A QUALITATIVE COMPARISON OF CODING LANGUAGES USED FOR IMAGE SYNTHESIS

A Thesis

by

CHRISTOPHER STEVEN POTTER

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Dr. Ergun Akleman
Committee Members,	Dr. Ann McNamara
	Dr. John Keyser
Head of Department,	Tim McLaughlin

May 2014

Major Subject: Visualization

ABSTRACT

In this thesis four different computer programming languages, C++, Python, Processing and Pixar's Renderman, were used to realize four different rendering programs. The goal was to identify the main challenges in implementation with each language and qualitatively evaluate each program once completed. A set of "ray tracing milestones" were established so that each language can address the challenges unique to that language. These milestones are related to the image synthesis process and include: preliminary preparations, direction illumination, distributed ray tracing and indirect illumination.

DEDICATION

Dedicating this Thesis to special people.

ACKNOWLEDGEMENTS

This is where I acknowledge important people. Ergun, Ann, John, Parents, Stefanie, VizLab.

NOMENCLATURE

GUI	Graphical User Interface
IDE	Integrated Developing Environment
MIT	Massachusetts Institute of Technology
PIL	Python Imaging Library
RIB	RenderMan©Interface Bytestream
RSL	RenderMan©Shading Language

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Image synthesis is the process of generating images. More specifically, we will be considering it as the process of generating images from virtual 3D scenes on a computer. Computer graphics studios like Pixar and Dreamworks rely on their propriety image synthesis pipeline to create photorealistic animations for their movies. The final product that is released to the public is a direct result of the image synthesis process. Implementing a ray tracer is a very daunting task, but can be worth the effort for artists who are looking to make a career in professional computer graphics lighting. Stepping through the tasks required to create images from a ray tracer can be invaluable to understanding how to optimize render time while still creating high quality images.

Writing a rendering program can be very intimidating for an artist. Rendering programs consist of two fundamental parts, theory and implementation. The complex vector/matrix math theory to compute intersections within the 3D scene can be difficult to understand by itself. Coupling those theories with proper computer language jargon can multiply the difficulty. In addition to remembering the linear algebra, artists need to worry about code syntax and segmentation faults (generally an attempt to access memory that the CPU cannot physically address), which can be demoralizing to even the most experienced computer scientist. Rather than gaining experience/experimentation in the mathematics of image synthesis, artists can become distracted from the task at hand by focusing on the logistics of implementing unnecessary code.

Most students, when implementing a ray tracer, are siphoned into implementing their image synthesis programs in C++, perhaps because of tradition or because of

introduction/familiarity from previous classwork. C++ can be a very tricky language, with many aspects of preparation needed before one can start programming at all. Though C++ can be very fast, which is needed for *professional* image synthesis, the process of *learning* ray tracing theory does not necessarily benefit from a high speed processing program. This thesis has attempted to provide guidelines to introduce programming languages and techniques related to rendering so artists can spend less time on the implementation aspects of the process and more time on the theoretical experience of image synthesis, especially since as an artist, the most beautiful code is not your end goal. In addition to the C++ language, Processing, Python and RenderMan were evaluated.

The qualitative results were generated from the researcher himself. My background in scripting languages was very informal before my admittance into the Visualization Department at Texas A&M. I hold a BFA from the Rochester Institute of Technology, with a minor in Computer Science. My experience with computer languages, however, was not at a point where I felt comfortable with writing simple scripts, nevermind an entire image synthesis program. Firsthand results recorded in this thesis are notes and "hiccups" encountered through each implementation step of the ray tracing process. My hope is that these results will show student and teacher alike the complications with each language and help lead them through overcoming those obstacles in order to produce great art.

2. BACKGROUND: THE IMAGE SYNTHESIS PROCESS

3D rendering can be described as the process of creating 2D images from 3D geometric shapes with the use of photorealistic/nonphotorealistic effects on a computer. One method of 3D rendering is based around the process of ray casting and ray tracing. For this thesis, ray casting and ray tracing will be described as the process of generating an image by tracing the path of virtual rays from an eye/camera source through pixels in an image plane and simulating the effects of its encounters with virtual objects.(See Figure 2.1) Each will be differentiated further by their illumination models, which is the shading algorithms used to interact with light rays. Four milestones have been established for the image synthesis process and out of those, three are associated directly with image synthesis theory: direct illumination, distributed ray tracing, and indirect illumination.

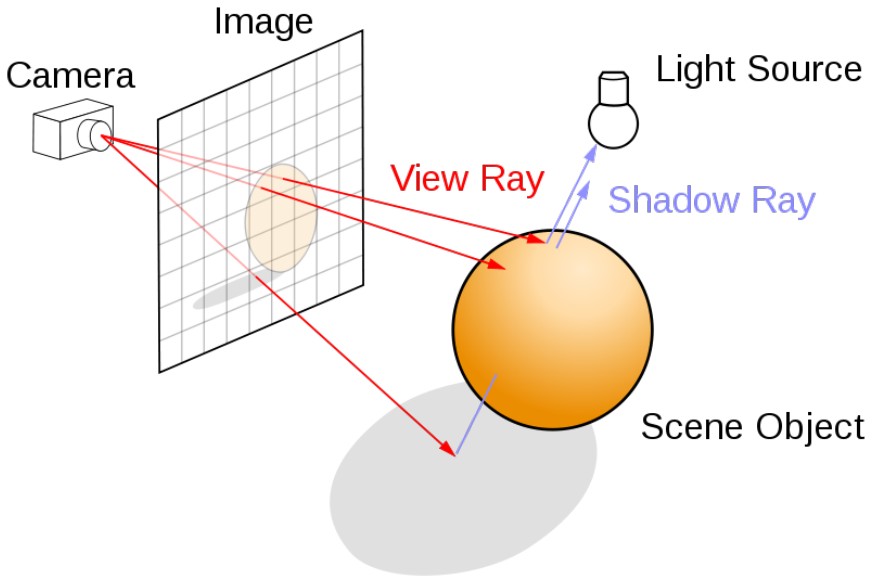

Figure 2.1: Ray Tracing Diagram[9]

2.1 Direct Illumination: Ray Casting

For this thesis, direct illumination will encapsulate the ray casting process. Direct illumination will be referred to as only illumination from a virtual light in a 3D scene contributing towards a 3D objects final color and shading, along with that object's material attributes. This also includes shadowing information, which is just an object obstructing the ray path from a light to the hit point.

Ray Casting, a term first introduced by Scott Roth in 1982, is the process of casting rays into a 3D scene, finding the closest intersection from the eye/camera through boolean operations, and returning the color value of that object[18]. By emulating the different properties of light based off of the cosine of the normal vector and the normalized vector from the high point of the object to a light, semi-realistic results were produced. Other more cartoon-ish effects were produced using similar methods as well, most notably Gooch and Gooch shading.

We will be using the basic Phong model of shading, or a variation similar, which has been altered slightly by Jim Blinn and will be discussed fully in following sections. The equation is as follows:

$$color_{output} = k_d O_d I_a + k_d O_d I_d[\hat{L_m} \cdot \hat{N}] + k_s O_s[\hat{R_m} \cdot \hat{V}]^n I_d[10] \tag{2.1}$$

where k_d and k_s are the diffuse and specular reflection coefficients, I_a is the ambient light color, I_d is the diffuse light color, O_d is the diffuse color of the object, L_m is the direction vector from point P towards the light, N is the surface normal for the surface at P, O_s is the specular color of the object, R_m is the direction that a perfectly reflected ray of light would take from P, and n is the specular exponent, or shininess. We will be referring to each part of this equation in the following sections.

2.1.1 Lambert Shading

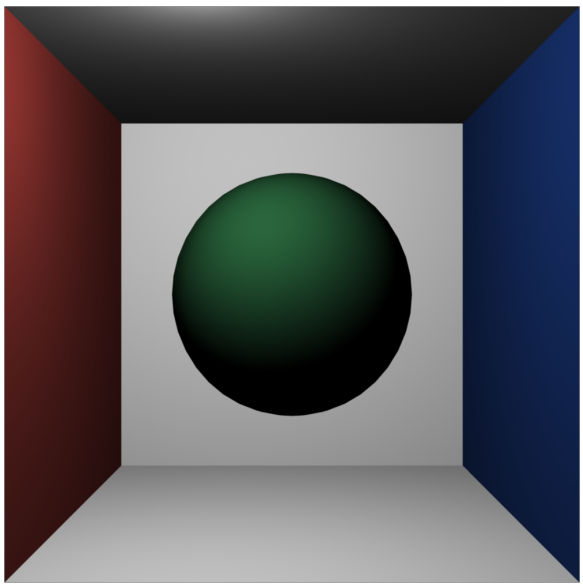

Figure 2.2: An example of Lambert Shading created in Autodesk Maya 2010©

The Lambertian model for reflectance is the diffuse component of a surface's material, or the diffuse reflection of a material's properties. Diffuse objects are generally thought of as unfinished wood, or anything that does not reflect light off of its surface. This effect can be see in Figure 2.2. This model follows Lambert's cosine law for optics, which says that the radian intensity or luminous intensity observed from an ideal diffusely reflecting surface or ideal diffuse radiator is directly proportional to the cosine of the angle θ between the observers line of sign and the surface normal. Attributing this to computer graphics we get the equation:

$$color_{output} = \hat{L_m} \cdot \hat{N} C I_L \tag{2.2}$$

where $\hat{L_m}$ is the normalized direction of the light-direction vector, $\hat{N}$ is the

normalized normal vector, C is the color, and I_L is the intensity of the incoming light. Since $L \cdot N = |N||L| \cos\theta$, and if we were to make the lengths of $|N||L| = 1$ by normalizing them, equation 2.2 satisfies the properties for Lambertian shading by making $L \cdot N = \cos(\theta)$. As we can see, Equation 2.2 is actually the term represented as $k_d O_d I_d[\hat{L_m} \cdot \hat{N}]$ in Equation 2.1, where $C = O_d$ and $I_L = I_d$.

2.1.2 Gouraud Shading

One of the first shading paradigms introduced to the computer graphics community was a vertex interpolation shading created by Henrik Gouraud at the University of Utah[8]. Gouraud's algorithm successfully perceived the Lambertian Shading effect described in Section 2.1.1. Gouraud's approach saved intensity values at each vertex as a weighted average of the normals of the surrounding polygons in the object's mesh. When a hit on an object was achieved, a weighted sum of the point's color would be returned depending on the intensity values at each closest vertex. This effect is demonstrated in Figure 2.3.

While this method works well, it is also dependent on the density of the object's mesh, since each vertex intensity is basically an average of the surrounding normals. This technique also benefits from the relatively simple calculations needed per hit point, because generally speaking a polygon is made up of 3 or 4 vertexes at maximum. Instead of calculating Equation 2.1 at each pixel of intersection within the scene, we only need interpolate between precalculated vertex intensities. However, since the smooth perception of shading is based off of interpolation, which in effect is characterized by the density of object meshes, high localized specular highlights will not be rendered correctly. Also, if a highlight lies in the middle of a polygon, but does not spread to the polygon's vertex, it will not be apparent in the render. If the highlight appears directly on a vertex, while it will be rendered correctly for

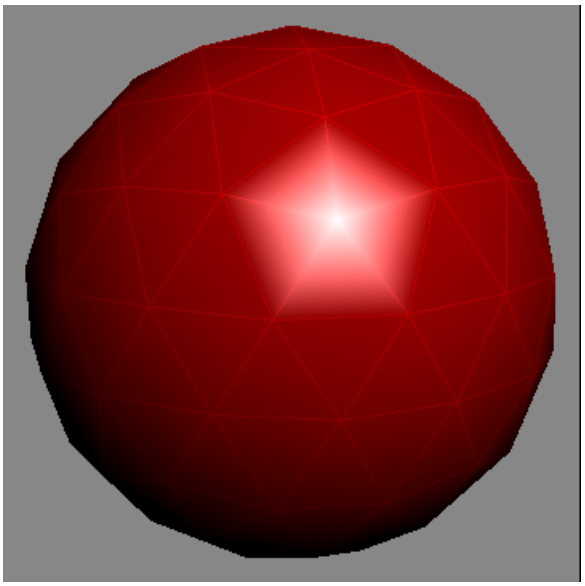

Figure 2.3: An example of the deficit of Gouraud shading being apparent in the highlight in a lowpoly model.[11]

that vertex, on neighboring polygons, it will be rendered incorrectly. An example of the deficit for Gouraud's shading can be see in Figure 2.3. For this reason, Equation 2.1 was introduced.

