliminary experiment” – Page 21
- Figure 9 – “Results of the vertex program power characterization experiment” – Page 22
- Figure 10 – “Results of the fragment program power characterization experiment” –  
Page 23

## GLOSSARY OF TERMS

And Gate – a digital logic element that outputs a logic high signal if both of its inputs are logic high signals.

Clock Gating – the practice of placing an *and* gate in front of the clock input to a circuit, effectively creating an enable signal for the clock signal going into the circuit, and allowing the designer of the chip to disable sections of the chip when they are not needed by lowering this enable signal (Li et al., 2003, p. 1)

Fragment Program – program that the graphics processor executes that tells the graphics processor how to color and order the pixels it processes

Index Rendering – using several asynchronous data flows that are converged by the index to render an image (Liang, Lee, Yeh, and Jen, 2002, p. 343)

Pixels – colored points that form images when a computer monitor displays them

Prefetching – guessing which pixels are likely to be used and placing these in the pixel cache (Park et al., 2003, p. 1501)

Programmability – ability to change the hardware operations of a graphics processor

Vertex Program – program that the graphics processor executes that tells the graphics processor how to move and color the vertices it processes

Rasterization – the process of converting triangles from descriptions of vertices and colors in the memory of a computer to sets of pixels

Rendering pipeline – the sequence of steps a program takes to transform a model of a scene in memory into a set of pixels to display

## ABSTRACT

This project determined the relationship between the different programmable operations, specifically vertex and fragment program operations, of a graphics processor and how much energy it consumes. It is necessary because modern graphics processors are consuming more power than ever before, and the need for power-conservative graphics processors is rapidly rising, as embedded applications of computer graphics are becoming widespread. This project is founded upon an experiment in which Tiwari repeatedly executed the operations of a computer processor and recorded the amount of power each type consumed. This opened the door for power models based upon empirical power characterizations of processors.

In order to reveal this relationship, programs were written that rendered arrays of millions of vertices repeatedly using arbitrary vertex and fragment programs. Test vertex and fragment programs that repeated each vertex and fragment program operation were then written. Finally, the power consumption of the NVIDIA GeForce FX 5900 was empirically observed while rendering the vertices using each of the test programs.

The results of the experiment were surprising. The experiment showed that there is no relationship between which vertex or fragment operation the graphics processor executes and how much power it consumes. This is probably because the processor does not use clock gating. This project is still useful, however, because it shows that the programmer's choice of operations does not affect power consumption, it reveals that the processor's architecture does not use clock gating, and it provides a framework to easily determine the relationship between choice of operation and power consumption on future clock-gated graphics processors.

## CHAPTER ONE: THE NEED FOR A POWER CHARACTERIZATION

Modern graphics processors have shattered expectations and unleashed imaginations in ways that were only dreamed about until recently, but have also consumed more power and produced more heat than ever before. Currently, researchers know little about the relationship between the operations that a graphics processor executes and the amount of power it consumes. This project determined that no relationship exists between which operation a programmable graphics card executes and how much energy the card consumes.

### HISTORY OF GRAPHICS PROCESSORS

To begin to understand why an examination of this relationship is important, we will first examine the history of computer graphics hardware, summarized in Figure 1. Primitive personal computers contained very little specialized graphics hardware and mainly displayed text. The first advances in graphics hardware came from companies that specialized in creating advanced graphics computers and displays, such as Silicon Graphics. These machines were too expensive for the mainstream of personal

<i>A History of Graphics Processors</i>		
<b>Generation:</b>	<b>New Features:</b>	<b>Example Cards:</b>
First (1998)	Rasterizing triangles, applying textures	RIVA TNT2, ATI Rage, 3Dfx Voodoo3
Second (1999)	3D vertex transformation and lighting	NVIDIA GeForce 2, ATI Radeon 7500
Third (2001)	Vertex and Fragment Processing Customization	NVIDIA GeForce 3/4, ATI Radeon 8500
Fourth (2003)	Vertex and Fragment Programmability	NVIDIA GeForce FX, ATI Radeon 9700

Figure 1. A History of Graphics Processors. [Adapted by Richard David McWhorter, III from Fernando and Kilgard, 2003, p. 10-12]



computing, but their developers were responsible for many of the foundational concepts of computer graphics (Fernando and Kilgard, 2003, p. 10).

The first mainstream graphics processors became widely available around 1998. These cards could rasterize triangles and apply textures to these images. Triangle rasterization refers to the process of converting triangles from descriptions of vertices and colors in the memory of a computer to sets of pixels, or colored points that form images when a computer monitor displays them. Texturization refers to the process of applying an image, or texture, to the surface of a geometric object to make it look real. The second generation of graphics processors, circa 1999, added the ability to transform a scene, or change the perspective of the viewer to the scene, and to light a scene, or change the brightness of objects in the scene and the background of the scene, in hardware.

The third set, released in 2001, offered highly configurable vertex and pixel level processing. This vertex and pixel level processing lets developers specify a few customized operations that the graphics processor applies to each vertex and pixel, respectively (Fernando and Kilgard, 2003, p. 10-11). Up to this point in time, graphics processors contained no real programmability, or ability to change the hardware operations of the processor. This lack of programmability meant that the decisions a developer made in implementing a program that used the graphics processor caused little to no variation in the power consumption of the graphics processor.

## **ENERGY AMBIGUITY OF MODERN GRAPHICS PROCESSORS**

The fourth and current (as of this writing) generation of graphics processors feature fully programmable vertex and fragment engines. This means that the programmer can write programs to tell the graphics processor how to move the vertices,

using vertex programs, and how to color the pixels of an image, using fragment programs (Fernando and Kilgard, 2003, p. 11).

This capability to program the graphics processor is very significant because it is enormously powerful, and represents a way in which processor behavior can vary, based upon the design of a graphics program. Prior to the introduction of programmable graphics processors, the set of operations that the processor performed was a relatively fixed part of the rendering pipeline, or the sequence of steps a program takes to transform a model of a scene in memory into a set of pixels to display. Before the programmer could precisely control the graphics processor through these vertex and fragment engines, it was reasonable to assume that the amount of power the graphics processor consumed was relatively constant. This was because the graphics processor was doing the same things for each vertex and each fragment it processed. Now the graphics processor's power consumption could vary easily, based upon the operations the programmer wants the graphics processor to execute.

## **POWER CONSUMPTION OF PROGRAMMABLE GRAPHICS PROCESSORS**

This variation or programmability of the graphics processor is the subject of investigation for this project. The aim of this project is to show the relationship between the different choices that the programmer makes and how much power the graphics card consumes. Knowing this relationship is significant for many reasons including enhancing the design of software and hardware for minimal power consumption and designing hardware to maximize thermal efficiency.

The scope of this project was limited in that it examined a specific graphics processor to gain an understanding of how different programmable operations of a

graphics processor consume power. The project examined only one processor because of the limited time frame of the project and the amount of work required to test each graphics processor. The original aim of the project was to determine empirically the amount of power that the NVIDIA GeForce FX 5900 graphics card consumes while performing each vertex and fragment operation; integrate these results into *Qsilver*, a computer graphics architecture simulator explained later in this report; and provide examples of how vertex and fragment programs could be optimized based upon these results. Because the outcome of the experiment was much different than expected, the report instead presents the results of the experiment to determine empirically the power consumption of the graphics card while performing each vertex and fragment program operation and discusses numerous implications of these results.

## **OVERVIEW OF PROJECT PROCEDURES**

To achieve a clear model of how a graphics processor consumes power, the following procedures were completed:

- Observed the relationship between power consumption and the number of vertices processed by the graphics card as a preliminary experiment
- Observed the amount of power the graphics processor consumes for each vertex and fragment program operation
- Inferred information about the probable architecture of the graphics processor from the above power consumption data

## **OVERVIEW OF THE REPORT**

The balance of this report takes a step back to examine the context of the project and then presents the details of the research. First, it considers the broader social and ethical dimensions of the research, including how it affects the computer graphics community, graphics hardware manufacturers, and graphics card consumers to place the project in the appropriate context. Next, it examines the existing research in the fields of computer architecture and computer graphics to determine the value of this power characterization. The relevant research includes architecture-level CPU power characterizations, existing power consumption minimization efforts, and power consumption research specific to graphics. The report also details the materials used and the methods followed in the actual experiments to determine the amount of power consumed by each of the vertex and fragment operations. Next, it reveals the surprising results of the power characterization. Finally, it explores the implications of these results.

## **CHAPTER TWO: THE EFFECTS OF A POWER CHARACTERIZATION**

There are at least three groups of people that affect and are affected by this research. First, it could enable graphics microarchitecture researchers to design more thermally efficient graphics processors and to design power optimized graphics software. Second, it could have economic impacts on graphics hardware manufacturers. Third, it could enable people who use graphics cards to have faster and more energy efficient cards to power their computing, communication, and entertainment.

### **COMPUTER GRAPHICS COMMUNITY**

To begin, the report will examine the effect that this power consumption model could have on the computer graphics community. First, practical applications of the project could include allowing researchers to build faster and cooler graphics processors using temperature-aware design (Skadron et al., 2003, p. 52) and though redesigning power bottlenecks in processor designs. A program that characterizes how a graphics processor consumes power based upon its architecture would be of enormous value to those designing graphics processors. These faster chips, explains Macedonia (2003), could also enable the community to make advances in the areas of using graphics processors to solve complex math problems and render audio (p. 107-108).

The computer graphics research community could also use the power model to start building power-optimized graphics software. When Tiwari et al. (1994) performed similar research on the main computer processor, they showed that it is possible to trade power consumption for performance by trading high power instructions for lower power

ones (p. 444). Thus, one could operate a high power graphics processor in a reduced power consumption mode just by changing the software that it runs.

## **GRAPHICS CARD MANUFACTURERS**

Next, the report will examine the impact of the power model on graphics card manufacturers. In a free market economic system such as that of the United States, financial profit primarily drives technical development. In particular, Takahashi (2003) notes, two corporations currently dominate the market for stand-alone graphics cards and battle each other constantly for business: NVIDIA Corporation, or NVIDIA, and Array Technology Inc., or ATI (p. 23). The implications of understanding how their graphics cards consume power could give either corporation an advantage, particularly in the areas of embedded and mobile computing.

For example, if ATI discovers a way to reduce their mobile chip's power consumption by even ten percent from an instruction-level power model, computer manufacturers could decide to buy ATI chips instead of NVIDIA chips and thus ATI would gain market share. Perhaps even more significantly, imagine a competitor that is currently not doing well in the consumer graphics market, such as 3DLabs or Matrox, according to Case, (2002, p. 2), making a breakthrough in their power models and is able to produce chips that consume far less power than those of ATI or NVIDIA. This could cause a significant shift in the main competitors in the graphics hardware market.

Realistically speaking, this is probably only likely to happen if substantial research is undertaken in response to this project. However, if ATI or NVIDIA determined that this power research would likely lead to an increase in market share and profit, either could jump at the chance to explore the possibilities.

Embedded computing is one area of particular interest to graphics hardware companies when considering power consumption. As mentioned above, areas such as cellular phones and personal digital assistants are a primary application of power-optimized graphics. In the past few years, many companies, including Sun and Microsoft, have been pushing forward various application-programmer interfaces for mobile computing, such as OpenGL Embedded Systems and Direct3D Mobile (Merritt, 2003, p. 1). If a hardware manufacturer is able gain an advantage in implementing these interfaces by looking carefully at the power consumption of its cards, this could again push the competitive advantage in its favor and result in economic gain for the company.

This project could also have an economic impact in the area of manufacturing costs and reliability. One of the most significant factors, according to Skadron et al. (2003), in keeping manufacturing costs low and chips reliable is the rise in heat density that results from speed increases (p. 52). If, for example, this research exposes that one particular operation of a graphics processor is producing a huge amount of heat, then ATI or NVIDIA could save money in manufacturing costs and increase the reliability of their chips by simply correcting this one operation. On the other hand, according to Bose et al. (2003), reducing power consumption might require additions to the microarchitectural design, and microarchitectural complexity often brings increased manufacturing cost and often reduces the ability to verify correctness (p. 10). Hence, the project could have positive or negative effects on manufacturing costs.

## **GRAPHICS CARD CONSUMERS**

The third and most obvious group that could affect and be affected by the development of this power model includes the consumers of graphics hardware.

According to Liang et al. (2002), graphics hardware has become a core component of multimedia systems, virtual reality, and entertainment (p. 343), as evidenced by the record sales of mobile graphics processors in the fourth quarter of 2003 (Jon Peddie Research Says PC Graphics Chips Hit Record Highs Q4 2003; Market Shares Shift., 2004, p. 1). Hence, consumers of graphics hardware demand faster graphics processors to run their mobile devices as well as their personal computers. This power consumption characterization could enable mobile multimedia devices to become more powerful as it enables graphics processors to become faster and cooler.

Possible negative effects that this speed and temperature improvement could have include causing consumers to become more dependent on technology and causing them to become less intellectually stimulated as they rely on technology to entertain them. For example, Krikke (2001) explains that in Japan, mobile phones are replacing the personal computers as the primary gateway to the Internet (p. 9) and that at every moment people have web sites available to keep them occupied instead of educational and service-oriented activities.

