Drawing with Microsoft DirectX

Federico Bacci
Student ID: 1724351
Supervised by Professor Ian Kenny

Mini-project 2
MSc Advanced Computer Science
School of Computer Science, University of Birmingham
AY 2016/17

April 23, 2017

Abstract

The presented project introduces the reader to Computer Graphics programming using Microsoft DirectX 12 Application Programming Interface (API). The creation of simple scenes, the manipulation of 3-dimensional (3D) objects and some lighting properties were studied, implemented and successfully experimented using DirectX API. In order to achieve these results, the following work describes the theory and the common practice behind all the features considered.

Keywords: Computer Graphics, Microsoft DirectX, 3D rendering

Contents

List of Figures

1 Aim

Firstly, the intent was to gain the proper knowledge to begin programming graphics application. This implied the study of Computer Graphics theory on books and online resources.

Secondly, it was necessary to learn the structure and how to take advantage of Microsoft DirectX API.

Finally, the main focus was on illustrate how to create a virtual environment, draw simple shapes, navigate with an interactive camera and apply some simple lighting effects. In fact, these can be considered basic features of professional rendering engines, useful to start working on video games development.

2 Introduction

First of all, a brief introduction on game engines and rendering engines, the history and the technologies they offer to games development will be presented. Furthermore, computer graphics theory and some of its techniques will be illustrated: specifically, the mathematical processes that allow an object to be drawn on the screen, along with special surface shading effects. Then, Microsoft DirectX graphics libraries will be discussed, focusing on their internal structure. Lastly, the implementation of the code will be illustrated and analysed, also discussing about further developments.

2.1 Game Engine and Rendering Engine

In the early era of video games development and before game engines existed, every game was necessarily hard-coded as a singular identity and on its own specific hardware. Unfortunately, this meant that all the code implemented for a game could not be reused and it had to be thrown away. However, this was not a waste of resources, as the rapid advanced in hardware allowed game developers to completely redesign games in order to exploits the latest technologies.

Through the 1980s and since game arcades spread widely, game industries started to build their in-house games engines, while commercial game engines were not common until the 1990s. In fact, the term 'game engine' arose in the mid-1990s, along with 3D computer graphics and 3D first person shooter (FPS) games (such as idSoftware's FPSs Doom and Quake). Game engines made a clear distinction between game-specific rules and data and the underlying basic functions that every game need to have (such as rendering techniques or collision detection), leading to the specialization of video games developers roles.

Nowadays, most game engines provide tools to ease every aspect of game development, such as graphics, physics, sound and artificial intelligence (AI). This kind of software is often called 'middleware', as allows flexibility, reusability and all the basic functions needed. At this point, game engines are crucial to the development industry business, since they reduce costs and save the time of developers.

In particular, a major feature of game engines is the impressive cutting-edge graphics that they can create, thanks to the rendering engine that deals with the generation of 3D or 2D animated graphics. Rendering engines are typically built upon specific APIs, which provide abstraction from the Graphic Processing Unit (GPU) hardware. For instance, Microsoft DirectX is a commonly used collection of libraries for video games development.

Rendering engines can render a scene using different techniques, such as rasterization and ray tracing. Moreover, rendering engines can use many particular algorithms to simulate a scene. they can manipulate lights, shadows, refelctions and many other properties of a surface or object. [1]

2.2 Microsoft DirectX

Microsoft DirectX provides a collection of APIs for implementing multimedia application on Microsoft platforms and it is especially used for video games programming. The collection includes APIs with different purposes; for instance, it is worth to mention the following: Direct2D is used for 2D graphics, Direct3D for drawing 3D graphics, DirectCompute for running general-purpose computing on GPUs (GPGPU), and DirectWrite for high-quality text rendering. The name 'DirectX' was coined for calling the collection of all these different APIs, as the 'X' stands for a specific API name. One of the most common used and publicized API of the collection is definitely Direct3D (or D3D); for this, the names 'DirectX' and 'Direct3D' are interchangeable.

Direct3D allows for hardware acceleration of the entire 3D rendering pipeline and it was chosen for this work because of its widely use in video games development and its crucial importance for building games and multimedia application on Microsoft Windows, which is still the most commonly used OS in the whole world.

Direct3D API provides various function calls to configure the renderer in order to draw objects with different effects. It is not really necessary to know of the internal rendering system. The render-state management is encapsulated by the API calls, allowing the programmers to focus on the higher-level details of the graphics system.

Only few decades ago, GPUs were rather simple devices compared to now, they were limited to the most basic graphical functions required; in fact, Direct3D was originally thought as a simple tool-kit. However, as the graphics hardware rapidly evolved and became more generalized, also Direct3D API was adjusted to take advantage of the latest technologies. With the introduction of programmable GPUs, graphics programmers were given the possibility to create and render sophisticated special effects with with particular programs, called 'shaders', without being forced to use the effects provided directly by the fixed-functions. In particular, the High-Level Shading Language (HLSL) was developed by Microsoft in order to implement shaders.

2.2.1 DirectX 12

The latest Microsoft DirectX version is DirectX 12, released on July 2015. The single greatest new feature of DirectX 12 is a change to Direct3D API, that will make possible for developers to optimise the performance of their video games. This latest change introduces advanced low-level programming for Direct3D, allowing a lower level of hardware abstraction than earlier versions and enabling future games to significantly improve resource usage through parallel computation and less CPU utilization. This choice to pass from a high-level type to a low-level type of API is justified by the recent developments in both the hardware and software sides.

Direct3D 12 provides direct access to hardware features, taking advantage of the full potential of modern GPUs and allowing games to significantly improve multi-thread scaling and CPU utilization. With these improvements, it is possible to render rich scenes with a larger number of objects. However, better performance are achieved at the expense of the overall API complexity.

Overall, Direct3D 12 was tested and found to increase the rendering performance and to decrease the power consumption of both CPU and GPU. [7, 8, 9]

3 Computer Graphics Background

First of all, it is important to define the field of computer graphics:

> *"The term Computer Graphics describes any use of computers to create and manipulate images."*[2]

Nowadays, computer graphics techniques can be applied to many different fields, such as art, education, design, engineering and medicine.

It is commonly accepted to divide the graphics field into three following major areas:

- **Modeling:** it is the process of developing a mathematical representation of the shape of an object, so that it can be efficiently stored on the computer.
- **Rendering:** it is the process for creating a shaded image from a given model.
- **Animation:** it is used to create the illusion of movement in sequences of images.

This section will briefly introduce the reader to the concept of 'graphics pipeline', that allows a collection of vertices of a model to be rendered correctly. Then, the focus will shift on the theory behind the steps that allow a model to be transformed from its own local coordinate to be projected on screen space. Finally, some basic surface shading techniques are presented, these help a model to appear to have more volume, making it more realistic in the eyes of the viewer. [2, 4]

3.1 Graphics Pipeline

The term 'graphics pipeline' refers to the sequence of steps used to create a 2D raster representation [1] of a 3D scene. The graphic pipeline defines all the stages that an object needs to go through in order to be drawn on the screen.

One of the key developments in computer graphics has been the evolution of the GPU from a fixed pipeline to a programmable pipeline. This means that various stages of the pipeline can be programmed, using programs called

[1]A raster graphics image represents a rectangular grid of 'pixels', or points of color, generally displayed on a monitor.

shaders: which evolved from fixed-function units for transforming vertices and texturing pixels to unified modules that can perform these tasks and much more. In this way, GPUs use their processing power more efficiently.

Two examples of graphics pipelines with different aims are the following:

The hardware pipelines: used for interactive applications, such as video games, where the speed of the scene rendering is a priority and the quality of animation and visual effects is of secondary importance.

The software pipelines: used in film production, where the most important feature is the quality of the scene.

Even if different pipelines are built for different purposes, they all share a number of common fundamental features. The core operations of a graphics pipeline can be resumed by these steps:

- Vertex processing [2]
- Rasterization
- Fragment processing (or pixel processing) [3]

The vertex processing and the fragment processing are implemented with program shaders. There are two types of shaders: 3D shaders and 2D shaders. The first type is used in the vertex processing step, the second one is used in the fragment processing step.

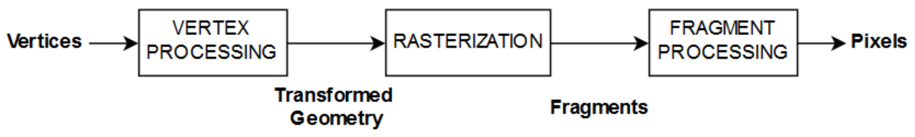

Figure 1: The core of graphics pipeline stages.

3.1.1 Vertex Processing

In this stage, it is executed the most established and common type of shaders: the vertex shader. Here, the vertices coordinates of a primitive are transformed and mapped from their origianl 3D coordinates to 2D screen space (where the position is measured in terms of pixels). In addition, vertex shaders can manipulate all of the vertex attributes, but can not create or delete vertices.

The vertex processing can include other complex type of shaders, which are often optional, such as the geometry shader and the tesselletion shaders.

[2]A 'vertex' can contain all sort of attributes, such as the coordinates of the vertex position in the world, the normal vector to the surface defined by the collection of vertices or the coordinates needed to map the texture.

[3]A 'fragment' is a collection of data needed to generate the final colour of one pixel. The output of the rasterization stage is a set of fragments, one for each pixel covered.

3.1.2 Rasterization

Rasterization is the central operation in graphics programs, given any different type of pipeline. This operation determines the pixels (fragments) on the screen that are covered by each primitive, interpolating per-vertex attributes across it. The output of the rasterizer is a set of fragments with their own set of attributes.

3.1.3 Fragment Processing

Also known as ‘pixel shaders’, fragment shaders compute the color and texture data of each pixel on the screen. Also, they can apply lighting values to the pixel, or manipulate shadows and specular highlights. In particular, fragment shaders are the only type of shaders that can be applied after the image has been rasterized.

3.2 3D Modeling

In computer graphics, 3D modeling is the process of developing a mathematical representation of any three-dimensional surface of an object.

There are different types of modeling surfaces, such as using implicit, parametric or volumetric representations. However, the polygonal representation is the most extensively used in computer graphics.

Polygonal representations approximate surfaces using polygons. Also called ‘polygon meshes’, the fundamental objects used to represent them are vertices; however, also edges and faces can be used. One disadvantage of using polygon meshes is that they can not accurately represent curved surfaces, so it is necessary to approximate them with a large number of polygons. This increase in model complexity leads to a decrease in the speed of the overall computation. On the other hand, the main advantage is that polygon meshes are generally faster than other types of representation and well suited for real-time graphics, such as video games.