2.1.3 Phong Shading

Another aspect of an object's illumination model is the specular component of an object. Specularity is the visual appearance of specular reflections. This represents an objects "shininess". One strategy to display this specular highlight was introduced by Bui Tuong Phong [17]. Whereas Phong's predecessors interpolated across surface patches [8], as stated in Section 2.1.2, Phong interpolated surface normals and evaluates a lighting model at each pixel. He also added a specular component to the lighting model to produce highlights.[12]

Phong's specular component from his paper *Illumination for Computer-Generated Images* introduces the equation:

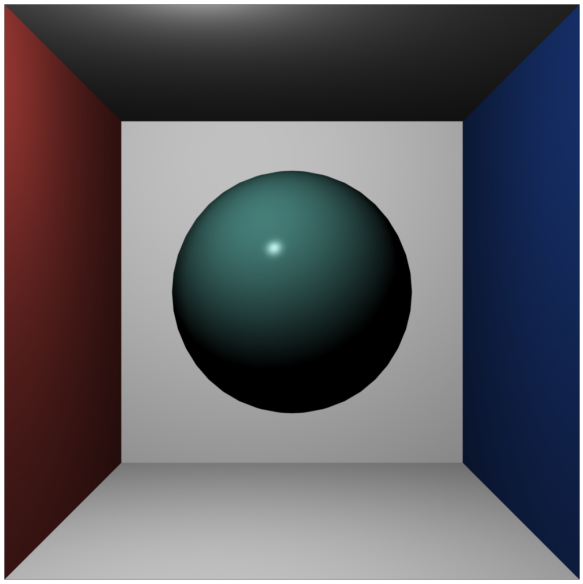

Figure 2.4: Phong Shading created in Autodesk Maya©2010

$$color = C_p[\cos i + d] + W(i)[\cos s]^n \tag{2.3}$$

where C_p is the reflection coefficient of the object at point P, i is the incident angle, d is the environmental diffuse reflection coefficient, $W(i)$ is a function which gives the ratio of the specular reflected light and the incident light as a function of the incident angle i, s is the angle between the direction of the reflected light and the light of sight.

The important aspect of Equation 2.3 is the term $[\cos(s)]^n$. Since the cosine of the angle between the direction of the reflected light, or R_m in Equation 2.1, and the line of sight, V, is equal to $R_m \cdot V$ as long as each vector is of unit length, we only need to know what R_m is.

To calculate R_m we have this equation:

$$R_m = 2[L_m \cdot N]N - L_m \tag{2.4}$$

where L_m is the direction vector from P towards the light, and N is the surface normal at P. The effect of Phong shading can be seen in Figure 2.4.

2.1.4 Blinn Shading

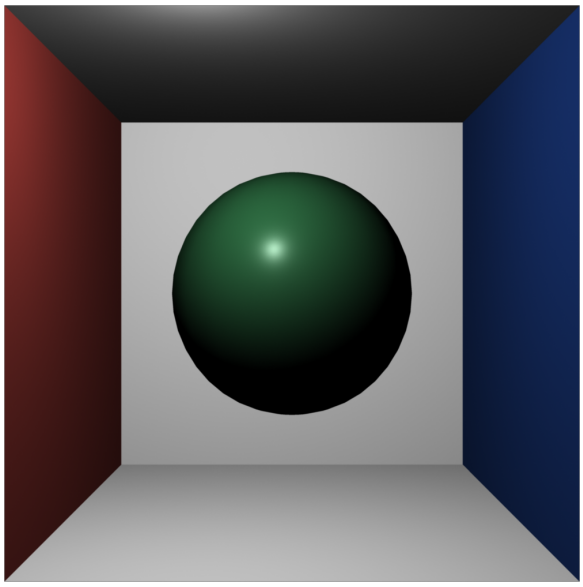

Figure 2.5: Blinn Shading created in Autodesk Maya 2010©

In Jim Blinn's paper *Models of Light Reflection for Computer Synthesized Pictures* he introduces the equation[1]:

$$color = p_a + max(0, N \cdot L)p_d + (N \cdot H)^n p_s \tag{2.5}$$

which is most similar to Equation 2.1, because this introduces an ambient term p_a, diffuse term p_d and the specular term p_s. Most importantly for Blinn's equation is the inclusion of H. Instead of using Phongs R_m from Equation 2.4, instead he introduces a new half-vector called H. Blinn describes

> "If the surface was a perfect mirror, light would only reach the eye if the surface normal, N, pointed halfway between the source direction, L, and the eye direction, E. We will name this direction of maximum hilights H..."

Relating this to the variables in this paper:

$$H = \frac{\hat{L} + \hat{V}}{len(L + V)} \tag{2.6}$$

Equation 2.6 then replaces $R \cdot V$ in Equation 2.1. Blinn's model produces more accurate models of empirically determined bidirectional reflectance distribution functions for many different types of surfaces[16]. While this equation produces better results, it also introduces a square root math function when determining $len(L+V)$. For this reason, Blinn's model has been considered slower. It will not be used in this thesis, but helps to show where the origins of Equation 2.1 come from.

2.1.5 Gooch and Gooch Shading

Ever since the introduction of Phong's model for photo-realistic rendering of geometric objects, there has been a trend towards non-photorealistic rendering(NPR). Most notably in this field is the work of Amy and Bruce Gooch, et al. Gooch and Gooch argues that "Phong shaded 3D imagery does not provide geometric information of the same richness as human-drawn technical illustration[6]." NPR techniques are referred to as computer graphics algorithms that imitate NPR techniques such as painting or pen-and-ink[6]. Gooch, et al., introduced a generalization of the classic computer graphics equation from Equation 2.2 by playing with the value of $L_m \cdot N$. Their equations is as follows:

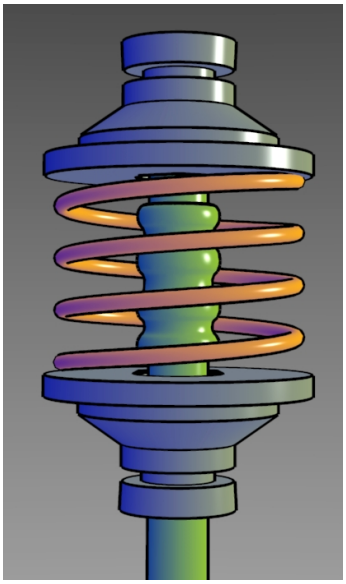

Figure 2.6: Gooch Shading Example[6]

$$color = \left(\frac{1 + (\hat{L_m} \cdot \hat{N})}{2} \right) k_{cool} + \left(1 - \frac{1 + (\hat{L_m} \cdot \hat{N})}{2} \right) k_{warm} \tag{2.7}$$

where k_{cool} and k_{warm} are two color values to interpolate between. In Equation 2.7, they use the large fractions to remap the values of $\hat{L_m} \cdot \hat{N}$ between zero and one, causing the colors to gracefully blend from k_{cool} to k_{warm}. As can be seen in Figure 2.6 the Gooch shading algorithm also includes outlines around each shape. Gooch referred to creating this effect from the paper *Real-Time Nonphotorealistic Rendering*[13] but for this thesis the outline can be determined from when the values of $\hat{L_m} \cdot \hat{N}$, after being remapped between 0 and 1, are between a certain threshold and zero. While this is not the best way to generate outlines around 3D shapes, it accomplishes some semi-reliable/believeable results and was therefore utilized in this thesis.

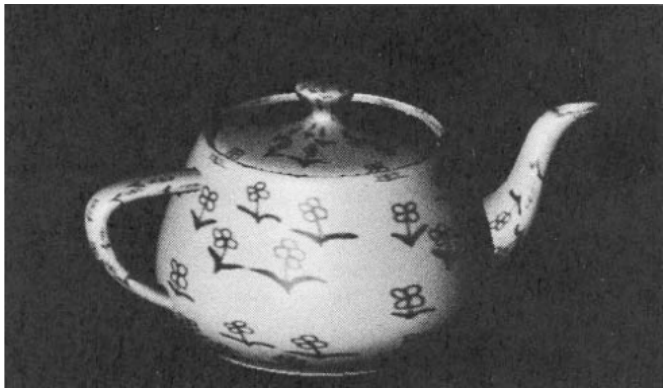

Figure 2.7: An example of texture mapping to the surface of a teapot. [3]

2.1.6 Texture Mapping and UV Coordinates

Until now we've focused on explaining basic color appropriation with solid colors represented as RGB, but in computer graphics is is also possible to map photographs to the surface of 3D objects. This idea was first introduced by Ed Catmull in 1974, where he first introduced a method of representing 3d curved patches as opposed to conventional polygons. He also presented a method to "map" photographs to the surfaces of these patches[5]. Jim Blinn extended this to include environmental mapping based off of reflections from the surface of objects[3]. As can be scene in Figure 2.7.

2.1.6.1 Bump/Normal Mapping

In addition to an extension of Catmull's texture mapping, Blinn also introduced a technique that could simulate a rough or textured surface on a 3D shape[3]. The technique he used for this was called bump mapping. Blinn discovered that by altering the Normal at each point of intersection on the surface of a 3D shape, simulations of wrinkled surfaces can be generated, as can be seen in Figure 2.8. The normal at each point can be mapped to a image texture or altered via a noise

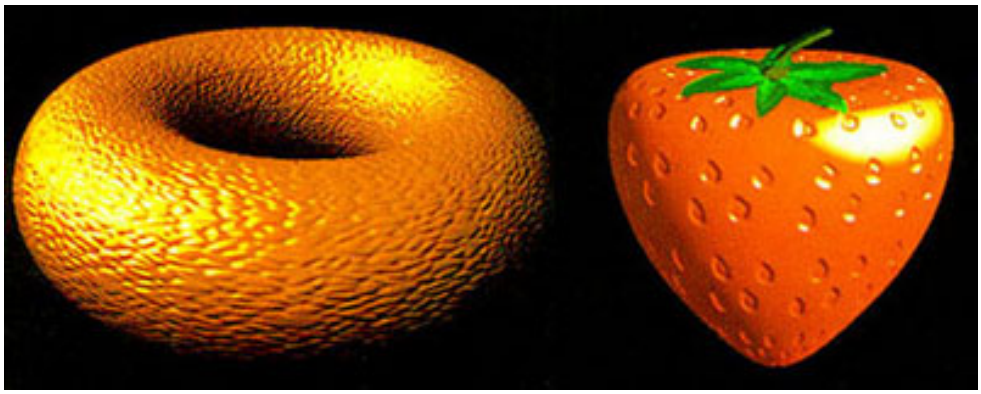

Figure 2.8: An example of bump mapping introduced by Jim Blinn. [2]

generating equation, such as Perlin noise. Blinns wrinkled surface simulations can be seen in Figure 2.8.

