The Creation and Illumination of 3D Scenes

Leon Johnson
963653

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Bachelor of Science

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

August 1, 2021

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..... (candidate)

Date .....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... (candidate)

Date .....

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date .....

I would like to dedicate this work to the family and the discord boys (FaZe Clan) for pushing me everyday to become the best Dota 2 player in Pontlliw.

R.I.P. Kobe Bean Bryant and Gianna Maria-Onore Bryant

Abstract

This work looks to explore the fundamentals of 3D graphics and lighting, and to implement a program exhibiting these fundamentals. The work will analyse the deployed techniques, and compare them to similar projects. This piece also briefly explores some modern techniques in the field of 3D graphics to discover the direction that the research field is headed.

Acknowledgements

Huge appreciation to my Mom for supporting me and ensuring my health and happiness to this point. Appreciation to Dr. Ulrich Berger for his support and patience during the process of this work-piece. Special thanks to Karen Baker and Pat Jagus for my previous education. Thanks to everyone and everything I have encountered in my life.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivations

Naturally, people preponderantly rely on vision to receive information about our surrounding world. As a consequence, art to model and capture scenes in our world have been created in any medium possible. The invention of computer graphics has provided a medium capable of the high-quality synthesis of scenes in our world, as well as the ability to model alternative realities. Advancement in computer graphical science and the growth of computational power has led to breakthroughs in the field; such as the ability to interact with the created environment. Evolution in computer graphical systems has enabled simulations to be implemented with increasing accuracy with minimized approximations.

At a fundamental level, for humans to visualise something, light emitted from sources and reflected from the world's surfaces have to be received by the retina of the eye [1]. This principle is replicated in computer graphical systems to receive an image of scenes in our virtual world. Properties of light can be divided into a multitude of categories, but structurally, light can be directly emitted onto an object to provide its received color, known as direct light. Alternatively, light can interact with diffuse surfaces in our world distorting its angle and color [2], among other factors; this behaviour is known as indirect light. The amalgamation of these properties in a world is known as Global Illumination. The light that propagates around the scene and is then received by a defined camera/eye viewpoint of the world is then the image that is displayed.

Global Illumination can be achieved using combinations of direct and indirect lighting algorithms. Simulating real-world lighting behaviours is practically impossible regarding the constraints of computational power. Consequently, these lighting algorithms are often tailored to suit the requirement of the system. Software yielding computational power in other areas will have lighting algorithms that use additional approximations – though more approximations results in lesser quality.

An example field of interest of which this applies is the video game industry – game studios attempt to generate the truest graphical output possible within time bounded restrictions. The more expensive the calculations, the slower the rendering speed of each frame. Target frame rates are dictated by the hardware the game/software deploys on [3]. Attaining a frame rate of at least 30 frames per second (FPS) whilst maintaining high graphical quality is a formality for modern gaming platforms, this number is recently tending towards 60 frames per second without a drop off in visual performance.

> "Many of today's popular games include entire outdoor environments and making these environments realistic and fast is a challenge for even the best programmers." [4]

1.1.1 Objective

In this document we explore the rendering 3D objects to a 2D display. In addition, this project touches methods of manipulating 3D objects within our constructed world and the world to achieve the desired visual effects. Furthermore, we will illuminate our world using a multitude of techniques. Implemented algorithms are to be measured with regard to performance, and possible optimisations for inefficient solutions will be considered.

More concretely the objectives for this project are as follows:

- **Implement a program capable of modelling and rendering 3D scenes**

  These scenes will be constructed of basic objects such as squares and spheres, so they can be easily defined and manipulated.

- **Manipulate the constructed 3D scene**

Implement methods of manipulating our constructed world, and individual objects within them.

- **Deploy variable lighting algorithms within the 3D world**

  Exploring different algorithms to generate the effects of illumination consistent across the whole scene, using one or multiple light sources.

- **Measure and analyse computational expense of rendering each scene**

  Record and analyse the performance of each rendering, to determine the efficiency of the rendering techniques and lighting algorithms used.

- **Explore alternative techniques for simulating lighting in the 3D environment**

  Research modern techniques for delivering the effects of ligting in 3D scenes.

Chapter 2

Background Reading

This chapter details and reviews the key literature used to gain knowledge of topics this paper explores, and to develop a foundation for the project.

2.1 3D Math Primer for Graphics and Game Development

Arguably the most essential subject area of this project is 3D Mathematics, therefore it is essential to have an understanding of 3D math to allow us to define, construct and manipulate our scene.

"3D Math Primer for Graphics and Game Development" [5] by *Fletcher Dunn* and *Ian Parberry* is a book that explains and demonstrates key concepts of mathematics related to graphics.

This book is written with an educational objective, where topics are often followed by examples or exercises to reinforce information. Although the book in parts can be perceived as informal, it effectively enables you to develop a clear understanding of some advanced concepts.
A chapter of extreme relevancy to this project is *Chapter 15 - 3D Math for Graphics* which applies the math discussed in preceding chapters to 3D graphics. Notable sections include *15.3 Coordinate Spaces* which explains the benefits of using multiple coordinate spaces for different calculations and *15.4 Lighting and Fog* which briefly explores the mathematics behind basic lighting algorithms.

2.2 GPU Gems Series

GPU Gems is a 3-book series published by world leading computer graphics company NVIDA Coorpation. The books explain how to fully utilize their GPUs, and also explains computer graphics theory. Large portions of this book series contain information beyond the scope of this project, but many chapters comprehensively cover important notions of 3D graphics.

Chapter 28. Graphics Pipeline Performance of the first iteration of *GPU Gems* [6] explores the graphical pipeline with respect to hardware, providing a low level perspective of the steps required to render 3D scenes.

Another notable chapter would be *Chapter 14. Dynamic Ambient Occlusion and Indirect Lighting* from *GPU Gems 2* [2] which explores lighting techniques and provides examples of code implementing these techniques.

2.3 songho.ca

The website *songho.ca* [7] created by Song Ho Ahn is a collection of webpages that concisely explains an array of important topics on 3D mathematics and 3D graphics, specifically using C++ *(See Section 3.1.1)* and OpenGL *(See Section 3.2.1)*.

Chapter 3

Tools

In order to accomplish the project goals in the provided time, I had to consider the best tools to complete the solution.

