Advanced Technologies: DirectX11 First Person Shooter

Alexander Hillman
Student Number: 19021645

University of the West of England

January 9, 2023

The aim of this report is to detail the process of creating a first person shooter within DirectX11. The final product, whilst no achieving all of the goals established below, demonstrates the steps required to produce a simple project within DirectX11. This combines DirectX11 and c++ to create an old school style level in which three dimensional cubes are rendered and a camera is able to traverse the world.

1 Introduction

The set task is to create a first person shooter video game using DirectX11. DirectX is a series of APIs that run on Microsoft Windows operating systems which provide low-level access to hardware components such as the graphics and sound cards as well of the operating system. As such, it allows for objects to be rendered to a screen or audio to be played which is primarily required for video games and other multimedia applications.

The final product should be in the style of old school retro shooters such as Wolfenstein 3D or Duke Nukem 3D (3D Realms, 1996). The game should include bill-boarded enemies and objects, the player should be able to move around a generate game level, shoot and be able to detect collision with other objects. Finally the game should include texturing of assets and lighting. Stretch goals for this project are to make a complete game with completion conditions, audio and other general polish.

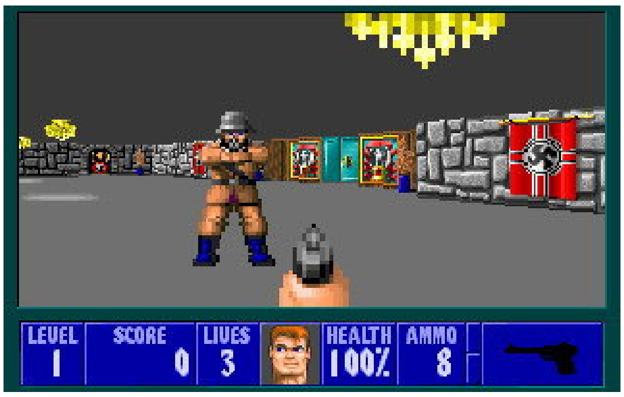


Figure 1: *Wolfenstein 3D gameplay*

2 Related Work

2.1 Similar work

Games which are similar to this work such as Wolfenstein 3D (id Software, 1992) and Doom (id Software, 1993) will be heavy inspiration for how the game looks, feels and plays.

The level layout for Wolfenstein 3D (id Software, 1992) could be replicated with simple cubes rendered with different textures and positions. For the character movement, the player is able to move in forward and backwards with the W and S keys and rotate with A and D keys, holding shift with A and D allows for the player to strafe left and right. Enemies are 2 dimensional textures that are bill-boarded to constantly be facing the direction of the player and shooting can be handled by ray-casting from the player's forward direction. These factors will be the main things to consider when creating the game.