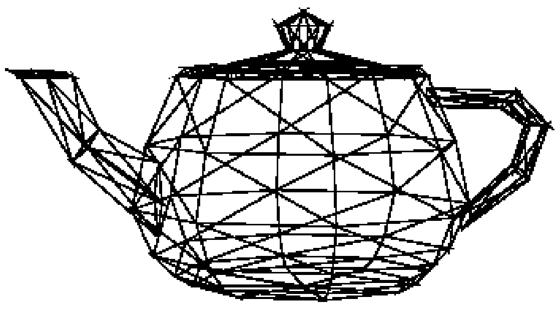

Figure 2: A triangle mesh that represents the famous Utah teapot.

3.3 Viewing Transformations

In order to be rasterized in the correct position on the screen, the vertices coordinates of a model need to be transformed through all the stages of the pipeline.

Here are discussed all the transformations, following the steps of the pipeline. However, it is necessary to briefly explain the mathematical foundations of these important operations: 'homogeneous coordinates' and 'affine transformations'.[4] These concepts will be especially useful when the model is projected on the screen view. [2, 3, 10]

3.3.1 Homogeneous Coordinates

It is obvious that inn Euclidean space two parallel lines on the same plane can not intersect. However, this is not true in projective space. Projective geometry has an extra dimension, called w, in addition to the x, y, and z dimensions: these are called 'homogeneous coordinates'. In 3D programs, the terms 'projective' and 'homogeneous' are basically interchangeable.

Homogeneous coordinates have the advantage that points, including points at infinity, can be represented using finite coordinates.

To simplify the concept, it is preferable to explain the concept using a 2D plane. This allows to draw some diagrams to illustrate a projective plane.

Now, once embedded the 2D plane in 3D space (called 'projective plane'), all lines and planes passing through the origin will be called 'points' and 'lines', respectively. So, a point through the origin can be written as $k(x, y, z)$, with $(x, y, z) \neq (0, 0, 0)$, and these are called the homogeneous coordinates of that point on a projective plane (which is the whole 3D space in this scenario).

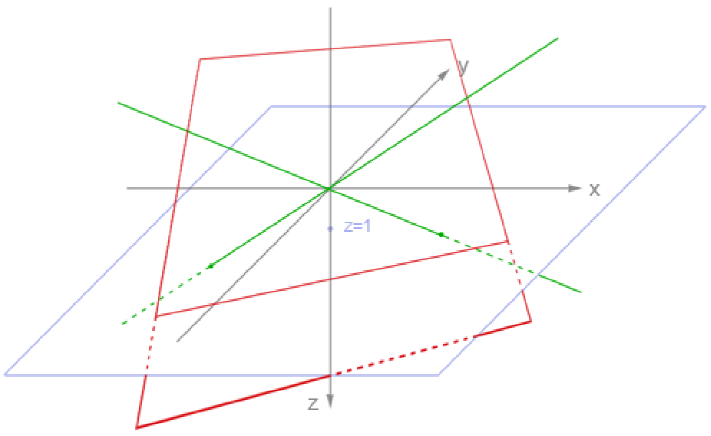

Figure 3: Visual representation of the 'projective plane'.

It is noticeable from Figure 3 that some points (green lines) and a line (red plane) intersect the plane $z = 1$: this is called an 'affine view' of the projective plane.

Points with coordinates $k(x, y, 0)$ are called 'vanishing points' (or 'points at infinity') and they do not intersect the plane $z = 1$. In this scenario, two projective lines (planes through the origin) will always intersect in a line through

[4]It will be assumed that the reader is familiar with the concepts of vector, matrix and all their properties.

origin. If the intersections of the two projective lines with $z = 1$ are not parallel, then the resulting point will also intersect the plane $z = 1$. Instead, if the two intersections are parallel they will converge in a projective point (line through the origin) that is parallel to $z = 1$: a vanishing point. In fact, if the plane $z = 1$ is tilted so that it intersects the x-axis, then it would be possible to see the vanishing point, which is the intersection of the two parallel lines in Figure 4. Finally, in a projective plane all lines intersect in exactly one point, even parallel lines.

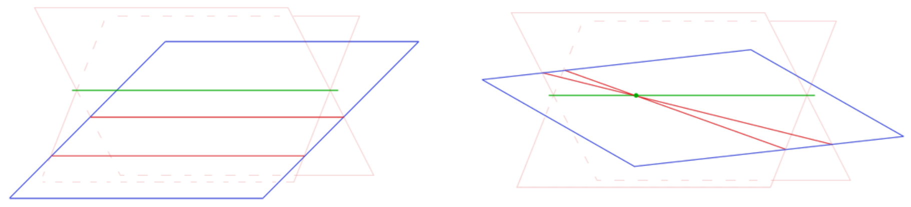

Figure 4: Visual representation of the concept of 'vanishing point' and the intersection of two parallel lines.

Homogeneous 3D coordinates are just a generalization of the 2D scenario above.

3.3.2 Linear and Affine Transformations

A linear transformation between two vector spaces V and W is a map $L : V \mapsto W$ such that:

1. $L(\boldsymbol{v_1} + \boldsymbol{v_2}) = L(\boldsymbol{v_1}) + L(\boldsymbol{v_2})$ for any vectors $\boldsymbol{v_1}, \boldsymbol{v_2} \in V$.
2. $L(\alpha \boldsymbol{v}) = \alpha L(\boldsymbol{v})$ for any scalar $\alpha \in \mathbb{R}$, with $\boldsymbol{v} \in V$.

Now, consider the case when $V = W$, when the dimension of the spaces is equal; this is called an 'endomorphism' of V.

Observe that a linear transformation can be written by a vector-matrix multiplication, so that a matrix $\boldsymbol{A}$ can represent a linear transformation L:

$$L(\boldsymbol{v}) = \boldsymbol{A}\boldsymbol{v} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y + a_{13}z \\ a_{21}x + a_{22}y + a_{23}z \\ a_{31}x + a_{32}y + a_{33}z \end{bmatrix}$$

An important property of linear transformations is that their composition is equal to their product. Suppose that:

$$\boldsymbol{v_2} = \boldsymbol{S}\boldsymbol{v_1} \qquad \text{and} \qquad \boldsymbol{v_3} = \boldsymbol{R}\boldsymbol{v_2}$$

with $\boldsymbol{S}$ and $\boldsymbol{R}$ two linear transformation matrices and with $\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3} \in V$. It is possible to rewrite the two equations:

$$\boldsymbol{v_3} = \boldsymbol{R}(\boldsymbol{S}\boldsymbol{v_1})$$
$$\boldsymbol{v_3} = (\boldsymbol{R}\boldsymbol{S})\boldsymbol{v_1}$$

So, it is possible to use just one matrix $\boldsymbol{M}$, which is the multiplication of the other two matrices: $\boldsymbol{M} = \boldsymbol{RS}$. [5]

For the sake of example, two important types of linear transformation will be presented: scaling and rotation. However, an object can be transformed in many different ways.

The scaling matrix $\boldsymbol{S}$:

$$\boldsymbol{Sv} = \begin{bmatrix} v_x & 0 & 0 \\ 0 & v_y & 0 \\ 0 & 0 & v_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} v_x x \\ v_y y \\ v_z z \end{bmatrix}$$

The rotation matrix $\boldsymbol{R}$, by an angle of θ about an axis in the direction of $\boldsymbol{u} = (u_x, u_y, u_z)$:

$$\boldsymbol{Rv} = \begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x\sin\theta \\ u_z u_x(1-\cos\theta) + u_y\sin\theta & u_z u_y(1-\cos\theta) - u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta) \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The combination of a linear transformation with a translation is defined as an ‘affine transformation’. An affine transformation preserves ratios of distances between points lying on a straight line, but it does not preserve angles or distances between points.

Translation does not make sense for vectors, because a vector only describes direction and magnitude, independent of location. Translations should only be applied to points. Fortunately, homogeneous coordinates allow to handle points and vectors in different ways. As explained before, in order to work with homogeneous coordinates it is necessary to add a fourth w-coordinate to the 3D space. Specifically, a new representation for vectors and points is:

- $(x, y, z, 0)$ for vectors
- $(x, y, z, 1)$ for points

In this way, translations of points work correctly.

For instance, let $\boldsymbol{T}$ be the transformation matrix for a translation of a point $\boldsymbol{p}$:

$$\boldsymbol{Tp} = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + a \\ p_y + b \\ p_z + c \\ 1 \end{bmatrix}$$

3.4 Pipeline of Viewing Transformations

Geometric transformations allow to move objects from their 3D locations to their positions in the 2D view of the 3D world. This 3D (world space) to 2D (image space) mapping is called ‘viewing transformation’ and it is crucial in object rendering. A visualization of the viewing pipeline is represented in Figure 5.

Most graphics system do this by using a sequence of three transformations: [2, 3]

[5] It is very important to notice that these transforms are applied from the right side first.

1. **Camera transformation:** is a rigid body transformation that places the camera at the origin in the desired orientation. It depends only on the pose of the camera.

2. **Projection transformation:** maps the points from camera space to the 'canonical view volume', so that all visible vertices fall in the cube $[-1;1]^3$ in 3D space (Figure 5). It depends only on the type of projection desired: for instance, orthographic or projective.

3. **Viewport transformation:** maps this unit image rectangle to the screen, in pixel coordinates. It depends only on the size and position of the output image.

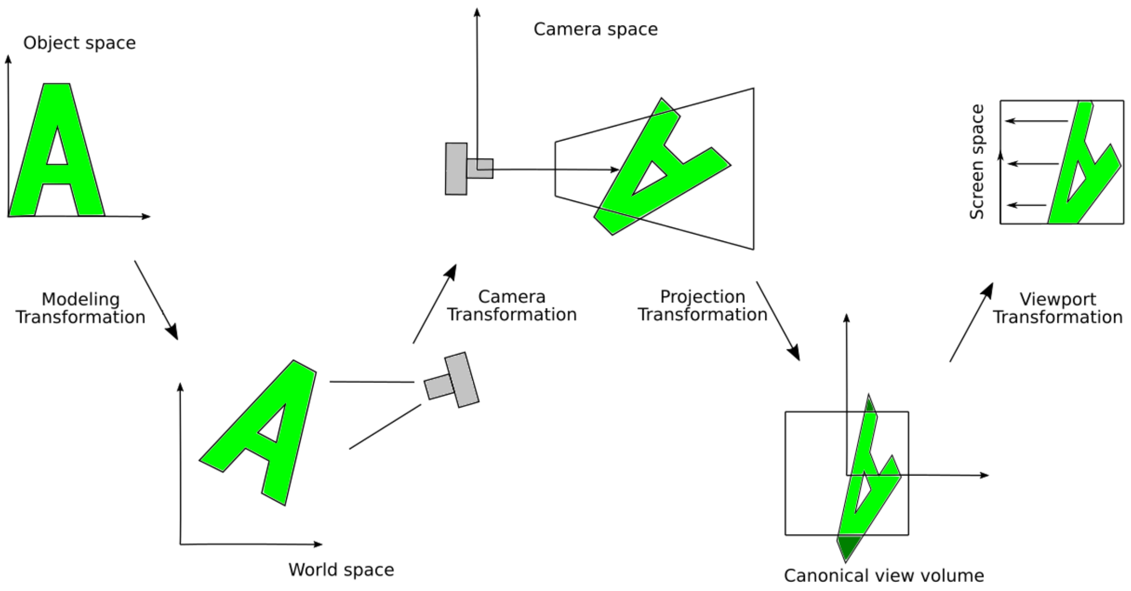

Figure 5: The sequence of transformations that allows an object to be drawn on the screen.