From an examination of the effect that this project could have on these three groups of people, it would seem that the potential for negative and positive effects exist. The project could bring faster and more energy-efficient graphics cards and increased competition among graphics card manufacturers. It could also increase widespread dependence upon technology. Nevertheless, it also seems reasonably clear that there is far greater potential for good to arise from the project than for the project to cause harm on those it affects.



## **CHAPTER THREE: THE BASIS FOR A POWER CHARACTERIZATION**

To set the stage for exactly what this project involves, the report now turns to examine the ideas that experts in the fields of computer graphics and computer architecture have produced that form the basis for this project. First, it steps back to the level of the computer processor and examines how researchers developed an architecture-level model of the power consumption of the central processor and the many advances that have come because of this model. Second, it turns to examine existing efforts at minimizing graphics processor power consumption. Finally, it examines the existing research relating graphics architectures to power consumption.

### **ARCHITECTURE-LEVEL CPU MODEL**

The foundational work for this project is Vivek Tiwari's (1994) architecture-level power characterization of a central processor. In it, Tiwari explains a method for developing a power model for essentially any processor. Tiwari measures the current drawn by a processor while it executes specific types of instructions and uses the supply voltage of the processor and execution time of the test program to determine how much energy each specific instruction consumes. For his sample processor, he used this data to develop a model of how a central processor consumes power (p. 437-444).

Tiwari (1994) went on to use this model to propose ways to optimize current software for minimal power consumption by using certain instructions liberally. He centered this optimization on replacing the most expensive operations with alternative operations that consumed less power (p. 437-444). Tiwari's research also inspired a host of other advances related to architecture-level power management; for example, Flinn

(2004) developed an application that optimizes the instructions a program executes at runtime based on power consumption (p. 137).

It is interesting to note that in addition to a strict instruction-level empirical power characterization, Brandolese et al. (2002) attempted to generalize the power model by breaking instructions into different fundamental types and characterizing the power consumption of these types (p. 1306-1315). This is significant because it represents a desire to generalize the power consumption model beyond having to take current measurements for each individual chip that one wants to characterize.

## **POWER CONSUMPTION MINIMIZATION EFFORTS**

Brooks, Tiwari, and Martonosi realized this desire to have a general power consumption model in 2000 when they developed *Wattch*. *Wattch* is a framework for characterizing a processor's power consumption based upon what components are present in the microarchitecture. Brooks, Tiwari, and Martonosi took the empirical power consumption data that comes from examining how a processor consumes power and used this to determine how the components of the architecture consume power. Brooks, Tiwari, and Martonosi then built this into a configurable tool that architecture designers can use to evaluate power consumption long before they actually build their design (p. 83-93). Brooks et al. developed a similar framework called *PowerTimer* in 2003 that they based upon a set of energy consumption functions (p. 653).

## **COMPUTER GRAPHICS POWER RESEARCH**

Next, the report will turn to examine some of the existing efforts to reduce the power of graphics processors. Akenine-Moller and Strom (2003) took the approach of

attempting to minimize memory accesses in the graphics processing of mobile phones by simplifying and approximating some aspects of the rendering process (p. 801). Park et al. (2003) took a similar approach in reducing memory bandwidth but also attempted to minimize pixel cache misses using prefetching, or guessing which pixels were likely to be used and placing these in the pixel cache (p. 1501). Woo et al. (2002) went further and designed a rendering engine that consumes only 120 milliwatts (mW) of power by integrating the memory and the processor into a single chip (p. 1352). In 2003, Kameyama et al. used clock gating, explained later in this report, to lower power consumption with a minimal in performance hit on a particular cellular phone. These mobile applications are increasingly important as Krikke (2001) asserts that the cellular phone is becoming the primary means of accessing the Internet and its applications (p. 9).

One particularly interesting effort to reduce graphics processor power came from Liang, Lee, Yeh, and Jen in 2002. They proposed a way to eliminate unnecessary operations in the traditional graphics pipeline through index rendering, or using several asynchronous data flows that are converged by the index to render the final image (p. 343). The efforts so far at reducing graphics power consumption for mobile applications have focused largely on the power consumed by memory access.

Curiously, there is not an abundance of research related to graphics processors and power consumption at the architecture level. Graphics processor architectures differ significantly from computer processor architectures, with graphics processors devoting most of their chip area to computational engines and computer processors tending to have large amounts of cache (Macedonia, 2003, p. 106-107). Some of the core components of a graphics architecture include pixel pipes, texturing units, and vertex pipelines (Salvator,

2004, p. 2) while fundamental central processor components include logic units and many cache levels. Thus, significant differences exist in the way the chips consume power and the way software can exploit architectures to maximize efficiency.

One interesting development came from Yoon et al. in 2001 when they created a specialized multimedia processor for video acceleration and 3-D rendering on embedded systems. They took the approach of designing their architecture from the ground up for power consumption optimization by using existing knowledge of various architectural components and existing knowledge of how those components consumed power (p. 1758). This project aims toward a similar goal, but with a particular thrust toward understanding the particular intricacies of graphics components and the ability to model arbitrary graphics processors.

Perhaps a few of the people researching power consumption in graphics processors at the architecture level are Shaeffer, Luebke, and Skadron (2004). They have created *Qsilver*, a microarchitectural simulator of a graphics processor. They use it to explore performance bottlenecks and estimate energy consumption. Their power model, however, is “admittedly crude” because it bases the power consumption assumptions of most of the components on estimates available from components in central processors. Even from crude models, they found ways to reduce energy consumption at minimal performance cost (p. 5). This further illustrates the importance of an accurate power model.

This project has created an architecture-level power characterization of the various operations of a graphics processor that enables detailed simulation of the power characteristics of an arbitrary graphics processor. The research is clearly valuable in the

area of graphics hardware development as is evidenced by the existing work in the field, including the way similar characterizations had enormous applications for central processors and the way it could be used to fill a gap in graphics architecture simulation.

## CHAPTER FOUR: THE METHOD FOR THE POWER CHARACTERIZATION

### MATERIALS

#### Provided by the UVA Computer Science Department:

- Computer with a 2.8 GHz Intel Pentium 4 Processor and 1Gb of RAM
- NVIDIA GeForce FX 5200 (used in preliminary power test)
- NVIDIA GeForce FX 5900 (used in main vertex and fragment program tests), illustrated in Figure 2.
- 2 Samsung 910T 19-inch LCD displays
- Fluke 189 Digital Multimeter (used to measure current going to the graphics processor), illustrated in Figure 3.
- Various Multimeter Test Leads
- Microsoft Visual Studio .NET 2003

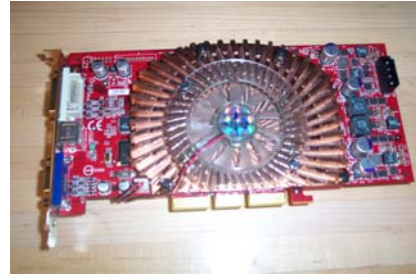


Figure 2. NVIDIA GeForce FX 5900. [Created by Richard David McWhorter, III.]



Figure 3. Fluke 189 Digital Multimeter [Created by Richard David McWhorter, III.]

#### Provided by Researcher:

- RadioShack 22-812 Digital Multimeter (used to measure voltage going to the graphics processor), illustrated in Figure 4 on the next page.
- Multimeter Test Leads
- Dell Inspiron 8000 Laptop Computer

- Various Wires
- Electrical Tape
- Wire strippers
- Molex PC Power Extension Cable (used to measure current and voltage to the graphics processor), illustrated in Figure 5.

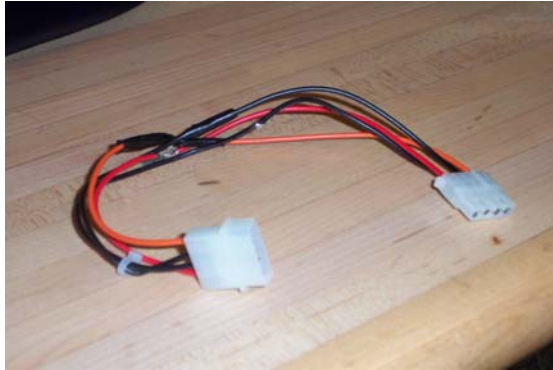


Figure 5. Molex PC Power Extension Cable  
[Created by Richard David McWhorter, III.]



Figure 4. RadioShack 22-812  
Digital Multimeter [Created by  
Richard David McWhorter, III.]

## **METHODS**

In order to accurately characterize how the NVIDIA GeForce FX 5900 consumes power, the project involved first writing an accurate test for each of the vertex and fragment operations and then running this test while recording power consumption. Before this main content could be undertaken, a preliminary test to confirm that power consumption was proportional to the number of vertices processed by the graphics processor was performed to test the feasibility of the project.

### **Preliminary Test**

In order to illustrate the way in which a stage of the rendering pipeline can be isolated and its power consumption characterized, the project started by constructing an example program that determines if a linear increase in the number of vertices a program

is processing per unit of time implies a linear increase in the amount of power it is consuming. This program drew quadrilaterals on the screen that covered a constant area, that is, contained the same number of fragments, but increased the number of vertices each time. The hope was to verify that as the number of vertices increases, power consumption increases. The source for this program is available in Appendix A.

This preliminary experiment also illustrated the method of determining power consumption. An ammeter was simply inserted into the external power source of the NVIDIA GeForce FX 5200 video card being used in the experiment to determine the current it was drawing from the external 5V and 12V lines. Current was then multiplied times the voltage to determine power consumption. This method is explained in more detail in the following tests.

### **Vertex and Fragment Program Tests**

The project then moved into the main test phase. Here the goal became developing test programs that would simply repeat each of the vertex and fragment program operations at a regular interval so that the power consumption could be recorded. It is important to understand that two types of programs were designed for this project: general programs that executed on the central processor and made calls to configure and control the graphics processor, and vertex and fragment programs which actually execute on the graphics processor, but are controlled by the general central processor programs. First, general programs that drew a huge number of vertices in close succession using either an arbitrary vertex or fragment program were designed. Then vertex and fragment programs for each vertex and fragment program operation were



designed. Finally, the test program was executed for each test vertex and fragment program and the power consumption recorded for each vertex and fragment operation.

Designing a program that simply rendered vertices in rapid succession using a vertex or fragment program proved to be quite a challenge. To begin, the program simply made the appropriate calls to render a small vertex array at a regular interval. As development progressed, however, a better strategy became apparent. The new strategy that emerged was to render an enormous vertex array a few times as quickly as possible. This would allow for recording of an operation's power consumption while the operation was being executed as frequently as possible. The final test programs were very similar for vertex and fragment programs; the source of the vertex program tester is available in Appendix B and the source of the fragment program tester is available in Appendix C.

The design of actual vertex and fragment programs that simply repeated an operation also proved to be more challenging than expected. The challenge arose because the graphics card driver, the software that controls the interaction between the general program and the graphics processor hardware, would optimize out some of the operations in a vertex or fragment program if it determined they did not change the output of the program. Thankfully, the UVA Computer Science department had a working relationship with NVIDIA and had a confidential method of preventing the driver from optimizing fragment programs, so these test programs were easy to design. An example test fragment program, for the addition operation, is included in Appendix D and a full listing of the fragment program operations is included in Appendix E. The only problem encountered for fragment programs was that four of the operations, listed in

the listing of instructions in Appendix E but not in the results in Appendix J, proved too difficult to design test cases for, so they are not included in the power characterization.

Unfortunately, no such optimization prevention method existed for vertex programs and these proved quite difficult to design. The main goal in keeping the driver from reducing the length of a vertex program was to make sure each operation, listed in Appendix F, was dependent on the operation before it. For some operations, such as addition, the design was trivial; simply adding a variable to itself repeatedly satisfied the dependency requirement. Designing vertex programs for other operations, however, such as absolute value, proved to be impossible, as taking the absolute value of a vector more than once produces the same result. Hence, the results for the absolute value, distance vector, move, and reciprocal vertex program operations are not significant. An example vertex test program, again for the addition operation, is included in Appendix G. Again, a two of the vertex program operations, listed in the listing of instructions in Appendix F but not in the results in Appendix I, proved too difficult to design test cases for and are not included in the power characterization.

Finally, after the general test program and specific vertex and fragment test programs were written, the test program was executed using each of the vertex and fragment test programs and the power recorded while the each vertex and fragment test program was executing. To record the power consumption of the board, the current through and the voltage across both the 5 volt and 12 volt wires of the card's external power supply were measured. This was done using an extension to the power supply cable by inserting an ammeter in series with the line under test, and by inserting a voltmeter in parallel with the line under test, as shown in Figure 6 on the next page for

the 5 volt line and Figure 7 for the 12 volt line. Since the card draws most, if not all, of



Figure 6. Power consumption test setup for 5V line. The red wire is the positive 5V lead, and is placed in series with the ammeter. The leads exiting the bottom of the picture are attached to the ammeter. The black wires are ground. The voltmeter, whose cords exit the picture to the right, is placed in parallel with the 5V line and ground. [Created by Richard David McWhorter, III.]