2.2 Ray Tracing and Distributed Ray Tracing

To improve upon the Phong specular shading model, a new recursive technique was introduced by Turner Whitted at Bell Laboratories. Turner hypothesized that a "tree" model should be used to calculate accurate, realistic representations of the physical world's reflective process. He based his algorithm on an R reflection ray that represents the reflection of a ray on a perfectly smooth surface, and P represents the transmitted ray through a perfectly smooth surface:

$$\hat{V}' = \frac{\hat{V}}{\hat{V} \cdot \hat{N}} \tag{2.8}$$

$$\hat{R} = \hat{V}' + 2\hat{N} \tag{2.9}$$

$$\hat{P} = k_f(\hat{N} + \hat{V}') - \hat{N} \tag{2.10}$$

where

$$k_f = (k_n^2|\hat{V}'|^2 - |\hat{V}' + \hat{N}'|^2)^{\frac{1}{2}} \tag{2.11}$$

and k_n is the index of refraction for the surface. The equations assume that $\hat{V} \cdot \hat{N}$ is less than zero so the sign of N must also be adjusted so that it points to the side of the surface the intersecting angle is incident from. This offers insight as the the calculation angle from a view angle that is then calculated recursively for each reflective object in a scene. When tracing a ray from the eye point, the intersection at each reflective surface would determine the next surface hit, forming a recursive "tree" formation. Upon achieving this "tree", the following equation was used to calculate the surface color:

$$color_{output} = I_a + k_d \sum j = 1^{j=ls}(\hat{N} \cdot \hat{L_j}) + k_s S + k_t T \tag{2.12}$$

where

S = the intensity of light incident from the $\hat{R}$ direction
k_t = the transmission coefficient
T = the intensity of light from the $\hat{P}$ direction

by keeping k_s and k_t constant Whitted acheived his results, but for ideal cirumstances their values should be mapped to a Fresnel algorithm that relates their values to realistic models of reflection and refraction.

To enhance upon the raytracing process, Cook, Loren and Carpenter introduced the term distributed ray tracing, which is defined as:

> ...The key is that no extra rays are needed beyond those used for oversampling in space. For example, rather than taking multiple time samples as every spacial location, the rays are distributed in time so that rays at different spatial locations are traced at different instants of time.

- Sampling the reflected ray according to the specular distribution function produces gloss (blurred reflection)
- Sampling the transmitted ray produces translucency (blurred transparency).
- Sampling the solid angle of the light sources produces penumbras.
- Sampling the camera lens area produces depth of field.
- Sampling in time produces motion blur.

By introducing randomization, or jittering, this distributed ray tracing effect can be achieved. By jittering the R value of T in 2.12 a glossy effect is produced. The same can be said of the P value from the T variable in 2.12. Distributed ray tracing can be important to a 3d images realism, because it is impossible to achieve the computed perfect reflection/refraction model introduced by Whitted.

2.3 Indirect Illumination/ Radiosity Effects

For this thesis, Indirect illumination will account for any shading or color values not calculated directly from the virual light in a scene. These terms are known as ambient occlusion, global illumination and caustic effects. In the natural world, diffuse reflection is the reflection of light from a surface such that an incident ray is reflected at many angles rather than at just one angle, which is the case of specular reflection. In the natural world, an object will base it's final color off of not only the light shining at it, but also from the color of the objects in closest proximity to it. This effect can be seen in Figure 2.9 within the shadow of the racquetball. Each of these surfaces, a racquetball and a piece of paper, are as close to lambertian surfaces that can be found in the natural world. In the figure, the diffuse reflection is most prominent in the shadow of the racquetball. It may seem like a trick on the eyes, but

the shadow has a purple tint to it because of the diffusely reflected light rays coming from the racquetball. Theoretically, if an object is in illuminated in a room with bright red walls, the color of the object will have a red tint because of the diffuse reflection of the red from the walls contributing to the overall color of it's surface. It is almost impossible to determine all the different contributors to an objects final color in the natural world, because of the infinite amount of light rays absorbed and reflected by an objects surface.

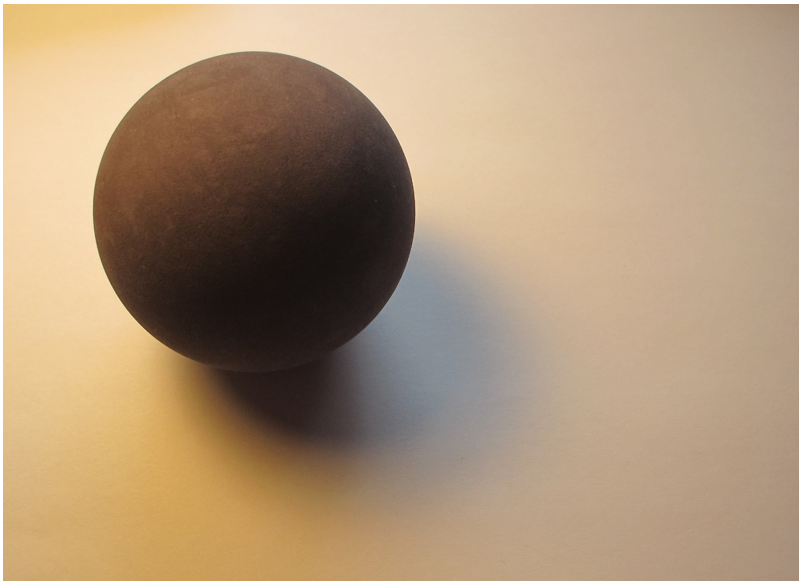

Figure 2.9: Real World Indirect Illumination/Radiosity Effect with a racquetball on a piece of paper.

To emulate this effect in a 3d environment, some new terms were introduced by Torrance, Greenberg et.al [7]. They determined a model of light interaction between diffuse surfaces based off of methods used in thermal engineering. To simplify their method this thesis will use a method inspired by their work. The process to calculate ambient occulsion, global illumination and caustic effects are fundamentally all the

same. At each point in a scene, a specific amount of sample rays will be cast at each intersection point. The average of the resulting color information from each sampling ray set will determine that pixels final color. Ambient occulsion is the simplest. Ambient occlusion informs the proximity of objects with other objects. It is fundamentally a black and while image that will be darker in pixels where images are closer together and light in places where images are further apart. To calculate this set, one only needs to determine if another object is hit but a ray in the set or not. For instance, if 255 rays are cast from an object, and 200 of them hit another object in a distance greater than 0, then the value for that pixel will be 200/255 or .7843. An ambient occlusion example can be seen in Figure 2.10.

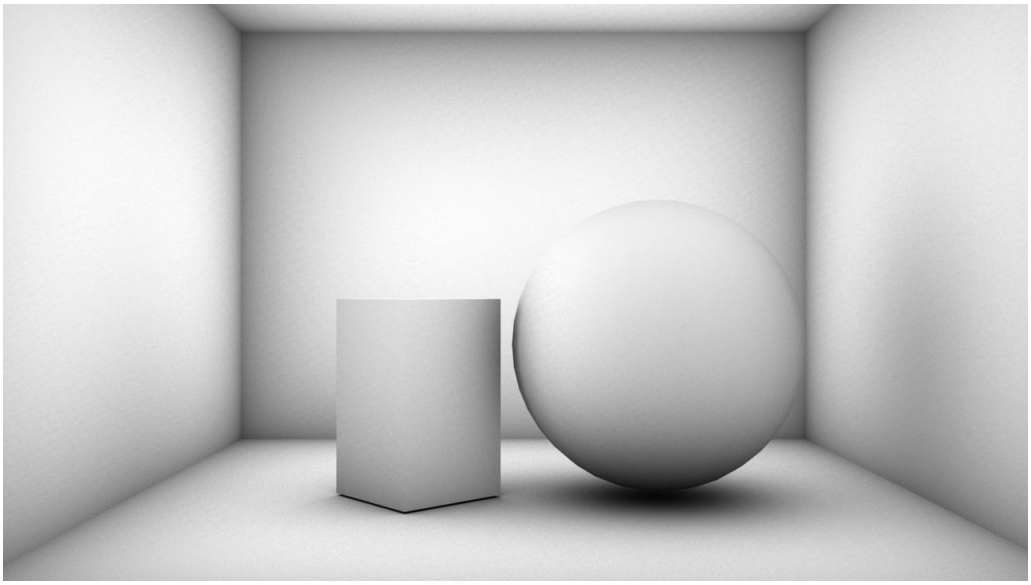

Figure 2.10: Example of Ambient Occlusion

To calculate global illumination. At each point in the sample set of data, instead of calculating the distance from the point, the Direct illumination shading value is

added instead. So the average color determined from 255 sample rays rather than the average number of intersections will be returned for a global illumination calculation. This can be seen in Figure 2.11. This process is where you can best see the color bleeding from nearby objects

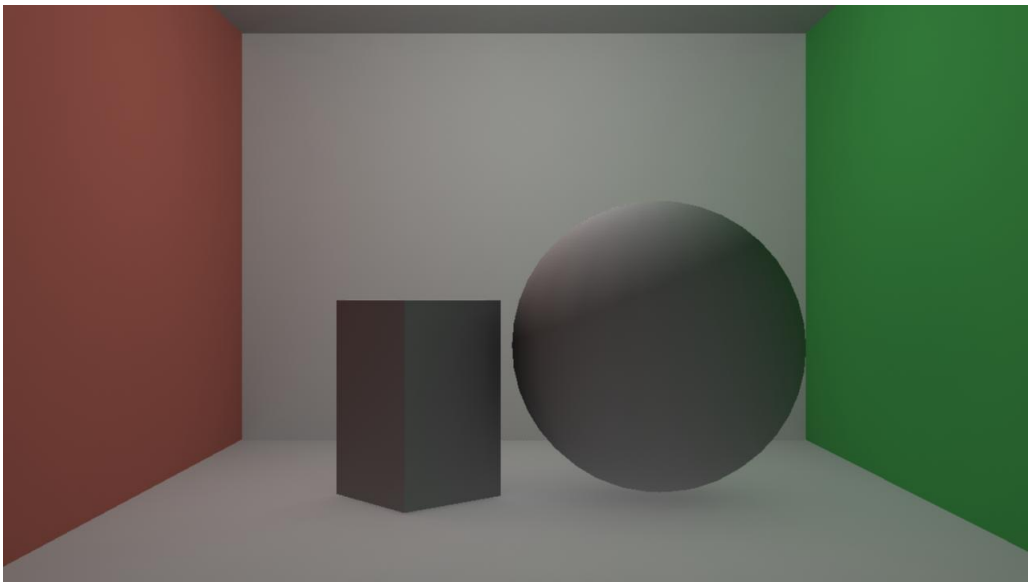

Figure 2.11: Example of Global Illumination

The final effect is a caustic effect, which is the ratio of specular color at each point on a surface. If a surface receives caustics, it calculates the reflective rays of nearby objects and returns the reflected color value average for each sample ray set. This effect can be seen in Figure 2.12.

Indirect illumination, added with Distributed ray tracing has made a different in the realistic perception of generated photorealistic images, and an effort is being made to incorporate these effects into realtime settings, such as video games, in order to provide a more immersive experience. However, since large sample ray sets are

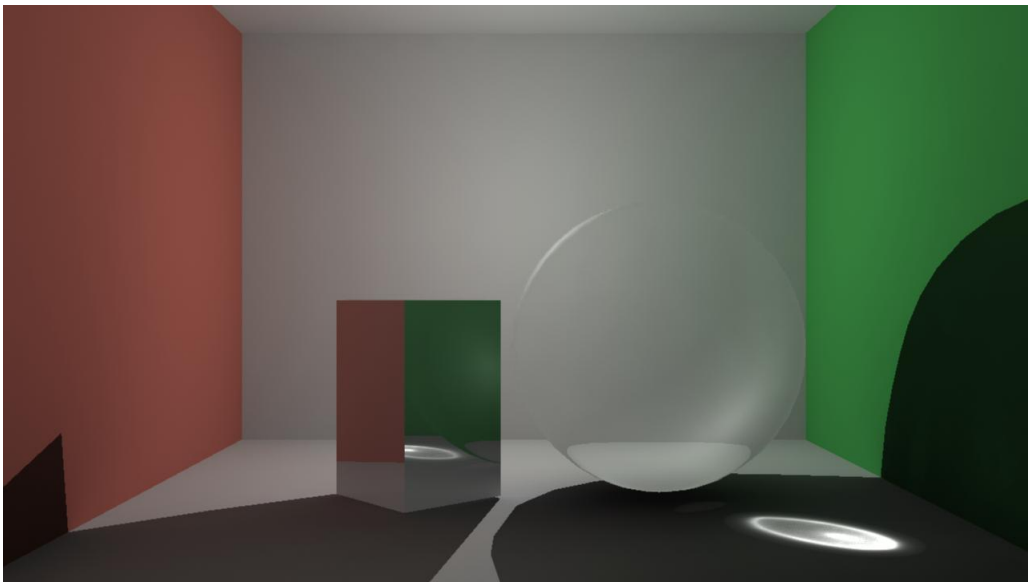

Figure 2.12: Example of Caustic Effect

needs for more realistic looking images, the computing power needed to produce this effect in realtime is a sizeable and expensive task to accomplish.