3.4.1 Camera Transformation

It is important to be able to change viewpoint in 3D space and look to any direction desired. For this, the viewer pose is defined by these vectors:

- $\boldsymbol{e}$, as the camera position in world space
- $\boldsymbol{g}$, as the gaze direction
- $\boldsymbol{t}$, as a view-up vector

From these, it is necessary to calculate a new orthonormal basis of the 3D space, composed by the unit vectors: [6]

$$\boldsymbol{w} = -\frac{\boldsymbol{g}}{\|\boldsymbol{g}\|}$$
$$\boldsymbol{u} = \frac{\boldsymbol{t} \times \boldsymbol{w}}{\|\boldsymbol{t} \times \boldsymbol{w}\|}$$
$$\boldsymbol{v} = \boldsymbol{w} \times \boldsymbol{u}$$

Here, a right-handed system was maintained (Figure 6). The coordinates of the origin $\boldsymbol{e}$ and the basis vectors $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ are stored in terms of the world space (with origin $\boldsymbol{o}$ and the axes $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$). So, it is necessary to transform the coordinate from the world space to the camera frame. This transformation is implemented by the matrix $\boldsymbol{M_{cam}}$:

$$\boldsymbol{M_{cam}} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

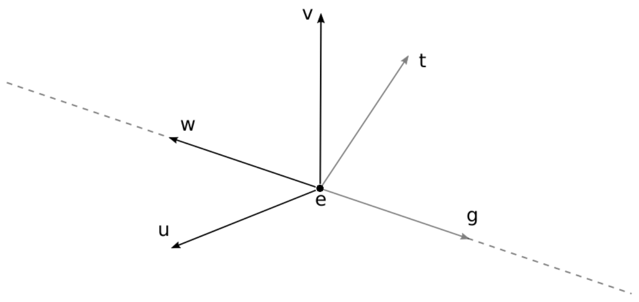

Figure 6: The new orthonormal basis that defines the pose of the camera.

3.4.2 Viewport Transformation

Suppose that all the geometry to render is in the canonical view volume and with an orthographic camera.

Assuming that the target window has n_x by n_y pixels, the viewport matrix $\boldsymbol{M_{vp}}$ is:

$$\boldsymbol{M_{vp}} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note that z-coordinate is ignored.

[6]The 'vector product' $\times$ returns a vector which is perpendicular to the plane defined by the two multiplied vectors.

3.4.3 Orthographic Projection Transformation

Usually, it is needed to render geometry in some region of space other than the canonical view volume. Keeping the view direction and orientation fixed looking along $-z$ with $+y$ up (right-handed system). The view volume is an axis-aligned box delimited by the planes l, r, b, t, n, f (left, right, bottom, top, near and far). The view volume is now called 'orthographic view volume'.

So, the transform from the orthogonal view volume to the canonical view volume is represented by the matrix $\boldsymbol{M_{orth}}$:

$$\boldsymbol{M_{orth}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.4.4 Projective Transformation

The key property of perspective is that the size of an object on the screen is proportional to z^{-1} for an eye at the origin looking up the negative z-axis.

Perspective is implemented in 3D computer graphics by using a transformation matrix that changes the w-coordinate of each vertex. After the camera matrix is applied to each vertex, but before the projection matrix is applied. The z-coordinate of each vertex represents the distance from the position of the camera; therefore, the larger z is, the more the vertex should be scaled down. The w-dimension affects the scale, so the projection matrix just changes the w value based on the z value.

Here is an example of a perspective projection matrix $\boldsymbol{P}$ being applied to a point $\boldsymbol{p}$ in homogeneous coordinate:

$$\boldsymbol{Pp} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} np_x \\ np_y \\ (n+f)p_z - fn \\ p_z \end{bmatrix} \sim \begin{bmatrix} \frac{np_x}{p_z} \\ \frac{np_y}{p_z} \\ n+f-\frac{fn}{p_z} \\ 1 \end{bmatrix}$$

The first, second, fourth rows of $\boldsymbol{P}$ simply implement the perspective equation. The third row is designed to bring the z-coordinate along so that it can be used later for hidden surface removal (depth ordering).

Finally, the full set of matrices for perspective view is:

$$\boldsymbol{M} = \boldsymbol{M_{vp}} \boldsymbol{M_{persp}} \boldsymbol{M_{cam}}$$

with $\boldsymbol{M_{persp}} = \boldsymbol{M_{orth}} \boldsymbol{P}$.

3.5 Surface Shading

In this section some basic shading techniques are presented. They help an object represented by polygon meshes to appear to have more volume, and so to be more realistic. These algorithms are executed in the fragment processing stage of the graphics pipeline. [7] [2, 4, 6]

[7]All the colour values in this section are represented in RGB, which allows to define any colour with only its Red, Green and Blue components.

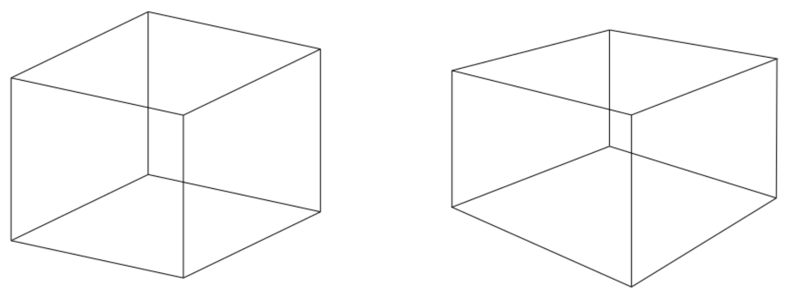

Figure 7: An orthographic (left) and a projective (right) representation of a cube.

3.5.1 Phong Lighting Model

Phong reflection model describes local illumination of an object and the way a surface reflects light as a combination of three parameters (Figure 8):

- **Ambient light:** background illumination
- **Diffuse reflectance:** non-shiny illumination
- **Specular reflectance:** shiny reflection

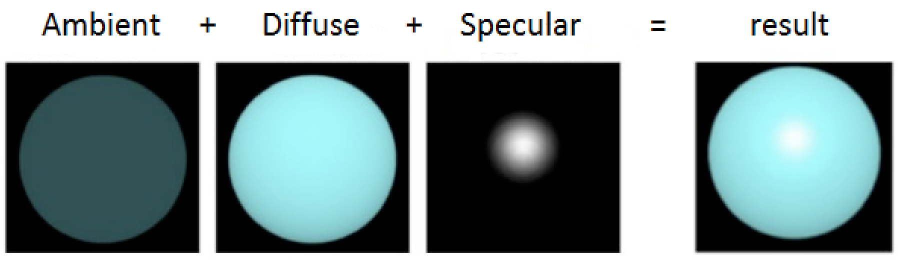

Figure 8: The three light components in Phong lighting model applied to a sphere. [11]

Firstly, in real life light is reflected all over and light rays arrive from every direction. For this, the ambient parameter consists of a constant colour term A:

$$A = k_a \boldsymbol{c_a}$$

where $0 \leq k_a \leq 1$ is the reflectivity of ambient light and $\boldsymbol{c_a}$ is its colour.

Secondly, the diffuse parameter is typical of Lambertian objects[8]. These kind o surfaces obey 'Lambert's cosine law', which says that the luminous intensity of a surface is proportional to the cosine of the angle θ_d between the surface

[8]The surface of many real object can be defined as 'matte'. This means that such objects are not shiny and do not change colour when the viewpoint changes. An object with these peculiarities is said to behave like a 'Lambertian' object.

normal $\boldsymbol{n}$ and the direction of the incident light source $\boldsymbol{l_d}$ (Figure 9). So, the diffuse parameter D is:

$$D = k_d \boldsymbol{c_d} \cos\theta_d$$

where $\boldsymbol{c_d}$ is the colour of diffuse light and $0 \leq k_d \leq 1$ is its reflectivity. It is noticeable that the diffuse parameter does not depend on the position of the viewer.

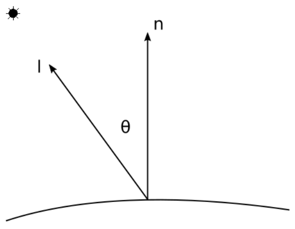

Figure 9: Illustration of the components needed to describe the diffuse reflectance D.

Thirdly, the specular parameter allows a surface to have 'highlights'. These kind of effect move across the surface as the viwer moves; so, this means that it will be necessary to take into account the viewer position with a vector $\boldsymbol{e_s}$ (Figure 10). The specular parameter S is:

$$S = k_s \boldsymbol{c_s} (\cos\varphi_s)^n$$

where $0 \leq k_s \leq 1$ is the reflectivity of specular light, $\boldsymbol{c_s}$ is its colour, and the exponent n defines the rate of decay of the highlight cone on the surface.

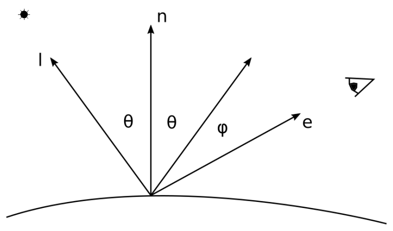

Figure 10: Illustration of the components needed to describe the specular reflectance S.

Finally, the complete shading model is:

$$\boldsymbol{c_{pixel}} = A + D + S = k_a \boldsymbol{c_a} + k_d \boldsymbol{c_d} \cos\theta_d + k_s \boldsymbol{c_s} (\cos\varphi_s)^n$$

where $\boldsymbol{c_{pixel}}$ is the final colour of the pixel. Note that the cosine function could be negative, so it is advisable to limit the values in the range $[0;1]$ with a $max(0, \cos\theta)$ function.

3.5.2 Flat Shading

Flat shading computes one lighting value per polygon surface and uses the resulting colour for the entire polygon. Moreover, the individual polygons can be seen.

This technique is usually used for high speed rendering where more advanced shading techniques are too computationally expensive.

The disadvantage of flat shading is that it gives low-polygon models a faceted look, which is not very realistic. However, sometimes this look can be advantageous: such as when modeling boxy objects.

3.5.3 Gouraud Shading

Gouraud shading is an interpolation method used to produce continuous shading of surfaces. It linearly interpolates the computed colours at each vertex across the surface of each polygon.