Figure 7. Power consumption test setup for 12V line. The orange wire is the positive 12V lead, and is placed in series with the ammeter. The leads exiting the bottom of the picture are attached to the ammeter. The black wires are ground. The voltmeter, whose cords exit the picture to the right, is placed in parallel with the 12V line and ground. [Created by Richard David McWhorter, III.]

its power from this external power supply, measuring the currents and voltages coming into the card through the accelerated graphics port bus was not necessary.

## CHAPTER FIVE: THE RESULTS OF THE POWER CHARACTERIZATION

Now that the report has described the experimental setup for the power characterization experiments, it moves on to reveal the results of these experiments. First, the report reveals the results of the initial test to assure that power consumption tracked linearly with the number of vertices the graphics card processed. Next, it moves to the heart of the project and reveals the results of the vertex program test for each operation. Finally, it reveals the results of the fragment program test for each operation.

### PRELIMINARY POWER CONSUMPTION TEST

Figure 8 shows the results of the preliminary experiment to verify that an increase

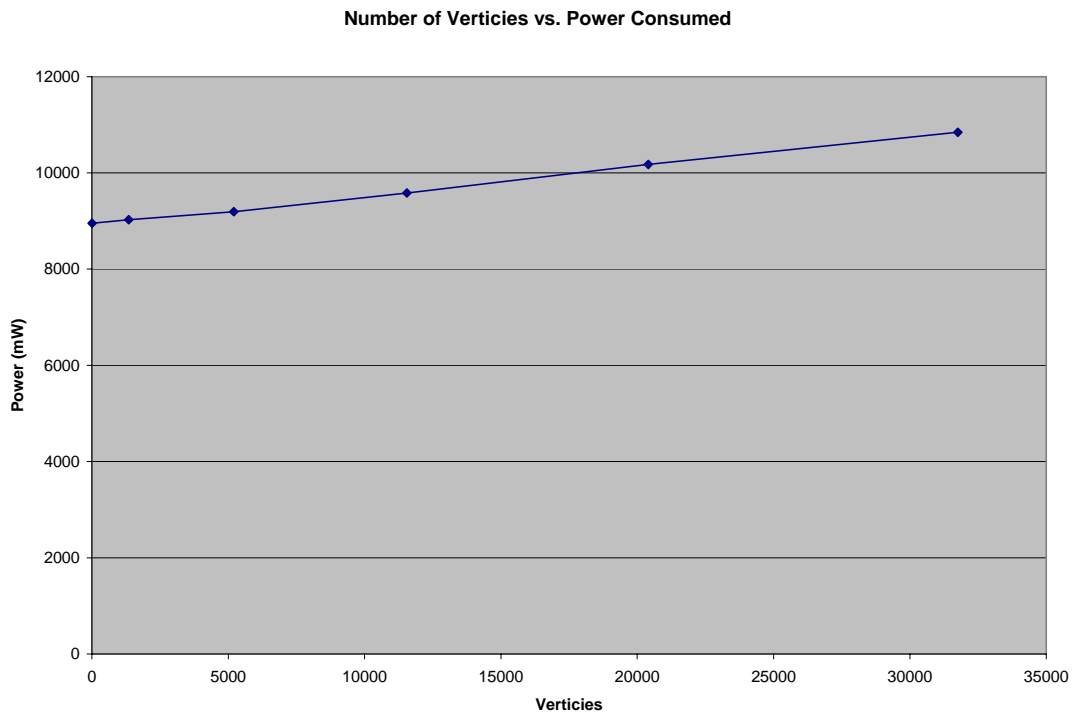


Figure 8. The results of the preliminary experiment that tested the amount of power the graphics card was consuming based upon the number of vertices it was processing per unit of time, while holding all other factors constant. The graph clearly shows that it takes a definite amount of power to process each vertex because of the linear relationship. [Created by Richard David McWhorter, III.]

in the number of vertices that the graphics processor is processing per unit of time

implied a linear increase in power consumption. As expected, the amount of power consumed by the card did track linearly with the number of vertices processed by the card. The full set of results for this experiment is available in Appendix H.

## VERTEX PROGRAM OPERATIONS TESTS

The results of the power characterization for vertex programs are illustrated in Figure 9. A full listing of the recorded voltages, currents, margins of error, and powers

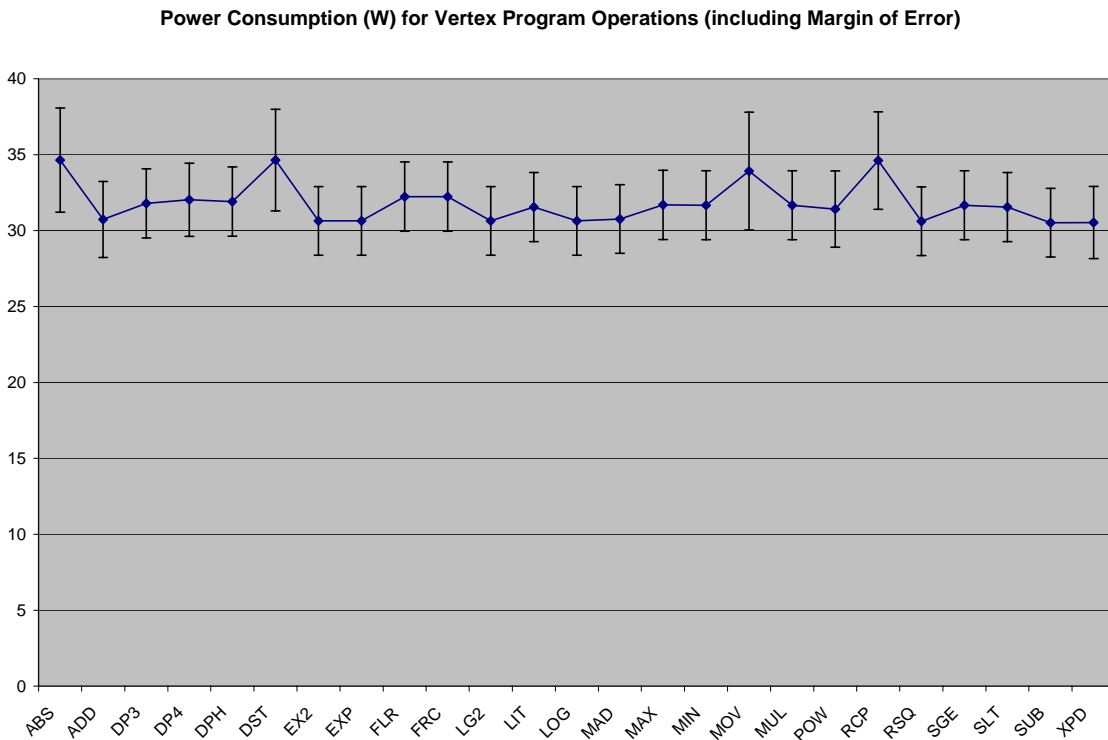


Figure 9. The results of the vertex program power characterization experiment. The graph includes the recorded power consumptions of each operation and includes error bars indicating the margin of error for each measurement. [Created by Richard David McWhorter, III.]

are included in Appendix I. What is significant to note about the results of this power characterization experiment is that there is no power difference, outside of the margin of error, between any of the vertex program operations.

## FRAGMENT PROGRAM OPERATIONS TESTS

Similarly, the results of the power characterization for fragment programs are illustrated in Figure 10. A full listing of the recorded voltages, currents, margins of error,

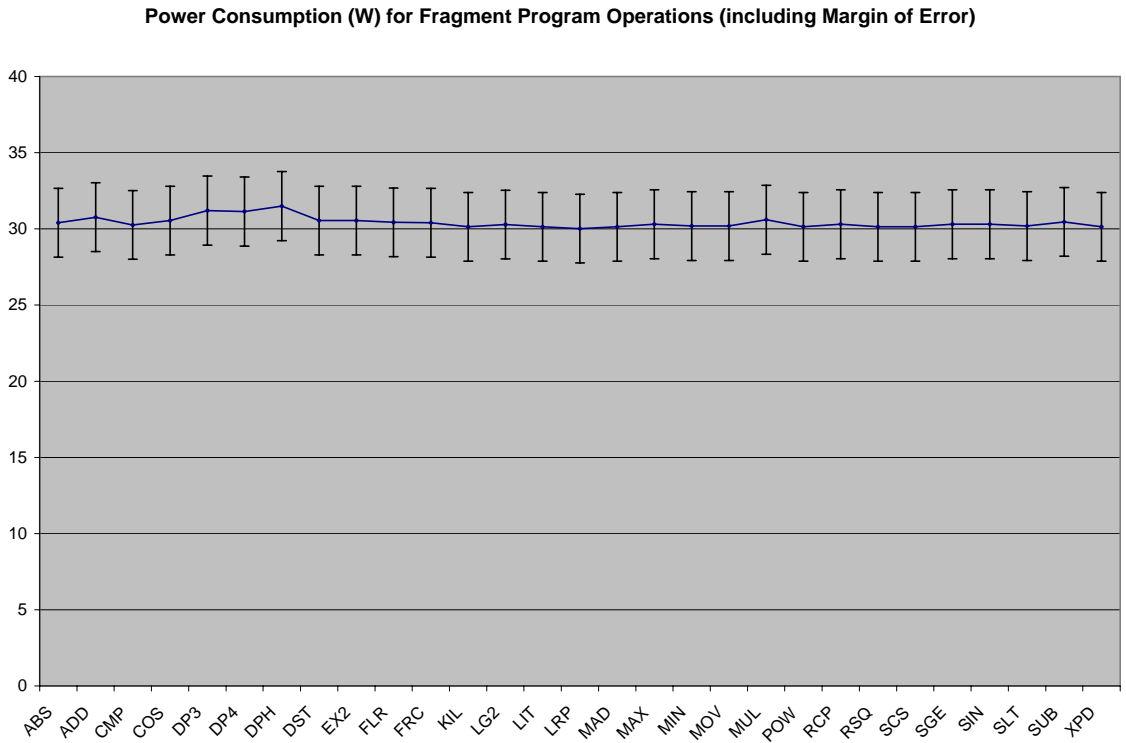


Figure 10. The results of the fragment program power characterization experiment. The graph includes the recorded power consumptions of each operation and includes error bars indicating the margin of error for each measurement. [Created by Richard David McWhorter, III.]

and powers are included in Appendix J. What is significant to note about the results of this power characterization experiment is that, as with vertex program operations, there is no power difference, outside of the margin of error, between any of the fragment program operations.

## **CHAPTER SIX: THE MEANING OF THE POWER CHARACTERIZATION**

The results of the power characterization experiments were very surprising. This chapter first looks at the results of the preliminary power characterization test as well as reasons why these results supported the feasibility of the project. It then examines the reason there is no power difference between the vertex and fragment program operations, specifically, because it is likely that the graphics processor in question does not use clock gating. Finally, it explains why this research is still valuable and particularly how someone could use the testing methodology to produce results that are more interesting for future generations of graphics cards.

### **PRELIMINARY POWER CONSUMPTION TEST**

The main purposes of the preliminary power consumption test were to evaluate the experimental setup for recording power consumption of the graphics card and to verify that changes to the scenes the graphics card is rendering do indeed produce logical changes in power consumption. This test took place before the project was proposed and confirmed its feasibility. It was useful because it provided a first try at recording power levels of the card. It mainly proved that changing one variable on the graphics card in a controlled manner produced a change in power consumption of the graphics card. Knowing this, the project proceeded directly to address its purpose, finding the relationship between the different vertex and fragment program operations and the amount of power the graphics processor consumed.

## **VERTEX AND FRAGMENT PROGRAM OPERATIONS POWER TEST**

The power characterization experiment for vertex and fragment program operations revealed that, for the NVIDIA GeForce FX 5900, there is no correspondence between which vertex or fragment operation the graphics processor is executing and how much power the graphics processor is consuming. While this result is surprising, it can be explained. The most likely reason for the lack of relationship between vertex and fragment program operation and power consumption is that the graphics processor does not use clock gating. This project still produces a valuable result, however, for a few different reasons.

### **Explanation for the Lack of Power Correspondence: Clock Gating**

According to Li et al. (2003), clock gating is a widely used technique for reducing power consumption. Specifically, clock gating refers to the practice of placing an *and* gate, a digital logic element that outputs a logic high signal if both of its inputs are logic high signals, in front of the clock input to a circuit, effectively creating an enable for the clock signal going into the circuit (p. 1). This allows the designer of the chip to disable sections of the chip when they are not needed, by lowering this enable signal going into the *and* gate with the clock. Hence, for one operation, certain sections of the chip that are not used for that operation can be disabled, while for other operations they are enabled. This results in the chip having varied power consumption when it is executing different instructions.

The converse is also true. If a chip does not use clock gating, it is likely to have the same power consumption regardless of what operations it is performing. A fundamental assumption behind this experiment was that the graphics processor used



clock gating. Since hardware manufacturers such as NVIDIA do not release any sort of architecture specifications for their chips, it is impossible to know for sure if architecture of the GeForce FX 5900 is clock-gated. The fact that the chip does not change power based upon which operations it is executing, however, is strong evidence that the chip does not use clock gating.