2.4 Languages

The four languages chosen by the researcher were determined for specific reasons. All languages needed to be object-oriented. They also needed to have potential for graphic capabilities. A brief history behind each language will be discussed to give insight into the basis for each languages creation, which might provide insight as to their feasibility in creating an image synthesis program.

2.4.1 C++

C++, whose commercial release was in October 1985[19] but originates from previous research with *C with Classes* starting in 1979, is an object-oriented coding language that is described as general purpose, or universally applicable to whatever task might be asked of it. It is considered an intermediate-level language, because of

it's capabilities with both high and low level computer functionality. Creator Bjarne Stroustrup claims to have drawn inspiration from not only C, but Simula, Algol68, BCPL, Ada, CLU and ML. C++ has become one of the most popular computer languages, and has gone on to influence the creation of other coding languages, such as C#[15].

According to Stroustrup, C++ was originally designed to combine Simula's facilities for program organization together with C's efficiency and flexibility for systems programming[19]. C++ is an extension of the C programming language, in fact it stemmed from an fundamental design called C with Classes[19]. When Stroustrup was designing the C++ language he had a variety of guidelines he deemed as "suitable" for computer languages. Stroustrup writes:

[1] " A good tool would have Simula's support for program organization-that is, classes, some form of class hierarchies, some form of support for concurrency, and strong(that is, static) checking of a type system based on classes. This I saw as support for the process of inventing programs, as support for design rather than just support for implementation.

[2] A good tool would produce programs that ran as fast as BCPL programs and share BCPL's ability to easily combine separately compiled units into a program. A simple linkage convention is essential for combining units written in languages such as C, Algol168, Fortran, BCPL, assembler, etc., into a single program and thus not get caught by inherent limitation in a single language

[3] A good tool should also allow for high portable implementations. My experience was the "good" implementation I needed would typically

not be available until "next" year and only on a machine I couldn't afford....[19]"

To further emphasize C++'s benefits, Stroustrup explained why he chose C over other languages of the time to implement from:

"C is clearly not the cleanest language ever designed nor the easiest to use so why do so many people use it?

[1] C is *flexible*: It is possible to apply C to most every application area, and to use most every programming technique with C. The language has no inherent limitations that preclude particular kinds of programs from being written.

[2] C is *efficient*:The semantics of C are "low level"; that is, the fundamental concepts of C mirror the fundamental concepts of a traditional computer. Consequently, it is relatively easy for a compiler and/or a programmer to efficiently utilize hardware resources for a C program.

[3] C is *available*: Given a computer, whether the tiniest micro or the largest super-computer, the chance is that there is an acceptable quality C compiler available and that that C compiler supports and acceptably complete and standard C language and library. There are also library and support tools available, so that a programmer rarely needs to design a new system from scratch.

[4] C is *portable*: A C program is not automatically portable from one machine (and operating system) to another nor is such a port necessarily easy to do. It is, however, usually possible and the level

> of difficulty is such that porting even major pieces of software with inherent machine dependencies is typically technically and economically feasible.
>
> Compared with these "first order" advantages, the "second order" drawbacks like the curious C declarator syntax and the lack of safety of some language constructs become less important. Designing "a better C" implies compensating for the major problems involved in writing, debugging, and maintaining C programs without compromising the advantages of C. C++ preserves all these advantages and compatibility with C at the cost of abandoning claims to perfection and of some compiler and language complexity. However, designing a language "from scratch" does not ensure perfection and the C++ compilers compare favorably in runtime, have better error detection and reporting, and equal the C compilers in code quality. [19]"

Stroustrup sought to create a universal, object-oriented language that was accessible and relatively low maintenance to begin programming with, since most computers were and still are compatible with C.

Object-oriented programming provides clean and modular code that is easier to upkeep and debug in the long run, because it separates functions and utilities in to class objects, which can be reused. This is C++'s main benefit to the Image synthesis process, besides it's efficiency to utilize hardware resources from a computer to provide faster results.

2.4.2 Processing

Processing is an open-source programming language and IDE developed by Casey Reas and Benjamin Fry in 2001. According to the Processing website(processing.org):

> "Processing is a programming language, development environment, and online community. Since 2001, Processing has promoted software literacy within the visual arts and visual literacy within technology. Initially created to serve as a software sketchbook and to teach computer programming within a visual context, Processing evolved into a development tool for professionals. Today there are tens of thousands of students, artists, designers, researchers and hobbyists who use Processing for learning, prototyping and production."

Andrew Glassner, a pioneer in Ray Tracing, also wrote a book on Processing called *Processing for Visual Artists*. Glassner helps to emphasize the importance and use for Processing in the visual world:

> "Processing is for artists, designers, visualization creators, hobbyists or anyone else looking to create images, animation, and interactive pieces for ary, education science or business....Processing offers you a 21st-century medium for exporessing new kinds of ideas and engaging audiences in new ways..."

To determine the usefulness and applicability for Processing and the ray tracing process, it can be found in the mission statement from its website. It was initially designed to teach computer programming within a visual context. It's goal is to promote software literacy within the visual arts and visual literacy within technology. Since that is also the aim and goal for this thesis, it seem that Processing is a strong option to investigate.

2.4.3 Python

Python, named after Monty Python's Flying Circus was first founded by Guido van Rossum. He began his work on Python at the the National Research Institute

for mathematics and Computer Science in the Netherlands in 1989. Python is a "high-level" and "interpreted" programming language. While it can be argued that all computer languages are interpreted because this is how the computer is able to function and read from the languages to accomplish tasks, Python is considered an interpreted language because unlike C or C++, Python does not require a compiler. Whereas C or C++ compiles its code into machine-language instructions for the computer to follow, Python is interpreted directly from it's written code for the computer to follow. van Rossum is quoted saying he was unhappy with the productivity of creating a script or utility in C, which was the direct result of his interest in establishing Python[20].

One of the main focus's of Python is readable syntax along with productivity. According to Jim Mcconnell's book *Code Complete*, Python has a ratio of 6:1 over one line of C code, which roughly translates to one line of Python makes up for six lines of C code[14]. According to van Rossum, Python's creation was heavily impacted from the coding language ABC. ABC's design was intended to be a programming language that could be taught to intelligent computer users who were not computer programmers or software developers. The main deficit with ABC's design was its inability to bridge the gap in GUI creation or an inability to directly access the file system and operating system in a computer. Pythons syntax eliminates the need for traditional curly braces() and instead uses a tabular system that denotes code blocks.

2.4.4 RenderMan

RenderMan© is a software and API that many companies in the computer industry use to render large projects for entertainment or video game use. RenderMan© specializes in network distribution of renderings throughout a "renderfarm" that has

the ability to render potentially ray traced images faster than a single computer. RenderMan© is referred to as the rendering engine that produces the image. In order to 'communicate" with this engine however, two things are needed. The first is a RenderMan© Interface Bytestream (RIB) file that acts as a scene description file. This file acts a descriptor for how the engine should work. It incorporates all aspects of ray casting, ray tracing, direct illumination, distributed ray casting and indirect illumination. The second needed file is a RenderMan© Shading file, which utilizes the RenderMan© Shading Language(RSL). This language acts as a description file for how objects in a scene interact with light. By utilizing the RIB and RSL files, each task will be accomplished within the scope of this thesis.

It is important to note that RenderMan© has already implemented all of the milestones needed for this thesis. Also RenderMan© is based off of a camera projection matrix algorithm that does not cast rays the same way that this thesis uses ray casting. RenderMan© also incorporates a variety of highly advanced rendering techniques that are too far outside the scope of this modest Master's thesis. However, the thesis will discuss the feasibility of utilizing the pre-constructed implementations of the milestones without the aid of a production software such as Autodesk Maya© or Side Effects Houdini© will present relatively the same issues of implementation as the other languages would in the theory of ray casting.

3. METHODOLOGY

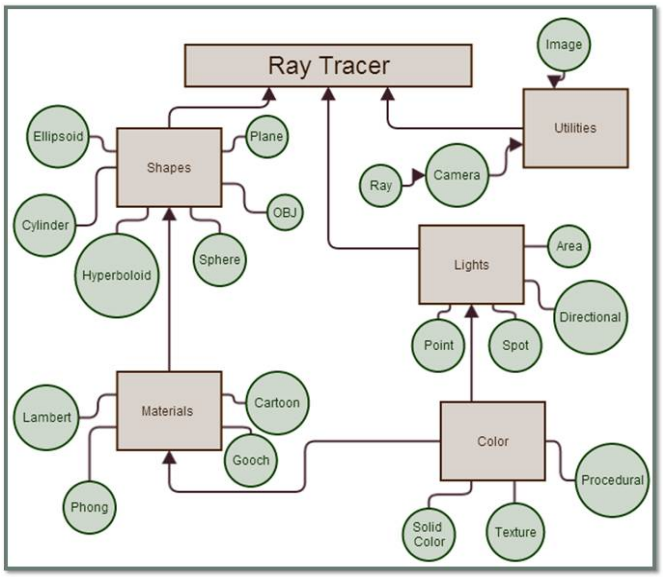

Figure 3.1: Diagram of the Parenting/Dependency Structure of the Ray Casting Program

The method used by the researcher are simple and straightforward. For each coding language, notes were kept that outlined the difficulties and roadblocks met specifically caused by each language. To provide continuity between each coding process, the same coding theories and fundamentals were be kept. It can be argued that an object oriented approach to software development is one of the best practices because it is a well organized and helpful for large coding projects, so this is the approach taken. This means that classes with specific variables and structures were defined, so that instances of objects could be made. Wherever possible, the image synthesis process was sub-categorized in order to better organize the information

needed for the process, which can lead to quicker development and troubleshooting after the code has been written. A Diagram of the structure for the ray tracing program can be seen in Figure 3.1.

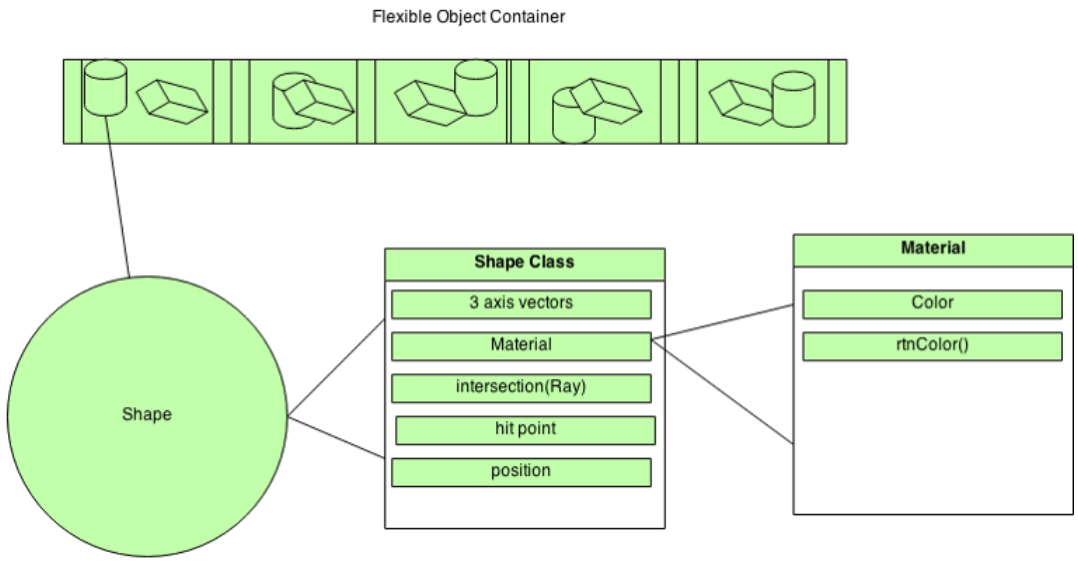

Figure 3.2: Diagram of the strived for each program Structure of a Ray Casting Program

Figure 3.1 demonstrates the parenting and dependency relationships between objects in the program. Since each program was modelled after this conceptualization, measurements of success and difficulty were more clearly defined. When building large projects it's important visualize the goals and relationships of the project to better understand the process and work smarter while implementing the task. For each language a similar data structure was also strived for, shown in Figure 3.2. A dynamic data structure that grows and shrinks with the amount of objects added to it along with a agnostic data structure type was desired for this project. As we will see in the next section, this was intuitively most similar to the functionality of

a ray casters. In addition to having a program architecture and data structure, the milestones established by the researcher that segments image synthesis theory within four categories also provided a structured approach to analyzing and reporting the conclusions used for each language. More details on the milestones can be found in the following section.

