

University of Pennsylvania Senior Project:  
**Achieving Real Time Ray Tracing Effects  
Through GPGPU Acceleration**

Matthew J. Vucurevich  
*mjvucure@seas.upenn.edu*

Advisor: Camillo J. Taylor  
*cjtaylor@central.cis.upenn.edu*

## **1 Abstract:**

The computer graphics industry has recently become one of the most influential stimuli for the production of faster and more advanced ASICs. GPU's have made their presence known as performance enhancements in your home computer as well as the primary component of video game consoles. They are recently emerging as a very important part of handheld devices (laptops, cell phones, i-pods) and their usefulness will only continue to grow. One could argue that computers are fast enough as is to run 99% of our non-graphical daily needs, but improvements in the GPU industry have been a driving force leading to improvements in the CPU industry as well. One major reason to upgrade to a more resource heavy OS such as Windows Vista is its ability to support Direct-X 10 graphics. As the demand for the GPU industry grows, research in this area becomes that much more important.

The current direction of GPU hardware has allowed for a certain amount of computational abstraction allowing functionality that is different from the traditional graphics pipeline. This new approach allows for individual programmability in two of its most important stages. Vertex and fragment shaders can now be given unique sets of instructions to execute in order to fully exploit the true level of data parallelism that is available in a GPU. Many languages have been developed to allow customization that is close to assembly language in scope, but with a clearer structure. Nvidia and Microsoft developed the Cg (C for graphics) programming language allowing individual programming of vertex and pixel shaders, and other languages include ARB low level assembly language, OpenGL shading language (GLSL), and DirectX High-Level Shader Language (HLSL). NVidia's CUDA technology (Compute Unified Device Architecture) focuses specifically on the potential programmable GPU shader blocks have to solve complex computational problems from a variety of different applications, and has its own Toolkit to develop in this environment.

The programmability of this newer generation of GPU's will allow complex functionality such as ray tracing image generation algorithm to take advantage of the GPU pipeline despite the fact that its approach to processing elements of the image is completely opposite of the traditional graphics pipeline. For this project, I utilize the programmability of the vertex and fragment shaders of modern GPU's to implement a real time ray tracing effect through the help of programmable GPU hardware. Graphical Processing Units no matter the approach will all share qualities that differ greatly from their CPU counterparts. GPU's can be improved simply by adding more vertex/pixel/shader processors because all graphics pipelines can parallelize most of their calculations. CPU's generally operate on sequential code making the cost/benefit ration of adding extra transistors much higher. With this project I hope to demonstrate the power of parallelism in GPU shader processors as well as produce a deliverable that could be used as an effects enhancement for modern video games.

## **2 Previous Work:**

“Ray Tracing on Programmable Graphics Hardware” from Stanford University provides an excellent blueprint on an approach to Ray Tracing on a GPGPU. This project was conducted during the initial onset of programmable vertex and fragment processors so it runs a simulation on many of its ideas as opposed to an actual implementation. Although their work did not include an implemented version, the theory clearly inspired later implementations of GPU ray tracing.

The most notable implementation of GPU ray tracing comes from Martin Christen’s (of University of Applied Sciences Basel) Master’s Diploma Thesis “Ray Tracing on GPU.” This paper gives a thorough breakdown on how to turn a shader program into a generalized stream processor and provides an actual HLSL GPU Ray Tracing example. While this pure ray tracer is very impressive, it still does not run in real time on even the most up to date graphics cards. As a result, the goal of my project is to take advantage of the normal graphics pipeline to do certain calculations quickly while only saving the most important for the costly ray tracer. My goal is to provide a ray tracing effect that is comparable to real ray tracing in quality but can be used in a real time environment.