This technique is definitely more realistic than flat shading. However, it is more expensive computationally speaking and is far from render a flawless representation of an object. For instance, Gouraud shading may not draw surface highlights correctly; in fact, if a highlight is localized at the centre of a polygon and it does not spread to at least one of its vertices, the highlight will not be rendered by the algorithm. So, the quality of the highlights depends on the number of vertices in the model (Figure 11).

3.5.4 Phong Shading

Phong shading can be regarded as an improved version of Gouraud shading that provides a better and smoother approximation to reality.

In Phong shading the vertices normals are linearly interpolated and normalized across the surface of the polygon. Then, the final colour is calculated in every pixel.

This is more computationally expensive than Gouraud shading, since the reflection model must be computed at each pixel instead of at each vertex. On the other hand, Phong shading is visually realistic and it is noticeable that no artifacts are visible on the polygons edges.

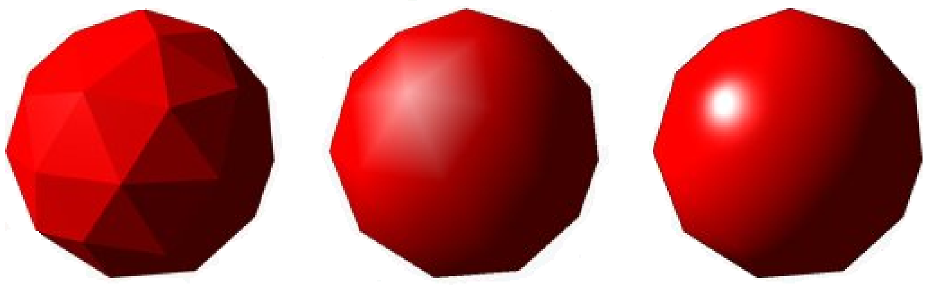

Figure 11: A low-resolution mesh representation of a sphere shaded with flat shading (left), Gauroud shading (middle) and Phong shading (right).

4 Programming with Direct3D 12

Direct3D 12 is a low-level graphics API used to control the GPU through an application. If supported by the system, Direct3D layer and drivers translate the instructions into appropriate machine language for the GPU, without worrying about the specifics of GPU hardware.

There are different hardware settings that defines how the geometry is managed and rendered by the GPU: such as the rasterizer state, the blend state, the depth/stencil state, the primitive topology type and the shaders needed.

In this section, the main focus will be on the graphics pipeline and the dispatch pipeline of the API. Moreover, some basics of how CPU and the GPU interact in Direct3D will be presented. [5]

4.1 CPU/GPU Interaction

In graphics applications, it is necessary to work with two different processor: the CPU and the GPU. They work in parallel and need to be synchronized. Unfortunately, synchronizations mean that one of the two units needs to stop its processing and wait for the other one to finish its computations; this is not a desirable behaviour for optimal performance.

In order to communicate and send commands to the GPU, the CPU needs to submit requests to the GPU command queue. Once submitted, the commands are not executed immediately by the GPU, as this unit is likely to be busy with previous processing; so, if the command queue is full the CPU will need to wait for the GPU. On the other hand, if the CPU is too slow and the queue gets empty, the GPU will need to wait for command from the CPU. Both these situations are undesirable because they waste computational time and for high-performance application it is necessary to maximize the use of available resources.

Unfortunately, sometimes it is necessary that one of the two processing units idles and waits for the other. For this, caution is needed when accessing to resources that are yet to be manipulated and not ready to be read.

4.2 The Graphics Pipeline

Direct3D 12 graphics pipeline is carefully designed to generate high-performance graphics for real-time applications, such as video games. Given a 3D scene with a virtual camera, the graphics pipeline refers to the steps needed to generate a 2D image based on what the camera sees.

Here, all the nine stages of the graphics pipeline will be described: four of these are fixed-function and the remaining five are programmable stages (called shaders). The output of each stage is taken as input into the next along with bound resources and lastly the output is sent to one or more render targets.

4.2.1 Input Assembler stage

The input assembler stage reads geometric data (vertices and indices) from user filled buffers and assembles them into primitives and feed them into the pipeline. There are several different primitive types: such as line lists, triangle

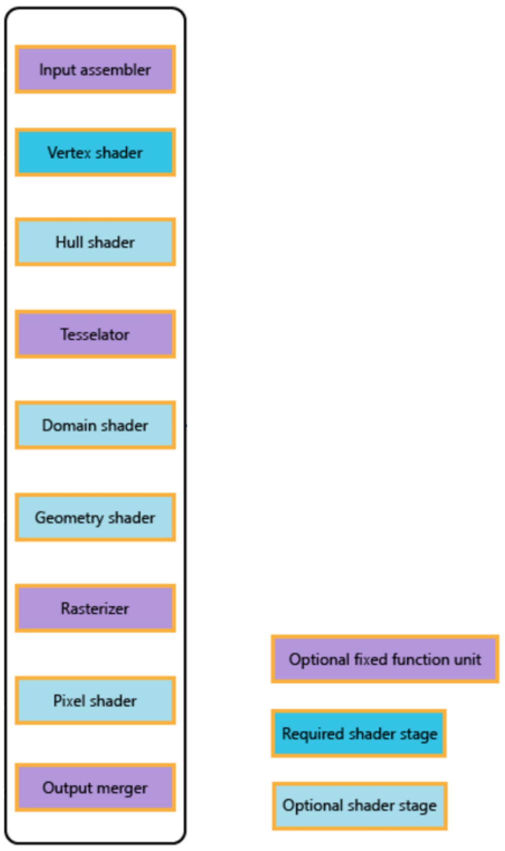

Figure 12: The stages of Direct3D 12 graphics pipeline. [12]

lists, triangle strips, or primitives with adjacency (this last type of information is visible to an application only in a geometry shader).

Usually, one or more vertex buffers, and optionally an index buffer, are provided as input and a layout description specifies the vertex structure to be expected (through text strings called ‘semantics’), so that the following stage knows what to do with each component.

4.2.2 Vertex Shader stage

Vertex shaders are the most common type of 3D shaders and allow per-vertex operations to be performed upon the vertices provided. They are executed by the GPU for each vertex, so the whole computation is very fast.

Vertex shaders manipulate attributes such as position, colour, normal and texture coordinates. Mainly, the vertices coordinates of a primitive are transformed and mapped from their original 3D coordinates into 2D screen space.

Furthermore, this stage must always be implemented in the pipeline. Even if no vertex modification or transformation is required, a shader must be provided that simply returns vertices without modifications.

4.2.3 Tessellation stages

The tessellation stages subdivide the triangles of a mesh to add new vertices (so new triangles), then offsetting them into new positions to create a finer and more detailed mesh.

From Direct3D 11, the pipeline supports new stages that implement tessellation in hardware. This off-loads the work from the CPU to the GPU and can lead to great performance improvements: hardware tessellation can generate an incredible amount of visual detail.

Some of the advantage of using this technique in graphics application are the following:

- Keep in memory low detailed meshes and create additional triangles when necessary, thus saving memory space.
- Operate on vertices (e.g.: physics or animation calculations) of simpler low detailed meshes and use tessellation only when rendering the meshes, thus saving computational time and achieving high quality results.

Tessellation is composed of one fixed-function stage and two optional stages, implemented only if tessellation is needed. When tessallation is implemented, the input assembler does not receive in input triangles as primitives, but patches with a number of control points; in fact, a triangle can be seen as a triangle patch with three control points. Eventually, these patches are tessellated into triangles.

So, the three sequential stages are the following:

1. **Hull Shader:** it actually consists of two shaders: the Constant Hull Shader and the Control Point Hull Shader. The first one outputs the so-called tessellation factors of the mesh per-patch. These factors define how much to tessellate the patch. The second shader operates on the control points, changing the number of control points of a patch as result.
2. **Tessellator:** it is the fixed-function stage that subdivides the polygons. The tessellator does not provide any method to control its state; all it needs is the output of the hull shader.
3. **Domain Shader:** it calculates the final position of the vertices in the output patch. It is run once per each new vertex created in the tessellator. Typically, the domain shaders operates as a vertex shader when the tessellation stages are not used in the application.

4.2.4 Geometry Shader stage

The geometry shader stage is an optional stage which was introduced in Direct3D 10. It is an optional stage and it can generate new primitives from the vertices that receives in input, unlike a vertex shader. The geometry shader evaluate primitives: it can output an amount of vertixes based on the input primitive (possibly with adjacency information), perhaps even discarding the current primitive.

Generally, the geometry shader is used for point sprite generation or shadow volume extrusion. It is useful for creating effects such as rain and explosions.

Moreover, especially prior to Direct3D 11, geometry shaders can be used to implement tessellation; however, they are limited on the number of output elements.

Furthermore, the geometry shader stage is the only one that can output to the rasterizer stage or to a vertex buffer through the stream output stage.

Overall, it is important to notice that the amount of data in input to the geometry stage can significantly affect the rendering performance of the application.

4.2.5 Stream Output stage

The stream output stage is an optional fixed-function stage and it is used only to store into vertex buffers the output of the geometry shader stage, so that the data are ready to be used for the next pipeline operations.

4.2.6 Rasterizer stage

The rasterizer is a fixed-function that converts the vector information into a raster image. Rasterization includes operations such as clipping, culling, perspective divide (to transform the vertices coordinates from clip space to normalized device coordinates) and mapping the vertices to screen space.

Moreover, if a pixel shader has been implemented, the rasterizer will call it for each pixel, interpolation vertex values across each primitive. It is even possible to set specific methods of interpolation, if desired.

Culling and Clipping Transforming primitives into screen space and rasterize them without culling and clipping them is not the right process, because objects outside the view of the screen could be drawn, leading to incorrect results.

Firstly, it is needed to cull (discard) all the geometry outside the view frustum of the camera, as seen in Figure 13, so to drastically reduce the number of polygons to compute.

Then, the models that are only partially outside the view are clipped and the geometry reshaped into new polygons.

For instance, good culling and clipping strategies is important in the development of video games, in order to maximize the game frame rate and visual quality.

Back-face Culling Every polygon has two sides: front-face and back-face. This distinction is determined by the order in which the vertices are drawn and in order to discriminate between the two it is necessary to compute the polygon normal, which arise from the front-face.

Therefore, back-face culling refers to the process of deleting from the rendering operations those polygons that are back-facing the camera view point. This can drastically decrease the number of vertices to be rendered, therefore saving computational time.

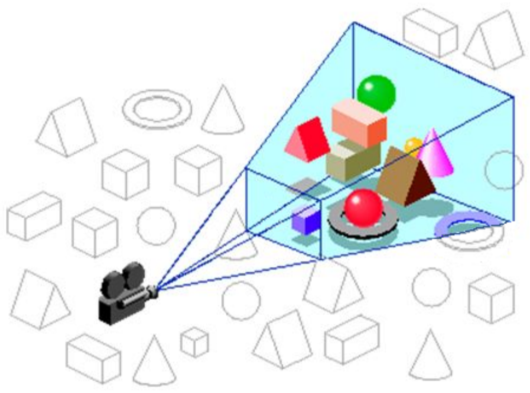

Figure 13: A representation of culling and clipping. Every geometry outside the view frustum is discarded and will not be rendered.