By determining the difficulty in implementing Figure 3.1 and Figure 3.2 along with using that class structure to further implement the image synthesis milestones of Preliminary Preparations, Direct Illumination, Ray Tracing/Distributed Ray Tracing, and Indirect Illumination, a detailed report has been collected that informs the results collected and reported in this thesis.

4. IMPLEMENTATION RESULTS

4.1 Image Synthesis Program Structure and Implementation

The basis for every ray tracing program can be broken down into a simple set of processes. First a program must be able to read and write images. Once the ability to read and write images has been realized, the process for casting rays is demonstrated by the following pseudocode, or code outline:

Algorithm 1: Ray Casting Pseudocode

```
foreach pixel in image do
    foreach object in scene do
        if object intersects ray then
            intersection = distance of intersection of object and ray from
            viewpoint;
            if intersection >previousIntersection then
                previousIntersection = intersection;
            end
        end
    end
    if intersection exists then
        foreach Light in scene do
            determine materialColor from Light;
        end
        pixel = materialColor;
    end
    else
        pixel = black or alpha(no color)
    end
end
write image;
```

Each pixel in an image will correspond to a position in 3D space, as demonstrated in the Figure 2.1, found on page 3. In this Figure 2.1 each box represents an image pixel. The basics of ray casting are determined as for each pixel in the image, determine if there is an intersection with each 3D object in the scene. The closest object's intersection point will determine the pixel's final color based off of that

object's illumination algorithm and the lights in the scene. Each ray tracer written follows the above pseudocode, but each ray tracer also adhered to specific structure and code organization. In order to better understand the ray tracing structure, the researcher needed to establish and learn some important Computer Science terms.

4.2 Milestone 1: Preliminary Preparations

Preliminary Preparations refers to anything that needed to be accomplished before being able to start working on the ray tracing theory coding. In this section, simple matters such as program language installation and accessibility will be discussed. Also discussed is the majority of the "Utilities" section of Figure **??**. The foundations of a ray tracing program are built around two fundamental procedures, image writing and vector math. It can be argued that ray tracing can also be accomplished through matrix operations, but for the purposes of this thesis, vector math will be enough. vector math is the basis for all ray tracing theory, and because of this, a class or data structure that can accomplish all aspects of vector math including addition, subtraction, dot product and cross product is crucial to accomplishing the image synthesis program. In addition to this, in order to see the final product after all the theory has been calculated, you must be able to write an image. Arguably the easiest image to write out is a Portable PixMap (PPM). Other image formats include JPEG, BITMAP, and PNG. Each of these image formats requires a more complicated compression algorithm. Each language may also require unique addition files in order to compile or execute the final product. Each of these languages will be assessed from the viewpoint of a student in the Visualization Department of Texas A&M, because this is directly relevant to the researcher's experience.

4.2.1 C++

The C++ language, as described in Section 2.4.1, is an object-oriented language that is considered both a high and low level language because of the level of control you can have over computer functionalities. Development and installation with C++ was accomplished via the provided computers at Texas A&M's Department of Visualization. When developing in a Linux/Unix environment, C++ can be executed straight from the terminal of the operating system. This means no external IDE is needed to compile your code. This development style was more appealing to the researcher because IDE environment's have their own learning curve to make the coding work, which would would have to be realized before learning the C++ language semantics. However, if the Department of Visualization was not available, installation of the C++ development tools presents it's own issues. To work on a Windows computer, you must first determine a compiler to work from, download that compiler, and then work from that specific compilers commandline. Common practice would be to download an IDE such as Visual Studio or Eclipse. Since computers are not typically sold with a Linux operating system on them, developing in Linux outside of an academic setting would mean to freshly install a new operating system, which is not a beginners task. After learning how to install Linux, you must then learn the proper way to download specific compiler packages, which with the help of Google can be significantly easier than installing a compiler in Windows since it is merely a written command into the computer's terminal. A Macintosh computer was not tested, but theoretically runs off of a Unix platform like Linux and is more commonly sold in stores, making it more accessible to students who are not computer scientists. Starting to develop in C++ on any computer is a complicated task unless it has been provided for you, like at Texas A&M.

What makes C++ most unique from other languages explored in this thesis is the concept of a makefile, or the make command. When forgoing the use of an IDE, in order to communicate with the compiler installed on the Unix platform, you need to create a file that will define which source code files are needed to make object files and which compilers and linkers are included in the make command. With an IDE, if you can get the IDE to work correctly, it will automatically generate a makefile and perform the make command. This further solidified the decision to work without a proper IDE and use a syntax highlighting text-editor, because nothing was generated automatically, only what was typed and developed by the researcher was executed. This allowed for easier debugging of the makefile and make command rather than debugging of the IDE's internal processes.

C++ also introduces another difference from other languages used in this thesis, which are optimum for processing speed, but also can be argued is just another confusing detail to learn about the language. When creating a class in C++, it is common practice to outline the class structure in a "header" file and then flesh out this outline in a ".cpp" file. This is a technique inherited from C coding, which was mostly used for older computers that could not support large quantities of memory at the same time. However, since computers have improved, this practice now further helps to segregate the interface of a class and the implementation of the class rather than save memory space during execution time. For this thesis, only header files were used because speed optimization was not a priority.

In addition to a makefile, another obstacle introduced by C++ was the vector math library. C++ does not have a standard vector library. This operation needs to be hand coded in order to be available for use. Either a developer has to implement their own vector math library, or borrow a pre-existing library. This requires additional research or work to write the code needed. If a developer is going to borrow

code from the Internet, they need to trust that the original writer knew what they were doing. They then need to learn how to use the classes or structures defined in this found vector class. The vector class used for this thesis was a library circulating amongst the Texas A&M students and was written by a former faculty member at A&M, so the source was reputable and could be trusted to be useful and correct.

The same obstacle applies to the image reading and writing functionality of an image synthesis program. There is no standard C++ image library. To read and write images, a developer needs again to write their own code or borrow an existing third-party library. For this thesis, a image class was written to process .ppm images. This is very limiting to the functionality of the final image synthesis program, because the .ppm images were always converted to .png or .jpg for web use after they were created, but for the developer .ppm was the most accessible and easiest to write, because of the format of a .ppm image. This was determined because the thought of including a third-party library, more specifically the semantics of linking the files correctly in the coding process seemed more difficult than writing an original class.

4.2.2 Processing

Processing handles the preliminary preparations issues significantly different from C++. Whereas the researcher tried to stay away from IDE's for C++, Processing itself is one giant IDE for the Java language, so that would have been impossible.

Installation of the Processing language is simple. A compressed file is downloaded from the processing.org website and once uncompressed an application file can be clicked and opened. This will start the Processing application and development can begin. The processing.org website provides downloads for the three major operating systems, Windows(32-bit and 64-bit), Linux(32-bit and 64-bit) and Mac OS X. This application file does not require installation, so it can be run from anywhere on

your computer. This is convenient when installed on an external hard drive because anywhere the hard drive can be read you can develop with Processing from. The availability of Processing is convenient for students who may not have a laptop and are switching between un-networked workstations that do not keep track of their workspaces. For other languages this would mean reinstallation of the IDE or compilers, or developing a new workflow for a new computer that you are working with (by using a different IDE or text editor).

Processing has a vector math class built into it's IDE called PVector. This library has extensive documentation on the processing.org website. The PVector implementation is something to be discussed however. The quirk of PVector operations can best be described with an example. Consider the C++ equation for determining the hit point along a ray from the rays origin:

$$p_{hit} = ray.Origin + t * ray.Direction \tag{4.1}$$

This same equation in PVector would be written:

$$p_{hit} = PVector.add(cast.Origin, PVector.mult(cast.Direction, t)) \tag{4.2}$$

Rather than have operator overloads for the math components of Vectors, each operation is a function. This can either make equations very long and confusing to look at, or create many more lines of code. Out of the two equations, since Equation 4.1 has more distinguishable characters, it will be easier to understand and therefore debug if a problem occurs than Equation 4.2.

Processing also has included it's own image class. This image class is very con-

venient and has no quirks to be noted. The PImage library has the ability to save any type of common image file, as long as the file type is specified within the image save command. The same is true for the PImage open command.

4.2.3 Python

Many of the same complications that arise with C++ occur with Python as well. Python is arguably much simpler to develop on a Linux environment because of the capability of running programs through the Unix terminal. However, unlike C++, Python does not need a header file or to be compiled before running on a computer. To install Python, one must go to Python.org and download the correct source. Python has been released for all major platforms just as Processing has been. Python is pre-installed on Unix based systems, so development can start right away with Linux or Max OS X. For windows, a similar process for use with IDE's and random Google searches on how to set up Python for a Windows environment is necessary.

Python does not have a vector class library. The same issues that were apparent for C++ programming are also relevant for Python. However, the researcher found it easier to find third party vector libraries written in Python, which might be as a direct result of the Python communities popularity in the past few years. For this thesis a vector class was written based off of the C++ vector library used. The decision to write an original Python vector class was not because a better library could not be found, but rather as an educational exercise to familiarize oneself with the Python language.

The imaging library that was used for this thesis is called the Python Imaging Library(PIL). PIL originally was a Python supported library until their most recent release. Now PIL is a third party image library that is still in development for

the latest 3.0 version of Python. If the latest version of Python is used, the same difficulties will occur as with C++, but into 3.0 are not significant in Python to effect the ray tracing process. It is recommended to use a version of Python that supports PIL, because it is more beneficial to have the PIL capabilities rather than write a image class from scratch.

4.2.4 RenderMan©

RenderMan© for this thesis is an outlier. It does not follow the same milestones as the other coding languages but there is a very important fact to point out. RenderMan© is a very expensive rendering engine that is not available to the "average joe". As of writing this thesis in 2014 a RenderMan© Floating Institutional license sells for $274.00. The same goes for a Pro Server license. A one year student subscription costs 199.95, and that does not allow students to produce anything for commercial value. Over at least 4 years of schooling that is $800 on top of tuition and other books expenses. It is not in the researchers budget, nor in most students budget, to have access to RenderMan© outside of an academic setting. Thankfully Texas A&M has an Institutional license for RenderMan© that was used for this, but that meant the accessibility of using RenderMan© outside of school hours or at anytime could be limited by school computer availability.

It is worth noting that there is an open source alternative to RenderMan© called Pixie©. This is based off of the RenderMan© syntax, but does not have every aspect that RenderMan© does, nor is it developed by the same people at Pixar. However, if it is most beneficial to learn the RenderMan© syntax and RIB/RSL file structure, Pixie© might be a safe alternative to use from home or outside of an academic environment.

4.3 Milestone 2: Direct Illumination- Ray Casting

Majority of the planning and learning occurred on an implementation level during this milestone, because it was the first milestone to implement code. Declaring a class for each language was a different learning process. C++ and Processing have a similar syntax, with relatively minor changes but Python is significantly different, not only in syntax but in theory. A class is fundamentally a descriptor of characteristics and functions needed by a virtual "object" in a computer program. Class declaration and planning for each language is important because this relates back to accomplishing our ray-tracing theory, which establishes our goal to iterate over every object in a scene from each pixel in an image to determine an intersection. Most logically, we want every object in our scene to be held in one place so that we only need to write one iteration loop. Furthermore, since we have no knowledge of how many objects might be in a scene at one time, ideally this container should be dynamic in size. This introduces our two main challenges for implementing Basic Ray Casting theory into computer programming, organization of classes(objects) and assembly of those classes into a dynamically sized data container (or data structure).

