Creating a Retro First Person Shooter With DirectX 11

Zoe Rowbotham
18014180

University of the West of England

February 19, 2022

Learning DirectX 11 through the creation of a retro first person shooter for Windows.

1 Introduction

Modern game engines such as Unity often hide complexities of making games, especially 3D games. DirectX is an application programming interface (API) that can be used to build games on Windows devices. By creating games through low level APIs programmers are able to begin to understand how modern game engines are structured.

First Person Shooters from the 1990s (for example: Doom) have an iconic look and feel that are still recognised today. The aim of this project is to create a retro-style first person shooter, inspired by these iconic games, using DirectX 11.

2 Related Work

DirectX 11 is an API which communicates with the hardware of a PC, for example graphics cards and sound cards (HEXUS, 2011). DirectX was originally designed to provide guaranteed compatibility through a single set of APIs (Sherrod and Jones, 2012).

To use DirectX, it is important to understand the fundamentals of Component Object Model architecture (which Windows and DirectX APIs are built upon). Microsoft, 2021 provides a useful explanation of how to start using the Windows API to create a window as well as explaining COM architecture.

Learning DirectX was the main objective of this project, Beginning DirectX 11 Game Programming (Sherrod and Jones, 2012) was helpful for this. Sherrod and Jones, 2012 explains how to initialize DirectX and also has more intermediate chapters for texturing, cameras and 3D principles, for example.

3 Method

3.1 Initializing DirectX

To initialise DirectX a Device, DeviceContext and SwapChain must be setup. The Device is used for creating resources while the DeviceContext is used to set the context for rendering. A SwapChain is a collection of buffers used to display frames, it avoids tearing of the screen when refreshing.

The DirectX Graphics pipeline shown in Figure 1 allows rendering of points, lines and triangles.

3.1.1 Vertex Buffer

After initialising the Graphics Pipeline, shapes can be rendered by adding vertices to the Vertex Buffer. The Vertex Buffer contains vertex data, and passes the information to the Vertex Shader.

3.1.2 Vertex Shader

The VertexShader handles individual vertices, allowing the shader to manipulate each vertex. Vertex Shaders typically apply transformations to move objects in world space. The data that defines a vertex is user-defined and can be different per Vertex Shader.

3.1.3 Rasterizer

The Rasterizer converts vertex information into pixel information. The raster image produced is manipulated