4.2.7 Pixel Shader stage

The pixel shader stage is the last programmable stage of the pipeline. Its job is to operate on per-pixel values in order to determine their final colour. Pixel shaders use the interpolated vertex attributes across the primitive for post-processing pixels. In particular, they can apply lighting values, manipulate shadows, and apply blur effects; moreover, pixel shaders can alter the depth of the fragments. However, they can only operate on a single pixel, not knowing anything about the scene; for this, pixel shaders alone are not powerful tools.

4.2.8 Output Merger stage

The final stage is the fixed-function output merger stage. After the colour of all fragments is calculated, the output merger selects those fragments which have passed the depth/stencil test and writes them to the back buffer, ready to be drawn on the screen. Also, blending is executed in this stage, so that a particular fragment can be blended with the current fragment in the back buffer, instead of fully overriding it.

4.3 The Dispatch Pipeline

The dispatch pipeline is composed of only one single stage: the compute shader stage. Although the graphics pipeline and the dispatch pipeline are theoretically separate, they can not run at the same time. Moreover, the context change cost to switch between them is not low, so all the calls to the dispatch pipeline need be grouped together.

4.3.1 Compute Shader stage

The compute shader stage is an optional and programmable stage that allow to to execute a shader across multiple threads. Even if not part of the graphics pipeline, compute shaders can manipulate GPU resources.

However, compute shader is not relevant only for graphics programmer, because some non-graphical application can benefit from the computational power of GPU parallel architecture. General purpose GPU (GPGPU) programming is how is defined the use of GPUs for handling non-graphical data.

5 Implementation

In this section, the methods and functions that allow to draw and shade simple shapes and to navigate the virtual environment with a dynamic camera will be presented. The focus of the explanation will be on the main techniques and the theory on which the code relies. [9]

Visual Studio was used, along with Win32 application environment for the implementation. The program was written in C++. [5]

5.1 Drawing Meshes

First of all, consider how an object can be computationally defined: 'polygon meshes' represent and approximate surfaces using polygons. The most common polygon mesh is the simplest one: the triangle mesh. However, it is possible to use more complex polygons.

Even if a mesh is composed by vertices, edges and faces, it is commonly represented just by a collection of vertices: these are defined by a 3D position vector. The vertices can be loaded and rendered in different ways. They can be read from tree-structured files, like BSP files (Binary Space Partitioning), where are stored all the information on the position and details of the objects that compose the map. However, it is possible for a simple scene to be hard-coded in the source. This last approach was used for its initial simplicity.

A builder class was implemented in order to define meshes of geometric objects. For each model, the builder provides a specific function which receives all the parameters necessary to create the vertices and connect them properly.

For instance, in order to create a square pyramid it is needed to pass the function three size parameters: width and depth of the base and height. In this way, the coordinates of the model are set in its local space; then, it will be necessary to transform them, so that the position of the vertices is arranged correctly in world space. The algorithm connects the vertices for each face of the pyramid, creating the mesh structure that will be rendered (Code 1). Specifically, a square pyramid has 5 faces: the function defines 3 vertices for each lateral face and 4 vertices for the base, so 16 vertices for the whole mesh. After this, the vertex indexes which compose the different triangles of the mesh are stored in clockwise order, so that the program will know which vertices need to be connected and which side is the front face of all the triangles.

Code 1: The implementation of a pyramid mesh.

```
// The two arrays that store the lists of vertices and indexes
Vertex v[16];
uint32_t i[18];

float w2 = width/2;
float d2 = depth/2;
```

[9]Direct3D initialization will not be presented for the purpose of simplicity and because it is not relevant to the work done.

```
// Instantiate 3 vertices for each lateral face and 4 vertices
// for the bottom face
v[0] = Vertex(0, height, 0);
v[1] = Vertex(w2, 0, d2);
v[2] = Vertex(w2, 0, -d2);

v[3] = Vertex(0, height, 0);
v[4] = Vertex(w2, 0, -d2);
v[5] = Vertex(-w2, 0, -d2);

v[6] = Vertex(0, height, 0);
v[7] = Vertex(-w2, 0, -d2);
v[8] = Vertex(-w2, 0, d2);

v[9] = Vertex(0, height, 0);
v[10] = Vertex(-w2, 0, d2);
v[11] = Vertex(w2, 0, d2);

v[12] = Vertex(-w2, 0, -d2);
v[13] = Vertex(-w2, 0, d2);
v[14] = Vertex(w2, 0, d2);
v[15] = Vertex(w2, 0, -d2);

// Define the order in which the vertices will be rendered
i[0] = 0; i[1] = 1; i[2] = 2;
i[3] = 3; i[4] = 4; i[5] = 5;

i[6] = 6; i[7] = 7; i[8] = 8;
i[9] = 9; i[10] = 10; i[11] = 11;

i[12] = 12; i[13] = 15; i[14] = 14;
i[15] = 12; i[16] = 14; i[17] = 13;
```

After the mesh structure is built, a colour value is added to every vertex and Direct3D resources are allocated in order to pass the data to the GPU. Respectively, a vertex buffer and an index buffer are created, taking into account all the different objects that need to be drawn.

Code 2: The Vertex structure which defines the attributes that need to be passed through the pipeline.

```
struct Vertex
{
  Vertex() {}
  Vertex(float x, float y, float z) : Position(x, y, z) {}

  XMFLOAT3 Position;
  XMFLOAT4 Colour;
};
```

The next step is to project the vertices in their correct position on the screen: a vertex shader is used for the implementation. It is observable from Code 3 that the coordinates of the vertex in input are transformed to clip space, ready to be rendered by the resterizer. Moreover, a simple pixel shader is used to only pass the initial set colour through the pipeline.

Code 3: Implementation of simple vertex shader and pixel shader using HLSL. All the constants (e.g.: worldMatrix and viewProjMatrix) needed for the calculation are passed through a constant buffer.

```
struct VInput
```

```
{
  float3 PosLocal : POSITION;
  float4 Color : COLOR;
};

struct VOutput
{
  float4 PosHomogeneous : SV_POSITION;
  float4 Color : COLOR;
};

VOutput VertexShader(VInput input)
{
  VOutput output;

  // Transform the vertex from local space to clip space
  float4 posWorld = mul(float4(input.PosLocal, 1.0f), worldMatrix);
  output.PosHomogeneous = mul(posWorld, viewProjMatrix);

  // Pass the colour without modifications
  output.Color = input.Color;

  return output;
}

float4 PixelShader(VOutput input) : SV_Target
{
  // Output the colour without modifications
  return input.Color;
}
```

5.2 Building an Interactive Camera

It is essential for a virtual environment to be navigable, so that the user can explore it all. For this, interactive camera systems are implemented to allow to control a camera, in order to display specific views.

A camera class was written to implement such a system. The class stores all the constants, parameters and functions needed to display the desired view of the virtual environment. Important pieces of information are the following:

- **Position vector:** it is the position of the camera in world coordinates and defines the origin of camera space.
- **Right vector:** it is expressed in world coordinates and defines the x-axis of the view[10] space.
- **Up vector:** it is expressed in world coordinates and defines the y-axis of the view space.
- **Look vector:** it is expressed in world coordinates and defines the z-axis of the view space.

Also, the properties that allow to characterize the 'view frustum' [11] of the camera are stored:

[10]'Camera space' and 'View space' are two names that define the same coordinates system.

[11]The exact shape of this region varies depending on what kind of camera lens is being simulated, but typically it is a frustum of a rectangular pyramid (hence the name).

- **Near and far planes:** they are the distances from the camera origin of the planes that cut the frustum perpendicularly to the viewing direction (look vector). Vertices behind the camera, closer than the near plane, and farther than the far plane are not drawn on the screen.

- **Aspect ratio:** it is defined by the ratio between the width and the height of the projection window, $r = w/h$. The 'projection window' is the projected 2D image of the scene.

- **Field of view:** it is measured in radians and it represents how much of the scene the camera can 'see'.

In order to control the view, it was necessary to implement some functions that allow to change the position vector of the camera and the orientation of its look vector. The methods adopted change the position of the camera, moving along the look vector and the right vector, and the orientation of the view relatively to the y-axis of world space (yaw) and around the right vector of the camera (pitch). All the functions are shown by the following pseudocode:

```
function MoveForwardsAndBackwards($amountMovement$)
    $displacement \leftarrow amountMovement \cdot lookVector$
    $positionVector \leftarrow positionVector + displacement$
    $changedFlag \leftarrow$ true
end function
function MoveLeftAndRight($amountMovement$)
    $displacement \leftarrow amountMovement \cdot rightVector$
    $positionVector \leftarrow positionVector + displacement$
    $changedFlag \leftarrow$ true
end function
function YawRotation($rotationAngle$)
    $rotMatrixAroundYaxis \leftarrow buildRotMatrix(rotationAngle)$
    $rightVector \leftarrow rightVector \cdot rotationMatrix$
    $lookVector \leftarrow lookVector \cdot rotationMatrix$
    $upVector \leftarrow upVector \cdot rotationMatrix$
    $changedFlag \leftarrow$ true
end function
function PitchRotation($rotationAngle$)
    $rotMatrixAroundRightVector \leftarrow buildRotMatrix(rotationAngle)$
    $lookVector \leftarrow lookVector \cdot rotationMatrix$
    $upVector \leftarrow upVector \cdot rotationMatrix$
    $changedFlag \leftarrow$ true
end function
```

As it is noticeable from the pseudocode, the last statement sets a flag of the camera class. This flag keeps track of the changes in the position and orientation of the view, it allows to update the view matrix needed to transform and project the vertices on the screen. The following algorithm was implemented in order to update the view matrix:

```
function UpdateViewMatrix
    if $changedFlag$ then
        $lookVector \leftarrow normalize(lookVector)$
        $upVector \leftarrow normalize(lookVector \times rightVector)$
        $rightVector \leftarrow upVector \times lookVector$
        $x \leftarrow -(positionVector \cdot rightVector)$
        $y \leftarrow -(positionVector \cdot upVector)$
        $z \leftarrow -(positionVector \cdot lookVector)$
        $buildViewMatrix()$        ▷ not specified for the sake of simplicity
        $changedFlag \leftarrow$ false
    end if
end function
```

The first lines of the function reorthonormalize the camera basis vectors, in order to be sure that they are mutually orthogonal and their length is unitary. This is necessary because of numerical errors that occurs in the computation of floating point values; they can accumulate and cause the basis vectors to become non-orthonormal, so that they no longer represent a rectangular coordinate system, but a skewed one.

After this, the view matrix is re-built, updating the position of the camera from the world origin with (x, y, z) and the basis vectors of camera space $rightVector$, $upVector$ and $lookVector$.