4.3.1 Classes

4.3.2 Data Structures

Accomplishing the data structure task in Python is the most simple and takes no extra effort. Simply declare an object and place it in a Python list, like so:

```
1 sphere = Sphere.Sphere();
2 cube = Cube.Cube();
3 objects =  [];
4 objects.append(sphere);
```

```
objects.append(cube);
```

Listing 4.1: Python List Example

That code will create a sphere object, cube object, and an objects list, and then add the sphere and cube into the data structure with the lists append method. Python lists can accept any combination of different variable types, which can be useful for a ray-tracers implementation but dangerous for software prosperity and robustness. If we are assuming that all variables in a list have the same type and therefore the same associated functions and an object that does not fall under these assumptions gets added to the list, this will cause the program to fail. More care needs to be taken to make sure all variables are of the correct type that are added to a list to be sure that the program will run successfully, since this is not required for the program to start running in the first place.

That is the reason why it is more difficult to accomplish the data structure task in C++ and Processing(Java). All objects must be of the same type to add them to a data structure. Furthermore the most basic data structure, an Array, must be declared a specific size upon initialization. The size issue can be easily circumvented however. For each of these there is a dynamic container that will grow and shrink in size. C++ has a vector data structure and Processing has an ArrayList. To declare each is simple:

```
vector <Sphere> sphere_list;
Sphere sphere_object = new Sphere();
sphere_list.push_back(sphere_object);
```

Listing 4.2: C++ Vector Example

```
ArrayList<Sphere> sphere_list = new ArrayList<Sphere>();
Sphere sphere_object = new Sphere();
```

```
3 sphere_list.add(sphere_object);
```

Listing 4.3: Java ArrayList Example

Each of these examples creates a *Sphere* list, an instance of a *Sphere* object, and then puts that *Sphere* object into the *Sphere* list. Note that each structure *must* be declared as the type of object that is being placed into it. Since Processing and C++ are both strong-typed languages, eliminating the data type issue is not possible, but there is a solution. Cardelli and Peter describe strong-typed, or static-typed, languages in their article entitled textitOn Understanding Types, Data Abstraction, and Polymorphism as follows

> Static typing allows type inconsistencies to be discovered at compile time and guarantees that executed programs are type-consistent. It facilitates early detection of type errors and allows greater execution-time efficiency. It enforces a programming discipline on the programmer that makes programs more structured and easier to read. [4]

Two Computer Science called Inheritance and Polymorphism will help to solve this problem. Inheritance is the idea that a child class can inherit the properties and functions of a parent class. Cardelli and Wegner also explain different types of polymorphism, and from their explanation the type used for this thesis is called subtyping.

> Subtyping is an instance of *inclusion polymorphism*. The idea of a type being a subtype of another type is useful not only for subranges of ordered types such as integers, but also for more complex structures such as a type representing Toyotas which is a subtype of a more general type such as Vehicles. Every object of a subtype can be used in a supertype context

in the sense that every Toyota is a vehicle and can be operated on by all operations that are applicable to vehicles. [4]

Subtyping will allow us to accomplish the example that if both a sphere class and a cube class need a move function that will reposition it within the scene, a parent Shape class can be established that will provide inherited logic for each class. If each object also needed a resize class, but the sphere needed to resize differently than all other objects, the size function can also be rewritten in the sphere class. The type Shape can be subtyped in to Sphere and Cube to accomplish shape specific logic as needed. This is accomplished in C++ as follows:

```
//shape.h file//
class Shape
{
    public:
        Shape(){};
        float move(){
            //calculate size for all objects
        }
        virtual float size(){
            //calculate size for all objects
        }
}
```

```
//sphere.h file//
#include "shape.h"
class Sphere: public Shape
{
    public:
        Sphere(){};
        float size(){
            //calculate sphere size logic.
        }
}
```

```
//cube.h file//
#include "shape.h"
class Cube: public Shape
{
```

```
    public:
        Cube(){};
}
```

Listing 4.4: C++ Inheritance Example

In the above example, all classes will have a move function and a size function, but the size function for the Sphere class will function differently than the size function for the Shape/Cube class because it has been overwritten. All children classes need to call their own constructors in order to be instantiated. It is important to note that a child class must contain "#include 'sphere.h'" or else it will not work correctly. In order to rewrite a parent function in a child class, the keyword virtual needs to be used as seen on the size function in the shape.h example. To accomplish this same procedure in Processing, classes need to be established as follows:

```
//shape.pde file//
abstract class Shape
{
    Shape(){};  //Constructor
    float size(){
        //calculate size for all object types;
    }
}
```

```
\label{listing:Java Abstract Class Example}
//sphere.pde file//
class Sphere extends Shape
{
    Sphere(){};
    float size(){
        //calculate size specific for sphere
    {
}
```

Listing 4.5: Java Abstract Class Example

The syntax for the cube.pde file is the same as the sphere.pde with the extends keyword showing that the sphere is the child of the Shape class.

Note:Python Classes While parent classes are not important for Python list functionality because Python can be considered a dynamic type language (as opposed to the static type for C++ and Java), for organizational purposed they were used and inheritance was declared as follows (note tabs and spacing), using the same example:

```
//shape.py file//
class Shape(object):
    __init__(self,...):
        //shape constructor
    def size(self,...):
        //size calculation for all child classes
```

```
//sphere.py file//
import Shape
class Sphere(Shape.Shape):
    def __init__(self,...):
        //sphere constructor contents
    def size():
        //calculate size specific for sphere
```

Listing 4.6: Python Class Inheritance Example

Now that the class structure has been established for each strict-type language to promote a single object data container, which will mirror the ray tracing structure theory, establishing the data structure for each language needs to be adjusted. To introduce how this is achieved in each language, the topic of "pass-by-value" and "pass-by-reference" must be discussed. In both C++, when declaring an object, two variations can be done in relation to the memory organization within the computer. The first, "pass-by-value" is the most common and everytime a variable changes or moves, a *copy* of the variable is made within the backend and saved in a new location. However another schema exists, "pass-by-reference", which rather than copies the data creates a reference or "pointer" back to the original variable. The "pass-by-reference" tactic in C++ is what will allow different variable types to be saved to a common parent data structure, as follows:

```
vector <Shape*> shape_list;
Sphere *sphere_object = new Sphere();
sphere_list->push_back(sphere_object);
```

Listing 4.7: C++ Vector Example

Declaring the vector with the "*" denotes a C++ pointer, or that the variable will use the pass-by-reference schema. Pass-by-reference and pass-by-value variables cannot be combined within a data structure. To then call any functions within the pointer variables, you must use "-¿" rather than ".". This allows us to save all different shape objects within the same data structure. To accomplish this is Processing you may have noticed that in **??** that the Shape class was declared as Abstract. This keyword signifies that an object of this type may not be instantiated within a program. This keyword demonstrates true inheritance because it acts as a placeholder for commonly used functions and variables between class types. In Java, it will allow us to declare child classes as common types without losing each classes individual functions and variables. This is accomplished as follows:

```
ArrayList<Shape> shape_list = new ArrayList<Shape>();
Shape sphere_object = new Sphere();
Shape cube_object = new Cube();
shape_list.add(sphere_object);
shape_list.add(cube_object);
```

Listing 4.8: Java ArrayList Example

Notice that we are declaring sphere_object and cube_object as a type Sphere but using the child classes individual constructors. This will allow us to call "shape_list.add()" on the next two lines without throwing an error. If the Shape class was not declared as abstract, when declaring Shape sphere_object and calling the child constructor,

and functions that are specific to the child class will be replaced with the parent classes functions. While Java still has “pass-by-value” and “pass-by-reference”, Processing/Java has simplified their language to effectively made all declared variables function as pointers, pointing back to the original variables declaration.

4.3.3 Milestone Results: Processing Time Overview and Images

Once the issues with Class and Object declaration were solved, implementing the Basic Ray Casting Milestone was very straight forward with each language. However there was one issue with the processing language that accounted for a majority of the Processing implementation, and accounts for a core theory of ray tracing implementation.

4.3.3.1 Left Hand vs. Right Hand Rule

In the 3-D virtual world, there is a set of governing rules that determine an objects position within the space, known as a coordinate plane. Just like on a two dimensional graph there is an X and Y coordinate, in a 3-D scene there is a new coordinate Z added. The order and direction for each of these axis, as they’re called, determines many important calculations when determining size and space of objects in the scene. Left-handed and right handed can be described as follows:

> In order to draw something at a point in three dimensions the coordinates are specified in the order you would expect: x, y, z. Cartesian 3D systems are often described as ”left-handed” or ”right-handed.” If you point your index finger in the positive y direction (up) and your thumb in the positive x direction (to the right), the rest of your fingers will point towards the positive z direction. It’s left-handed if you use your left hand and do the same.

For the C++ and Python implementations of the ray tracers, a coordinate system needed to be created from scratch, and since the researcher is right handed a right handed, a right handed coordinate system was used. This means, when looking at a computer monitor, the positive y axis pointed towards the top of the screen, the positive X axis pointed to the right of the screen and the Z axis points directly out of the screen, perpendicular to the screens surface. However, this is not typically a traditional schema for coordinate planes with computers, nor was it the scheme used by Processing. Processing used a left-handed coordinate system where the positive y axis points towards the bottom of the screen, the positive x axis points towards the right of the screen, and the positive z points away from the user, towards the back of the computer, perpendicular with the computer screen. Many of the calculations associated with cross products and other Vector Math needed to have the y value and z value negated so that it behaved more similarly to the other two ray tracing programs

4.3.4 Basic Ray Casting Images

Here are a few Image from the Basic Ray Casting Milestone. While these are just a few examples from each language and each task, each language accomplished the same types of images that are presented.

4.3.5 Performance Results

For each test in Basic Ray Casting, a variety of levels of complexity were tested in order to see how each language can process a different amount of load. For each rendering a 1280 x 720 image size was used to generate a high quality image for testing. The results are graphed and labeled as follows:

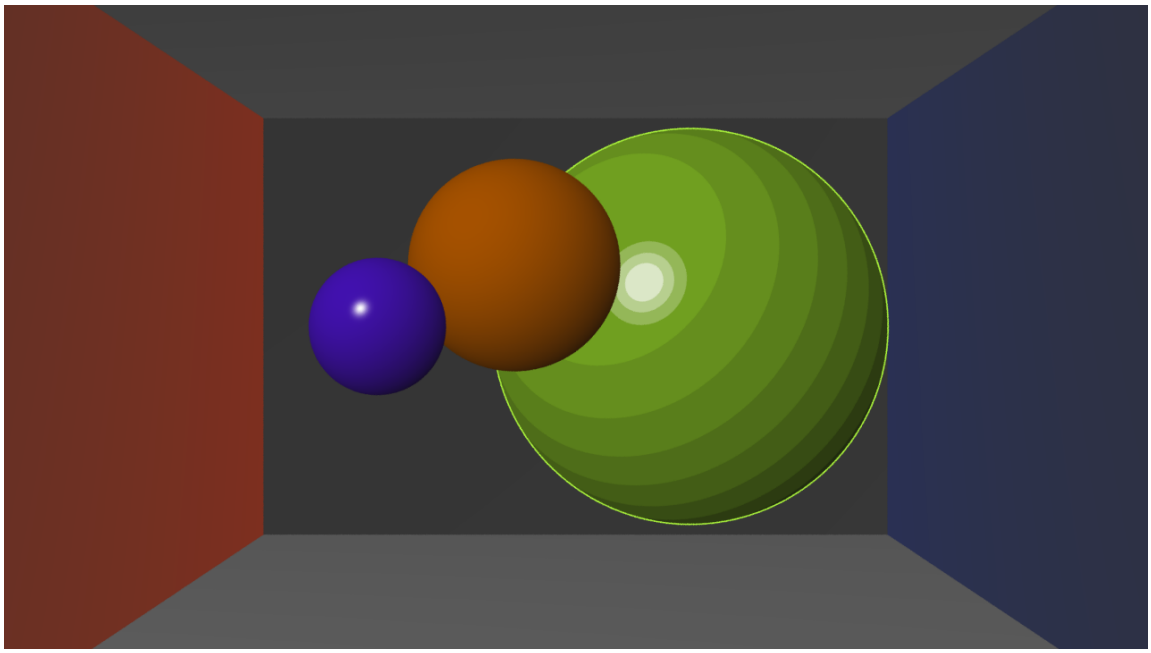

Figure 4.1: Ray Casting Accomplished with Processing. Shown are three spheres with a different texture type for each, and five planes all with flat textures applied.