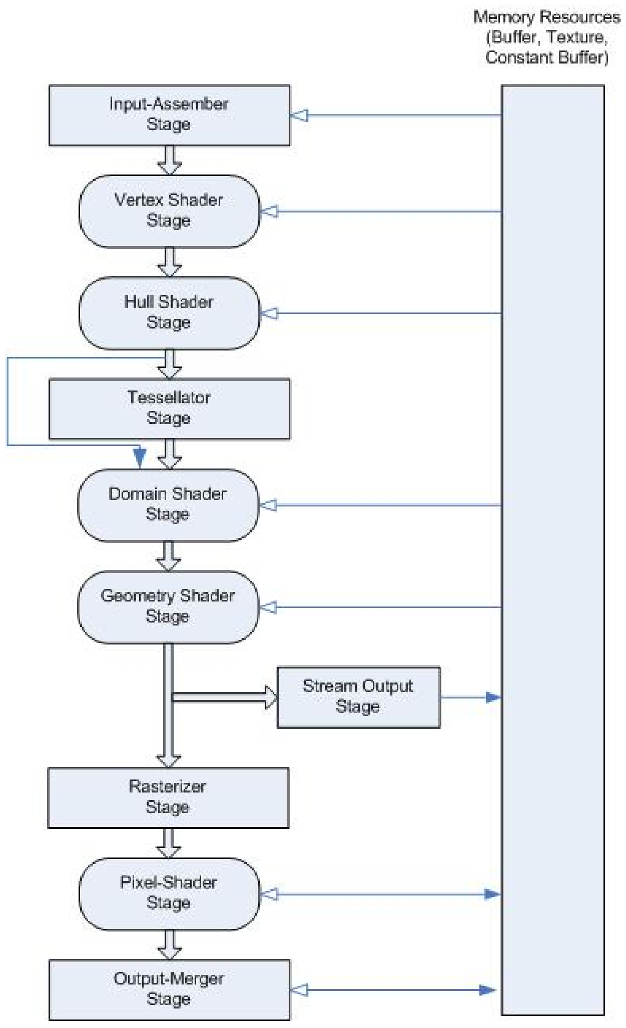


Figure 1: *The DirectX 11 Graphics Pipeline (Microsoft, 2020)*

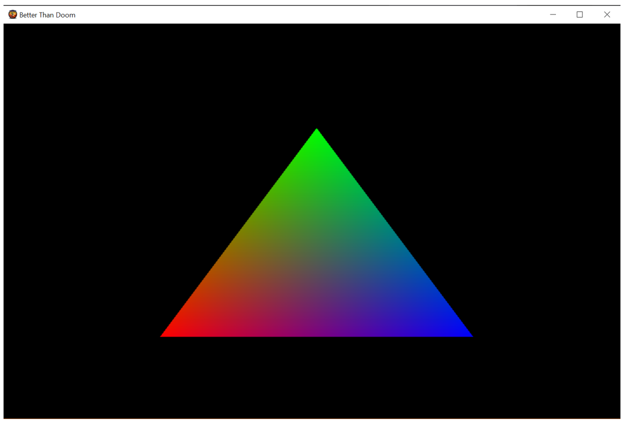


Figure 2: *Rendering of a triangle after initialising the DirectX Graphics Pipeline*

by the perspective, depth buffer and the view port.

3.1.4 Pixel Shader

The Rasterizer sends the pixels to the Pixel Shader where the pixel colour can be manipulated. The Pixel Shader runs on every pixel in an object.

Sampling a texture at a coordinate gives a colour allowing the Pixel Shader to display textures. The Pixel Shader can also be used to calculate lighting, manipulating the colour based on the lights in the scene.

3.1.5 Constant Buffers

Constant Buffers pass constant information into shaders, for example the Vertex Shader could need the transformation matrix of an object. Constant Buffer information needs to be bound to the pipeline through the Device Context before each object is drawn.

3.2 From 2D To 3D

It is logical to start with rendering a triangle (seen in Figure 2), as all shapes are formed with multiple triangles.

To aid in creating 3D shapes, an Index Buffer is useful to eliminate duplicate vertices from the Vertex Buffer. The Index Buffer indicates in which order the defined vertices should be drawn.

Figure 3 shows a 3D rotating cube; to achieve this a Depth Buffer is needed to compare the positions Z value of each object at each pixel and select the nearest one to draw.

3.2.1 Transformations

In the Vertex Shader the vertices of an object can be manipulated; the vertex position is multiplied by the object's transform matrix to give the world position of the vertex. The vertex can be multiplied further by the

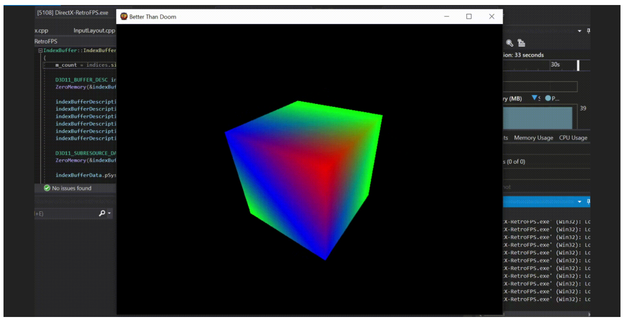


Figure 3: *A 3D rotating cube rendered after the setup of the game framework*

view and projection matrices of the camera to give the on screen coordinates of the vertex.