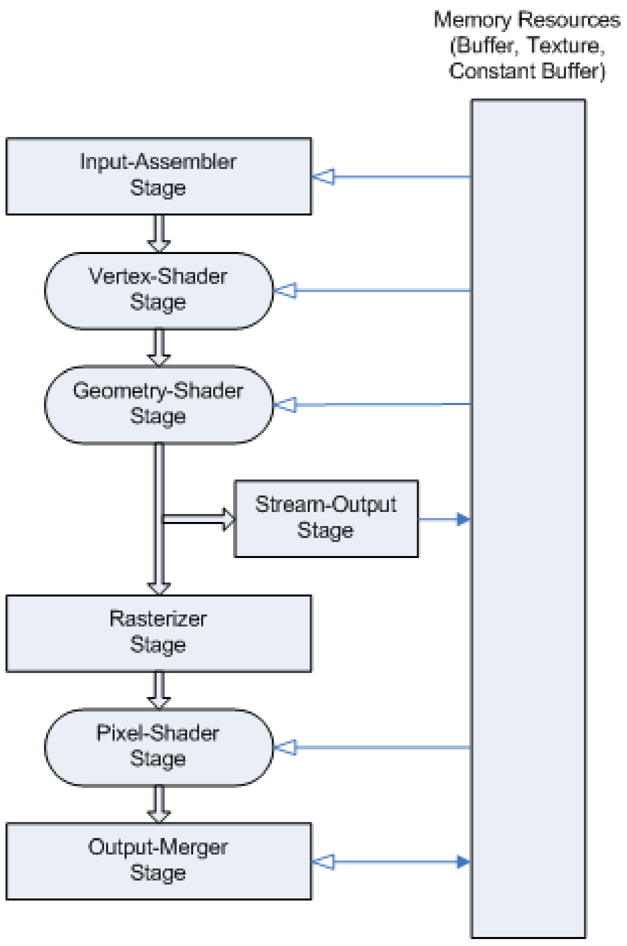


Figure 2: *Render stages for Direct3D*

2.2 DirectX breakdown

The first stage of DirectX 11's rendering pipeline is the input assembler stage. This stage is where primitive data is read in from data buffers such as index or vertex buffers and assembled into primitives such as line lists or triangle strips.
This data is then passed to a vertex shader to execute transformations, skinning, morphing and vertex lighting.
The Rasterizer stage then takes the primitives and converts it to pixel based raster image. This also clips vertices to the view frustum, provides perspective, maps primitives to a two dimensional viewport and determines how to carry out the pixel shader.
A pixel shader allows for pixel lighting and post-processing. It combines constant variables and, texture data and vertex values to produce per-pixel outputs such as colour to which is written into a render target.
The final stage for rendering is the output merger stage, this generates the final rendered pixel colour using the pixel data generated by the pixel shaders, the contents of the render targets and the contents of the depth and stencil buffers. This also determines which pixels are visible using the depth-stencil buffer and blends the final pixel colours.

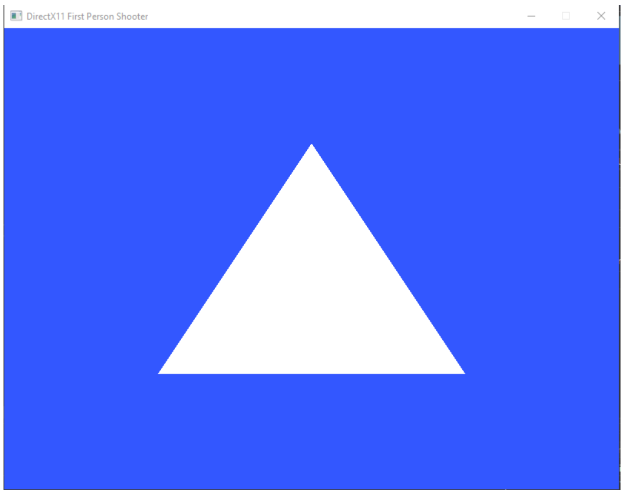


Figure 3: *A triangle rendered using DirectX11*

3 Method

3.1 Creating and Clearing a Window

After a window was set up and cleared using in c++, DirectX's Windows API and DirectX3D, a renderer class was created to handle what was being shown to the screen and . A swap chain allowed for frames to be drawn to the back buffer and then moved to the screen buffer to update and draw frames to the window. The window was cleared to a static colour and then presented to the screen as a single frame. An application class was added that contains an update function which runs while the window is active to allowed for more than one frame to be displayed. In combination with a timer class to track the elapsed time, the update function constantly updates the colour value that the screen cleared and rendered the frame so that more than one could be presented over the lifetime of the window.

3.2 Rendering a Triangle

To render an object to screen such as a two dimensional triangle, structures were needed to store vertex information as well as data concerning each of its individual vertices. A vertex was defined as two floats to store x and y positions. Then an array of three vertices was used to store the three points of the triangle. A vertex buffer then holds the data concerning the triangle's vertices and is bound to the input-assembler. Pixel and vertex shaders are then created and set to the pipeline stages.The end result was the back buffer cleared to a block colour with a block white triangle (figure: 3).