5.3 Lighting

Both a directional light and a point light was added to the scene to experiment their effects on the objects. Phong lighting model was used to characterise how the light is reflected by the surfaces in the scene.

Obviously, in order to work with lighting, it was necessary to calculate and add normal vectors to the vertices attributes.

Code 4: The updated Vertex structure with the Normal attribute.

```
struct Vertex
{
  Vertex() {}
  Vertex(float px, float py, float pz, float nx, float ny,
    float nz) : Position(px, py, pz), Normal(nx, ny, nz) {}

  XMFLOAT3 Position;
  XMFLOAT3 Normal;
  XMFLOAT4 Colour;
};
```

Code 5: The implementation of a pyramid mesh considering the vertex normals.

```
// The two arrays that store the lists of vertices and indexes
Vertex v[16];
uint32_t i[18];

float w2 = width/2;
float d2 = depth/2;

// The vertex normal is calculated for one face of the pyramid and
// then adapted for every other face
XMVECTOR v1 = XMVectorSet(0.0f, height, 0.0f, 0.0f);
XMVECTOR v2 = XMVectorSet(w2, 0.0f, -d2, 0.0f);
XMVECTOR v11 = XMVectorSubtract(v2, v1);

XMVECTOR v22 = XMVectorSet(0.0f, 0.0f, d2, 0.0f);

XMVECTOR normal = XMVector3Cross(v22, v11);
```

```
normal = XMVector3Normalize(normal);

XMFLOAT3 n;
XMStoreFloat3(&n, normal);

// Instantiate 3 vertices for each lateral face and 4 vertices
// for the bottom face
v[0] = Vertex(0, height, 0, n.x, n.y, n.z);
v[1] = Vertex(w2, 0, d2, n.x, n.y, n.z);
v[2] = Vertex(w2, 0, -d2, n.x, n.y, n.z);

v[3] = Vertex(0, height, 0, n.z, n.y, -n.x);
v[4] = Vertex(w2, 0, -d2, n.z, n.y, -n.x);
v[5] = Vertex(-w2, 0, -d2, n.z, n.y, -n.x);

v[6] = Vertex(0, height, 0, -n.x, n.y, n.z);
v[7] = Vertex(-w2, 0, -d2, -n.x, n.y, n.z);
v[8] = Vertex(-w2, 0, d2, -n.x, n.y, n.z);

v[9] = Vertex(0, height, 0, n.z, n.y, n.x);
v[10] = Vertex(-w2, 0, d2, n.z, n.y, n.x);
v[11] = Vertex(w2, 0, d2, n.z, n.y, n.x);

// The normal of the base is just the unitary vector along
// the y-axis, pointing in the negative direction
v[12] = Vertex(-w2, 0, -d2, 0.0f, -1.0f, 0.0f);
v[13] = Vertex(-w2, 0, d2, 0.0f, -1.0f, 0.0f);
v[14] = Vertex(w2, 0, d2, 0.0f, -1.0f, 0.0f);
v[15] = Vertex(w2, 0, -d2, 0.0f, -1.0f, 0.0f);

// Define the order in which the vertices will be rendered
i[0] = 0; i[1] = 1; i[2] = 2;
i[3] = 3; i[4] = 4; i[5] = 5;

i[6] = 6; i[7] = 7; i[8] = 8;
i[9] = 9; i[10] = 10; i[11] = 11;

i[12] = 12; i[13] = 15; i[14] = 14;
i[15] = 12; i[16] = 14; i[17] = 13;
```

5.3.1 Directional Light

A so-called ‘directional light’ illuminates all objects in the scene from a given direction. It approximates a light source far away, like the Sun, in fact it is useful to simulate the light of natural environment. In this way, it is possible to suppose all the light rays as parallel to each other, so that the same light vector is used for the reflectance computation of all surfaces.

As it is possible to see from Code 6, all the necessary calculations were implemented in the pixel shader stage of the pipeline, because interpolating the colour per pixel leads to better and more realistic results. Moreover, in this situation per pixel calculations can be considered as demanding as per vertex computations because of the relatively simplicity of the application.

Code 6: The vertex shader and pixel shader implemented to manage directional light. All the constants (e.g.: lightDirection and lightStregth) needed for the calculation are passed through a constant buffer.

```
struct VInput
{
```

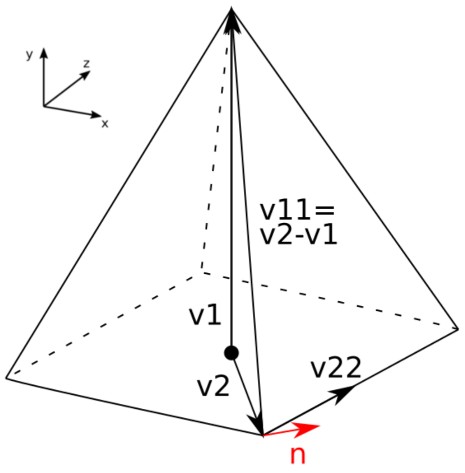

Figure 14: A visual representation of the calculation of vertex normals for a square pyramid.

```
  float3 PosLocal : POSITION;
  float3 NormalLocal : NORMAL;
  float4 Color : COLOR;
};

struct VOutput
{
  float4 PosHomogeneous : SV_POSITION;
  float3 PosWorld : POSITION;
  float3 NormalWorld : NORMAL;
  float4 Color : COLOR;
};

VOutput VertexShader(VInput input)
{
  VOutput output;

  output.PosWorld = mul(float4(input.PosLocal, 1.0f), worldMatrix);

  // Transforming the coordinates of the normal vector
  // to world space
  output.NormalWorld = mul(input.NormalLocal,
    (float3x3)worldMatrix);

  output.PosHomogeneous = mul(posWorld, viewProjMatrix);

  output.Color = input.Color;

  return output;
}

float4 PixelShader(VOutput input) : SV_Target
{
  // Re-normalize the normal vector
  input.NormalWorld = normalize(input.NormalWorld);

  // Calculate the eye vector
  float3 toCameraWorld = normalize(cameraPosWorld - input.PosWorld);
  // Normalize the light vector
  float3 light = normalize(lightDirection);

  // Scale the strength of the light down by Lambert's cosine law
  lightStrength *= max(dot(-light, input.NormalWorld), 0.0f);

  // Calculate the reflection vector of the light
  float3 R = reflect(light, input.NormalWorld);

  // Parameters of Phong reflectance model
  float4 Ka = 0.5f;
  float4 Kd = 0.7f * saturate(dot(input.NormalWorld, -light));
  float4 Ks = 0.5f * pow(saturate(dot(R, toCameraWorld)), 0.5f);

  return (Ka + Kd + Ks) * input.Color * float4(lightStrength, 0.0f);
}
```

5.3.2 Point Light

A 'point light' originates from a single point and radiates spherically in all directions, the same way the light originated from a light bulb does. For this, the direction with the light hits a surface depends on its position in the world. Furthermore, it was taken into account the way light intensity weakens when distance from the source increases. So, two terms were introduced to simulate this behaviour: $falloffStart$ and $fallffEnd$ control the strength of the light.

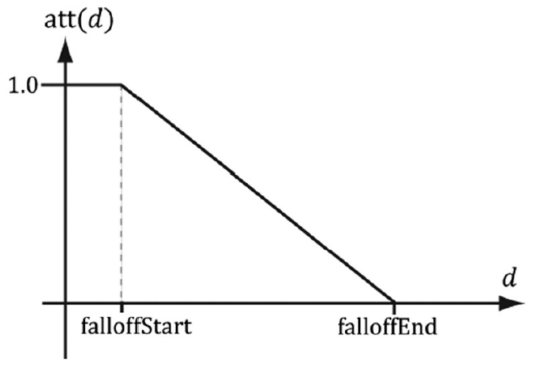

Figure 15: The strength of the light depends on the distance d from its source. The parameter $att(d)$ is multiplied with the light strength.

Code 7: The vertex shader and pixel shader implemented to manage directional light. All the constants (e.g.: lightPosition and lightStregth) needed for the calculation are passed through a constant buffer.

```
struct VInput
{
  float3 PosLocal : POSITION;
  float3 NormalLocal : NORMAL;
  float4 Color : COLOR;
};

struct VOutput
```

```
{
  float4 PosHomogeneous : SV_POSITION;
  float3 PosWorld : POSITION;
  float3 NormalWorld : NORMAL;
  float4 Color : COLOR;
};

VOutput VertexShader(VInput input)
{
  VOutput output;

  output.PosWorld = mul(float4(input.PosLocal, 1.0f), worldMatrix);

  // Transforming the coordinates of the normal vector
  // to world space
  output.NormalWorld = mul(input.NormalLocal,
    (float3x3)worldMatrix);

  output.PosHomogeneous = mul(posWorld, viewProjMatrix);

  output.Color = input.Color;

  return output;
}

float4 PixelShader(VOutput input) : SV_Target
{
  // Re-normalize the normal vector
  pin.NormalW = normalize(pin.NormalW);

  // Calculate the eye vector
  float3 toCameraWorld = normalize(cameraPosWorld - input.PosWorld);

  // Calculate the vector from the surface to the light source
  float3 light = lightPosWorld - input.PosWorld;

  // Distance from the surface to the light source
  float distance = length(light);

  // Range test
  if (distance > falloffEnd)
    return 0.0f;

  // Normalize the light vector
  light /= distance;

  // Scale the strength of the light down by Lambert's cosine law
  lightStrength *= max(dot(light, input.NormalWorld), 0.0f);

  // Attenuate the strength of the light by distance
  lightStrength *= saturate(
    (falloffEnd - d) / (falloffEnd - falloffStart));

  // Calculate the reflection vector of the light
  float3 R = reflect(-light, input.NormalWorld);

  // Parameters of Phong reflectance model
  float4 Ka = 0.5f;
  float4 Kd = 0.7f * saturate(dot(pin.NormalW, light));
  float4 Ks = 0.5f * pow(saturate(dot(R, toCameraWorld)), 0.5f);

  return (Ka + Kd + Ks) * input.Color * float4(lightStrength, 0.0f);
}
```

6 Evaluation

Because of the intrinsic nature of the application, a qualitative evaluation of its features was performed.

6.1 Building and Navigating the Environment

Methods It was decided to set a scene up, in order to test the ability of the application to render objects and correctly draw different views of the environment. Specifically, the scene contained a grid, two different-sized cubes and a square pyramid.

Analysis As it is possible to see from Figure 16, Figure 17 and Figure 18, the scene did not show evidence of errors representing the scene. The objects perspective view is always drawn correctly, even when really close to the object surfaces.

However, the objects look flat; this is because here no shading technique was applied.