### **Implications of the Lack of Power Correspondence**

This research has numerous useful outcomes. It reveals that all of the vertex and fragment program operations have the same cost. This means that programmers are free to choose any vertex or fragment program operation to accomplish their goals at the same cost, and that the only way for the programmer to reduce power consumption is to shorten the vertex or fragment program. The fact that the architecture of the graphics processor uses no clock gating is a useful result, as it provides information to improve simulators of graphics architectures, such as *Qsilver*. Finally, it provides a framework that can easily be used to determine the power differences among vertex and fragment program operations in future graphics processors that use clock gating, which is likely as graphics architectures continue to mature as central processor architectures did.

## **CHAPTER SEVEN: THE FUTURE OF THE POWER CHARACTERIZATION**

This project has examined the relationship between different programmable operations of a graphics processor, the NVIDIA GeForce FX 5900, and how much power the processor consumes. This work is important to and has potential impact upon the computer graphics community; graphics processor manufacturers; and graphics processor consumers. It is based upon a similar experiment performed by Vivek Tiwari in 1994 in which Tiwari wrote tests for a central processor that executed the different operations of the processor in a loop and then empirically recorded the power consumption of the processor as it was executing the instructions (p. 437-444).

To determine the relationship between the different vertex and fragment program operations and the graphics processor's power consumption, a preliminary experiment was first performed that confirmed that power consumption increases linearly as the number of vertices the graphics card is processing increases linearly. Next, general programs that caused millions of vertices to be rendered as quickly as possible using vertex or fragment program were written, as well as vertex and fragment programs for each of the vertex and fragment program operations that executed the respective operation repeatedly. Finally, the power consumption of the graphics processor was recorded while executing each of these vertex and fragment programs. These observations revealed that there is no relationship between which vertex or fragment program operation the graphics processor executes and the amount of power the graphics processor consumes. This is probably because the GeForce FX 5900 almost certainly does not use clock gating.

The set of power consumption values for the GeForce FX 5900 is quite accurate and close to complete. It is not as significant of a finding as was hoped before the experiment, but it is valuable because it reveals a key fact about the architecture of the graphics processor, namely, that it does not use clock gating. Knowing this fact is significant to programmers because they can know with confidence that it does not matter which operations their programs use, only how long they are, when trying to minimize graphics processor power consumption. It is also valuable to graphics architecture simulators, such as *Qsilver*, as they become much more accurate when they know that the architecture they are simulating does not use clock gating.

It would be reasonable to say that this project was ahead of its time. It has constructed a detailed model of how to determine the power consumption of various vertex and fragment programs if they were to be implemented using clock gating. Given that this practice has become widely used in central processor design, it is reasonable to think that future graphics processors will take advantage of clock gating. Hence, when they do, the framework this project provides will become very valuable in determining how much power the different vertex and fragment program operations consume.

In conclusion, while the finding that there is no relationship between which operations a modern programmable graphics processor is executing, and the amount of power it consumes is not terribly interesting, it is important for those who desire to know how the vertex and fragment engines of those processors are likely implemented in hardware. This project also provides a great framework for future research on graphics processors that use clock gating. While the outcomes of this research are clearly valuable now, only time will tell how valuable they truly become.

## WORKS CITED

- Akeley, K., Akin, A., Ashbaugh, B., Beretta, B., Carmack, J., Craighead, M., Dyke, K., Glanville, S., Gold, M., Hart, E., Kilgard, M., Licea-Kane, B., Lichtenbelt, B., Lindholm, E., Lipchak, B., Mark, B., McCombe, J., Morris, J., Paul, B., Poddar, B., Roell, T., Sandmel, J., Schelter, J., Stahl, G., Stauffer, J., & Triantos, N. (2003 August 17). OpenGL ARB\_vertex\_program extension specification. Silicon Graphics. Retrieved October 26, 2004 from the World Wide Web: [http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex\\_program.txt](http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt)
- Akenine-Moller, T. & Strom, J. (2003 July). Graphics for the Masses – A Hardware Rasterization Architecture for Mobile Phones. ACM Transactions on Graphics, 22(3), 801-808.
- Beretta, B., Brown, P., Craighead, M., Everitt, C., Hart, E., Leech, J., Licea-Kane, B., Poddar, B., Sandmel, J., Schelter, J., Seetharamaiah, A., & Triantos, N. (2003 August 22). OpenGL ARB\_fragment\_program extension specification. Silicon Graphics. Retrieved October 26, 2004 from the World Wide Web: [http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment\\_program.txt](http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt)
- Bose, P., Albonesi, D., & Marculescu, D. (2003 September-October). Guest Editors' Introduction: Power and Complexity Aware Design. IEEE Micro, 23(5), 8-11.
- Brandolese, C., Salice, F., Fornaciari, W., & Sciuto, D. (2002 November). Static power modeling of 32-bit microprocessors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 21(11), 1306-1316.
- Brooks, D., Bose, P., Srinivasan, V., Gschwind, M., Emma, P., & Rosenfield, M. (2003 September-November). New methodology for early-stage, microarchitecture-

- level power-performance analysis of microprocessors. IBM Journal of Research & Development, 47(5-6), 653-670.
- Brooks, D., Tiwari, V., & Martonosi, M. (2000 June 10-14). Watch: a framework for architectural-level power analysis and optimizations. Proceedings of the 27<sup>th</sup> International Symposium on Computer Architecture. Vancouver, BC, Canada.
- Case, L. (2002, November 25). Making the Right Graphic Choice. ExtremeTech.com  
Retrieved October 26, 2004 from Lexis-Nexis database on the World Wide Web:  
<http://www.lexis-nexis.com>
- Fernando, R. and Kilgard, M. (2003). The CG Tutorial. Boston: Addison-Wesley.
- Flinn, J. & Satyanarayanan, M. (2004). Managing Battery Lifetime with Energy-Aware Adaptation. ACM Transactions on Computer Systems, 22(2), 137-139.
- Jon Peddie Research Says PC Graphics Chips Hit Record Highs Q4 2003; Market Shares Shift. (2004 February 4). Business Wire. Retrieved October 26, 2004 from Lexis-Nexis database on the World Wide Web: <http://www.lexis-nexis.com>
- Kameyama, M., Kato, Y., Fijimoto, H., Negishi, H., Kodama, Y., Inoue, Y., & Kawai, H. (2003 July). 3D graphics LSI core for mobile phone "Z3D". Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. San Diego, California.
- Krikke, J. (2001, May-June). Graphics applications over the wireless Web: Japan sets the pace. IEEE Computer Graphics and Applications, 21(3), 9-15.
- Li, H., Bhunia, S., Chen, Y., Vijaykumar, T., & Roy, K. (2003 February 8-12). Deterministic clock gating for microprocessor power reduction. Proceedings of

the Ninth International Symposium on High-Performance Computer Architecture, 2003. West Lafayette, Indiana.

Liang, B., Lee, Y., Yeh, W., & Jen, C. (2002 June). Index rendering: hardware-efficient architecture for 3-D graphics in multimedia system. IEEE Transactions on Multimedia, 4(3), 343-360.

Macedonia, M. (2003). The GPU Enters Computing's Mainstream. IEEE Computer, 36(10), 106-108.

Merritt, R. (2003, July 28). Mobile specs duel for graphics prize. Electronic Engineering Times. Retrieved October 5, 2004 from Lexis-Nexis database on the World Wide Web: <http://www.lexis-nexis.com>

Park, W., Lee, K., Kim, I., Han, T., & Yang, S. (2003 November). An Effective Pixel Rasterization Pipeline Architecture for 3D Rendering Processors. IEEE Transactions on Computers, 52(11), 1501-1508.

Salvator, D. (2004 April 13). Preview: Nvidia's GeForce 6800 Ultra. ExtremeTech. Retrieved October 5, 2004 from the World Wide Web: <http://www.extremetech.com/article2/0,1558,1566805,00.asp>

Shaeffer, J., Luebke, D., & Skadron, K. (2004 August). A Flexible Simulation Framework for Graphics Architecture. Proceedings of the 2004 Graphics Hardware Workshop. Grenoble, France.

Skadron, K., Stan, M., Huang, W., Velusamy, S., Sankaranarayanan, K., & Tarjan, D. (2003, November-December). Temperature-aware computer systems: Opportunities and challenges. IEEE Micro, 23(6), 52-61.

- Takahashi, D. (2003, April 15). Graphics chip challenge: Nvidia and ATI battle for technology leadership. (Semiconductors). Electronic Business, 29(5), 23.
- Tiwari, V., Malik, S., & Wolfe, A. (1994). Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2(4), 437-445.
- Woo, R., Yoon, C., Kook, J., Lee, S., & Yoo, H. (2002 October). A 120-mW 3-D rendering engine with 6-Mb embedded DRAM and 3.2-GB/s runtime reconfigurable bus for PDA chip. IEEE Journal of Solid-State Circuits, 37(10), 1352-1355.
- Yoon, C., Woo, R., Kook, J., Lee, S., Lee, K., & Yoo, H. (2001 November). An 80/20-MHz 160-mW Multimedia Processor Integrated With Embedded DRAM, MPEG-4 Accelerator, and 3-D Rendering Engine for Mobile Applications. IEEE Journal of Solid-State Circuits, 36(11), 1758-1767.

## APPENDICES

### APPENDIX A – SOURCE OF THE PRELIMINARY POWER CONSUMPTION

#### TEST

```
// Power test to determine that the number of vertices
// being processed directly corresponds to the amount
// of power consumed by the GPU.
//
// Author: David McWhorter
//
// Adapted from OpenGL Win32 Tutorial
// http://www.nullterminator.net/opengl32.html

#include <string>
#include <windows.h>
#include <iostream>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glxt.h>
#include <GL/gh.h>
#include <GL/glut.h>
#include <ctime>

using namespace std;

bool initialized = false;

float state = 2;
const int MAX_STATE = 252;

void display()
{
    if( !initialized )
    {
        /*
        //VERTEX PROGRAM CODE

        glhInit("GL_ARB_vertex_program");

        string ProgramStr( "!!ARBvpl.0\n#Input\nATTRIB InPos = \"
        \"vertex.position;\nATTRIB InColor = vertex.color;\n\"
        \"#Output\nOUTPUT OutPos = result.position;\nOUTPUT\"
        \"OutColor = result.color;\n\"
        \"PARAM MVP[4] = { state.matrix.mvp }; # Modelview\" \"Projection
        Matrix.\nTEMP Temp;\n\"
        \"#Transform vertex to clip space\nDP4 Temp.x,\"
        \"MVP[0], InPos;\nDP4 Temp.y, MVP[1], InPos;\n\"
        \"DP4 Temp.z, MVP[2], InPos;\nDP4 Temp.w, MVP[3],\"
        \"InPos;\n#Output\nMOV OutPos, temp;\n\"
        \"MOV OutColor, InColor;\nEND\n;TEMP Temp2;\n\"
        \"ADD InColor, OutColor, Temp2;\n\" );

        const char * Program = ProgramStr.c_str();

        unsigned int VP;
        glGenProgramsARB(1,&VP);
        glProgramStringARB(GL_ARB_vertex_program,
            GL_PROGRAM_FORMAT_ASCII_ARB, strlen(Program), Program);

        glBindProgramARB(GL_ARB_vertex_program, VP);

        glEnable(GL_ARB_vertex_program);
```



```

        if ( GL_INVALID_OPERATION == glGetError() )
        {
            GLint errPos;
            glGetIntegerv( GL_PROGRAM_ERROR_POSITION_ARB,
                &errPos );
            const GLubyte *errString =
                glGetString( GL_PROGRAM_ERROR_STRING_ARB );
            printf("Error at position ");
            printf((char *) errPos);
            printf(" :");
            printf((char *) errString);
            exit(445);
        }

        initialized = true;
        printf("this is working");
        */
    }

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode( GL_PROJECTION );
    glBegin(GL_QUADS);

    float squareWidth = 1.0 / state;
    bool drawLevel1 = true;
    bool drawLevel2 = true;

    for( float horiz = -0.5; horiz < 0.49; horiz += squareWidth )
    {
        if( drawLevel1 ) drawLevel2 = true;
        else drawLevel2 = false;
        for( float vert = -0.5; vert < 0.49; vert += squareWidth )
        {
            if( drawLevel2 )
            {
                glColor3f(1.0, 1.0, 1.0);
                glVertex3f(horiz, vert, 1);
                glColor3f(1.0, 1.0, 1.0);
                glVertex3f(horiz + squareWidth, vert, 0);
                glColor3f(1.0, 1.0, 1.0);
                glVertex3f(horiz + squareWidth,
                    vert + squareWidth, -1);
                glColor3f(1.0, 1.0, 1.0);
                glVertex3f(horiz, vert + squareWidth, 0);
            }
            drawLevel2 = !drawLevel2;
        }
        drawLevel1 = !drawLevel1;
    }

    glEnd();
    glFlush();
}

LONG WINAPI WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    static PAINTSTRUCT ps;

    switch(uMsg) {
    case WM_PAINT:
        display();
        BeginPaint(hWnd, &ps);
        EndPaint(hWnd, &ps);
        return 0;

    case WM_SIZE:
        glViewport(0, 0, LOWORD(lParam), HIWORD(lParam));
        PostMessage(hWnd, WM_PAINT, 0, 0);
        return 0;

    case WM_CHAR:

```

```

        switch (wParam)
        {
        case 27:                                /* ESC key */
            PostQuitMessage(0);
            exit(0);
        case 32:                                /* SPACE */
            state += 50;
            if( state > MAX_STATE ) exit(0);
            display();
            BeginPaint(hWnd, &ps);
            EndPaint(hWnd, &ps);
            break;
        }
        return 0;

    case WM_CLOSE:
        PostQuitMessage(0);
        return 0;
    }

    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

HWND CreateOpenGLWindow(char* title, int x, int y, int width, int height, BYTE type,
DWORD flags)
{
    int          pf;
    HDC          hDC;
    HWND        hWnd;
    WNDCLASS    wc;
    PIXELFORMATDESCRIPTOR pfd;
    static HINSTANCE hInstance = 0;

    /* only register the window class once - use hInstance as a flag.*/
    if (!hInstance)
    {
        hInstance = GetModuleHandle(NULL);
        wc.style      = CS_OWNDC;
        wc.lpfnWndProc = (WNDPROC)WindowProc;
        wc.cbClsExtra = 0;
        wc.cbWndExtra = 0;
        wc.hInstance  = hInstance;
        wc.hIcon      = LoadIcon(NULL, IDI_WINLOGO);
        wc.hCursor    = LoadCursor(NULL, IDC_ARROW);
        wc.hbrBackground = NULL;
        wc.lpszMenuName = NULL;
        wc.lpszClassName = "OpenGL";

        if (!RegisterClass(&wc))
        {
            MessageBox(NULL, "RegisterClass() failed: "
                "Cannot register window class.",
                "Error", MB_OK);
            return NULL;
        }
    }

    hWnd = CreateWindow("OpenGL", title, WS_POPUP | WS_MAXIMIZE |
        WS_CLIPSIBLINGS | WS_CLIPCHILDREN, x, y, width, height, NULL, NULL, hInstance,
        NULL); // fullscreen window

    if (hWnd == NULL)
    {
        MessageBox(NULL, "CreateWindow() failed: "
            "Cannot create a window.", "Error", MB_OK);
        return NULL;
    }

    hDC = GetDC(hWnd);

    /* there is no guarantee that the contents of the stack that become

```

```

    the pfd are zeroed, therefore _make sure_ to clear these bits.*/
memset(&pfd, 0, sizeof(pfd));
pfd.nSize      = sizeof(pfd);
pfd.nVersion   = 1;
pfd.dwFlags    = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | flags;
pfd.iPixelFormat = type;
pfd.cColorBits = 32;

pf = ChoosePixelFormat(hDC, &pfd);
if (pf == 0) {
    MessageBox(NULL, "ChoosePixelFormat() failed: "
        "Cannot find a suitable pixel format.",
        "Error", MB_OK);
    return 0;
}

if (SetPixelFormat(hDC, pf, &pfd) == FALSE) {
    MessageBox(NULL, "SetPixelFormat() failed: "
        "Cannot set format specified.", "Error", MB_OK);
    return 0;
}

DescribePixelFormat(hDC, pf, sizeof(PIXELFORMATDESCRIPTOR), &pfd);
ReleaseDC(hWnd, hDC);

return hWnd;
}

int APIENTRY WinMain(HINSTANCE hCurrentInst, HINSTANCE hPreviousInst,
    LPSTR lpszCmdLine, int nCmdShow)
{
    HDC     hDC;                /* device context */
    HGLRC   hRC;                /* opengl context */
    HWND    hWnd;               /* window */
    MSG     msg;                /* message */

    hWnd = CreateOpenGLWindow("minimal", 0, 0, 256, 256, PFD_TYPE_RGBA,
        0);
    if (hWnd == NULL) exit(1);

    hDC = GetDC(hWnd);
    hRC = wglCreateContext(hDC);
    wglMakeCurrent(hDC, hRC);

    ShowWindow(hWnd, nCmdShow);

    static PAINTSTRUCT ps;

    clock_t LastRefresh = clock();
    clock_t CurrentTime = clock();

    while(true)
    {
        CurrentTime = clock();

        // refresh the display every 5ms
        if( ((float)(CurrentTime - LastRefresh) /
            (float)CLOCKS_PER_SEC) >= 0.005 ) {
            LastRefresh = CurrentTime;
            PostMessage(hWnd, WM_PAINT, 0, 0);
        }
        if(PeekMessage(&msg, hWnd, 0, 0, 0)) {
            GetMessage(&msg, hWnd, 0, 0);
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    wglMakeCurrent(NULL, NULL);
    ReleaseDC(hWnd, hDC);
    wglDeleteContext(hRC);
    DestroyWindow(hWnd);
}

```

```
    return msg.wParam;  
}
```

## APPENDIX B – SOURCE OF THE VERTEX PROGRAM TESTER

```
/*
/ Vertex Program Tester
/ David McWhorter
/ mcwhorter@virginia.edu
/
/ Some window initialization code adapted
/ from OpenGL Win32 Tutorial
/ http://www.nullterminator.net/opengl32.html
/
#pragma comment(lib, "opengl32.lib")
#pragma comment(lib, "glu32.lib")
#pragma comment(lib, "glaux.lib")
#pragma comment(lib, "glh.lib")

#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glh.h>
#include <cassert>
#include <fstream>
#include <string>
#include <ctime>

#include "glenumlookup.h"

using namespace std;

HINSTANCE globalHInstance;
HDC      globalDeviceContext;          // Handle to device context
HGLRC    globalRenderingContext;     // Rendering Context for OpenGL
HWND     globalHWnd;
RECT     globalRect;

const int SCREEN_WIDTH   = 1280;
const int SCREEN_HEIGHT = 1024;
const int SCREEN_DEPTH  = 16;
const int MAX_VP_SIZE   = 50000;

const int    NUMBER_OF_VERTICIES = 1000002; // should be divisble by 3

// b/c we're drawing triangles
double    verticies[NUMBER_OF_VERTICIES*3];
unsigned int globalVertexProgramID;

const char * VP_SOURCES[27] = { "AbsTest.asm",
                                "AddTest.asm",
                                "ArlTest.asm",
                                "Dp3Test.asm",
                                "Dp4Test.asm",
                                "DphTest.asm",
                                "DstTest.asm",
                                "Ex2Test.asm",
                                "ExpTest.asm",
                                "FlrTest.asm",
                                "FrcTest.asm",
                                "Lg2Test.asm",
                                "LitTest.asm",
                                "LogTest.asm",
                                "MadTest.asm",
                                "MaxTest.asm",
                                "MinTest.asm",
                                "MovTest.asm",
                                "MulTest.asm",
                                "PowTest.asm",
                                "RcpTest.asm",
```

```

        "RsqTest.asm",
        "SgeTest.asm",
        "SltTest.asm",
        "SubTest.asm",
        "SwzTest.asm",
        "XpdTest.asm" };

HWND SetupWindow(HINSTANCE hInstance);
void InitializeOpenGL(int width, int height);
void PopulateArrays();
bool SetupRendering(const char * vpSrc, ofstream &errStream);
void render(ofstream &errStream);
void DeInit();

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hprev, PSTR cmdline, int ishow)
{
    // Integer argument to program invocation indicates which
    // vertex program is being tested
    int i = atoi(cmdline);

    // Windows setup
    globalHwnd = SetupWindow(hInstance);
    if(globalHwnd == NULL) return -1;
    GetClientRect(globalHwnd, &globalRect);

    // Initialize OpenGL
    InitializeOpenGL(globalRect.right, globalRect.bottom);

    // Initialize the vertex array
    PopulateArrays();

    clock_t TestStartTime, TestEndTime;

    // Error and message output file
    ofstream fout("Output.txt");

    fout << VP_SOURCES[i] << endl;
    fout << "Initialization errors: ";

    SetupRendering(VP_SOURCES[i], fout);

    fout << endl;
    fout << "Rendering errors: ";

    TestStartTime = clock();
    render(fout);
    TestEndTime = clock();

    fout << endl;
    fout << "<<Start time: " << (long double) TestStartTime / CLK_TCK
        << ">>" << endl;
    fout << "<<End time: " << (long double) TestEndTime / CLK_TCK
        << ">>" << endl;
    fout << "<<Test length: " << (long double)
        (TestEndTime - TestStartTime) / CLK_TCK << ">>" << endl << endl;

    DeInit();
    return 0;
}

void render(ofstream &errStream)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glDisable(GL_DEPTH_TEST); // Disable depth testing so the card
    // renders every triangle
    glClearColor(0.0f, 0.0f, 1.0f, 1);
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    gluLookAt(50, 50, 150, 50, 50, 149, 0, 1, 0);

    glEnableClientState(GL_VERTEX_ARRAY);

```

```

glVertexPointer(3, GL_DOUBLE, 0, &verticies);

glEnable(GL_VERTEX_PROGRAM_ARB);
glBindProgramARB( GL_VERTEX_PROGRAM_ARB, globalVertexProgramID );

// Main testing loop.  Draws an array of 10000000 verticies
// 20 times.
for( int i = 0; i < 20; i++ ) {
    glDrawArrays(GL_TRIANGLES, 0, NUMBER_OF_VERTICIES);
}

glDisable(GL_VERTEX_PROGRAM_ARB);
glDisableClientState(GL_VERTEX_ARRAY);

errStream << glEnumLookup(glGetError()) << endl;
GLint errPos;
glGetIntegerv( GL_PROGRAM_ERROR_POSITION_ARB, &errPos );
const GLubyte *errString = glGetString( GL_PROGRAM_ERROR_STRING_ARB );
errStream << "error at position:" << errPos << (char *) errString << endl;

SwapBuffers( globalDeviceContext );
}

bool SetUpRendering(const char * vpSrc, ofstream &errStream)
{
    char Program[MAX_VP_SIZE];

    ifstream fin(vpSrc);
    if( !fin.is_open() ) {
        errStream << "Could not find file: " << vpSrc << endl;
        return false;
    }

    fin.read(Program, MAX_VP_SIZE);
    fin.close();
    char * temp = strstr(Program, "END");
    temp[3] = 0;

    assert( glGetError() == GL_NO_ERROR );
    glEnable(GL_VERTEX_PROGRAM_ARB);
    assert( glGetError() == GL_NO_ERROR );
    glBindProgramARB(GL_VERTEX_PROGRAM_ARB,globalVertexProgramID);
    assert( glGetError() == GL_NO_ERROR );
    glProgramStringARB(GL_VERTEX_PROGRAM_ARB,GL_PROGRAM_FORMAT_ASCII_ARB,
        (GLsizei) strlen((char *) Program),Program);
    errStream << glEnumLookup(glGetError()) << endl;
    GLint errPos;
    glGetIntegerv( GL_PROGRAM_ERROR_POSITION_ARB, &errPos );
    const GLubyte *errString = glGetString( GL_PROGRAM_ERROR_STRING_ARB );
    errStream << "error at position:" << errPos << (char *) errString << endl;
    assert( glGetError() == GL_NO_ERROR );
    glDisable(GL_VERTEX_PROGRAM_ARB);
    return true;
}

void PopulateArrays()
{
    for( int i = 0; i < ( NUMBER_OF_VERTICIES * 3 ); i = i + 9 )
    {
        verticies[i] = ((double) i) /
            (((double) NUMBER_OF_VERTICIES ) / 33.333333333333333);
        verticies[i+1] = ((double) i) /
            (((double) NUMBER_OF_VERTICIES ) / 33.333333333333333);
        verticies[i+2] = 0;
        verticies[i+3] = ((double) i) /
            (((double) NUMBER_OF_VERTICIES ) / 33.333333333333333) + 1;
        verticies[i+4] = ((double) i) /
            (((double) NUMBER_OF_VERTICIES ) / 33.333333333333333);
        verticies[i+5] = 0;
        verticies[i+6] = ((double) i) /
            (((double) NUMBER_OF_VERTICIES ) / 33.333333333333333) + 1;
    }
}