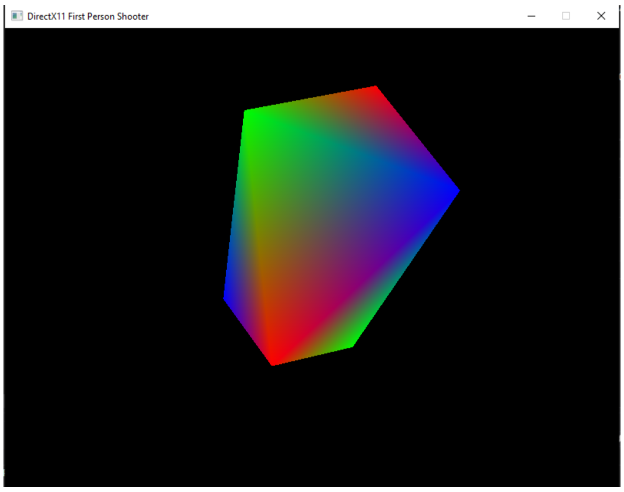


Figure 4: *A complex shape rendered in DirectX11*

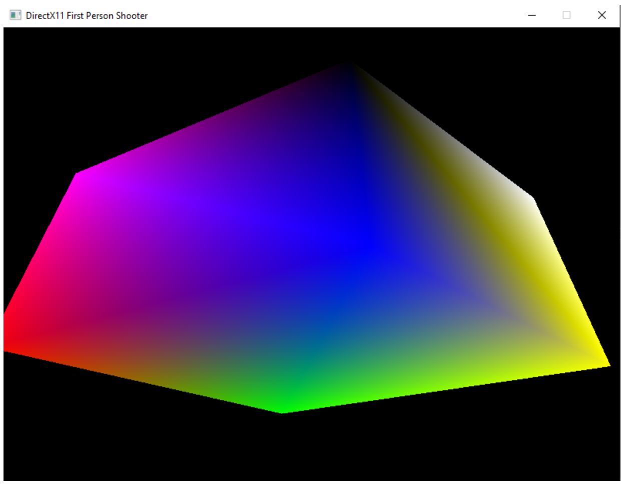


Figure 5: *A 3 dimensional cube rendered in DirectX11*

3.3 Rendering a More Complex Shape

Colour values were added to the vertex structure so that each vertex of the rendered object could be coloured differently. More vertices were also added to generate a more complex shape, however this time each vertex in the array took 6 values, the x and y coordinates as well as the RGBA values. With an increased number of vertices, a index array was needed to track and reuse vertices. This also meant that an index buffer had to be created and bound to store the index values. A constant buffer also was used to store the transformation matrix. The result from this was a multi-coloured polygon that rotated around a point on screen (figure: 4).

3.4 Rendering a 3D Cube

The next step was to transition to rendering a three dimensional cube. This required more vertices and indices to be added since each face of the cube is composed of two triangles. The constant buffer was also updated to use the DirectXMath library so that the object could be transposed on the x, y and z planes to rotate the object in three dimensional space (figure: 2).

3.5 Rendering Multiple Objects

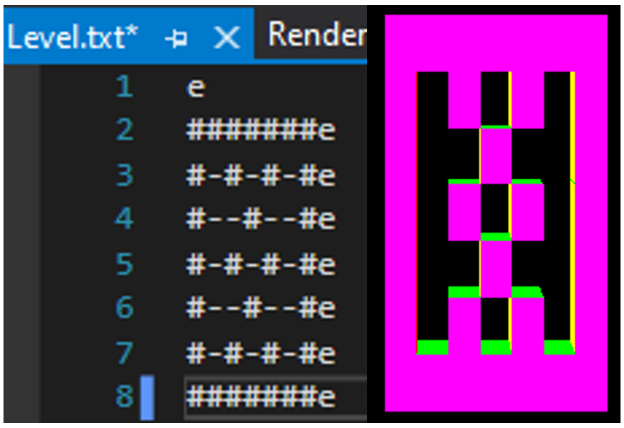
The colouration of the cube changed from per vertex colour to per face so that each face was a uniform colour. The colours were stored in a second separate constant buffer, independent from the vertex position. The position of the drawable was then taken out of draw function within the renderer and instead was used a arguments within the function call and as a result of this the draw function can be called numerous times with different parameters to draw multiple cubes with different positions. With multiple objects being rendered to screen a depth stencil is needed so that objects which are obscured by other objects can be culled from rendering.

3.6 Generating a Game Level

To generate the map for the game, a text file was used to store layout on a two dimensional plane. Different characters were used to distinguish between different aspects of the map. The hash character was used to denote where a cube should be placed and the letter e was used to identify the end of a line. The result is a text file that can be altered to quickly and easily change the layout of a level (figure: 6). To create the objects from the text file, the level text file is opened then iterated through until is reaches the end of the file. If the character read is the same as the endline character then the column index is incremented to go to the line below and the line index is reset to zero to start from the beginning of the next line. If character being read is a block character then the draw function is called with the line and column indexes used as two of the cube's positional values, each cube is drawn with the same zero rotation and one positional value so that the cubes are aligned and only offset of two dimensions (figure: 6).