Rotation, movement and scaling are calculated through matrices; a matrix for a 3D transformation consists of 16 floats arranged in a 4x4 grid, as seen by the translation and scale matrices below.

$$\begin{Bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{Bmatrix} \begin{Bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{Bmatrix}$$

Understanding matrix math is important for translating objects, Pal, 2019 provides a useful explanation of matrices and how to manipulate them. The order in which the aspects of a transform are multiplied is very important, to achieve local rotation and scale the matrices should be multiplied in the order: scale, rotation, position. How to multiply two matrices is shown in Equation 3.2.1.

$$\begin{Bmatrix} a1 & a2 \\ a3 & a4 \end{Bmatrix} * \begin{Bmatrix} b1 & b2 \\ b3 & b4 \end{Bmatrix}$$

$$= \begin{Bmatrix} a1 \cdot b1 + a2 \cdot b3 & a1 \cdot b2 + a2 \cdot b4 \\ a3 \cdot b1 + a4 \cdot b3 & a3 \cdot b2 + a4 \cdot b4 \end{Bmatrix}$$

(1)

3.2.2 Projection Matrix

View and Projection matrices are 4x4 matrices that when multiplied with a 3D point finds the coordinates of the point in 2D space.

The Projection Matrix is a cuboid shape with values from -1 to 1 on each axis; transforming an object into projection space allows easy clipping of anything outside the -1 to 1 range as is it therefore out of the camera view.

3.3 Texturing and Lighting

Texturing requires loading a texture from a file and mapping the coordinates of the texture to vertices on the object. The WICTextureLoader included in the DirectXToolkit library provides an easy way to load a texture as seen below.

```
HRESULT hResult = DirectX::
    CreateWICTextureFromFile(graphics
    .GetDevice(), StringConversion::
    StringToWide(filename).c_str(),
    nullptr, m_pTextureView.
    GetAddressOf());
if (FAILED(hResult))
{
  ErrorLogger::Log(hResult, "Failed
    to load texture from file: " +
    filename);
}
```

A point light is created with a position, colour and intensity. The amount of light affecting a pixel is determined by the dot product of the direction to the light and the pixel normal, meaning a light directly over a pixel provides the most light. Lighting is applied in the Pixel Shader with a constant buffer for light information.

$$Intensity = k_{\mathrm{a}} i_{\mathrm{a}} + \sum_{m\,\in\,\mathrm{lights}} (k_{\mathrm{d}}(\hat{L}_m \cdot \hat{N}_p) i_{\mathrm{d}}).$$

Where:
$\mathrm{k}_a = AmbientColour$
$\mathrm{i}_a = AmbientIntensity$
$\mathrm{k}_d = DiffuseIntensity$
$\mathrm{L}_m = DirectiontoLight$
$\mathrm{N}_p = PixelNormal$
$\mathrm{i}_d = LightIntensity$
(2)

Equation 2 shows the calculation for ambient lighting plus the sum of the diffuse lighting in the scene. Where L is the direction to the light, i is the intensity and N is the pixel normal. The constant multipliers (k) in this report refer to attenuation and colour.

Attenuation is the loss of light over distance. The calculation for attenuation is below, where D is the distance to the light.

$$Attenuation = \frac{1}{D^2} \tag{3}$$

The results of texturing and lighting can be seen in Figure 4.

3.4 Level Generation

Reading level data from a file allows for quick and iterative level design. The file structure for level data

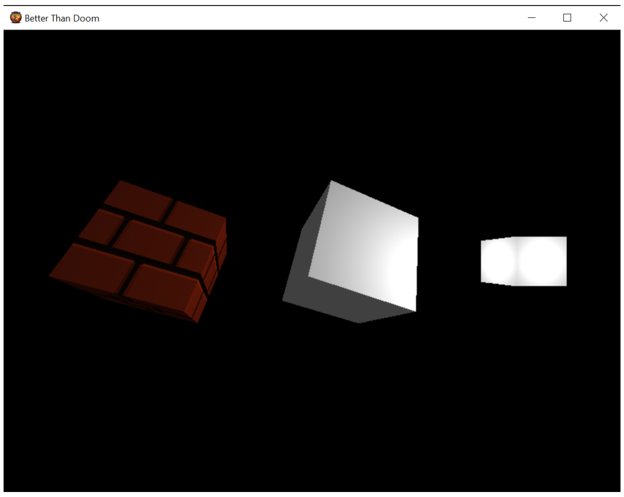


Figure 4: *From left to right: a textured cube, a solid colour cube and a point light.*