```

```

        vertices[i+7] = ((double) i) /
            (((double) NUMBER_OF_VERTICIES ) / 33.333333333333333) + 1;
        vertices[i+8] = 0;
    }
}

// This function handles all the initialization for OpenGL.
void InitializeOpenGL(int width, int height)
{
    globalDeviceContext = GetDC(globalHwnd);
    // This sets our global HDC
    // We don't free this hdc until the end of our program

    PIXELFORMATDESCRIPTOR pfd;
    int pixelformat;

    pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
    // Set the size of the structure
    pfd.nVersion = 1;
    // Always set this to 1

    // Pass in the appropriate OpenGL flags
    pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
    pfd.dwLayerMask = PFD_MAIN_PLANE; // We want the standard mask
    pfd.iPixelFormat = PFD_TYPE_RGBA; // want RGB and Alpha pixel type
    pfd.cColorBits = SCREEN_DEPTH; // Here we use our #define for the color
bits
    pfd.cDepthBits = SCREEN_DEPTH; // Depthbits is ignored for RGBA
    pfd.cAccumBits = 0; // No special bitplanes
needed
    pfd.cStencilBits = 0; // We desire no stencil bits

    // This gets us a pixel format that best matches the one passed in from the device
    if ( (pixelformat = ChoosePixelFormat(globalDeviceContext, &pfd)) == FALSE )
    {
        MessageBox(NULL, "ChoosePixelFormat failed", "Error", MB_OK);
        PostQuitMessage(0);
    }

    // This sets the pixel format that we extracted from above
    if (SetPixelFormat(globalDeviceContext, pixelformat, &pfd) == FALSE)
    {
        MessageBox(NULL, "SetPixelFormat failed", "Error", MB_OK);
        PostQuitMessage(0);
    }

    globalRenderingContext = wglCreateContext(globalDeviceContext);
    // This creates a rendering context from our hdc

    wglMakeCurrent(globalDeviceContext, globalRenderingContext);
    // This makes the rendering context we just created the one we want to use

    if (height==0) // Prevent A Divide By Zero error
    {
        height=1; // Make the Height Equal One
    }

    glViewport(0,0,width,height); // Make our viewport the whole window

    glMatrixMode(GL_PROJECTION); // Select The Projection Matrix
    glLoadIdentity(); // Reset The Projection Matrix

    // Calculate The Aspect Ratio Of The Window. The parameters are:
    // (view angle, aspect ration of the width to the height, the closest distance
    // to the camera before it clips, FOV, Ratio,
    // the farthest distance before it stops drawing)
    gluPerspective(45.0f, (GLfloat)width/(GLfloat)height, .5f, 150.0f);

    glMatrixMode(GL_MODELVIEW); // Select The Modelview Matrix
    glLoadIdentity(); // Reset The Modelview Matrix

```



```

        glhInit("GL_ARB_vertex_program"); // Initialize Vertex Program functionality

        assert( glGetError() == GL_NO_ERROR );
    }

// Our window proc. does nothing because we do not want to be interrupted.
LRESULT CALLBACK WinProc(HWND hWnd,UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    return 0;
}

// This function cleans up and then posts a quit message to the window
void DeInit()
{
    if (globalRenderingContext)
    {
        {
            wglMakeCurrent(NULL, NULL); // This frees our rendering memory
                                        // and sets everything back to normal

            // Delete our OpenGL Rendering Context
            wglDeleteContext(globalRenderingContext);
        }

        if (globalDeviceContext)
            // Release our HDC from memory
            ReleaseDC(globalHWnd, globalDeviceContext);

        ChangeDisplaySettings(NULL,0); // If So Switch Back To The Desktop
        ShowCursor(TRUE); // Show Mouse Pointer

        // Free the window class
        UnregisterClass("VertexPowerText", globalHInstance);

        PostQuitMessage (0); // Post a QUIT message to the window
    }
}

HWND SetUpWindow(HINSTANCE hInstance)
{
    HWND hWnd;
    WNDCLASS wndclass;

    // Init the size of the class
    memset(&wndclass, 0, sizeof(WNDCLASS));
    // Regular drawing capabilities
    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    // Pass our function pointer as the window procedure
    wndclass.lpfnWndProc = WinProc;
    // Assign our hInstance
    wndclass.hInstance = hInstance;
    // General icon
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    // An arrow for the cursor
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    // A white window
    wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    // Assign the class name
    wndclass.lpszClassName = "VertexPowerTest";

    // Register the class
    RegisterClass(&wndclass);

    DWORD dwStyle = WS_POPUP | WS_CLIPSIBLINGS | WS_CLIPCHILDREN;

    /*** Changing to full screen mode ***/
    DEVMODE dmSettings; // Device Mode variable
    memset(&dmSettings,0,sizeof(dmSettings)); // Makes Sure Memory's Cleared

```

```

// Get current settings -- This function fills our the settings
// This makes sure NT and Win98 machines change correctly
if(!EnumDisplaySettings(NULL,ENUM_CURRENT_SETTINGS,&dmSettings))
{
    // Display error message if we couldn't get display settings
    MessageBox(NULL, "Could Not Enum Display Settings", "Error", MB_OK);
    return NULL;
}

dmSettings.dmPelsWidth = SCREEN_WIDTH;           // Selected Screen Width
dmSettings.dmPelsHeight = SCREEN_HEIGHT;       // Selected Screen Height

// This function actually changes the screen to full screen
// CDS_FULLSCREEN Gets Rid Of Start Bar.
// We always want to get a result from this function to check if we failed
int result = ChangeDisplaySettings(&dmSettings,CDS_FULLSCREEN);

// Check if we didn't recieved a good return message From the function
if(result != DISP_CHANGE_SUCCESSFUL)
{
    // Display the error message and quit the program
    MessageBox(NULL, "Display Mode Not Compatible", "Error", MB_OK);
    PostQuitMessage(0);
}
//*** end changing to full screen mode ***//

ShowCursor(FALSE);                               // Hide the cursor

globalHInstance = hInstance;

RECT rWindow;
rWindow.left = 0;                                // Set Left Value To 0
rWindow.right = SCREEN_WIDTH;                   // Set Right Value To Requested Width
rWindow.top = 0;                                // Set Top Value To 0
rWindow.bottom = SCREEN_HEIGHT;                // Set Bottom Value To Requested Height

// Adjust Window To True Requested Size
AdjustWindowRect( &rWindow, dwStyle, false);

hWnd = CreateWindow("VertexPowerTest", "Vertex Power Test Program",
    dwStyle, 0, 0, rWindow.right - rWindow.left,
    rWindow.bottom - rWindow.top, NULL, NULL, hInstance, NULL);

if(!hWnd) return NULL;

ShowWindow(hWnd, SW_SHOWNORMAL);                // Show the window
UpdateWindow(hWnd);                             // Draw the window

SetFocus(hWnd);                                 // Sets Keyboard Focus To The Window

return hWnd;
}

```

## APPENDIX C – SOURCE OF THE FRAGMENT PROGRAM TESTER

```

/*****
/ Fragment Program Tester
/ David McWhorter
/ mcwhorter@virginia.edu
/
/ Some window initialization code adapted
/ from OpenGL Win32 Tutorial
/ http://www.nullterminator.net/opengl32.html
/*****/

#pragma comment(lib, "opengl32.lib")
#pragma comment(lib, "glu32.lib")
#pragma comment(lib, "glaux.lib")
#pragma comment(lib, "glh.lib")

#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glh.h>
#include <cassert>
#include <fstream>
#include <string>
#include <ctime>

#include "glenumlookup.h"

using namespace std;

HINSTANCE globalHInstance;
HDC        globalDeviceContext;           // Handle to device context
HGLRC      globalRenderingContext;       // Rendering Context for OpenGL
HWND       globalHWnd;
RECT       globalRect;

const int  SCREEN_WIDTH   = 1280;
const int  SCREEN_HEIGHT  = 1024;
const int  SCREEN_DEPTH   = 16;
const int  MAX_FP_SIZE    = 50000;

const int  NUMBER_OF_VERTICIES = 1000002; // should be divisble by 3

// b/c we're drawing triangles
double    verticies[NUMBER_OF_VERTICIES*3];
unsigned int globalFragmentProgramID;

const char * FP_SOURCES[33] = {
    "AbsTest.asm",
    "AddTest.asm",
    "CmpTest.asm",
    "CosTest.asm",
    "Dp3Test.asm",
    "Dp4Test.asm",
    "DphTest.asm",
    "DstTest.asm",
    "Ex2Test.asm",
    "FlrTest.asm",
    "FrcTest.asm",
    "KilTest.asm",
    "Ilg2Test.asm",
    "LitTest.asm",
    "LrpTest.asm",
    "MadTest.asm",
    "MaxTest.asm",
    "MinTest.asm",
    "MovTest.asm",
    "MulTest.asm",
    "PowTest.asm",

```

```

        "RcpTest.asm",
        "RsqTest.asm",
        "ScsTest.asm",
        "SgeTest.asm",
        "SinTest.asm",
        "SltTest.asm",
        "SubTest.asm",
        "SwzTest.asm",
        "TexTest.asm",
        "TxbTest.asm",
        "TxpTest.asm",
        "XpdTest.asm" };

HWND SetupWindow(HINSTANCE hInstance);
LRESULT CALLBACK WinProc(HWND hWnd,UINT uMsg, WPARAM wParam, LPARAM lParam);
void InitializeOpenGL(int width, int height);
void PopulateArrays();
bool SetupRendering(const char * vpSrc, ofstream &errStream);
void render(ofstream &errStream);
void DeInit();

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hprev, PSTR cmdline, int ishow)
{
    // Integer argument to program invocation indicates which
    // vertex program is being tested
    int i = atoi(cmdline);

    // Windows setup
    globalHwnd = SetupWindow(hInstance);
    if(globalHwnd == NULL) return -1;
    GetClientRect(globalHwnd, &globalRect);

    // Initialize OpenGL
    InitializeOpenGL(globalRect.right, globalRect.bottom);

    // Initialize the vertex array
    PopulateArrays();

    clock_t TestStartTime, TestEndTime;

    // Error and message output file
    ofstream fout("Output.txt");

    fout << FP_SOURCES[i] << endl;
    fout << "Initialization errors: ";

    SetupRendering(FP_SOURCES[i], fout);

    fout << endl;
    fout << "Rendering errors: ";

    TestStartTime = clock();
    render(fout);
    TestEndTime = clock();

    fout << endl;
    fout << "<<Start time: " << (long double) TestStartTime / CLK_TCK
        << ">>" << endl;
    fout << "<<End time: " << (long double) TestEndTime / CLK_TCK
        << ">>" << endl;
    fout << "<<Test length: " << (long double)
        (TestEndTime - TestStartTime) / CLK_TCK << ">>" << endl << endl;

    DeInit();
    return 0;
}

void render(ofstream &errStream)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

```

```

    glDisable(GL_DEPTH_TEST); // Disable depth testing so the card
                              // renders every triangle
    glClearColor(0.0f, 0.0f, 1.0f, 1);
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    gluLookAt(50, 50, 150,      50, 50, 149,      0, 1, 0);

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_DOUBLE, 0, &verticies);

    glEnable(GL_FRAGMENT_PROGRAM_ARB);
    glBindProgramARB( GL_FRAGMENT_PROGRAM_ARB, globalFragmentProgramID );

    // Main testing loop.  Draws an array of 10000000 vertices
    // 20 times.
    for( int i = 0; i < 20; i++ ) {
        glDrawArrays(GL_TRIANGLES, 0, NUMBER_OF_VERTICIES);
    }

    glDisable(GL_FRAGMENT_PROGRAM_ARB);
    glDisableClientState(GL_VERTEX_ARRAY);

    errStream << glEnumLookup(glGetError()) << endl;
    GLint errPos;
    glGetIntegerv( GL_PROGRAM_ERROR_POSITION_ARB, &errPos );
    const GLubyte *errString = glGetString( GL_PROGRAM_ERROR_STRING_ARB );
    errStream << "error at position:" << errPos << (char *) errString << endl;

    SwapBuffers( globalDeviceContext );
}

bool SetUpRendering(const char * fpSrc, ofstream &errStream)
{
    char Program[MAX_FP_SIZE];

    ifstream fin(fpSrc);
    if( !fin.is_open() ) {
        errStream << "Could not find file: " << fpSrc << endl;
        return false;
    }

    fin.read(Program, MAX_FP_SIZE);
    fin.close();
    char * temp = strstr(Program, "END");
    temp[3] = 0;

    assert( glGetError() == GL_NO_ERROR );
    glEnable(GL_FRAGMENT_PROGRAM_ARB);
    assert( glGetError() == GL_NO_ERROR );
    glBindProgramARB(GL_FRAGMENT_PROGRAM_ARB,globalFragmentProgramID);
    assert( glGetError() == GL_NO_ERROR );
    glProgramStringARB(GL_FRAGMENT_PROGRAM_ARB,GL_PROGRAM_FORMAT_ASCII_ARB,(GLsizei)
strlen((char *) Program),Program);
    errStream << glEnumLookup(glGetError()) << endl;
    GLint errPos;
    glGetIntegerv( GL_PROGRAM_ERROR_POSITION_ARB, &errPos );
    const GLubyte *errString = glGetString( GL_PROGRAM_ERROR_STRING_ARB );
    errStream << "error at position:" << errPos << (char *) errString << endl;
    assert( glGetError() == GL_NO_ERROR );
    glDisable(GL_FRAGMENT_PROGRAM_ARB);
    return true;
}

void PopulateArrays()
{
    for( int i = 0; i < ( NUMBER_OF_VERTICIES * 3 ); i = i + 9 )
    {
        verticies[i] = ((double) i) /
            (((double) NUMBER_OF_VERTICIES ) / 33.333333333333333);
        verticies[i+1] = ((double) i) /