3.1 Base Language

Our project will be constructed using C++. Although I posses greater proficiency in Java, I concluded that C++ was far superior to complete this task. Its efficient optimizations for native code generation and seamless integration's with graphical APIs make it a perfect fit.

3.1.1 Brief Introduction to C++

C++ was developed by Bjarne Stroustrup in 1979. The language is a robust extension of the powerful language C.

> "The C++ language features most directly support four programming styles:
>
> - Procedural programming
> - Data abstraction
> - Object-oriented programming
> - Generic programming" [8]

These paradigms, especially Object-oriented will provide us with superfluous features to complete this task.

Despite its age, C++ remains one of the worlds leading languages due to its robustness and its ability to enable low-level access to memory and registers. This provides the programmer a large amount of control on how the software interacts with the target machines – especially applicable in the video game and graphical development industries.

3.2 Graphical API

Graphical APIs provide features to communicate with graphical hardware. For this project the program should be able to run on a multitude of machines, so the API needs to be able to deploy on multiple platforms. The field leading APIs are DirectX, Vulcan and OpenGL. For this system OpenGL will provide more than enough control to implement the specification and is a lot simpler to assimilate with C++ than its some of its company in the graphics API field.

3.2.1 Brief Introduction to OpenGL

Open Graphics Library is an "industry-standard, cross platform Application Programming Interface (API)" [9] developed and supported by Silicon Graphics Inc. in 1992.

OpenGL is constantly evolving in parallel with modern GPUs. Its library enables the drawing of 2D and 3D graphics and contains a large amount of extension utilities for creating software.

3.3 Shader Language

Shaders are an essential part of the *graphical pipeline (See Section 4.1)*. These shaders are programmable using shader languages. It is important that our choice of language can be integrated with the other languages used, especially the Graphical API.

3.3.1 Brief Introduction to GLSL

OpenGL Shader Language (GLSL) is OpenGL's accompanying shader language. It was created to give developers greater control over the *graphics pipeline (See Section 4.1)* - its natural link to OpenGL makes it an obvious choice. GLSL has very similar syntax to the C programming language making at a great fit for this project.

3.4 Other Libraries and APIs

To produce a complete program, other libraries will need to be linked into the program to enable the implementation of features that would be cumbersome and time consuming to develop.

3.4.1 A Brief Introduction to GLM

OpenGL Mathematics is a C++ mathematics library, based on the specifications of GLSL [10]. The library contains key mathematical elements relavent to our project, most notably matrices and their associated operations as well as vectors.

3.4.2 A Brief Introduction to GLFW

Graphics Library Framework is a lightweight cross-platform windowing and input handling tool. It enables the developer to create and handle windows, as well as the ability to receive input from keyboard and mouse.

Chapter 4

Brief Introduction to 3D Graphics

The following chapter acts as an introduction on the concepts and techniques required to develop a 3D graphics program. The content in this chapter is included and supported in our graphics library, however it is essential to have a competence in this field to understand the functions of the library to correctly integrate them into the systems implementation.

4.1 Graphical/Rendering Pipeline

In order to render a 3D scene to a 2D display there are a series of computations required to be executed, known as the *Graphical Pipeline*. Named pipeline due to the nature in which output of a phase will be used as the input for the next phase. Different GPUs and API's can have slightly different pipeline models, however the general outline remains constant.
For this project, we will be rendering using OpenGL's rendering pipeline:

- Vertex Specification – Vertices are defined as arrays of coordinates.
- Vertex Processing – Each vertex retrieved from the vertex array is processed by a Vertex Shader, and dependant on the primitives chosen, a Geometry Shader is also used.
- Vertex Post-Processing – Transforms the results of the Vertex Processing stage.
- Primitive Assembly – Primitives are separated into sequences of base primitives.
- Rasterization – Primitives are further deconstructed into discrete fragments.
- Fragment Shading – Each fragment is processed.

- Per-Sample Processing – A series of tests to determine how the processed fragments are written to various buffers. These tests include:
  - Depth Testing: Test that determines the value that is stored inside the depth buffer.
  - Stencil Testing: Test used to determine fragments that require to be culled.
  - Blending: Test that takes fragment colors and combines them with colors currently stored in the color buffers.

4.2 Vectors

Vectors are fundamentally directions with a magnitude. They are used all around a 3D graphics program. Most commonly applied as light rays or general rays cast through the scene.

Figure 4.1 displays how vectors hold values irrespective of their position. The vectors in this figure are represented in 2D space but we could easily imagine these vectors to be in 3D space with a z-coordinate of 0.

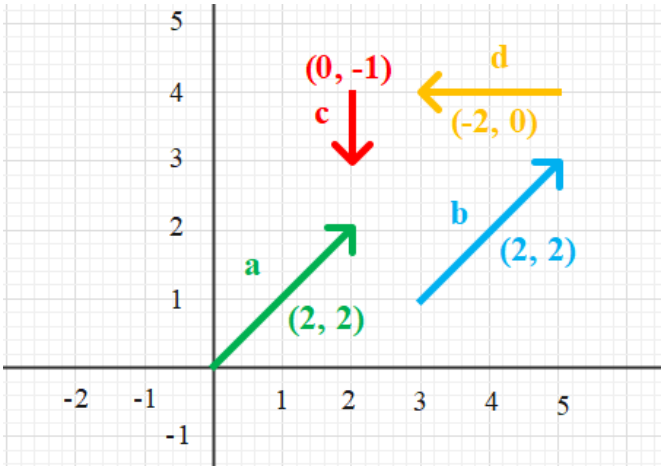


Figure 4.1: Image of 4 vectors in 2D space

Vectors need to be manipulated mathematically to achieve some of the phenomena we wish to mimic in our 3D scene. The following subsections demonstrate important manipulations of vectors which we will use in our program and shaders.

4.2.1 Length

To calculate the length of a vector, which can be interpreted as the vectors' magnitude, we apply Pythagoras' theorem $||\bar{a}|| = \sqrt{x^2 + y^2}$, where $||\bar{a}||$ represents the length of a vector $\bar{a}$, and where x and y represent the vectors' x-coordinate and y-coordinate respectively.

The length of vectors is used in many common vector calculations so it is important we know how to obtain this value.

4.2.2 Unit Vectors

Another important vector concept is that of a unit vector. A unit vector is a normalised vector, in which its length is 1.