### **3 Technical Approach:**

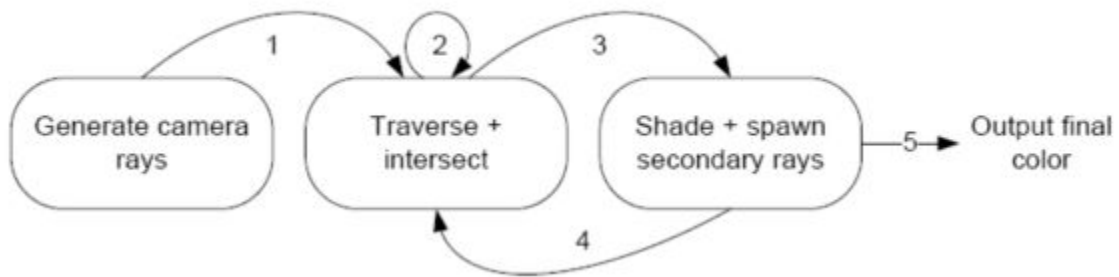
#### **3.1 GPGPU Ray Tracing**

In order to achieve the final goal of encapsulating the visual effects of ray tracing in a real time application, it was necessary to minimize calculations wherever possible. I will begin this section with a breakdown of how to properly implement a full out ray tracer on the GPU and follow with my project description of the shortcuts I used to achieve renderings that run in real time. Full GPGPU ray tracing closely follows the research of the pioneers listed in my previous work’s section. For a complete and detailed report on ray tracing using DirectX HLSL, see Martin Christen’s paper “Ray Tracing on GPU.”

In general, when programming with GPU, the shader programs have a single main method and are compiled into one executable set. The compiled shader suite serves a single function; to render the scene. We do not explicitly call this shader program like a normal function. It simply is run automatically when on a stream of input data when we decide to render the scene. When we consider a ray tracing application, we need to abstract the use of shader programs into generic stream processors. The following definitions will come in handy when describing the shader program’s new use:

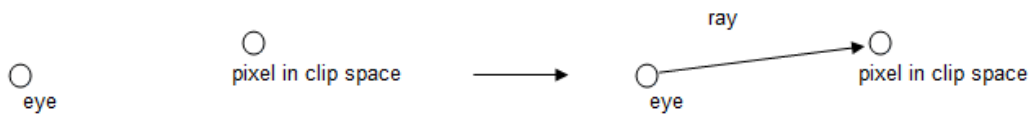
*kernel*: a small program that operates on a stream (i.e. the shader)  
*stream*: set of data of the same type that is read-only or write-only

The following is an abstracted diagram of Martin Christen’s approach that summarizes the operation of kernels in GPU ray tracing: (each circle represents a kernel or a shader program)



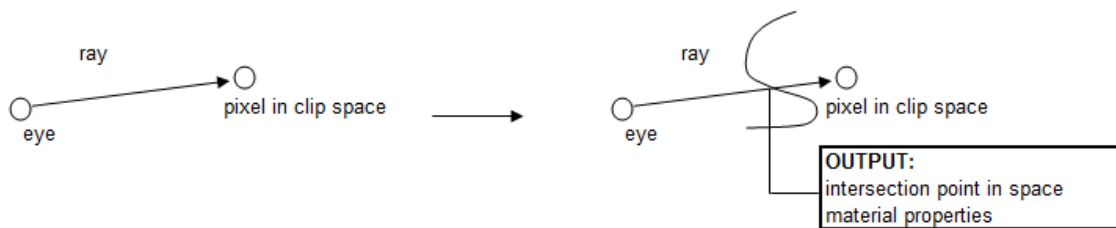
### 3.1.1 Ray Generator

The Ray Generator is a shader that generates a ray for each pixel that will be traversed in the following kernels. Output is stored in a texture. Each pixel in the texture corresponds to a pixel on screen. The fragment processor will operate on a pixel on the screen so by storing into a texture matching this coordinate, future passes in the other fragment shaders will be able to reference the correct information.