```

```

        (((double) NUMBER_OF_VERTICIES ) / 33.333333333333333);
        vertices[i+2] = 0;
        vertices[i+3] = ((double) i) /
            (((double) NUMBER_OF_VERTICIES ) / 33.333333333333333) + 1;
        vertices[i+4] = ((double) i) /
            (((double) NUMBER_OF_VERTICIES ) / 33.333333333333333);
        vertices[i+5] = 0;
        vertices[i+6] = ((double) i) /
            (((double) NUMBER_OF_VERTICIES ) / 33.333333333333333) + 1;
        vertices[i+7] = ((double) i) /
            (((double) NUMBER_OF_VERTICIES ) / 33.333333333333333) + 1;
        vertices[i+8] = 0;
    }
}

// This function handles all the initialization for OpenGL.
void InitializeOpenGL(int width, int height)
{
    globalDeviceContext = GetDC(globalHwnd);
    // This sets our global HDC
    // We don't free this hdc until the end of our program

    PIXELFORMATDESCRIPTOR pfd;
    int pixelformat;

    pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
    // Set the size of the structure
    pfd.nVersion = 1;
    // Always set this to 1

    // Pass in the appropriate OpenGL flags
    pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
    pfd.dwLayerMask = PFD_MAIN_PLANE; // We want the standard mask
    pfd.iPixelFormat = PFD_TYPE_RGBA; // We want RGB and Alpha pixel type
    pfd.cColorBits = SCREEN_DEPTH; // Here we use our #define for the color bits
    pfd.cDepthBits = SCREEN_DEPTH; // Depthbits is ignored for RGBA
    pfd.cAccumBits = 0; // No special bitplanes needed
    pfd.cStencilBits = 0; // We desire no stencil bits

    // This gets us a pixel format that best matches the one passed in from the device
    if ( (pixelformat = ChoosePixelFormat(globalDeviceContext, &pfd)) == FALSE )
    {
        MessageBox(NULL, "ChoosePixelFormat failed", "Error", MB_OK);
        PostQuitMessage(0);
    }

    // This sets the pixel format that we extracted from above
    if (SetPixelFormat(globalDeviceContext, pixelformat, &pfd) == FALSE)
    {
        MessageBox(NULL, "SetPixelFormat failed", "Error", MB_OK);
        PostQuitMessage(0);
    }

    globalRenderingContext = wglCreateContext(globalDeviceContext);
    // This creates a rendering context from our hdc

    wglMakeCurrent(globalDeviceContext, globalRenderingContext);
    // This makes the rendering context we just created the one we want to use

    if (height==0) // Prevent A Divide By Zero error
    {
        height=1; // Make the Height Equal One
    }

    glViewport(0,0,width,height); // Make our viewport the whole window

    glMatrixMode(GL_PROJECTION); // Select The Projection Matrix
    glLoadIdentity(); // Reset The Projection Matrix

    // Calculate The Aspect Ratio Of The Window. The parameters are:

```

```

// (view angle, aspect ration of the width to the height, the closest distance
// to the camera before it clips, FOV, Ratio,
// the farthest distance before it stops drawing)
gluPerspective(45.0f, (GLfloat)width/(GLfloat)height, .5f, 150.0f);

glMatrixMode(GL_MODELVIEW);           // Select The Modelview Matrix
glLoadIdentity();                     // Reset The Modelview Matrix

glhInit("GL_ARB_vertex_program"); // Initialize Vertex Program functionality

assert( glGetError() == GL_NO_ERROR );
}

// Our window proc. does nothing because we do not want to be interrupted.
LRESULT CALLBACK WinProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    return 0;
}

// This function cleans up and then posts a quit message to the window
void DeInit()
{
    if (globalRenderingContext)
    {
        wglMakeCurrent(NULL, NULL); // This frees our rendering memory
                                   // and sets everything back to normal

        // Delete our OpenGL Rendering Context
        wglDeleteContext(globalRenderingContext);
    }

    if (globalDeviceContext)
        // Release our HDC from memory
        ReleaseDC(globalHWnd, globalDeviceContext);

    ChangeDisplaySettings(NULL, 0); // If So Switch Back To The Desktop
    ShowCursor(TRUE);              // Show Mouse Pointer

    // Free the window class
    UnregisterClass("VertexPowerText", globalHInstance);

    PostQuitMessage (0);          // Post a QUIT message to the window
}

HWND SetupWindow(HINSTANCE hInstance)
{
    HWND hWnd;
    WNDCLASS wndclass;

    // Init the size of the class
    memset(&wndclass, 0, sizeof(WNDCLASS));
    // Regular drawing capabilities
    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    // Pass our function pointer as the window procedure
    wndclass.lpfnWndProc = WinProc;
    // Assign our hInstance
    wndclass.hInstance = hInstance;
    // General icon
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    // An arrow for the cursor
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    // A white window
    wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
    // Assign the class name
    wndclass.lpszClassName = "VertexPowerTest";

    // Register the class
    RegisterClass(&wndclass);
}

```

```

DWORD dwStyle = WS_POPUP | WS_CLIPSIBLINGS | WS_CLIPCHILDREN;

/** Changing to full screen mode ***/
DEVMODE dmSettings; // Device Mode variable
memset(&dmSettings,0,sizeof(dmSettings)); // Makes Sure Memory's Cleared

// Get current settings -- This function fills our the settings
// This makes sure NT and Win98 machines change correctly
if(!EnumDisplaySettings(NULL,ENUM_CURRENT_SETTINGS,&dmSettings))
{
    // Display error message if we couldn't get display settings
    MessageBox(NULL, "Could Not Enum Display Settings", "Error", MB_OK);
    return NULL;
}

dmSettings.dmPelsWidth = SCREEN_WIDTH; // Selected Screen Width
dmSettings.dmPelsHeight = SCREEN_HEIGHT; // Selected Screen Height

// This function actually changes the screen to full screen
// CDS_FULLSCREEN Gets Rid Of Start Bar.
// We always want to get a result from this function to check if we failed
int result = ChangeDisplaySettings(&dmSettings,CDS_FULLSCREEN);

// Check if we didn't recieved a good return message From the function
if(result != DISP_CHANGE_SUCCESSFUL)
{
    // Display the error message and quit the program
    MessageBox(NULL, "Display Mode Not Compatible", "Error", MB_OK);
    PostQuitMessage(0);
}
/** end changing to full screen mode ***/

ShowCursor(FALSE); // Hide the cursor

globalHInstance = hInstance;

RECT rWindow;
rWindow.left = 0; // Set Left Value To 0
rWindow.right = SCREEN_WIDTH; // Set Right Value To Requested Width
rWindow.top = 0; // Set Top Value To 0
rWindow.bottom = SCREEN_HEIGHT; // Set Bottom Value To Requested Height

// Adjust Window To True Requested Size
AdjustWindowRect( &rWindow, dwStyle, false);

hWnd = CreateWindow("VertexPowerTest", "Vertex Power Test Program",
    dwStyle, 0, 0, rWindow.right - rWindow.left,
    rWindow.bottom - rWindow.top, NULL, NULL, hInstance, NULL);

if(!hWnd) return NULL;

ShowWindow(hWnd, SW_SHOWNORMAL); // Show the window
UpdateWindow(hWnd); // Draw the window

SetFocus(hWnd); // Sets Keyboard Focus To The Window

return hWnd;
}

```





```
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
ADD Test, Test, Test;
```

```
#Output
MOV OutPos, InPos;
MOV OutColor, Test;
```

```
END
```

## APPENDIX E – LISTING OF OPENGL FRAGMENT PROGRAM OPERATIONS

There are thirty-three vertex program instructions. The instructions and their respective input and output parameters are summarized in the table below. “v” indicates a floating-point vector input or output, “s” indicates a floating-point scalar input, “ssss” indicates a scalar output replicated across a 4-component result vector, “ss--” indicates two scalar outputs in the first two components, “u” indicates a texture image unit identifier, and “t” indicates a texture target.

<u>Instruction</u>	<u>Inputs</u>	<u>Output</u>	<u>Description</u>
ABS	v	v	absolute value
ADD	v,v	v	add
CMP	v,v,v	v	compare
COS	s	ssss	cosine with reduction to [-PI,PI]
DP3	v,v	ssss	3-component dot product
DP4	v,v	ssss	4-component dot product
DPH	v,v	ssss	homogeneous dot product
DST	v,v	v	distance vector
EX2	s	ssss	exponential base 2
FLR	v	v	floor
FRC	v	v	fraction
KIL	v	v	kill fragment
LG2	s	ssss	logarithm base 2
LIT	v	v	compute light coefficients
LRP	v,v,v	v	linear interpolation
MAD	v,v,v	v	multiply and add
MAX	v,v	v	maximum
MIN	v,v	v	minimum
MOV	v	v	move
MUL	v,v	v	multiply
POW	s,s	ssss	exponentiate
RCP	s	ssss	reciprocal
RSQ	s	ssss	reciprocal square root
SCS	s	ss--	sine/cosine without reduction
SGE	v,v	v	set on greater than or equal
SIN	s	ssss	sine with reduction to [-PI,PI]
SLT	v,v	v	set on less than
SUB	v,v	v	subtract
SWZ	v	v	extended swizzle
TEX	v,u,t	v	texture sample
TXB	v,u,t	v	texture sample with bias
TXP	v,u,t	v	texture sample with projection
XPD	v,v	v	cross product

(Beretta et al.)

## APPENDIX F – LISTING OF OPENGL VERTEX PROGRAM OPERATIONS

There are twenty-seven vertex program instructions. The instructions and their respective input and output parameters are summarized in the table below. “v” indicates a floating-point vector input or output, “s” indicates a floating-point scalar input, “ssss” indicates a scalar output replicated across a 4-component result vector, and “a” indicates a single address register component.

<u>Instruction</u>	<u>Input</u>	<u>Output</u>	<u>Description</u>
ABS	v	v	absolute value
ADD	v,v	v	add
ARL	v	a	address register load
DP3	v,v	ssss	3-component dot product
DP4	v,v	ssss	4-component dot product
DPH	v,v	ssss	homogeneous dot product
DST	v,v	v	distance vector
EX2	s	ssss	exponential base 2
EXP	s	v	exponential base 2 (approximate)
FLR	v	v	floor
FRC	v	v	fraction
LG2	s	ssss	logarithm base 2
LIT	v	v	compute light coefficients
LOG	s	v	logarithm base 2 (approximate)
MAD	v,v,v	v	multiply and add
MAX	v,v	v	maximum
MIN	v,v	v	minimum
MOV	v	v	move
MUL	v,v	v	multiply
POW	s,s	ssss	exponentiate
RCP	s	ssss	reciprocal
RSQ	s	ssss	reciprocal square root
SGE	v,v	v	set on greater than or equal
SLT	v,v	v	set on less than
SUB	v,v	v	subtract
SWZ	v	v	extended swizzle
XPD	v,v	v	cross product

(Akeley et al.)





**APPENDIX H – RESULTS OF PRELIMINARY POWER CONSUMPTION TEST**

<b><u>8 vertices</u></b>	Current (mA)	Voltage (V)	Power (mW)
5V EXT #1	826	5	4130
12V EXT #1	402	12	4824
Total:			8954
<b><u>1352 vertices</u></b>	Current (mA)	Voltage (V)	Power (mW)
5V EXT #1	838	5	4190
12V EXT #1	403	12	4836
Total:			9026
<b><u>5202 vertices</u></b>	Current (mA)	Voltage (V)	Power (mW)
5V EXT #1	871	5	4355
12V EXT #1	403	12	4836
Total:			9191
<b><u>11552 vertices</u></b>	Current (mA)	Voltage (V)	Power (mW)
5V EXT #1	947	5	4735
12V EXT #1	404	12	4848
Total:			9583
<b><u>20402 vertices</u></b>	Current (mA)	Voltage (V)	Power (mW)
5V EXT #1	1053	5	5265
12V EXT #1	409	12	4908
Total:			10173
<b><u>31752 vertices</u></b>	Current (mA)	Voltage (V)	Power (mW)
5V EXT #1	1182	5	5910
12V EXT #1	411	12	4932
Total:			10842

## APPENDIX I – VERTEX PROGRAM OPERATION TEST RESULTS

This appendix includes the full results from the vertex program power characterizations. In the tables below, Error from Measurement indicates how much the observed current or voltage was moving in the steady state while rendering the vertices for the corresponding operation and Error for Ammeter or Voltmeter Accuracy indicates the error from the stated accuracy of the Ammeter or Voltmeter. The accuracy of the Ammeter, or the Fluke 189 Digital Multimeter was  $\pm 0.3\%$  plus ten significant digits for current measurements in the range of up to 5 amps. The accuracy of the Voltmeter, or the RadioShack 22-812 Digital Multimeter, was  $\pm 0.3\%$  plus four significant digits for voltage measurements in the range of up to 40 V.