A vector $\bar{a}$ has its unit vector denoted as $\hat{a}$, and it can be derived by dividing each vector component by its length.

4.2.3 Cross Product

The equation for calculating the cross product between vectors is only defined in 3D space with 2 input vectors that are not parallel. The result of the algorithm is a vector orthogonal to both input vectors, as depicted in *Figure 9.4*.

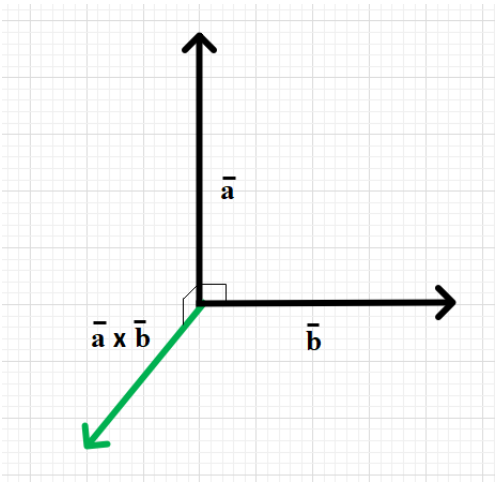


Figure 4.2: Image displaying cross product between 2 vectors

4.2.4 Dot Product

The formula to calculate the dot product of two vectors is $\bar{a} \cdot \bar{b} = ||\bar{a}|| \cdot ||\bar{b}|| \cdot cos\theta$. However, if the lengths of $\bar{a}$ and $\bar{b}$ are both 1, i.e. unit vectors *(See Section 4.2.2)* then we can simplify this equation to $\hat{a} \cdot \hat{b} = cos\theta$. This is a useful property, making dot products extremely easy to calculate within algorithms in our program, particularly for lighting algorithms.

4.3 Matrices

Matrices can be visualised as $n \times m$ array of numbers or mathematical expressions. Similarly to *Vectors (See Section 4.2)*, matrices have a useful set of associated equations and operations applicable to the field of 3D graphics.

4.3.1 Transformation Matrix

In 3D graphics, transformations matrices are critical for manipulating objects. Transformation matrices are matrices designed to be multiplied to an original matrix or vector to achieve some desired effect.

4.3.1.1 Identity Matrix

An identity matrix is an $n \times n$ matrix which contains all 0's except for its diagonal. Multiplying a vector to an identity matrix would leave the vectors values unchanged, this is due to a property of matrix multiplication.

4.3.1.2 Scaling

A scaling matrix is a matrix capable of growing or shrinking the values in a vector.

4.3.1.3 Translation

A translation matrix, is a matrix responsible for moving the position of the original vector by adding or subtracting values from it.

4.3.1.4 Rotation

A rotation matrix uses linear algebra to perform rotations. For 3D rotations, each axis x, y, z has its own predefined rotation matrix, and the passed in angle to the matrix will affect its degree of rotation.

4.4 Coordinate System

Throughout the graphical pipeline, objects and the world are transformed *(See Section 4.3.1)* into different states, know as coordinate spaces. It is easier to rotate and move a single object, when we focus on just that object, rather than viewing that object along with the rest of the scene. There are specific matrices *(See Section 4.3)* that allow us to convert to these coordinate spaces.

4.4.1 Object Space

This coordinate space, is specific to an object, and is the coordinate space we will likely define our objects in. It is convenient to perform transformations *(See Section 4.3.1)* of our vertices in this coordinate space.

4.4.2 Scene/World Space

This space is where the true location of all objects are defined in the scene relative to to each other.
Coordinates in object space are converted to world space by using a model matrix.

4.4.3 Camera Space

Aptly named camera space, this is the perspective in which the scene will be visible from, as if a camera is being used to view the scene.
To convert the scene to world space to camera space, a camera/view matrix is used to transform the coordinates of the vertices.

4.4.4 Clip Space

In the graphical pipeline *(See Section 4.1)* after each iteration of the vertex shader is execution, coordinates are to be inside a pre-defined range by OpenGL- coordinates that fall outside of

this range are clipped, leaving only the remainder of coordinates to be visible.
Conversion from camera space to clip space is done using a projection matrix.

Chapter 5

Brief Introduction to Lighting

In order for objects in a scene to be visible and have color, there has to be light present. This section serves as an introduction of some illumination topics important to the project.

5.1 Ambient Lighting

Ambient lighting is a fundamental model of global illumination. To deploy this method, a constant is applied to the objects in the scene. The ambient color of a scene is determined by taking the lights color and multiplying it by a constant ambient factor. This will provide objects in the world a base color.

5.2 Diffuse Lighting

Diffuse lighting is a method of direct lighting, based on the premise that objects illumination intensity is affected by how close its surface is aligned to light rays from the source. Using the surface normal and shooting a ray from the light source to an object, you can calculate the 'angle of incidence' between the two vectors.

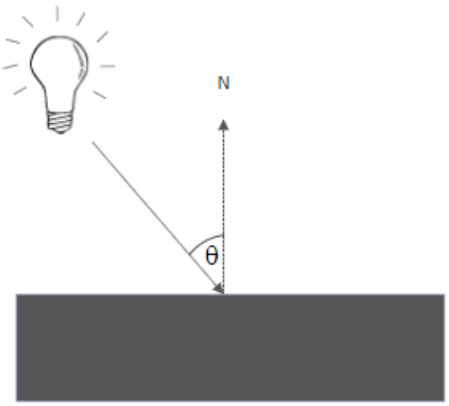


Figure 5.1: Diffuse Lighting Model

5.3 Specular Lighting

Specular lighting considers the reflective element of light. Alike diffuse lighting, this method also utilises the lights direction vector, and the objects normal vector; however, in addition, specular accounts for the view direction of the camera. The light direction vector is reflected about the normal vector of surface it hits – using this new reflection vector, the angle between the reflected vector and the camera direction vector is computed. These reflection vectors are calculated using the dot product *(See Section 4.2.4)*. The result of this calculation will then be scaled by a 'specular' factor based on the objects material to determine the intensity of the light received.