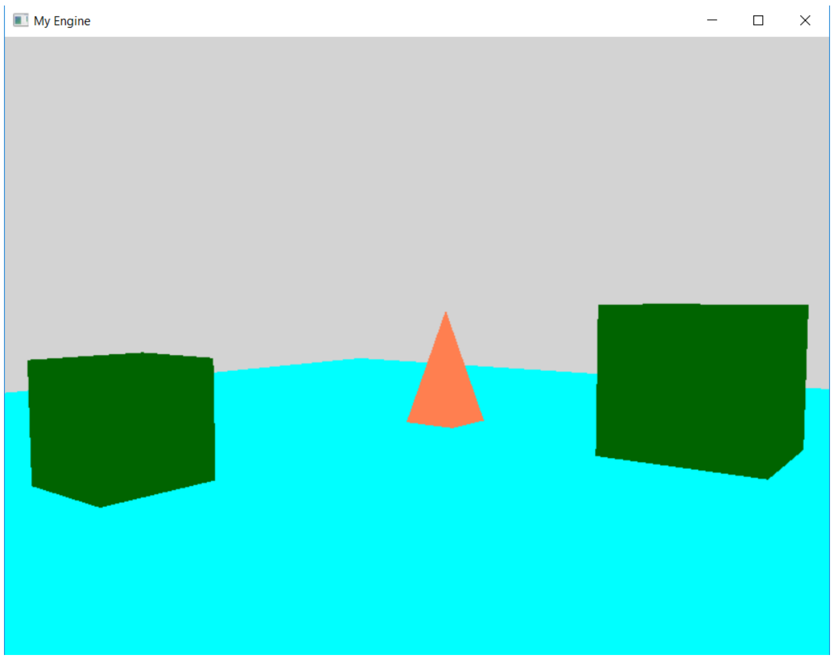

Figure 16: A view of all the objects rendered.

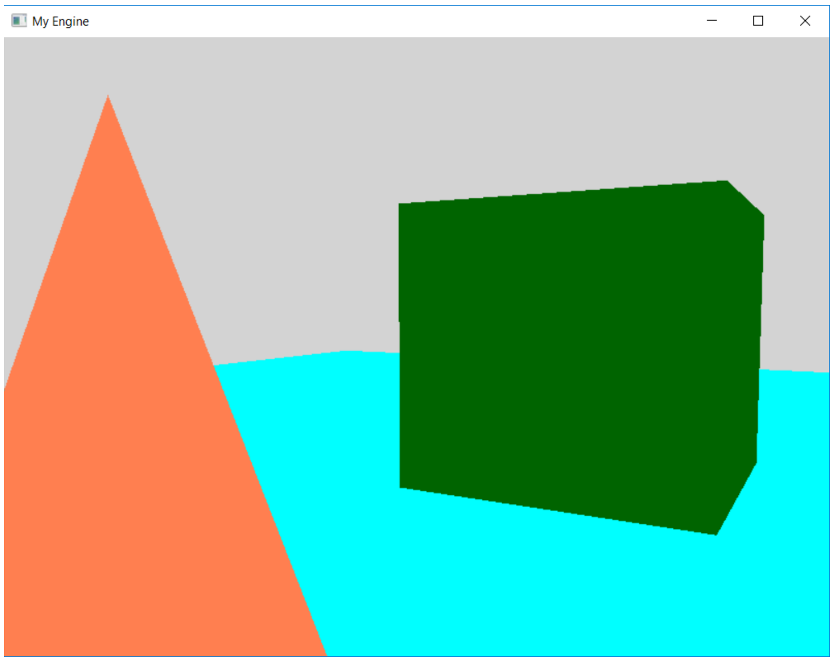

Figure 17: A close-up of the square pyramid.

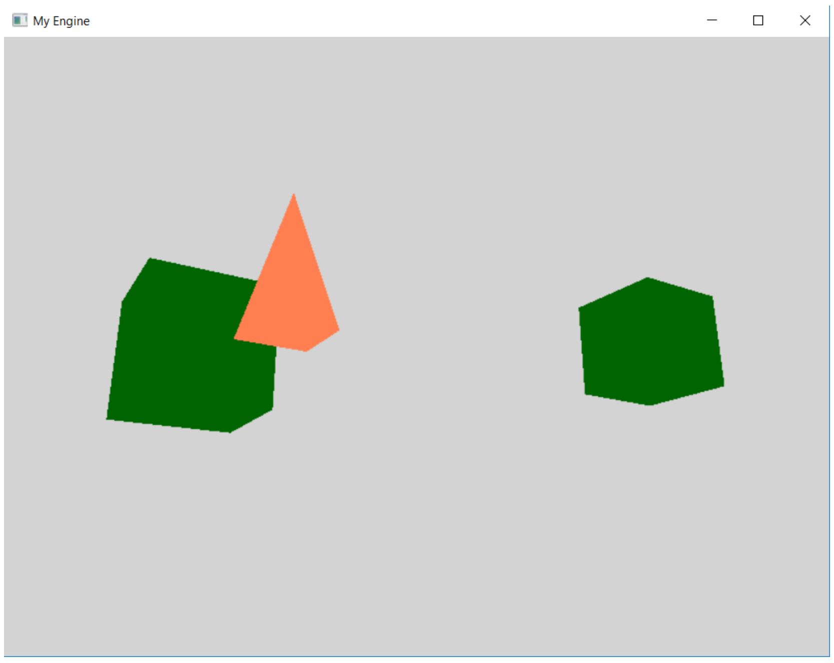

Figure 18: A view from under the plane; from this perspective it is not rendered because the camera is facing its back face.

6.2 Directional Lighting

Methods A simple scene was tested with a directional light. The direction of the light was defined by the vector $d = [-1, -1, 0]$, while the strength of the light was set to $s = [2, 2, 2]$. Moreover, Phong reflectance model coefficients were defined by the following values:

$$K_a = 0.5$$
$$K_d = 0.7$$
$$K_s = 0.5$$
$$n = 0.5$$

Analysis It is noticeable from Figure 19 and Figure 20 that only the surfaces that face the direction of the light are properly illuminated. Instead, the light vector and the normals of the darker cube-faces form an angle of 90° and for this the faces are illuminated only by the ambient coefficient of Phong reflectance model.

Moreover, the cubes look faceted because the vertices normals were not bent in order to achieve a smooth gradient of colour on the surface; however, in this situation it is correct that an object like a cube (which has faces that form angles of 90°) looks faceted.

In addition, the cubes do not cast any shadows, because this feature (shadow mapping) was not taken into account. Overall, the results were as expected.

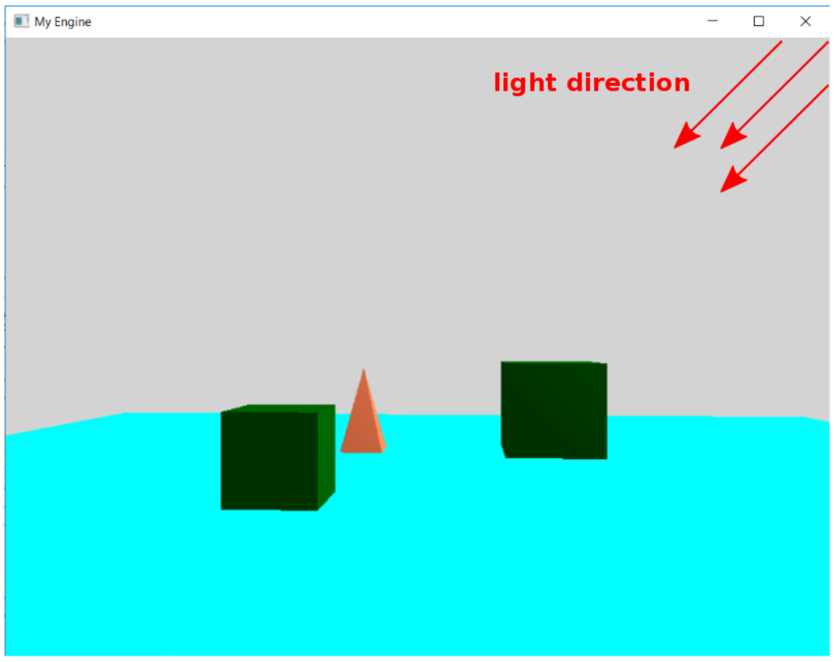

Figure 19: A view of the scene illuminated by a directional light. The camera is looking at the origin of world space.

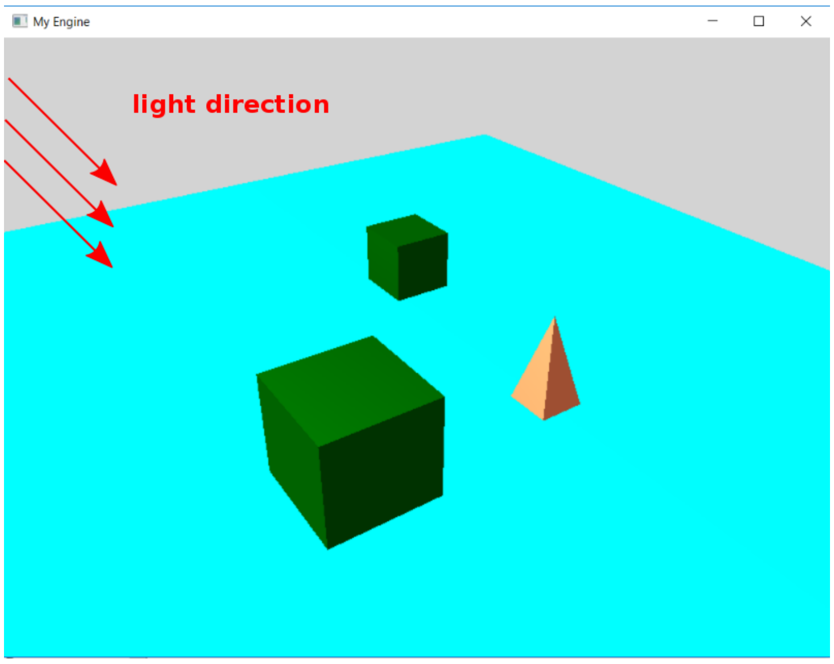

Figure 20: Another view of the scene where it is possible to clearly notice the surfaces more illuminated by the light.

6.3 Point Lighting

Methods Once again, a simple scene was tested. The position of the point light was set to $p = [0, 2, 0]$, at the centre of the scene and just above the blue plane, while the strength of the light was set to $s = [2, 2, 2]$. The two falloff parameters were defined as $falloffStart = 0.5$ and $falloffEnd = 15$. Once again, Phong reflectance model coefficients were defined by the following values:

$$K_a = 0.5$$
$$K_d = 0.7$$
$$K_s = 0.5$$
$$n = 0.5$$

Finally, the strength of the light was set to $s = [5, 5, 5]$ in order to inspect better the specular highlights created by the reflectance model.