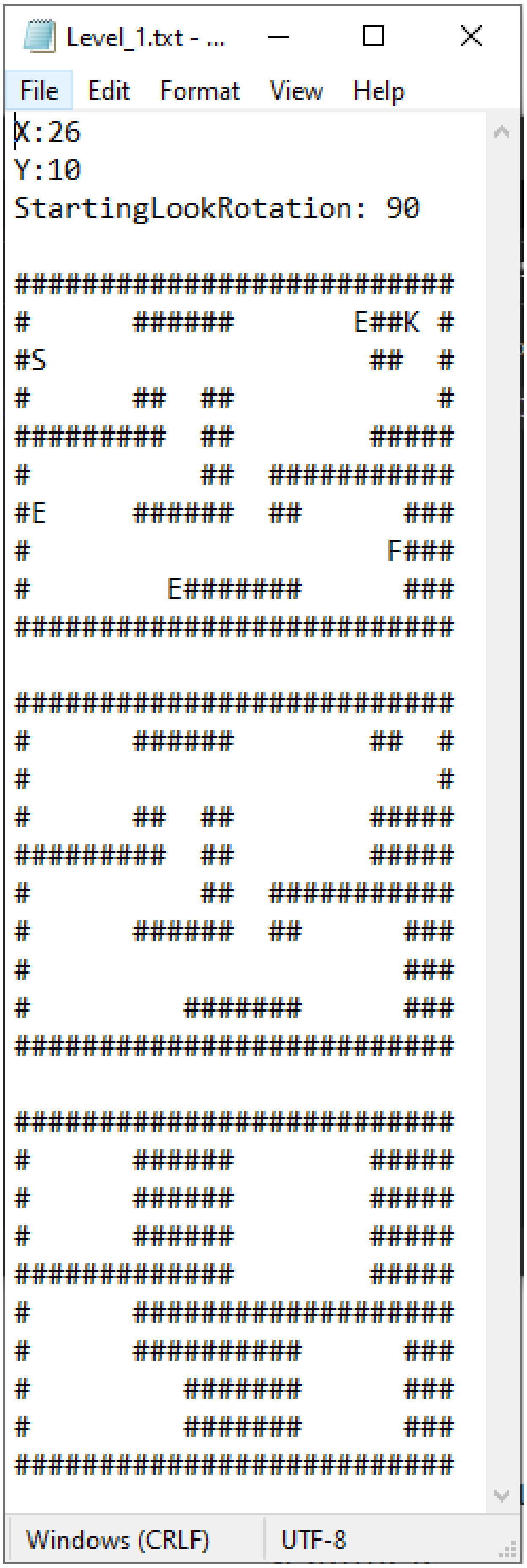
can be seen in Figure 5. The first lines dictate important information about the level, followed by the level sections; each section corresponds to a Y position, increasing by one Y unit with each section. Sections consist of key characters representing objects in the level.

3.5 Collision

Collision detection was implemented using the Separating Axis Theorem (SAT) which states if objects have a separation on at least one axis there is no collision. Using SAT means the collision detection can handle convex polygons with rotation.