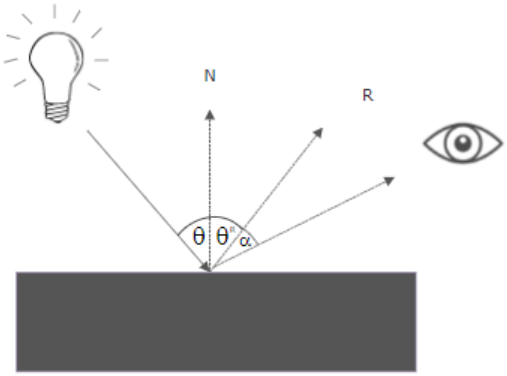


Figure 5.2: Specular Lighting Model

5.4 Phong Shading Model

The combination of the three above lighting elements construct what is known as the Phong-Shading model. Describing light as the amalgam of diffuse reflection of rough surfaces and reflected light from shiny surfaces, factored by a global ambient lighting constant. The model "makes use of the fact that, for any real surface, more light is reflected in a direction making an equal angle of incidence with reflectance" [11] – making its visual effects more prominent when modelling curved surfaces compared to its shading rivals that don't facilitate for angles of incidence. This calculation is performed in the Fragment shader within the shader program.

5.5 Blinn-Phong

An extension to the Phong shading model was proposed by James Blinn in 1977. This revised model uses an alternative approach to the specular component of the technique [12]. This technique uses a unit vector commonly named the 'halfpoint/halfway' vector which is a calculated half point between the light direction vector and the view vector. This is to correct a visual error in the original model when the angle between the view vector and reflection vector exceeds 90 degrees.

5.6 Gouraud Shading

Older implementations of lighting to 3D scenes used to perform Phong Shading in the vertex shader rather than in the fragment shader. This is efficient as a result of there typically being much less vertices compared to fragments, resulting in less iterations of the expensive lighting calculations. The consequence of this however is less realistic results, as essentially less samples are being taken for each calculation run.

Chapter 6

Method

6.1 Plan

In order to implement the program a top-down design approach was used, to modularise elements of the system into smaller components to make the overall assembly of the system more intuitive. In order to conduct this design method I had to identify each core component of the system, and recursively break them down into smaller pieces until they were atomic enough to implement.

The below image depicts the system in the top-down design model.

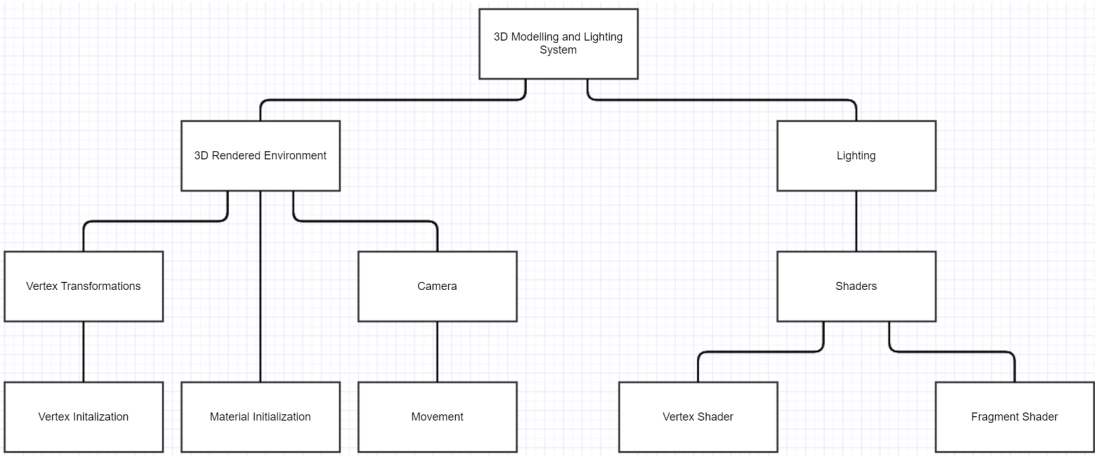


Figure 6.1: Top Down Design Model of System

Chapter 7

Implementation

The following sections within this chapter detail how components depicted in *Figure 6.1* were implemented and combined to complete the program. Only important functionality of the program have been included in this document.

7.1 3D Rendered Environment

This section concerns the elements of the program regarding 3D modelling and rendering, implementing and elaborating on topics introduced in *Chapter 4* and *Chapter 5.3*.

7.1.1 Vertex Transformation

In order to define a shape, we must define its vertices. For our project, we are also using lighting calculations on our shape(s) so along with the vertex definition, we must also make definitions of normals in order for us to perform our lighting calculations further in the program. These were defined in 3D space, thus possessing x, y and z coordinates.
We will be using a cube as our shape for our program, in which these vertices can be defined once, and used to draw two cubes, our main cube, and a cube representing the light source. Once defined, these vertices are stored in buffers and array data structures provided by OpenGL. These structures allow us to easily manipulate these vertices and perform transformations.

The vertices were manipulated using transformed using matrices *(See Section 4.3)* and converted to various coordinate spaces *(See Section 4.4)* to change position and size of our

cube - these matrices are defined and then passed to our shader program *(See Section 7.2)* before drawing our vertices to the screen (See

7.1.2 Materials

For the system, we defined multiple materials, each with differing ambient, diffuse and specular components to be used in the lighting shaders *(See Section 7.2)*. These material definitions attempt to mimic real-life materials and their colour properties. The component values were provided by OpenGL's Teapot Demo [13].

```
1  //----GOLD MATERIAL----
2  glm::vec3 GOLD_AMBIENT(0.24725f, 0.1995f, 0.0745f);
3  glm::vec3 GOLD_DIFFUSE(0.75164f, 0.60648f, 0.22648f);
4  glm::vec3 GOLD_SPECULAR(0.628281f, 0.555802f, 0.366065f);
5  glm::vec1 GOLD_SHININESS(0.4f);
6
7  //----SILVER MATERIAL----
8  glm::vec3 SILVER_AMBIENT(0.19225f, 0.19225f, 0.19225f);
9  glm::vec3 SILVER_DIFFUSE(0.50754f, 0.50754f, 0.50754f);
10 glm::vec3 SILVER_SPECULAR(0.508273f, 0.508273f, 0.508273f);
11 glm::vec1 SILVER_SHININESS(0.4f);
12
13 //----BRONZE MATERIAL----
14 glm::vec3 BRONZE_AMBIENT(0.2125f, 0.1275f, 0.054f);
15 glm::vec3 BRONZE_DIFFUSE(0.714f, 0.4284f, 0.18144f);
16 glm::vec3 BRONZE_SPECULAR(0.393548f, 0.271906f, 0.166721f);
17 glm::vec1 BRONZE_SHININESS(0.2f);
18
19 //----PEARL MATERIAL----
20 glm::vec3 PEARL_AMBIENT(0.25f, 0.20725f, 0.20725f);
21 glm::vec3 PEARL_DIFFUSE(1.00f, 0.829f, 0.829f);
22 glm::vec3 PEARL_SPECULAR(0.296648f, 0.296648f, 0.296648f);
23 glm::vec1 PEARL_SHININESS(0.088f);
24
25 //----EMERALD MATERIAL----
26 glm::vec3 EMERALD_AMBIENT(0.0215f, 0.1745f, 0.0215f);
27 glm::vec3 EMERALD_DIFFUSE(0.07568f, 0.61424f, 0.07568f);
28 glm::vec3 EMERALD_SPECULAR(0.633f, 0.727811f, 0.633f);
29 glm::vec1 EMERALD_SHININESS(0.6f);
```