Analysis It is noticeable from Figure 21 the effect of the light falloff; bright at the centre of the scene, the light gradually dims outwards. Moreover,

From Figure 22, it is possible to see the specular highlight created on the right face of the cube. The red arrows represent the vertex normals.

Overall, the results were as expected; however, other techniques should be studied and applied in order to achieve more complex lighting effects, so that objects of different materials could be properly represented.

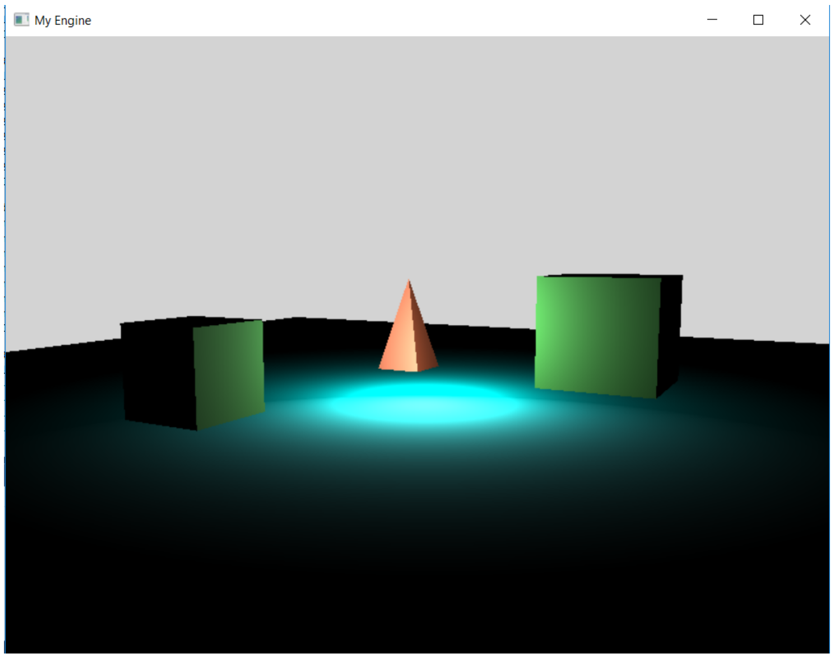

Figure 21: A view of the scene illuminated by a point light. The camera is looking at the origin of world space.

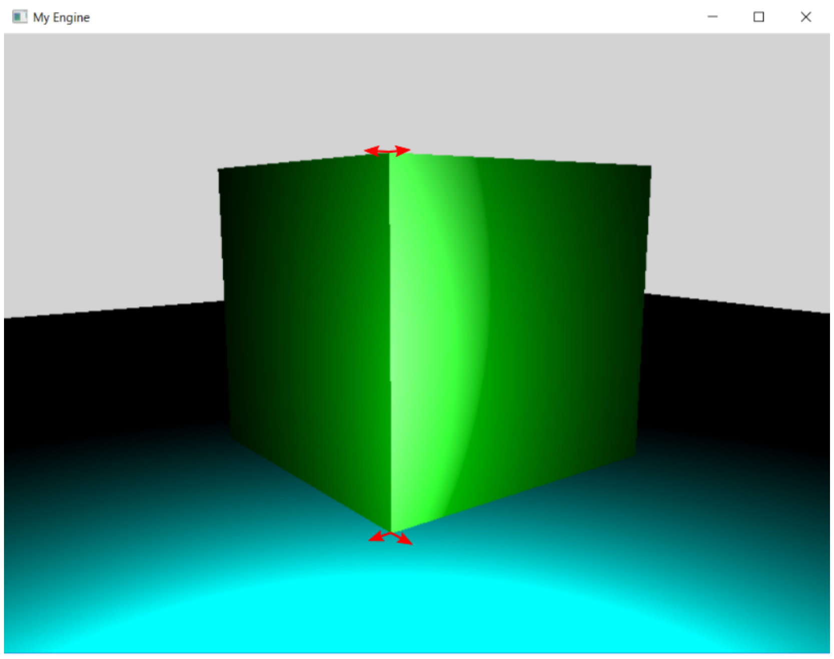

Figure 22: A close-up of the illuminated edge of one cube. The red arrows represent the vertices normals.

7 Further Developments

In the future, new features can be added. For instance, other types of light sources and reflectance effects could be implemented. Different kinds of objects and materials could be properly characterized by reflectance properties and peculiar surface looks.

Specifically, one feature that was not implemented for lack of time, nevertheless it was researched, was a map file reader. The intent was to read a particular file format: Binary Space Partitioning (BSP) file [13], which contains the information necessary to render a complete environment. A BSP file includes the geometry of all the polygons in the map and the references to the textures that need to be drawn on those polygons. Moreover, it contains the properties of all the model entities, the player initial position and physical behaviour, and the visibility table used to locate the player in the map.

By implementing a map reader, it would be possible to display many different maps and with implement editor features, in order to modify the environments and share them in the community.

8 Conclusions

This project could have been wider and covered more topics, but mainly for lack of time many features were not implemented.

Overall, the work presented many challenges. First of all, it was necessary to learn about Computer Graphics, the theory and all the techniques needed to understand and implement a project like this one. Moreover, it was crucial to learn about DirectX pipeline, its structure, and how to initialize and manipulate the API. In fact, the level of complexity of DirectX API is not indifferent.

In conclusion, the aims of the work were achieved. Basic Computer Graphics topics and concepts were analysed and implemented correctly.

References

[1] *Game Engine Architecture*; Jason Gregory; 2nd edition; 2014.

[2] *Fundamentals of Computer Graphics*; Steve Marschner, Peter Shirley; 4th edition; 2015.

[3] *Schaum's Outline of Computer Graphics*; Zhigang Xiang, Roy A. Plastock; 2nd edition; 2000.

[4] *3D Computer Graphics*; Dr. Alan Watt; 3rd edition; 1999.

[5] *Introduction to 3D Game Programming with DirectX 12*; Frank D. Luna; 2016.

[6] *Illumination for computer generated pictures*; B. T. Phong; Communications of ACM; 1975.

[7] Microsoft DirectX 12 relevance;
`http://www.pcgamer.com/what-directx-12-means-for-gamers-and-developers/`
Accessed 23-April-2017.

[8] SIGGRAPH DirectX 12 demo;
`https://software.intel.com/en-us/blogs/2014/08/11/siggraph-2014-directx-12-on-intel`
Accessed 23-April-2017.

[9] Microsoft DirectX 12 demo;
`https://blogs.msdn.microsoft.com/directx/2014/08/12/directx-12-high-performance-and-high-power-savings/`
Accessed 23-April-2017.

[10] Homogeneous coordinates;
`http://www.tomdalling.com/blog/modern-opengl/explaining-homogenous-coordinates-and-projective-geometry/`
Accessed 23-April-2017.

[11] Phong lighting model image,
`http://pages.cpsc.ucalgary.ca/~eharris/past/cpsc453/w15/tut19/`
Accessed 23-April-2017.

[12] Direct3D 12 graphics pipeline,
`https://msdn.microsoft.com/en-us/library/windows/desktop/dn899200(v=vs.85).aspx`
Accessed 23-April-2017.

[13] Binary Space Partitioning (BSP) file,
urlhttps://developer.valvesoftware.com/wiki/Source_BSP_File_Format
Accessed 23-April-2017.

Appendices

A Mini-Project Declaration

The mini-project declaration can be found attached at the last four pages of this document.

B Statement of Information Search Strategy

Relevant literature useful for the research was mainly retrieved from books, articles and web-pages.

Specifically, some books were advised by the supervisor and online material was retrieved using search engines like Google Scholar. Moreover, dedicated Computer Graphics forums (such as the Games Development section of Stack Exchange) were consulted.

The University of Birmingham

School of Computer Science

Second Semester Mini-Project: Declaration

This form is to be used to declare your choice of mini-project. Please complete all three sections and upload an electronic copy of the form to Canvas: https://canvas.bham.ac.uk/courses/21891

Deadline: 17:00, 27 January 2017

1. Project Details

Name: Federico Bacci

Student number: 1724351

Mini-project title: Implementation Of A Graphic Engine Using DirectX

Mini-project supervisor: Ian Kenny

2. Project Description

The following questions should be answered in conjunction with a reading of your programme handbook.

Aim of mini-project	Implement a graphic engine which can load a scene file and draw the polygons and objects contained in it. This will be achieved using C++ programming language and DirectX libraries.

Objectives to be achieved	Learn about Computer Graphics theory, graphic pipeline and how to manipulate scene files. Good understanding of C++ and DirectX.

Project management skills **Briefly explain how you will devise a management plan to allow your supervisor to evaluate your progress**	The project will be reviwed during periodic meetings.

Systematic literature skills **Briefly explain how you will find previous relevant work**	It will be necessary to study various Computer Graphic books in order to acquire the proper knowledge.

Communication skills **What communication skills will you practise during this mini-project?**	Writing skills, since a final report is mandatory.

3. Project Ethics Self-Assessment Form

Please answer YES/NO to the following questions:

- **Does the research involve contact with NHS staff or patients?**
  NO

- **Does the research involve animals?**
  NO

- **Will any of the research be conducted overseas?**
  NO

- **Will any of the data cross international boarders?**
  NO

- **Are the results of the research project likely to expose any person to physical or psychological harm?**
  NO

- **Will you have access to personal information that allows you to identify individuals, or to corporate or company confidential information (that is not covered by confidentiality terms within an agreement or by a separate confidentiality agreement)?**
  NO

- **Does the research project present a significant risk to the environment or society?**
  NO

- **Are there any ethical issues raised by this research project that in the opinion of the PI or student require further ethical review? If you are unsure, consider whether the project has the potential to cause stress or anxiety in the people you are involving.**
  NO

- **Human subjects can be involved as users, providers of system requirements, testers, for evaluation, or similar such activities. Does the experiment involve the use of human subjects in any other capacity? If you are unsure, answer YES.**
  NO

- **Answer YES if ANY of the following are true**

  *** the project has the potential to cause stress or anxiety in the people you are involving, e.g. it addresses potentially sensitive issues of health, death, religion, self-worth, financial security or other such issues**

  *** the project involves people under 18**

  *** the project involves a lack of consent or uninformed consent**

  *** the project involves misleading the subjects in any way**

  If the project's involvement of people relates only to straightforward information gathering, requirements specification, or simple usability testing, then you can indicate NO.
  NO

- **If any of the above questions is answered YES, or you are unsure if further review is needed (the first point is usually a good indicator - may cause stress or anxiety) then you should refer it for review.**

- **Further review will involve the School Ethics Officer meeting with the supervisor and ideally the student, reviewing the project, and suggesting any procedures necessary to ensure ethical compliance.**

DECLARATION

By submitting this form, I declare that the questions above have been answered truthfully and to the best of my knowledge and belief, and that I take full responsibility for these responses. I undertake to observe ethical principles throughout the research project and to report any changes that affect the ethics of the project to the University Ethical Review Committee for review.