SAT projects both shapes onto the normal of each vertex by using a dot product calculation, which can be seen in the pseudo code below. An overlap is calculated from the minimum and maximum values of the projection as explained by Pikuma, 2021. This can be seen in Figure 6 with vertices projected on to the axis created by the normal shown in red; since the minimum and maximum values on the axis do not overlap there is a separation and the objects are not colliding.

```
separation = -infinity
FOR i = 0 to numVertsA
  minSep = infinity
  FOR j = 0 to numVertsB
    dir = bVerts[j]-aVerts[i]
    proj = dot(dir, aNors[i])
    IF proj < minSep THEN
      minSep = proj
    END IF
  END FOR
  IF minSep > separation THEN
    separation = minSep
  END IF
END FOR
```



```
X:26
Y:10
StartingLookRotation: 90

##########################
#      ######       E##K #
#S                   ##  #
#      ##  ##            #
#########  ##        #####
#          ##  ###########
#E     ######  ##      ###
#                     F###
#        E#######      ###
##########################

##########################
#      ######        ##  #
#                        #
#      ##  ##        #####
#########  ##        #####
#          ##  ###########
#      ######  ##      ###
#                      ###
#         #######      ###
##########################

##########################
#      ######        #####
#      ######        #####
#      ######        #####
#############        #####
#      ###################
#      ##########      ###
#         #######      ###
#         #######      ###
##########################
```

Figure 5: *Example of the LevelData files for generating a level in the game.*

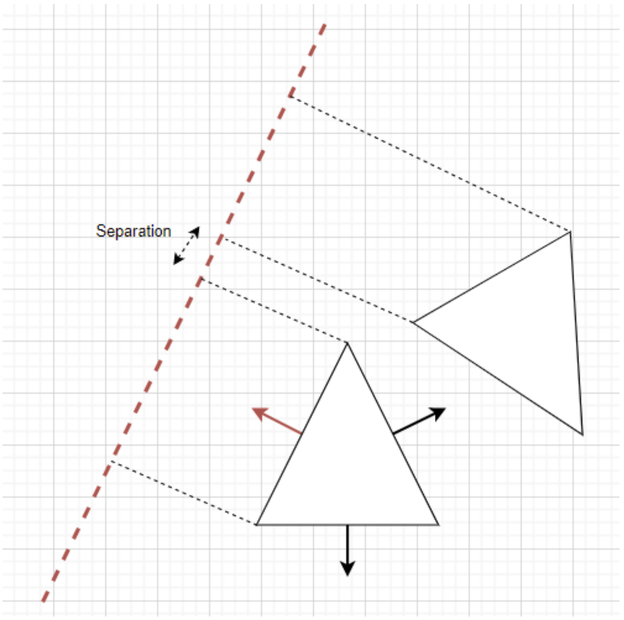


Figure 6: *The Separating Axis Theorem, showing a separation found.*

Collision resolution for a static and a moving object requires pushing the moving object out of the static one; this is done by translating the moving object along the normal of the face of the colliding object, with a magnitude of the overlap value.

3.6 Billboards

Enemies and pickups are presented as billboards in the game. Billboards are 2D shapes that rotate to face the player. This is the typical style of enemy for retro FPS games.

Rotation to the player is calculated by the angle between the forward vector of the billboard and the vector from the billboard to the player as seen in Equation 4. Where the angle should be greater than 180° the angle should be subtracted from 2π.

$$a = \arccos\left(\frac{f \cdot p}{|f||p|}\right) \tag{4}$$

3.7 Shooting

Shooting with rays instead of spawned projectiles means game play with a faster pace.

A ray is created and collisions are detected between the level objects and the ray; only the closest object collided with is considered 'hit'.

To calculate a ray and an oriented bounding box collision, each axis needs to be checked against the other axes to determine if the ray has overlapped.

The code below shows how to compare the X and Y axis to determine an overlap, where the min and max point of the collider are transformed by the transformation matrix.