Listing 7.1: Example of Material Definitions within the program

7.1.3 Camera

The camera/eye of the system is the viewpoint that the user gets to experience the 3D scene. This was implemented in a way that would enable the user to not only move around, but to look around the scene. In order to do this, not only positional translation was required, but also angular translation of the camera. Using the facilities that GLFW *(See Section 3.4.2))* provides, keyboard input and mouse input was used to invoke movement in the system.

7.1.3.1 Directional Movement (Keyboard Input)

Directional movement within the system is input via a keyboard peripheral. Using keys W, A, S, D to move Up, Left, Down, Right respectively.

At the conclusion of each iteration of the game loop *(See Section)* inputs are polled. However the rate in which this loop is iterated is dependant on the hardware.

For example, imagine two computers named A, and B, if A is twice as fast as computer B, then computer A iterates the loop two times more often, and therefore polls inputs twice as frequent. This results in computer A moving twice as fast as computer B. This is a very detrimental side effect regarding consistency of how our program performed. To avert this, we use "delta time" which is the difference in time between the current and last frame. This value is then multiplied to the desired movement speed to calculate a true, consistent velocity. This technique is common in video games to ensure that the game running consistently on different hardware to provide fairness. The code listing below displays how this variable is calculated.

```
1 //Delta time calculation
2 float live_frame = glfwGetTime();
3 delta_time = live_frame - last_frame;
4 last_frame = live_frame;
```

Listing 7.2: Implementation of delta time

Now that we have a consistent measure for polling inputs, we can now implement our movement knowing it will be executed consistently on different hardware. The below segment of code demonstrates how positional movement of the camera was performed.

```
1 /*
```

```
2   * Recieves input commands from the keyboard. Takes in a direction specified by
        Movement ENUM, and the delta time value
3   * to ensure movment speed is consistent across different hardware.
4   */
5  void process_keyboard(Movement direction, float delta_time)
6  {
7    //Use the product of the preset movement speed, and the delta time to generate
         true velocity.
8    float true_velocity = movement_speed * delta_time;
9
10   //Translate position by the true velocity respective to the input direction.
11   if (direction == FORWARD)
12     this->pos += front * true_velocity;
13   if (direction == BACKWARD)
14     this->pos -= front * true_velocity;
15   if (direction == LEFT)
16     this->pos += left * true_velocity;
17   if (direction == RIGHT)
18     this->pos -= left * true_velocity;
19 }
```

Listing 7.3: Method implementing camera positional translation

7.1.3.2 Angular Movement (Mouse Input)

To enable the user to look around, the camera must be able to have its viewing angles altered. This was implemented using Euler Angles, which contain yaw, pitch and roll values which represent orientations in x,y and z axis of a coordinate space. Offsets in the mouse position along the *x* and *y* axis are added to the current yaw and pitch values of the camera respectively. After this calculation the yaw and pitch values are used to calculated new updated normals for the camera.

The below two code listings displays the methods handling the mouses positional offset and also the method updating camera normals:

```
1  //Translate the yaw and pitch of the camera relative to offsets in x and y using
       the mouse last and current location.
2    void process_mouse_movement(float x_offset, float y_offset)
3    {
4      x_offset *= mouse_sensitivity;
5      y_offset *= mouse_sensitivity;
6
7      yaw += x_offset;
```

```
8       pitch += y_offset;
9
10      //Stops the screen from fliping the world direction, so contrains pitch to <
            90 degrees.
11      if (pitch > 89.9f)
12        pitch = 89.9f;
13      if (pitch < -89.9f)
14        pitch = -89.9f;
15
16      update_camera_vectors();
17    }
```

Listing 7.4: Method implementing camera angular translation

```
1   //Updates the cameras front,left and up vectors using the cameras newly generated
        Euler angles.
2     void update_vectors()
3     {
4       glm::vec3 new_front;
5
6       new_front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
7       new_front.y = sin(glm::radians(pitch));
8       new_front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
9       front = glm::normalize(new_front);
10
11      left = -glm::normalize(glm::cross(front, true_up));
12
13      up = glm::normalize(glm::cross(-left, front));
14    }
```

Listing 7.5: Method updating vectors of the camera

7.2 Lighting

Implementing lighting is commonly done using shaders, following the *Graphical Pipeline (See Section 4.1)* we will be using two shaders, a vertex and a fragment shader per lighting technique.

The lighting techniques implemented in the system are listed as follows:

- Phong Shading *(See Section 5.4)* ‘
- Blinn-Phong Shading *(See Section 5.5)*

- Gouraud Shading *(See Section 5.6)*

7.2.1 Phong Shading (Fragment Shader)

The most important calculations for Phong reside in the fragment shader, this shader makes use of the mathematics and logic discussed in *Section 5.4* - the implementation of this shader is provided in A.2 within this documents appendix.