4.4 Milestone 3: Ray Tracing and Distributed Ray Tracing

4.4.1 C++

4.4.2 Processing

4.4.3 Python

4.4.4 RenderMan©

4.5 Milestone 4: Indirect Illumination

4.5.1 C++

4.5.2 Processing

4.5.3 Python

4.5.4 RenderMan©

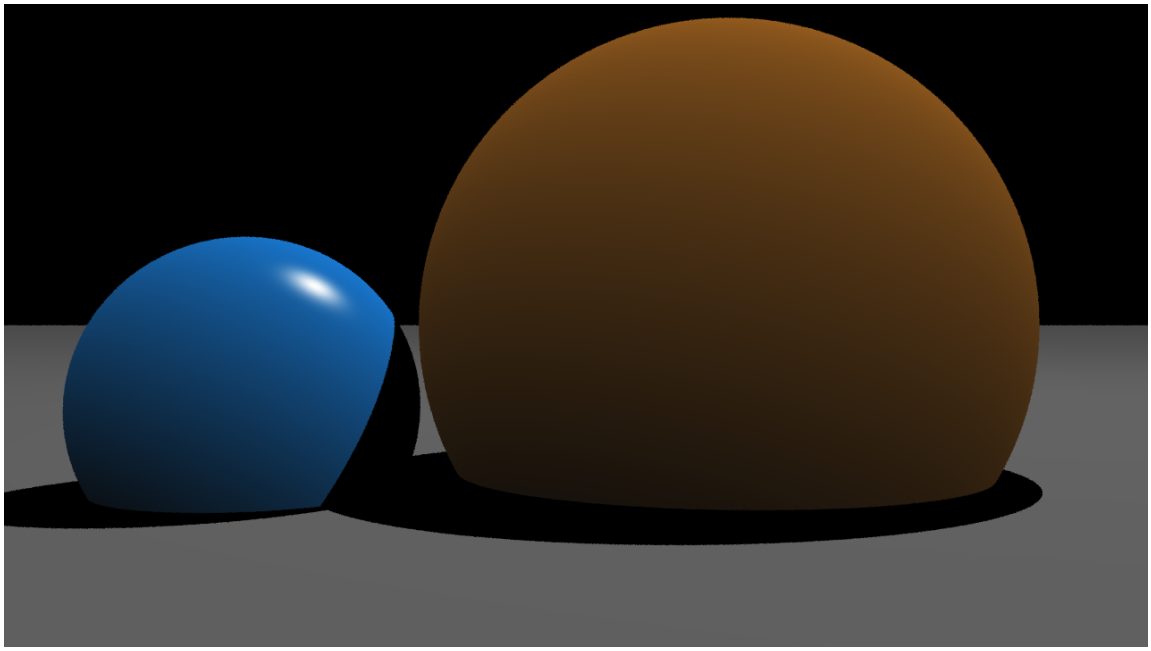

Figure 4.2: Ray Casting Accomplished with Python. Shown are two spheres, and one plane that all cast shadows from a simple point light.

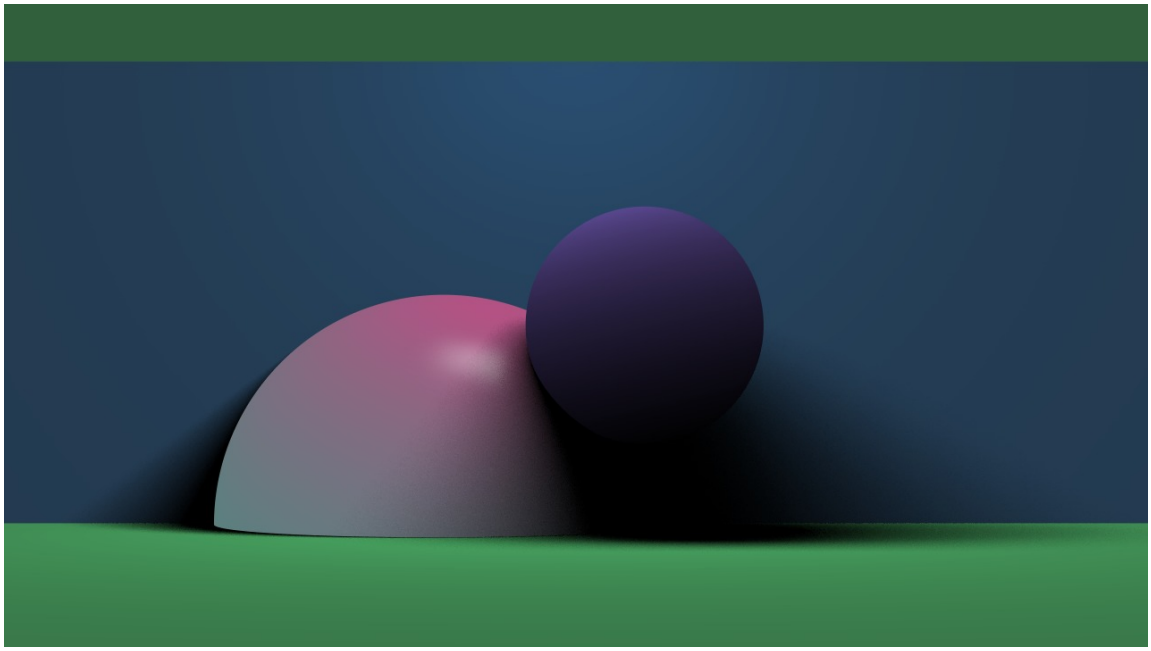

Figure 4.3: Ray Casting Accomplished with C++. Shown are two spheres, five planes and an area light that all cast shadows, which accounts for the soft shadow effect.

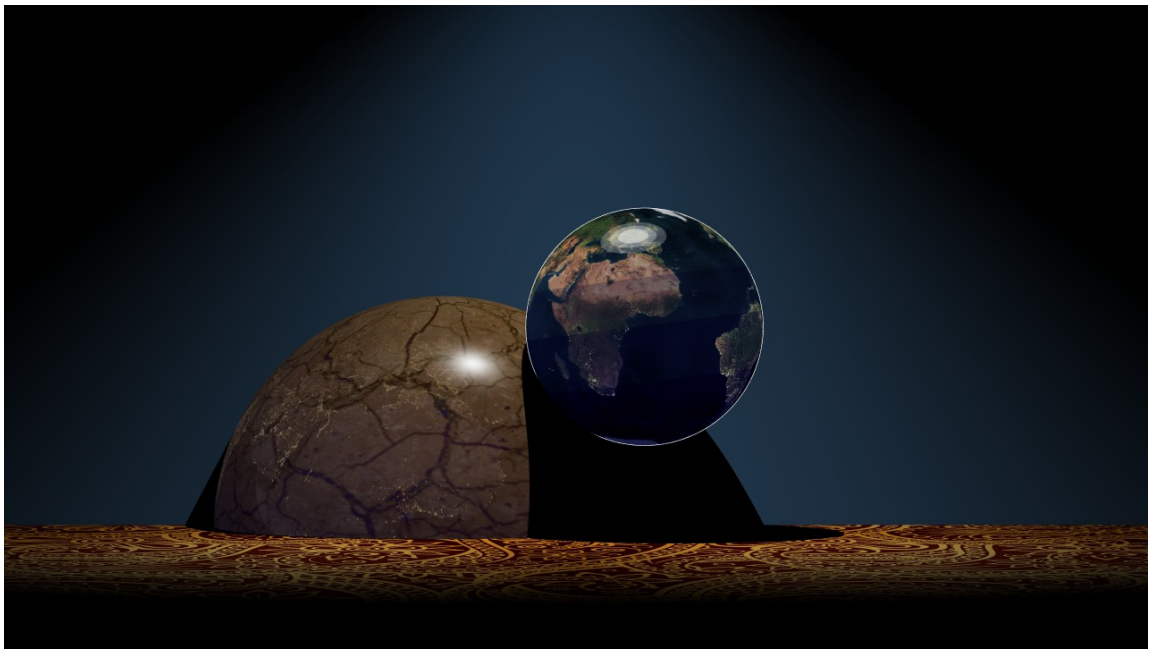

Figure 4.4: Ray Casting Accomplished with C++. Shown are two spheres with 2 separate shader types with two separate images mapped to them, and two planes, one with a repeated image texture and one without any image texture.

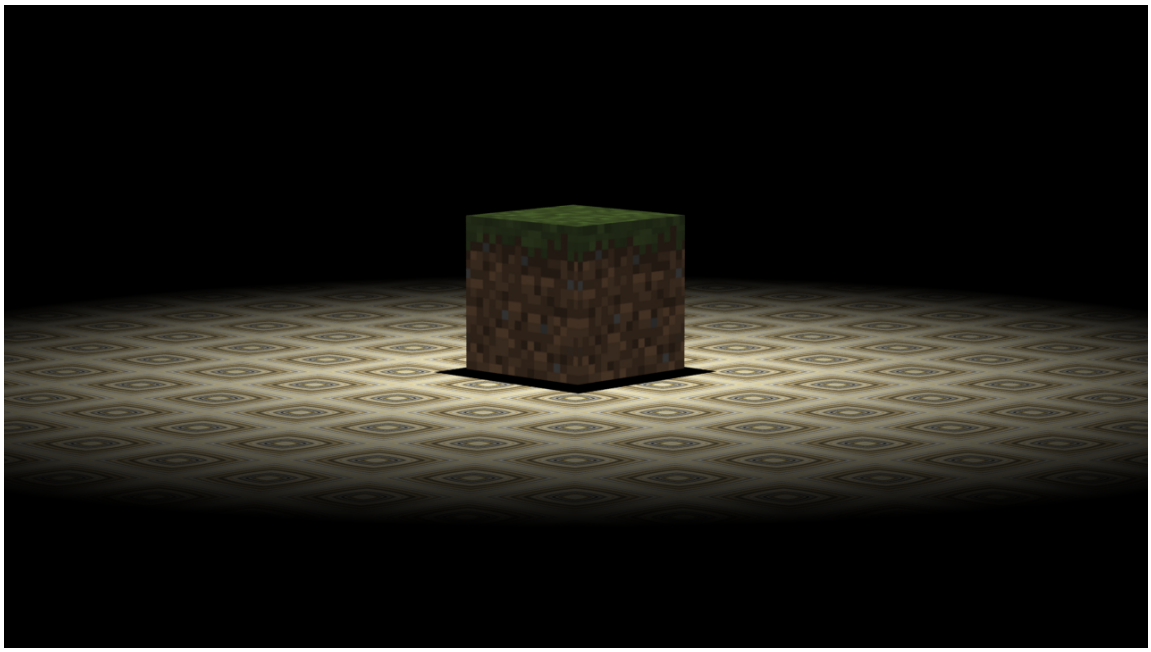

Figure 4.5: Ray Casting Accomplished with Processing. One cube with twelve triangles and an image mapped to it on a plane with a repeated image texture illuminated by a spotlight.