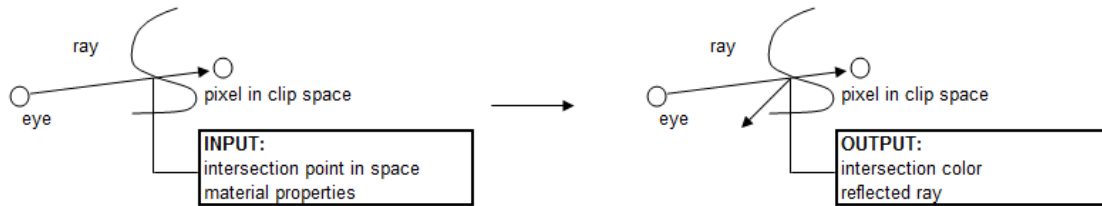
### 3.1.2 Ray Traverse + Intersect

The Traverse and Intersect program takes in the information of the world geometries along with the rays we are testing, and generates a point of intersection along with the material that was intersected. This stage will incorporate any acceleration structures used to reduce calculations and may actually consist of multiple shaders as a result. Output is once again stored in a texture.



### 3.1.3 Shade + Spawn Secondary Rays

The Shade and Spawn secondary rays program does all the lighting calculations necessary to determine the final color, and if necessary generates additional rays that need to be traversed. The shader returns a color along with a new ray for each pixel. The color is passed into the “traverse and intersect” section so that future traversals can blend with the current pixel.



## 3.2 Modified GPGPU Ray Tracing

I will now go into detail on my approach at handling the following effects that have unique properties in ray tracing: lighting, shadowing, and reflections.

### 3.2.1 Lighting

In this section I will deal with optimizing lighting in a GLSL program to simulate ray tracing properties. I begin with lighting because its implementation is both the simplest and farthest removed from real ray tracing. This is a direct result of the impressive capabilities that a shader program gives to us.

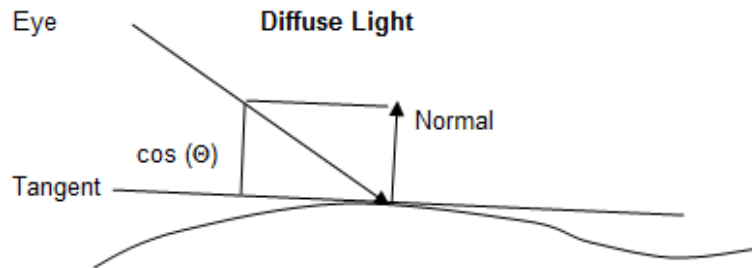
Shader programs allow for lightning quick per pixel lighting through the use of vertex and fragment shaders. The calculations involved are the same as the ones involved in ray tracing. It might seem baffling at first that every geometry-rendered pixel on the screen can have its own calculations done on it and still run in real time, especially when we consider how slow ray tracing is as a result of doing calculations in every pixel. The difference in run time between the lighting posed here and ray tracing comes from obtaining information about the geometry in the scene that the pixel represents. Ray tracing involves casting a ray through the scene in order to find out what it intersects. The GPU hardware is set up so that each pixel processed in a fragment shader automatically knows which vertex it came from. This allows computation at each point to be simple. Combined with the parallel nature of fragment processors, lighting calculations on complex scenes can be done in real time.

Specular light, ambient light, and diffuse light are the three components I use to make up my lighting equation. Specular lighting is the brightest light that shows a reflection from the source. Ambient lighting is the strength of light in the absence of direct light. Diffuse light is the normal color of the light source.

Let us review the basic diffuse and specular shading equations:

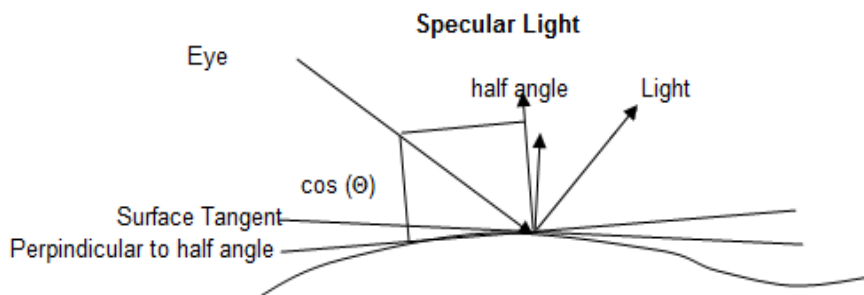
Diffuse shading is based on the principal that more light hits the surface of an object where the angle between the surface normal and the vector from the intersection to the light is least. As a result we get basic shading from the dot product of the light direction with the normal of the surface. This is quite easy in a shader program because the light position, point position, and surface normal are all known.