7.2.2 Blinn-Phong Shading (Fragment Shader)

The specular component of Blinn-Phong is the only differing component between itself and regular Phong Shading, the implementation of the Blinn-Phong specular compment is implmeneted as follows:

```
1 //Specular component.
2
3 vec3 view_direction = normalize(view_pos - fragment_shader_input.fragment_position
      );
4 vec3 half_direction = normalize(light_direction + view_direction);
5 float spec = pow(max(dot(norm, half_direction), 0.0), material.shininess * 2); //
      Double shininess to get results closer to regular phong.
6 vec3 specular = (spec * material.specular) * light.specular;
```

Listing 7.6: Blinn-Phong Specular Component (Fragment Shader)

7.2.3 Gouraud Shading (Vertex Shader)

Unlike the previous two techniques, Gouraud shading performs its important lighting calculations in the vertex shader rather than the fragment shader. The calculations themselves are almost identical to that of Phong Shading but iterated fewer times, as there are lesser vertices than fragments. The implementation of the vertex shader is provided as A.3 in the appendix.

7.3 Program Overview

The above sections are placed within a game loop (render loop) which results in a newly rendered frame each iteration, resulting in a dynamic world which reacts to changes within it. The variables of the system are toggle-able via input handlers, where materials, shader type

and movement can be input. The main method of the program is as displayed below.

```
int main()
{
  //Initalize the window and window handlers.
  init_GLFW();

  //Initialize open gl settings
  init_open_gl_settings();

  //Initialize the vertex buffers with their data.
  prepare_buffers();

  //Run the render loop.
  game_loop();


  //Free the OpenGL buffers.
  glDeleteVertexArrays(1, &cube_vao);
  glDeleteVertexArrays(1, &light_vao);
  glDeleteBuffers(1, &vbo);


  glfwTerminate();
  return 0;
}
```

Listing 7.7: Main method of program

Chapter 8

Measurables

In order to compare the most efficient method there were multiple options available. One method to consider was to analyse the algorithms and calculator its algorithmic complexity using *O* notation. Another method is to use our delta time variable (time differential between current and last frame) to determine how quick each frame is being rendered. A problem with using delta time is that the results are not particularly readable. However delta time can be extended to calculate how many frames are rendered in a second. The number of frames per second can then be used to see the time it takes to draw one frame averaged on one second, the unit returned will be in milliseconds. This was implemented in the manner displayed below.

```
1 float live_frame = glfwGetTime();
2 num_frames++;
3
4 if (live_frame - last_time >= 1.0)
5 {
6   std::cout <<  (double)1000.0/(double)num_frames << std::endl;
7   num_frames = 0;
8   last_time = live_frame;
9 }
```

Listing 8.1: Implementation of frames per second counter (ms)

The visual outputs of the program will be compared to example renderings of the same technique to evaluate the accuracy of the output.

Chapter 9

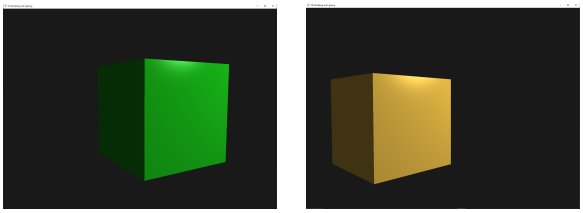
Results

This chapter looks at the results from the project, both the visual output and performance of the constructed system and the results from the measurables detailed in *Chapter 8*.

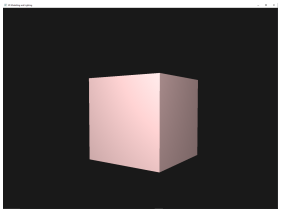
9.1 Visual Results

9.1.1 Materials

The below images display the results of different materials being rendered with our cube in the world.



A. Emerald B. Gold



C. Pearl

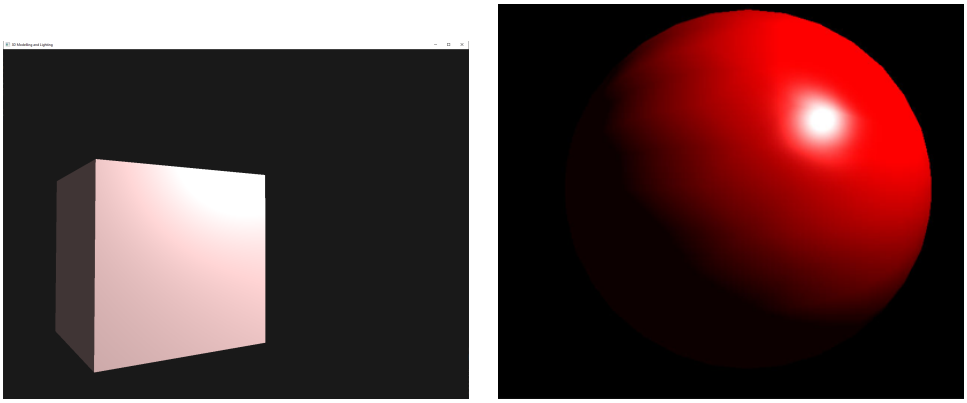
Figure 9.1: Rendering of variable materials within the system.

As can be seen from the above images, the program was capable of successfully rendering the cube object. This suggests not only the methods of rendering are correct, but also indicates that at least the ambient component of the lighting shader is functioning as colour is visible.

9.1.2 Lighting

9.1.2.1 Phong

The below images display the result of a pearl cube being rendered using the Phong shading program, along with an example Phong rendering of a sphere.



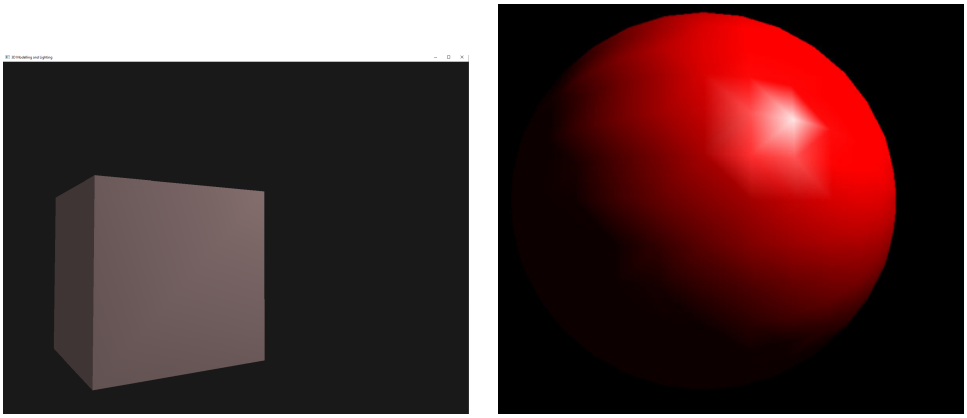
A. My Phong rendered cube B. Example Phong rendered sphere

Figure 9.2: Phong shading comparison of my rendered cube vs an Phong shading example

As can be seen from the above image, my rendering does display circular specular blemishes like the rendered example. It also exhibits diffuse darkening as the cube face that is hidden from the light source is significantly darker than that of the face that is positioned towards the light source. This is a similar result to that of the example.