3.7 Camera and Movement Around the Map

A keyboard class was implemented to handle input for the game. This helps to keep track of keyboard actions such as which keys have been pressed or released. This class was then used in conjunction with the created camera class to maneuver around the scene. The camera handles the transformation, rotation, view and projection matrices. The camera can be moved forward and backward using the W and s keys and rotated using the A and D keys. The camera is set to



```
e
#######e
#-#-#-#e
#--#--#e
#-#-#-#e
#--#--#e
#-#-#-#e
#######e
```

Figure 6: *A text file detailing the layout of a level and a top down perspective of the map generated*

an initial position within the level and then then the application constantly checks if one of the four keys have been pressed which then in turn either moves or rotates the camera around the scene.

4 Evaluation

From analysing the memory usage of the game running, it is evident that the game runs at a constant memory usage on average of 14MB with little to no impact on frame-rate, there is however a spike in overall CPU usage in which it increases to 43 percent, however it is unclear what is causing this. This analysis of suggests that more can be done for this game, such as introducing more memory taxing tasks. Aspects such as a larger map or the introduction of enemies and bullets on screen would be such possible next steps as there would be memory free to allocate towards this.

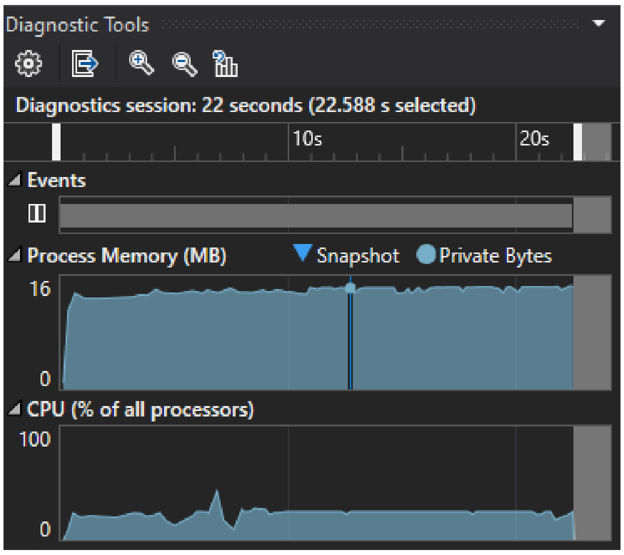

Diagnostic Tools
Diagnostics session: 22 seconds (22.588 s selected)
10s
20s
Events
Process Memory (MB)
Snapshot
Private Bytes
16
0
CPU (% of all processors)
100
0

Figure 7: *Memory and and CPU usage of game running*

5 Conclusion

The final result from this project is a level that is composed of numerous three dimensional cubes placed using a text file and a camera that is able to traverse said level. Whilst not achieving all the goals that were set out, the task has accomplish some.

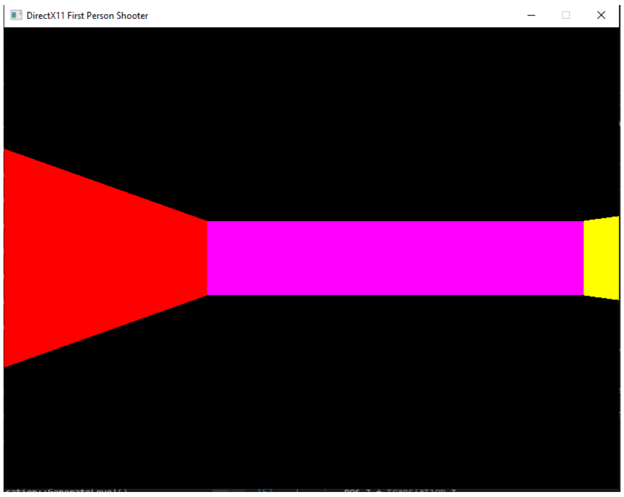


Figure 8: *The final product, a three dimensional level created in DirectX11*

References

3D Realms (1996). *Duke Nukem 3D*.
id Software (1992). *Wolfenstein3D*.
– (1993). *Doom*.

6 Appendix

`https://youtu.be/CE7_X3_P5Ig` - Week 1 video

`https://youtu.be/8dCKehABoNE` - Week 2 video

`https://youtu.be/EgcOvUDnLLY` - Week 3 video

`https://youtu.be/mTFzQuANbIQ` - Week 4 video

`https://youtu.be/9WpUKG6XT6k` - Week 5 video