The dot product of the eye and normal are used as shown below:



Specular lighting is a function of light direction, eye vector direction, and the eye position of the camera. As a result, the only additional piece of information needed is the eye position. We calculate specular lighting based on the half vector of the light direction and eye direction for a given pixel. The half vector could be found using simple vector addition (light direction + eye direction) but GLSL includes a built in function that calculates the half vector based on the light position combined with the implicitly stored eye and vertex positions. As a result per-pixel smooth lighting can be calculated extremely quickly in less than 30 lines of shader code.

The dot product of the eye and half angle vector are used as shown below:



### 3.2.2 Shadowing

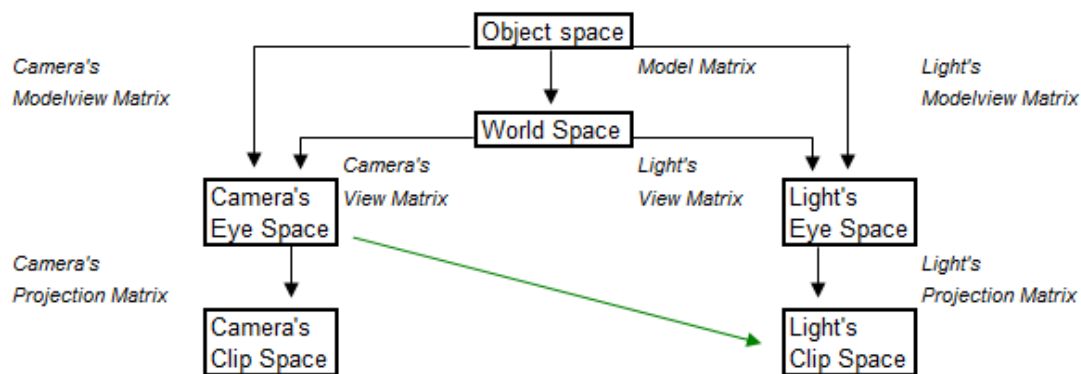
Shadowing is a fairly costly step within ray tracing because you have to trace an additional ray for each pixel back to the light source. In this section I will discuss some of the key elements of shadowing that allow us to utilize a faster approach for a similar effect.

There are two important concepts to keep in mind when speeding up shadowing. The first is that every ray is being traced back to the same point. The second is that we only care about whether or not something is blocking the path to the light. These facts have lead to the concept of shadow-mapping.

Shadow mapping was created by Lance Williams in 1978. The procedure allows us to shadow the world with two passes of rendering. I will first discuss the concept and follow with a review of its constraints compared to normal ray tracing.

Shadow mapping begins with our ability to quickly calculate what is visible from a given eye view. This ability is incorporated in graphics hardware and is called Z-buffering. In addition to simply determining what we see, we also can determine its depth. The first step in shadow mapping involves drawing the scene from the light's point of view and storing the depth buffer of the resulting image into a texture. This texture can be created quickly and in real time for reasonable scenes so shadow mapping will remain within the same time domain even if the light moves between renderings. One advantage of the shadow map is that you can render only the models in the scene you want to shadow in this stage (for example: Guild Wars will render only your main character's shadow if you turn down shadowing).

The next step is to draw the scene from the camera viewpoint and apply the shadow map. The diagram below gives us a roadmap of relationship between camera and light.



We can see from the diagram above the general path needed to be taken to convert a point in the camera's eye space to the light's clip space. Simply multiply by the inverse of the camera's modelview matrix to enter the object space. From here multiply by the lights projection matrix along with the light's modelview matrix. The result should be a direct transform from camera eye space to light clip space. In actual application the matrix values will be in the range [-1,1] but texture maps expect a [0,1] range so a simple "bias" matrix must also be multiplied in the mix.