9.1.2.2 Gouraud

Below is the result of a pearl material cube rendered using the Gouraud shading program. Besides it is an example rendering of a sphere deploying a Gouraud shader.



A. My Gouraud rendered cube

B. Example Gouraud rendered sphere

Figure 9.3: Gouraud shading comparison of my rendered cube vs a Gouraud shading example

Comparing my output to the example rendered sphere, I was disappointed with the lack of specular blemish displayed. The ambient component of the pearl seems to be slightly darker than expected but there is some diffuse effects present as the away facing cube-face is darker than that of the forward face.

Looking closer at the my rendering, it is faintly visible that it exhibits the same polygonal shape as seen in the example. Showing that some aspects of the shader are correct. This is visible in the below image, where the outline of the shape is marked for clarity.

Figure 9.4: Zoomed Gouraud Shaded Cube displaying rendering pattern

Despite this, I am unsatisfied with the output from the rendering using the Gouraud shader program.

9.1.2.3 Blinn-Phong

The below figure shows the output of a pearl cube rendered with a Blinn-Phong shader active.

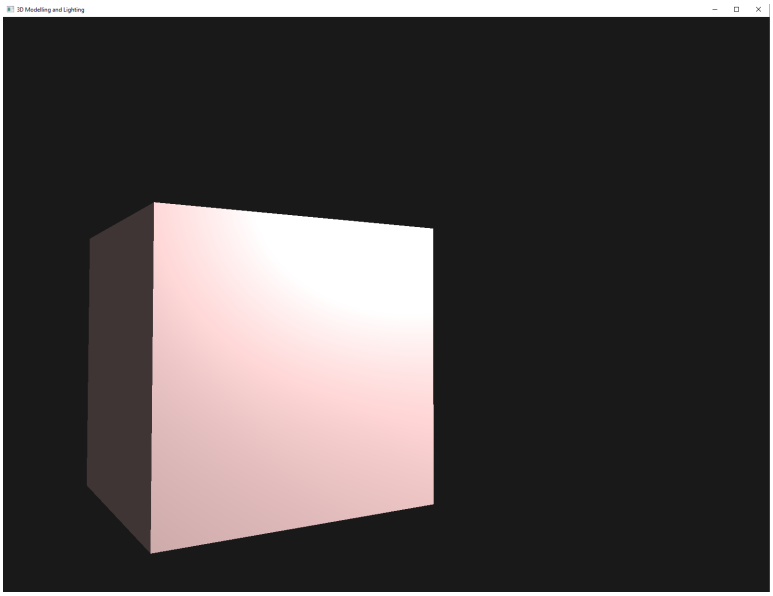


Figure 9.5: My Blinn-Phong rendred cube

Although there is no comparative rendering to mark my rendering against, I am confident that the blinn-phong shader is behaving as expected. It seems to be displaying the effects expected by the mathematics and compared to my rendering in *Figure 9.2 A.* it certainly removes the distinct 'cut-off' of the specular blemish and instead displays a more gradual descent of the specular highlight.

9.2 Measured Results

The below figures display the result of measuring the frame rates in milliseconds of the program whilst running the three different lighting methods.

A. Phong frame rate

B. Gouraud frame rate

C. Blinn-Phong frame rate

Figure 9.6: Results of Phong, Gouraud and Blinn-Phong frame rates respectively

As the results show, the frame rates are constant despite using variable shader programs. This is attributed to the program running to the maximum frame rate and being bounded due to hardware restriction, namely the frame rate of the target monitor. This makes the testing of efficiency between the three techniques inconclusive.

Chapter 10

Conclusions and Future Work

10.1 Overview

Overall, the results of the project were positive - all goals *(See Section 1.1.1)* specified for the project were achieved. The project goals are again listed below, with a brief description on how each goal was completed.

- **Implement a program capable of modelling and rendering 3D scenes**

  As evidenced in the results, we were successfully able to render a cube into the scene.

- **Manipulate the constructed 3D scene**

  The system enabled the use of a dynamic camera, which could be controlled via a keyboard and mouse, defined vertices (shapes) were also able to be translated in position and size in the scene. Variable materials were additional features in the program, where the user could toggle between the active material.

- **Deploy variable lighting algorithms within the 3D world**

  Three different lighting techniques were implemented. Two techniques were accurately implemented where the third technique was partially implemented.

- **Measure and analyse computational expense of rendering each scene**

  The system was able to record the frame rates when running each lighting shader program, however the results yielded were inconclusive as a result of the simplicity of the rendered scene.

• **Explore alternative techniques for simulating lighting in the 3D environment**

10.2 Future Work

In future, this project can be expanded to where variable shapes can be rendered, more advanced lighting algorithms are included, as well as an interactive user interface to aid the use of the system. An example of a lighting technique I would like to implemented in future is Radiosity, which approaches the concept of light as the transfer of energy rather than as a ray.

Fundamental understanding of rendering and lighting was gained during the construction of this system, serving as a useful introduction to the broad field of 3D graphics. With further provided time, the Gourard shader program would be amended, until a satisfactory result was obtained.

Another future goal would be to implement some of the modern rendering and lighting techniques discussed in *(See Section)*

Bibliography

[1] G. Westheimer and F. W. Campbell, "Light distribution in the image formed by the living human eye," vol. 52, no. 9, pp. 1040–1045, 1962.

[2] M. Bunnell, *Dynamic Ambient Occlusion and Indirect Lighting*. Nvidia Corporation, 2005.

[3] R. Hagström, "Frames that matter," pp. 9–11, 2015.

[4] G. Snook, *Real-Time 3D Terrain Engines Using C++ and DirectX 9*. Charles River, 2003.

[5] I. P. Fletcher Dunn, *3D Math Primer for Graphics and Game Development*. Wordware Publishing Inc., 2002.

