AT Task 1 Report DirectX 11 Old School FPS

Ryan Manning
18026856

University of the West of England

April 27, 2021

This report outlines the making of a old school first person shooter.

1 Introduction

Retro first person shooters are some of the most iconic games in history(reference to doom/ wolfenstien article). The task is to create a first person shooter similar to these iconic games using the directX 11 library. This solutions should have a working camera, a basic level and some enemies that are billboards.

2 Related Work

Alot of games use directX to get the most out of the graphics for their game. Some of the research i did for this was to try and work out how this is done. I found this Allen Sherrod, 2012 where it gives a nice beginners guide to the render pipeline and allowing somebody to start in directX. As well as this i found Stenning, 2014 this that goes into some more details on directX. This includes tesellation, rendering and animating meshes.

3 DirectX 11

To start to use directX first the directX render pipeline needs to be set up. The first thing to set up is a swap chain this will allow the program to be able to draw and render images to the screen. This is done as while the frame that is on the screen is done rendering the swapchain draws the next frame underneath this image then once this has been completed the swapchain will stop drawing and switch to the frame behind the first one. This then carries on and is how directX updates the screen. Once this is set up the next thing that needs to be initialized is the actuall pipe line. This consisits of the imput assembler which is how directX finds your graphics card so that it can use it. The vertex shader is something that needs to be programmed and once this is done can be used to start drawing objects to the screen.Next is the rasterizer this determines which pixels to render. This can be seen in figure 1. Next is the pixel shader this uses the data from the rasterizer and then outputs a colour and a depth. The next stage is the output merger performs the actuall render to the render target so that this can all be shown on the screen. Finally this is then brought to the render target as said previously this is just the target at which all of this is going to show up on. Some of the stages of the pipeline have been missed out such as tessellation and geometry shading this is because for a basic implementation this isn't needed or can be set to a default in which everything will still run as intended. As well as this to use directX certain buffers have to be set up. All these do is store all the data related to that thing. The three buffers that will need to be set up are vertex, index and constant. With the vertex buffer storing information on the vertex data, the index on which index of the vertex buffer to render and the constant buffer to constantly apply a shader or texture to the object.

4 Method

Firstly i had to create the window that my game would run in using the WIN32 library this was fairly simple. Once this done next was setting up the directX libraries and then initalizing the directX render pipeline. To

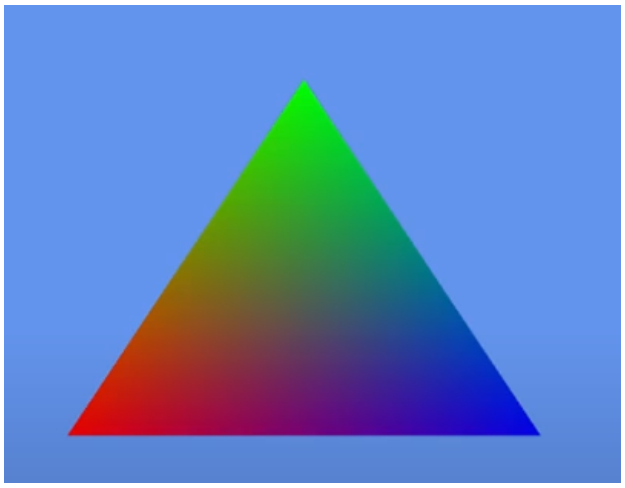

Figure 1: *Rasterizer example*

```
//Textured Square
Vertex v[] =
{
    Vertex(-10.0f,  -6.0f, -1.0f, 0.0f, 1.0f), //FRONT Bottom Left   - [0]
    Vertex(-10.0f,   6.0f, -1.0f, 0.0f, 0.0f), //FRONT Top Left      - [1]
    Vertex(10.0f,    6.0f, -1.0f, 1.0f, 0.0f), //FRONT Top Right     - [2]
    Vertex(10.0f,  -6.0f, -1.0f, 1.0f, 1.0f), //FRONT Bottom Right   - [3]
    Vertex(-10.0f,  -6.0f, 1.0f, 0.0f, 1.0f), //BACK Bottom Left    - [4]
    Vertex(-10.0f,   6.0f, 1.0f, 0.0f, 0.0f), //BACK Top Left       - [5]
    Vertex(10.0f,   6.0f, 1.0f, 1.0f, 0.0f), //BACK Top Right      - [6]
    Vertex(10.0f,  -6.0f, 1.0f, 1.0f, 1.0f), //BACK Bottom Right   - [7]
};

//Load Vertex Data
HRESULT hr = vertexBuffer.Init(this->device, v, ARRAYSIZE(v));
if (FAILED(hr))
{
    return false;
}

DWORD indices[] =
{
    0, 1, 2, //FRONT
    0, 2, 3, //FRONT
    4, 7, 6, //BACK
    4, 6, 5, //BACK
    3, 2, 6, //RIGHT SIDE
    3, 6, 7, //RIGHT SIDE
    4, 5, 1, //LEFT SIDE
    4, 1, 0, //LEFT SIDE
    1, 5, 6, //TOP
    1, 6, 2, //TOP
    0, 3, 7, //BOTTOM
    0, 7, 4, //BOTTOM
};

//Load Index Data
hr = indexBuffer.Init(this->device, indices, ARRAYSIZE(indices));
```