The bias matrix looks as follows:

```
// Bias matrix to map [-1,1] to [0,1] ratio after clip space
const float mBias[] = {0.5, 0.0, 0.0, 0.0,
                      0.0, 0.5, 0.0, 0.0,
                      0.0, 0.0, 0.5, 0.0,
                      0.5, 0.5, 0.5, 1.0};
```

Having this transformation matrix allows us to take a point seen by the camera's eye and determine what depth the light's clip space mapped to that point. We can then check to see if the distance from the point seen by the camera's eye to the light is the same as the depth map. If the depth map marked a depth that is smaller, this means the light sees an occluding object when it looks at the same point, and we must shadow it.

The following snippet is the code I use for setting up the texture application:

```

glMatrixMode(GL_TEXTURE);
// The bias matrix to convert to a 0 to 1 ratio
glLoadMatrixf(mBias);
glMultMatrixf(g_mProjection); //light's projection matrix
glMultMatrixf(g_mModelView); //light's modelview matrix
glMultMatrixf(g_mCameraInverse); //inverse modelview matrix

```

Shadow mapping has some obvious speed advantages but it consequently has limitations. The first limiting factor is how large a viewing frustum you use for the light's view on the initial depth map. This will determine how much of the world can be shadowed. A limiting factor on quality will be what texture resolution you store the initial depth map in relation to the viewing frustum you use. More pixels per area of view frustum gives improved quality at the cost of speed.

One way around these quality issues that does not affect speed is to blend nearby pixels. While testing a value on the texture map, simply grab nearby values as well and blend them together. Blending allows us to simulate soft lighting in the same way we would attain it in normal ray tracing using a Monte-Carlo like approach. When checking nearby points you can assume the light is a sphere and have a predefined scaling based on this assumption. The problem with this approach is that the blending becomes less accurate around the edges and thus gives a less crisp picture than actual ray tracing.

### 3.2.3 Reflections

Reflections are the most complicated of the three primary effects in ray tracing and the only one that I am forced to use actual ray tracing. The trick in improving run time with reflections is to only ray trace when absolutely needed (and only on the first recursion). I will discuss three ways of dealing with reflection, all of which are used in my program: faking reflections, simulating reflections, and ray tracing reflections.

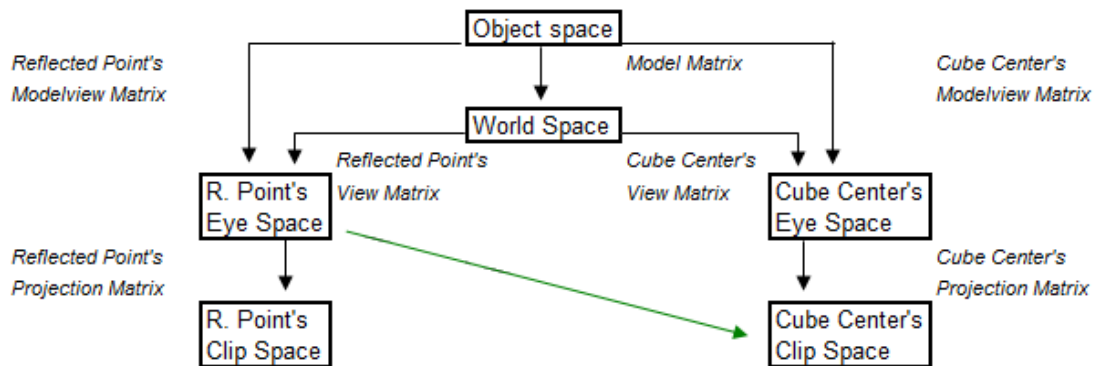
Even faking reflections is a consuming task that requires six different scene renderings to do fully. The technique I use to fake reflections involves creating a cube map around an object by rendering the scene on six faces of a box that surround the reflective object. Now, instead of ray tracing a reflected ray into the scene, simply trace the ray into the cube map we created. There is no need to store vertices or material properties in this situation, just intersect a ray with a 2D image to get the color. This technique would give perfect reflections if the box's faces were infinitely far from the center of the box and is used in many applications today. Unfortunately error is introduced as the depth of the objects in the faces become closer and closer to the reflecting object. As a result I set the cube to be fairly large and cull away the objects that are inside of it. I then keep track of the culled objects to properly reflect them using a more accurate method.