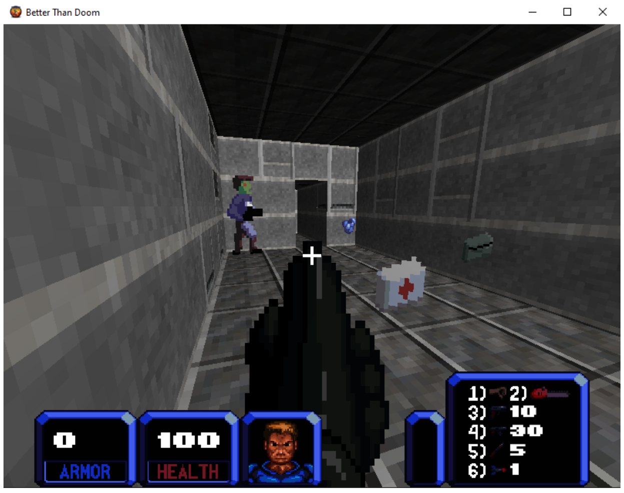


Figure 7: *Screenshot showing the final game produced.*

```
Vector min = (collider.GetMinPoint()
    - ray.Origin) / ray.Direction;
Vector max = (collider.GetMaxPoint()
    - ray.Origin) / ray.Direction;

float tmin = min.X;
float tmax = max.X;

if((tmin > max.Y) || (min.Y > tmax))
{
  return false;
}

tmin = std::max(tmin, min.Y);
tmax = std::min(tmax, max.Y);
```

3.8 User Interface

UI elements are 2D objects where their position is updated to always be drawn in front of the camera. To avoid clipping issues with world objects, the depth stencil should be reset. UI objects should also be unlit as there is no need for lighting to affect the UI.

4 Evaluation

The final game is shown in Figure 7, with enemies, weapons, pickups and levels; players collect the key of each level and collide with exit door to complete it.

Collision resolution is very simplistic, for games with a higher physics demand the current solution would not suffice. Calculations for acceleration, impulse and mass would need to be included to improve the collision resolution for moving objects with physics attributes.

With the current system it is difficult to align UI elements and the elements do not scale with the screen size. Improving this would allow much easier UI creation and more flexibility.

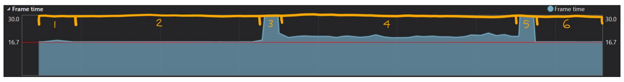


Figure 8: *Time needed to draw frames in the final game.*

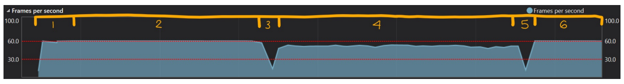


Figure 9: *The frames per second of the final game (limited to 60FPS).*

The frame time over both of the levels was evaluated to determine the difference in performance.

Figures 8 and 9 show the frame time and the frames per second respectively. Figures 8 and 9 are annotated with the following game stages:

1. Main Menu
2. Level 1
3. Loading Level 2
4. Level 2
5. Unloading Level 2
6. Main menu

There is a large drop in frame rate when levels are loaded, this is when previous level objects are disabled and new level objects enabled. The frame rate also drops while playing Level 2, this is because Level 2 is larger than Level 1. The table below shows the number of objects in each level.

Type	Level 1	Level 2
Wall	837	1377
Enemy	4	10
Pickup	11	24

For optimisation, walls could be merged to dramatically reduce the number of objects drawn each frame.

5 Conclusion

This project aimed to create a retro style first person shooter similar to Doom using the DirectX 11 API. Creation of an FPS game was successful with design decisions creating a visually similar experience to the first 3D first person shooters. Overall, the project achieved a good level of understanding of DirectX and a complete game was produced.

Bibliography

HEXUS (2011). *DirectX 11*. Last accessed 21 October 2021. URL: `https://hexus.net/tech/tech-explained/graphics/32104-directx-11/1`.

Microsoft (2020). *Graphics Pipeline*. Last accessed 21 October 2021. URL: `https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline`.

– (2021). *Get Started with Win32 and C++*. Last accessed 21 October 2021. URL: `https://docs.microsoft.com/en-us/windows/win32/learnwin32/learn-to-program-for-windows`.

Pal, Shakunt (2019). *Understanding 3D matrix transforms*. URL: `https://medium.com/swlh/understanding-3d-matrix-transforms-with-pixijs-c76da3f8bd8`.

Pikuma (2021). *Math for Game Developers: Collision Detection with SAT*. Youtube. URL: `https://www.youtube.com/watch?v=-EsWKT7Doww&ab_channel=Pikuma`.

Sherrod, Allen and Wendy Jones (2012). *Beginning DirectX 11 Game Programming*. Course Technology.