[6] C. Cebenoyan, *Dynamic Ambient Occlusion and Indirect Lighting*. Nvidia Corporation, 2004.

[7] S. H. Ahn. (2005-2018) songho.ca. [Online]. Available: http://www.songho.ca

[8] B. Stoustrup, *The C++ Programming Language*. Addison-Wesley, 1985.

[9] R. J. R. et al., *OpenGL Shader Language*, 2004, vol. 3.

[10] (2020) Opengl mathematics. [Online]. Available: https://glm.g-truc.net/0.9.9/index.html

[11] J. F. Blinn, "Models of light reflection for computer synthesized pictures," vol. 11, no. 2, pp. 192–198, 1977.

[12] T. L. Linus Kallberg, "Optimized phong and blinn-phong glossy highlights," vol. 3, no. 3, pp. 1–6, 2014.

[13] S. G. Inc. (1993-1997) Opengl teapots demo. [Online]. Available: https://www.opengl.org/archives/resources/code/samples/redbook/teapots.c

Appendix A

Implementation Code Listings

A.1 Matrices definitions and drawing

```
1  //---Main Cube---
2  glm::mat4 projection_matrix = glm::perspective(glm::radians(camera.zoom), (float)
       WINDOW_WIDTH / (float)WINDOW_HEIGHT, 0.1f, 100.0f);
3  glm::mat4 view_matrix = camera.get_view_matrix();
4  glUniformMatrix4fv(glGetUniformLocation(lighting_shader.id, "projection"), 1,
       GL_FALSE, &projection_matrix[0][0]);
5  glUniformMatrix4fv(glGetUniformLocation(lighting_shader.id, "view"), 1, GL_FALSE,
       &view_matrix[0][0]);
6  glm::mat4 model_matrix = glm::mat4(1.0f);
7  glUniformMatrix4fv(glGetUniformLocation(lighting_shader.id, "model"), 1, GL_FALSE,
        &model_matrix[0][0]);
8  //Draw the main cube
9  glBindVertexArray(cube_vao);
10 glDrawArrays(GL_TRIANGLES, 0, 36);
11 //----Light Source Cube----
12 glUseProgram(light_source_shader.id);
13 glUniformMatrix4fv(glGetUniformLocation(light_source_shader.id, "projection"), 1,
       GL_FALSE, &projection_matrix[0][0]);
14 glUniformMatrix4fv(glGetUniformLocation(light_source_shader.id, "view"), 1,
       GL_FALSE, &view_matrix[0][0]);
15 model_matrix = glm::mat4(1.0f);
16 model_matrix = glm::translate(model_matrix, light_pos);
17 model_matrix = glm::scale(model_matrix, glm::vec3(LIGHT_SIZE_SCALE));
18 glUniformMatrix4fv(glGetUniformLocation(light_source_shader.id, "model"), 1,
       GL_FALSE, &model_matrix[0][0]);
19 //Draw the light source
20 glBindVertexArray(light_vao);
21 glDrawArrays(GL_TRIANGLES, 0, 36);
```

Listing A.1: Matricies definition and object drawing)

A.2 Phong Shader Program (Fragment Shader)

```
#version 460 core

out vec4 fragment_colour;

struct Light{
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};


in VERTEX_SHADER_OUTPUT
{
    vec3 normal;
    vec3 fragment_position;
} fragment_shader_input;

uniform vec3 view_pos;
uniform Material material;
uniform Light light;

void main()
{
    //Ambient Component
    vec3 ambient = material.ambient * light.ambient;

    //Diffuse Component
    vec3 norm = normalize(fragment_shader_input.normal);
    vec3 light_direction = normalize(light.position - fragment_shader_input.
        fragment_position);

    float diff = max(dot(norm, light_direction), 0.0);
```

```
40      vec3 diffuse =  (diff * material.diffuse) * light.diffuse;
41
42      //Specular Component
43      vec3 view_direction = normalize(view_pos - fragment_shader_input.
            fragment_position);
44      vec3 reflect_direction = reflect(-light_direction, norm);
45      float spec = pow(max(dot(view_direction, reflect_direction), 0.0), material.
            shininess);
46      vec3 specular = (spec * material.specular) * light.specular;
47
48      //Combination
49      vec3 result = ambient + diffuse + specular;
50      fragment_colour = vec4(result, 1.0);
51  }
```

Listing A.2: Phong Shading (Fragment Shader)

A.3 Gouraud Shader Program (Vertex Shader)

```
1   #version 460 core
2
3   layout (location = 0) in vec3 a_pos;
4   layout (location = 1) in vec3 a_normal;
5
6   struct Light{
7       vec3 position;
8       vec3 ambient;
9       vec3 diffuse;
10      vec3 specular;
11  };
12
13  struct Material {
14      vec3 ambient;
15      vec3 diffuse;
16      vec3 specular;
17      float shininess;
18  };
19
20  out VERTEX_SHADER_OUTPUT {
21
22      vec3 colour;
23
24  } vertex_shader_output;
25
26
```

```
27 uniform vec3 view_pos;
28
29 uniform Material material;
30 uniform Light light;
31
32
33 uniform mat4 model;
34 uniform mat4 view;
35 uniform mat4 projection;
36
37 void main()
38 {
39     gl_Position = projection * view * model * vec4(a_pos, 1.0);
40
41     vec3 pos = vec3(model * vec4(a_pos, 1.0));
42     vec3 normal = mat3(transpose(inverse(model))) * a_normal;
43
44     //Ambient component
45     vec3 ambient =  light.ambient;
46
47
48     //Diffuse component
49     vec3 norm = normalize(normal);
50     vec3 light_direction = normalize(light.position - pos);
51     float diff = max(dot(norm, light_direction), 0.0);
52     vec3 diffuse = diff * light.diffuse;
53
54     //Specular component
55     vec3 view_direction = normalize(view_pos - pos);
56     vec3 reflect_direction = reflect(-light_direction,norm);
57     float spec = pow(max(dot(view_direction, reflect_direction), 0.0), material.
           shininess);
58     vec3 specular = material.specular * spec * light.specular;
59
60     //Combination
61     vertex_shader_output.colour = ambient + diffuse + specular;
62 }
```

Listing A.3: Gourau Shading (Vertex Shader)