Figure 2: *Code for initializing a basic cube*

do this the swap chain needed to be set up along with the directX device. This is just a lot of initializing of different buffers such as the back buffer, render target and the depth stencil. This is all needed to start setting up the render pipeline as i mentioned in the previous section. This is all being doing in the graphics class as this is where most of the directX initialization is as its directly involved in the graphics of the project.

At the start i used some of the buffers and set everything up in the graphics file this ended up that the file got very messy and hard to manage therefore i created different files for some of the functionality that i could move out of that class such as the constant buffer, vertex buffer, index buffer.For the constant buffer and the vertex buffer i used a template class as this allowed them to provide all the functionality that they do anyway just in a condensed and more manageable place.

Once the directX render pipeline had been completely set up i started work on getting a cube drawn to the screen as once this was done it would mean i can start to build my level with instantiated cubes. To create a 3d cube figure 2 first i set up the respective class and started to add in the struct of vertexes. This will draw the vertices of the cube. Then set up the draw order of the indices this is in what order the vertices are drawn in. Once all of this has been completed the cube class needs to update the world matrix this is so that the cube can be calculated so its ready to be drawn in the scene. Next i need to actually draw the cube in the render function. This is done through updating the constant buffer with vertex matrix data of the cube. Then its just using the device context element to call the DrawIndexed function to actually render the cube in the scene.

Next was setting up the camera. To do this i needed to check when ever the player presses a button or moves the mouse. Then apply these transforms and rotations into the constant buffer for the camera based off of the base vectors that are in the class. As well as this i have added some functions to set the position and

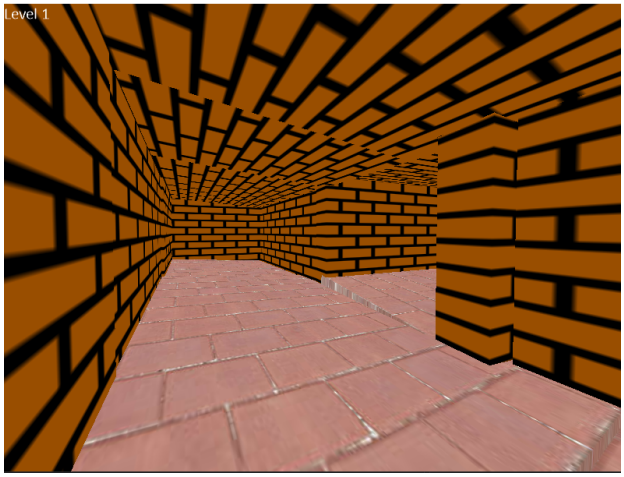

Figure 3: *Screenshot of the level running*

rotation of the camera. This was mainly about getting input working and then creating some form of delta time in the timer class so that i can apply the delta time value to everything that moves which should just be the camera.

Finally i needed to create the level to do this i created two arrays of cubes and used two different textures for each of them. Then i just used for loops to initialize them all. Once they have been initialized then i set all of the walls rotation and position to build the level figure 3.

5 Evaluation

A lot of the system that was implemented for the prototype is not needed. However this system now holds as a good base to improve and develop the prototype after the basics have been completed. This is the reason for adding some of the extra classes. An example of this is the error logger class as it is not needed in the project

however it meant that when debugging error messages and such i could understand what the problem was quicker.

The prototype is a good slice of a game and has the basics completed well. The improvements that could be made to this would be to add collision detection for the level, shooting and enemies that can shoot back and are billboards. As well as the mouse movement only works when moving around so the player can't look around on the spot this is something that has a small impact on the prototype.

6 Conclusion

I created a simple level using directX 11 which has a camera that is the player and the player can move around with. The level has two types of texture cube in it that makes up the small level.

Bibliography

Allen Sherrod, Wendy Jones (2012). "begining DirectX 11 Game programming". In: *ACM siggraph computer graphics*. Vol. 1. 1. Course Technology.

Stenning, Justin (2014). *Direct3D Rendering Cookbook*. Vol. 1. 1. Packt Publishing Ltd.