This led me to a preliminary solution of simulating the reflections. The idea behind simulating the reflections is strongly influenced by the shadow mapping explained above. Starting with an initial cube map, let us assume that our reflecting object culled another

object in the process of making the cube map. This object now needs to be added back into the reflection.

I start this process by taking a snapshot of the object from the center of the cube map. This is similar to taking the depth map from the light's clip space in shadow mapping. The simulation of the reflection involves a process that mirrors the transformation from camera eye space to light clip space used in shadow mapping. We are now converting an eye space from the point on the reflecting object to the clip space of the center of the cube map.

Thus the transformation looks as follows:



This will work assuming there are no objects between the reflecting object and the target object. Additionally we need to make sure the object is entirely within our initial cube center's view frustum. These are fair assumptions since the objects that are rendered with this reflection simulation strategy are a small subset of all the objects reflected.

The final option for simulating reflections is a complete ray trace. This requires taking a list of potential objects to intersect with, passing it into a shader program, and then checking for intersection. From the intersection I would use the lighting equations to determine color. Because of the slowness of ray tracing, I limit the use of actual ray tracing to when the clipped object is known to not be a complex geometry (i.e. does not have many vertices).

I combine these three methods to create a balance of speed and accuracy. For a final example of how to combine the tools available, let us consider a case of recursive reflection. In this situation I calculate the deep reflections using the quick and easy method at a low texture resolution. The first bounce of the reflection needs to be the most accurate as it is the easiest to see and for this I would either simulate a reflection or ray trace a reflection. In the case of a complex object I would simulate a reflection but in the instance of a simple object I would ray trace.

### 3 Conclusion:

In conclusion I was able to set up a system that takes advantage of the power of GPU shaders in order to create ray tracing effects in real time. I created a toolbox that allowed me to render both simulated ray tracing effects as well as actual ray tracing effects. With this toolbox I was able to create a balance of speed and quality that allowed my final program to render some stunning visual effects in real time.

The overall project was successful at achieving an excellent rendering but looking back at the history of my project goals, a real time ray tracer of complex images is still out of reach. The project experience involved sticking to the main goal of a real time rendering and adjusting my use of ray tracing to achieve the goal. While the end product as very little ray tracing in its computations, it is filled with ray casting and the fundamental theory behind ray tracing. In creating shortcuts to avoid actual ray tracing wherever possible, I discovered the true power of the graphics pipeline and found its structure to be more compatible to the theory of ray tracing than I originally had thought.

## Works Cited:

Purcell, Timothy J., & Buck, Ian, & Mark, William R., & Hanrahan, Pat, (2002) Ray

Tracing on Programmable Graphics Hardware. Stanford University.

This paper was published at Stanford University and can be found in the Computer Graphics Lab's archive of technical publications. While the article is not relatively current, its influence can clearly be seen in future papers dealing with GPGPU ray tracing. The paper was written as part of a research project and provides an objective view outlining the basic creating of a GPU based ray tracer. The report is an original work of research and was a breakthrough in field when published. The document served as the basic outline of my implementation as I began my work on the project. I followed the theories presented in work when creating an initial ray tracer but found I had to drastically change most of the concepts to eventually achieve a real time ray tracing effect.

Christen, Martin, (2005) Master's Diploma Thesis: Ray Tracing GPU. University of Applied Sciences Basel (FHBB).