Test Program	12V current (A)	Err. from Meas. ( $\pm A$ )	Err. from Ammeter Acc. ( $\pm A$ )	12V Voltage (V)	Err. from Meas. ( $\pm V$ )	Err. from Voltmeter Acc. ( $\pm V$ )
ABS	2.57	0.07	0.1132	12.13	0.02	0.06645
ADD	2.35	0.01	0.1118	12.13	0.01	0.06642
DP3	2.42	0.01	0.11215	12.13	0.01	0.06642
DP4	2.44	0.02	0.1123	12.13	0.01	0.06642
DPH	2.43	0.01	0.1122	12.13	0.01	0.06642
DST	2.57	0.07	0.1132	12.13	0.01	0.06642
EX2	2.34	0.01	0.11175	12.14	0.01	0.06645
EXP	2.34	0.01	0.11175	12.14	0.01	0.06645
FLR	2.44	0.01	0.11225	12.13	0.01	0.06642
FRC	2.44	0.01	0.11225	12.13	0.01	0.06642
LG2	2.34	0.01	0.11175	12.14	0.01	0.06645
LIT	2.4	0.01	0.11205	12.13	0.01	0.06642
LOG	2.34	0.01	0.11175	12.14	0.01	0.06645
MAD	2.35	0.01	0.1118	12.14	0.01	0.06645
MAX	2.41	0.01	0.1121	12.14	0.01	0.06645
MIN	2.41	0.01	0.1121	12.13	0.01	0.06642
MOV	2.51	0.11	0.1131	12.13	0.01	0.06642
MUL	2.41	0.01	0.1121	12.13	0.01	0.06642
POW	2.41	0.03	0.1122	12.13	0.01	0.06642
RCP	2.58	0.06	0.1132	12.13	0.01	0.06642
RSQ	2.34	0.01	0.11175	12.13	0.01	0.06642
SGE	2.41	0.01	0.1121	12.13	0.01	0.06642
SLT	2.4	0.01	0.11205	12.13	0.01	0.06642
SUB	2.33	0.01	0.1117	12.14	0.01	0.06645
XPD	2.34	0.02	0.1118	12.14	0.01	0.06645
Test Program	5V current (A)	Err. from Meas. ( $\pm A$ )	Err. from Ammeter Acc. ( $\pm A$ )	5V Voltage (V)	Err. from Meas. ( $\pm V$ )	Err. from Voltmeter Acc.
ABS	0.67	0.07	0.1037	5.17	0.01	0.04554
ADD	0.43	0.05	0.1024	5.18	0	0.04554
DP3	0.47	0.005	0.102375	5.18	0.01	0.04557
DP4	0.47	0.005	0.102375	5.18	0.01	0.04557
DPH	0.47	0.005	0.102375	5.18	0.01	0.04557



DST	0.67	0.06	0.10365	5.17	0.01	0.04554
EX2	0.43	0.005	0.102175	5.18	0	0.04554
EXP	0.43	0.005	0.102175	5.18	0	0.04554
FLR	0.51	0.005	0.102575	5.17	0	0.04551
FRC	0.51	0.005	0.102575	5.17	0	0.04551
LG2	0.43	0.005	0.102175	5.18	0	0.04554
LIT	0.47	0.005	0.102375	5.18	0.01	0.04557
LOG	0.43	0.005	0.102175	5.18	0	0.04554
MAD	0.43	0.005	0.102175	5.18	0	0.04554
MAX	0.47	0.005	0.102375	5.18	0.01	0.04557
MIN	0.47	0.005	0.102375	5.18	0.01	0.04557
MOV	0.67	0.07	0.1037	5.18	0	0.04554
MUL	0.47	0.005	0.102375	5.18	0.01	0.04557
POW	0.42	0.005	0.102125	5.18	0	0.04554
RCP	0.64	0.06	0.1035	5.18	0	0.04554
RSQ	0.43	0.005	0.102175	5.18	0	0.04554
SGE	0.47	0.005	0.102375	5.18	0	0.04554
SLT	0.47	0.005	0.102375	5.18	0	0.04554
SUB	0.43	0.005	0.102175	5.18	0	0.04554
XPD	0.41	0.005	0.102075	5.18	0	0.04554
Test Program	Power (W)	Mar. of Err. ( $\pm W$ )	Time to draw 1000000 triangles 20 times (s)		(Trial 1)	(Trial 2)
ABS	34.638	3.430603176	4.1		4.1	4.1
ADD	30.7329	2.498531704	26.95		27	26.9
DP3	31.7892	2.279539364	14.05		14.1	14
DP4	32.0318	2.40573859	13.75		13.8	13.7
DPH	31.9105	2.280917706	14		14	14
DST	34.638	3.34807623	4.1		4.1	4.1
EX2	30.635	2.260063774	26.85		26.9	26.8
EXP	30.635	2.260063774	26.8		26.9	26.7
FLR	32.2339	2.277206317	10		10	10
FRC	32.2339	2.277206317	9.9		9.9	9.9
LG2	30.635	2.260063774	26.8		26.9	26.7
LIT	31.5466	2.27678268	13.95		14	13.9
LOG	30.635	2.260063774	26.7		26.8	26.6
MAD	30.7564	2.261442919	28.3		29.9	26.7
MAX	31.692	2.279461648	13.95		14	13.9
MIN	31.6679	2.278161022	13.95		14	13.9
MOV	33.9169	3.8782142	4.3		4.6	4
MUL	31.6679	2.278161022	13.95		14	13.9
POW	31.4089	2.514583293	30.1		30.1	30.1
RCP	34.6106	3.215518668	4.25		4.3	4.2
RSQ	30.6116	2.258768769	26.85		26.8	26.9
SGE	31.6679	2.271292979	14.05		14	14.1
SLT	31.5466	2.269914637	14.05		14	14.1
SUB	30.5136	2.258684629	27		27	27
XPD	30.5314	2.382169511	40.35		40.2	40.5

## APPENDIX J – FRAGMENT PROGRAM OPERATION TEST RESULTS

This appendix includes the full results from the fragment program power characterizations. In the tables below, Error from Measurement indicates how much the observed current or voltage was moving in the steady state while rendering the verticies for the corresponding operation and Error for Ammeter or Voltmeter Accuracy indicates the error from the stated accuracy of the Ammeter or Voltmeter. The accuracy of the Ammeter, or the Fluke 189 Digital Multimeter was  $\pm 0.3\%$  plus ten significant digits for current measurements in the range of up to 5 amps. The accuracy of the Voltmeter, or the RadioShack 22-812 Digital Multimeter, was  $\pm 0.3\%$  plus four significant digits for voltage measurements in the range of up to 40 V.

Test Program	12V current (A)	Err. from Meas. ( $\pm A$ )	Err. from Ammeter Acc. ( $\pm A$ )	12V Voltage (V)	Err. from Meas. ( $\pm V$ )	Err. from Voltmeter Acc. ( $\pm V$ )
ABS	2.34	0.01	0.11175	12.13	0.01	0.06642
ADD	2.37	0.01	0.1119	12.13	0.01	0.06642
CMP	2.33	0.01	0.1117	12.14	0.01	0.06645
COS	2.35	0.01	0.1118	12.14	0.01	0.06645
DP3	2.41	0.01	0.1121	12.13	0.01	0.06642
DP4	2.4	0.01	0.11205	12.13	0.01	0.06642
DPH	2.43	0.01	0.1122	12.13	0.01	0.06642
DST	2.35	0.01	0.1118	12.14	0.01	0.06645
EX2	2.35	0.01	0.1118	12.14	0.01	0.06645
FLR	2.34	0.01	0.11175	12.14	0.01	0.06645
FRC	2.34	0.01	0.11175	12.13	0.01	0.06642
KIL	2.32	0.01	0.11165	12.14	0.01	0.06645
LG2	2.33	0.01	0.1117	12.13	0.01	0.06642
LIT	2.32	0.01	0.11165	12.14	0.01	0.06645
LRP	2.31	0.01	0.1116	12.14	0.01	0.06645
MAD	2.32	0.01	0.11165	12.14	0.01	0.06645
MAX	2.33	0.01	0.1117	12.14	0.01	0.06645
MIN	2.32	0.01	0.11165	12.14	0.01	0.06645
MOV	2.32	0.01	0.11165	12.14	0.01	0.06645
MUL	2.35	0.01	0.1118	12.14	0.01	0.06645
POW	2.32	0.01	0.11165	12.14	0.01	0.06645
RCP	2.33	0.01	0.1117	12.14	0.01	0.06645
RSQ	2.32	0.01	0.11165	12.14	0.01	0.06645
SCS	2.32	0.01	0.11165	12.14	0.01	0.06645
SGE	2.33	0.01	0.1117	12.14	0.01	0.06645
SIN	2.33	0.01	0.1117	12.14	0.01	0.06645
SLT	2.32	0.01	0.11165	12.14	0.01	0.06645
SUB	2.34	0.01	0.11175	12.13	0.01	0.06642
XPD	2.32	0.01	0.11165	12.14	0.01	0.06645

<u>Test Program</u>	<u>5V current (A)</u>	<u>Err. from Meas. (±A)</u>	<u>Err. from Ammeter Acc. (±A)</u>	<u>5V Voltage (V)</u>	<u>Err. from Meas. (±V)</u>	<u>Err. from Voltmeter Acc. (±V)</u>
ABS	0.39	0.005	0.101975	5.18	0	0.04554
ADD	0.39	0.005	0.101975	5.18	0	0.04554
CMP	0.38	0.005	0.101925	5.18	0	0.04554
COS	0.39	0.005	0.101975	5.18	0	0.04554
DP3	0.38	0.005	0.101925	5.18	0	0.04554
DP4	0.39	0.005	0.101975	5.18	0	0.04554
DPH	0.39	0.005	0.101975	5.18	0	0.04554
DST	0.39	0.005	0.101975	5.18	0	0.04554
EX2	0.39	0.005	0.101975	5.18	0	0.04554
FLR	0.39	0.005	0.101975	5.18	0	0.04554
FRC	0.39	0.005	0.101975	5.18	0	0.04554
KIL	0.38	0.005	0.101925	5.18	0	0.04554
LG2	0.39	0.005	0.101975	5.18	0	0.04554
LIT	0.38	0.005	0.101925	5.18	0	0.04554
LRP	0.38	0.005	0.101925	5.18	0	0.04554
MAD	0.38	0.005	0.101925	5.18	0	0.04554
MAX	0.39	0.005	0.101975	5.18	0	0.04554
MIN	0.39	0.005	0.101975	5.18	0	0.04554
MOV	0.39	0.005	0.101975	5.18	0	0.04554
MUL	0.4	0.005	0.102025	5.18	0	0.04554
POW	0.38	0.005	0.101925	5.18	0	0.04554
RCP	0.39	0.005	0.101975	5.18	0	0.04554
RSQ	0.38	0.005	0.101925	5.18	0	0.04554
SCS	0.38	0.005	0.101925	5.18	0	0.04554
SGE	0.39	0.005	0.101975	5.18	0	0.04554
SIN	0.39	0.005	0.101975	5.18	0	0.04554
SLT	0.39	0.005	0.101975	5.18	0	0.04554
SUB	0.4	0.005	0.102025	5.18	0	0.04554
XPD	0.38	0.005	0.101925	5.18	0	0.04554
<u>Test Program</u>	<u>Power (W)</u>	<u>Mar. of Err. (±W)</u>	<u>Time to draw 1000000 triangles 20 times (s)</u>		<u>(Trial 1)</u>	<u>(Trial 2)</u>
ABS	30.4044	2.25589295	19.625		20.5	18.75
ADD	30.7683	2.26002798	7.85		8.2	7.5
CMP	30.2546	2.25508986	70.25		73.4	67.1
COS	30.5492	2.2585671	20.2		20.5	19.9
DP3	31.2017	2.26482239	20.05		20.4	19.7
DP4	31.1322	2.26416301	21		21.3	20.7
DPH	31.4961	2.26829803	21.25		21.6	20.9
DST	30.5492	2.2585671	20.05		20.4	19.7
EX2	30.5492	2.2585671	20.25		20.6	19.9
FLR	30.4278	2.25718796	20.2		20.5	19.9

FRC	30.4044	2.25589295	20.2	20.5	19.9
KIL	30.1332	2.25371071	39.15	39.4	38.9
LG2	30.2831	2.25451461	20.4	20.5	20.3
LIT	30.1332	2.25371071	39.45	39.6	39.3
LRP	30.0118	2.25233157	40.95	41.2	40.7
MAD	30.1332	2.25371071	9.1	9.1	9.1
MAX	30.3064	2.25580881	20.75	20.8	20.7
MIN	30.185	2.25442967	20.75	20.8	20.7
MOV	30.185	2.25442967	20.45	20.5	20.4
MUL	30.601	2.25928606	8.2	8.2	8.2
POW	30.1332	2.25371071	39.35	39.4	39.3
RCP	30.3064	2.25580881	20.5	20.5	20.5
RSQ	30.1332	2.25371071	39.7	39.7	39.7
SCS	30.1332	2.25371071	39.65	39.6	39.7
SGE	30.3064	2.25580881	20.85	20.8	20.9
SIN	30.3064	2.25580881	20.5	20.5	20.5
SLT	30.185	2.25442967	20.75	20.8	20.7
SUB	30.4562	2.25661191	8.2	8.2	8.2
XPD	30.1332	2.25371071	39.9	39.9	39.9