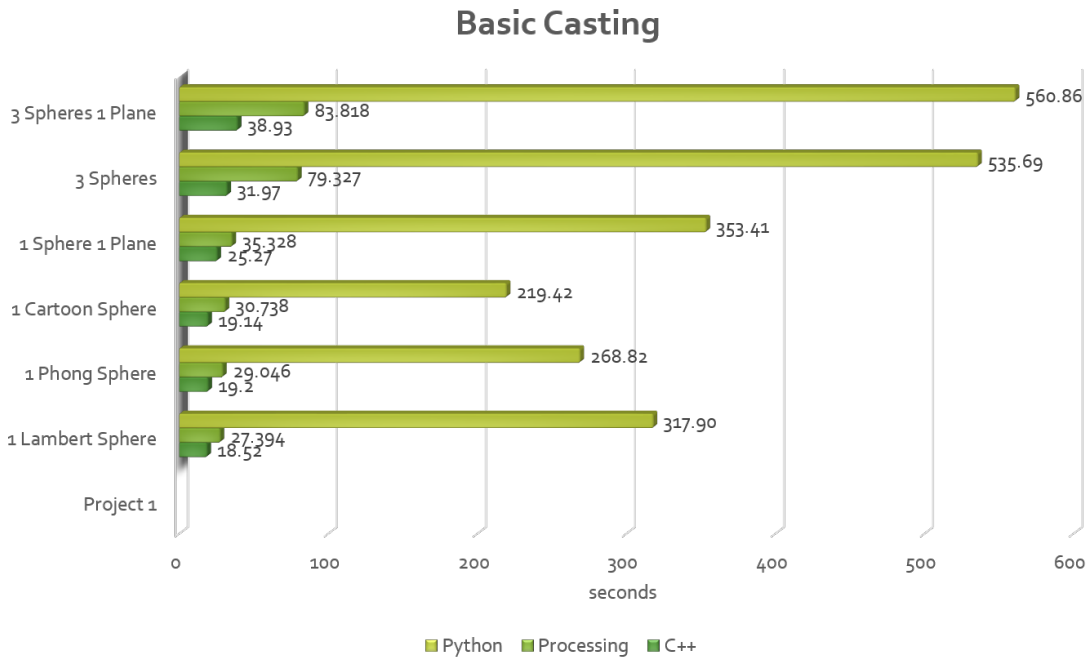

Figure 4.6: Basic Performance Graph filler

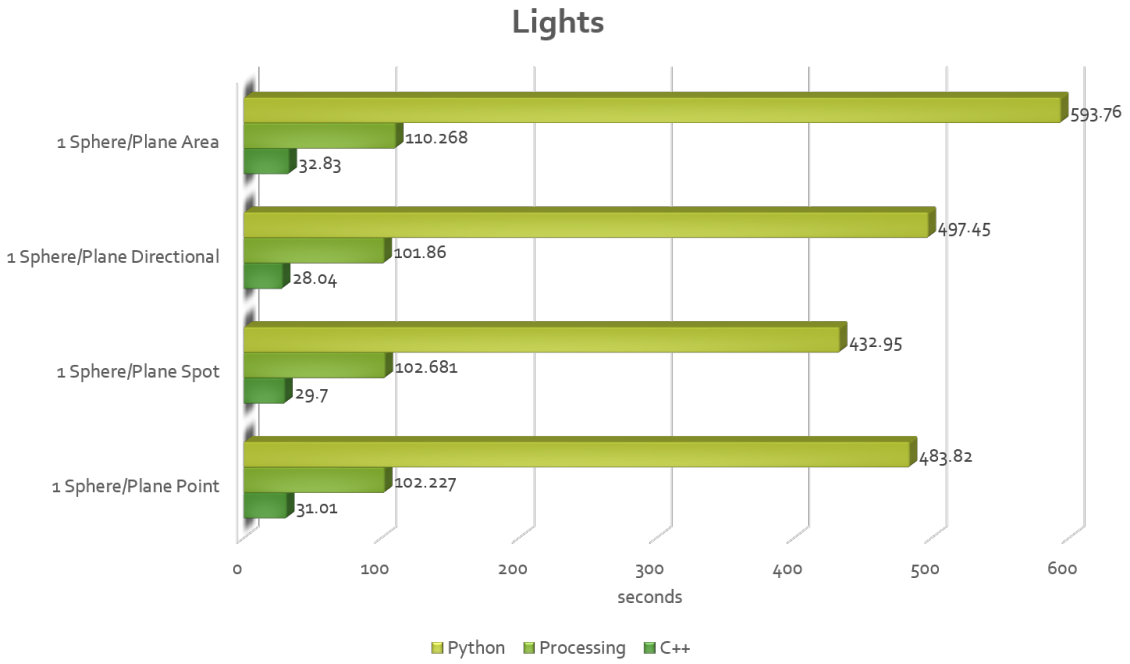

Figure 4.7: Basic Performance Graph filler

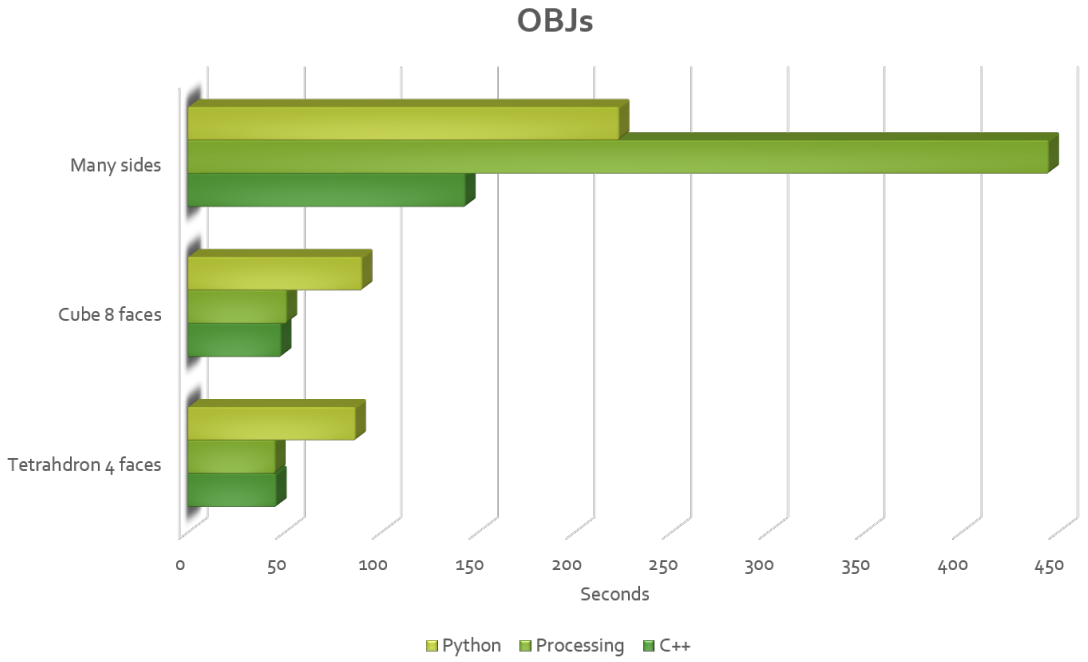

Figure 4.8: Basic Performance Graph filler

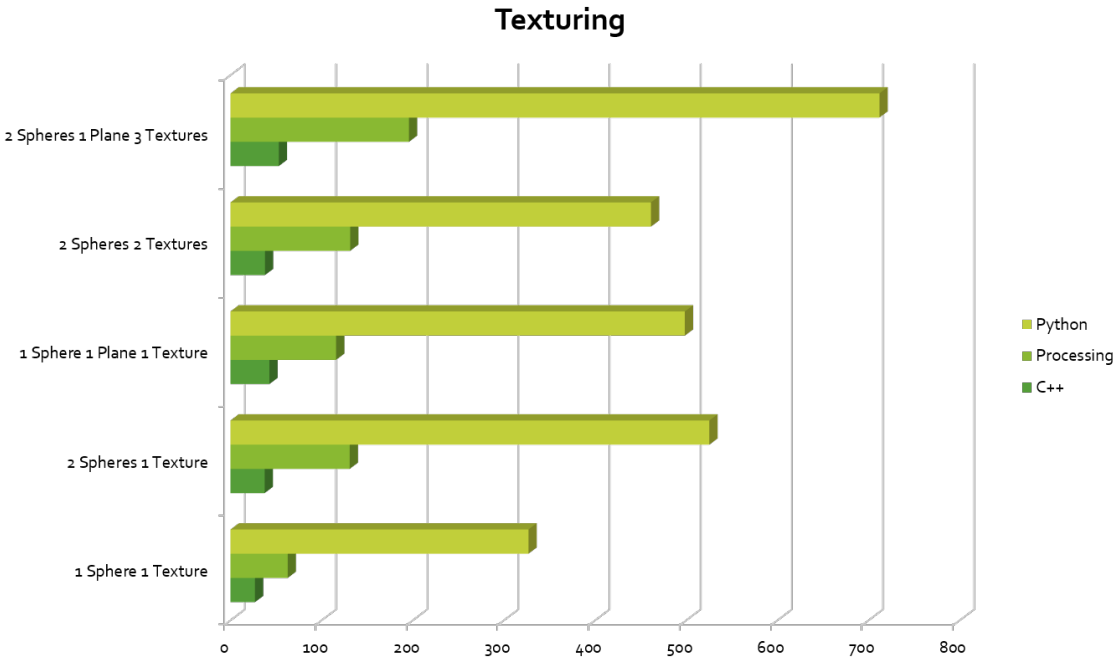

Figure 4.9: Basic Performance Graph filler

5. RESULTS AND CONCLUSIONS

Text goes here [**?**].

5.1 Computing Speed Graphs

5.2 Conclusions

5.2.1 C++

5.2.2 Processing

5.2.3 Python

5.2.4 Renderman

5.3 Another Section

Text between the figures. Text between the figures. Text between the figures. Text between the figures. Text between the figures. Text between the figures. Text between the figures. Text between the figures. Text between the figures. Text between the figures. Text between the figures. Text between the figures.

5.3.1 Subsection

5.3.2 Subsection

A table example is going to follow.

5.4 Another Section

Figure 5.1: TAMU figure

Table 5.1: This is a table template

Product	1	2	3	4	5
Price	124.-	136.-	85.-	156.-	23.-
Guarantee [years]	1	2	-	3	1
Rating	89%	84%	51%		45%
Recommended	yes	yes	no	no	no

REFERENCES

[1] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977.

[2] James F Blinn. Simulation of wrinkled surfaces. In *ACM SIGGRAPH Computer Graphics*, volume 12, pages 286–292. ACM, 1978.

[3] James F Blinn and Martin E Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976.

[4] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[5] Edwin Catmull. A subdivision algorithm for computer display of curved surfaces. Technical report, DTIC Document, 1974.

[6] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 447–452. ACM, 1998.

[7] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18(3):213–222, January 1984.

[8] Henri Gouraud. Continuous shading of curved surfaces. *Computers, IEEE Transactions on*, 100(6):623–629, 1971.

[9] Henrik. Ray tracing diagram.

[10] John F Hughes, Andries van Dam, Steven K Feiner, Morgan McGuire, and David F Sklar. *Computer graphics: principles and practice*. Pearson Education, 2013.

[11] Kri. Gouraud low.

[12] Richard F Lyon. Phong shading reformulation for hardware renderer simplification. *Apple Computer*, 43, 1993.

[13] Lee Markosian, Michael A Kowalski, Daniel Goldstein, Samuel J Trychin, John F Hughes, and Lubomir D Bourdev. Real-time nonphotorealistic rendering. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 415–420. ACM Press/Addison-Wesley Publishing Co., 1997.

[14] Steve McConnell. *Code complete*. O'Reilly Media, Inc., 2004.

[15] David R Naugler. C# 2.0 for c++ and java programmer: conference workshop. *Journal of Computing Sciences in Colleges*, 22(5):1–1, 2007.

[16] Addy Ngan, Frédo Durand, and Wojciech Matusik. Experimental validation of analytical brdf models. In *ACM SIGGRAPH 2004 Sketches*, page 90. ACM, 2004.

[17] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[18] Scott D Roth. Ray casting for modeling solids. *Computer graphics and image processing*, 18(2):109–144, 1982.

[19] Bjarne Stroustrup. A history of c++: 1979–1991. In *History of programming languages—II*, pages 699–769. ACM, 1996.

[20] Guido van Rossum. Personal history- part 1, cwi. http://python-history.blogspot.com/2009/01/personal-history-part-1-cwi.html, 2009.