This paper was published by the University of Applied Sciences Basel as Martin Christen's Diploma Thesis for his Master's Degree. This is a research paper that outlines his implementation of a ray tracer in Direct X's HLSL shader language. The paper itself is out of date enough to where he was forced to write in HLSL because his model GLSL attempt did not have enough driver support for 2005 graphics cards. Martin's report was a very important driving force in my project. My initial goal was to construct a ray tracer in the exact way that Martin did. Reading Martins report showed me that even with a great acceleration structure, ray tracing is so computationally intense that a full out ray tracer at a reasonable frame rate is not yet possible (needless to say Martin's frame rates were very impressive). As a result of this report I adopted my own approach to GPU ray tracing and began planning my methodology for simulating ray tracing wherever possible.

Medvedev Alexander, & Budankov, Kirill. (2004) NVIDIA GeForce 6800 Ultra. from

<http://www.digit-life.com/articles2/gffx/nv40-part1-a.html>

This article was published by the Digit-Life website. This website is renowned for its collection of articles on statistical properties on the most up to date digital devices. The article gives statistical information on a recent NVidia graphics card and additionally provides a flow map of its hardware structure. The information is based off of released data from N'Vidia.

Understanding the hardware I was using for this project was a necessary step in determining what optimizations I could make. This straightforward article is an excellent technical report on NVidia's graphics cards.

Owens, John D., & Luebke, David, & Govindaraju, Naga, & Harris, Mark, & Krüger, Jens, & Lefohn, Aaron E., & Purcell, Timothy J., (2005) A Survey of General-Purpose Computation on Graphics Hardware. Retrieved November 7, 2006 from Publications at the Institute of Data Analysis and Visualization.

This paper was published in 2005 by the European Association for Computer Graphics. Eurographics is responsible for many well known publications in the field and hosts the primary graphics events in Europe. The paper provides original ideas on using the graphics processor as a general purpose processor. This is closely related to using the GPU for ray tracing since this implementation requires an abstraction of shader programs into general purpose kernels. The paper played an influential role in the project as it gives the theory behind using the GPU for functionality that is different from the normal graphics pipeline. This paper shows us how broad a potential the GPU has for general purpose implementations.

Nuno Moutinho de Carvalho, Goncalo, & Gill, Tony, & Parisi, Toni., (2004) X3D Programmable Shaders. Retrieved November 10, 2006 from Publications at the Institute of Data Analysis and Visualization.

This paper's publication was sponsored by SIGGRAPH. SIGGRAPH is the forerunner in all research related to computer graphics. The SIGGRAPH is the largest in the graphics world and companies and enthusiasts alike attend with interest. The paper's publication is fairly out of date but it explains concepts about shader processors that are up to date in today's graphics cards. The paper outlines how shaders work and gives examples of how they are used. In order to be able to fully exploit the power of shader programs it was necessary to understand how they work. This paper provided me necessary information to optimize the design for my project and have it run most efficiently.

Shirley, Peter, & Jensen, Henrik W., (2003) Monte Carlo Ray Tracing. University of California, San Diego, 11-118.

This work was an instructional course book used for a class at the 2003 SIGGRAPH convention. The book was designed for a class to teach the intricacies of Monte Carlo Ray Tracing and based mostly on the teachings of Peter Shirley. The chapters I read as research for my project are titled: Fundamentals of Monte Carlo Integration, Direct Lighting via Monte Carlo Integration, Stratified Sampling of 2-Manifolds, and Monte Carlo Path Tracing. The book was extremely helpful in explaining the concept of statistical sampling combined with ray tracing that is the Monte Carlo method. I used many of the concepts I learned in this book to create a model for soft lighting when working on the lighting simulation section of my project.

Paul's Projects. Shadow Mapping Tutorial. from

<http://www.paulsprojects.net/tutorials/smt/smt.html>

This article is a tutorial on implementing Shadow Mapping in Open GL. The article is not from a distinguished site, but since it simply relates factual information describing an algorithm, all that is important is its clarity. Although I implement my Shadow Mapping through GLSL, the tutorial itself gives an excellent description on how to make a shadow map. The diagrams I use to show the transformation from eye space to clip space in my report were remodeled from the diagram on this website.