Creating A 3D Platformer Game In DirectX 11

Luke Tolchard
17024063

University of the West of England

July 31, 2020

This report outlines the principles behind the DirectX API and how it was used to create a game-like scenario

1 Introduction

DirectX 11 is a rendering library for writing high performance 3D graphics applications using modern graphics hardware, with the predominant consumer of the technology being the games industry (Luna, 2012). As computer hardware has evolved to have more computational strength, Many non-3D applications are offloading heavy calculations to the GPU to reduce the duress. The Direct3D library provides the compute shader API for writing programs that allow a developer to fully utilise the GPU and create powerful programs.

2 Related Work

Zioma and Green, 2012, gave a talk at the Game Developers Conference for Nvidia outlining the DirectX implementation into Unity, and the new features it offered to game developers, including the Catmull-Clark subdivision method. It is also widely used in the development of Windows applications, and used as a basis for the Xbox API (Carter, 2007). The implementation of this report is based on the lessons provided by Iedoc, 2015

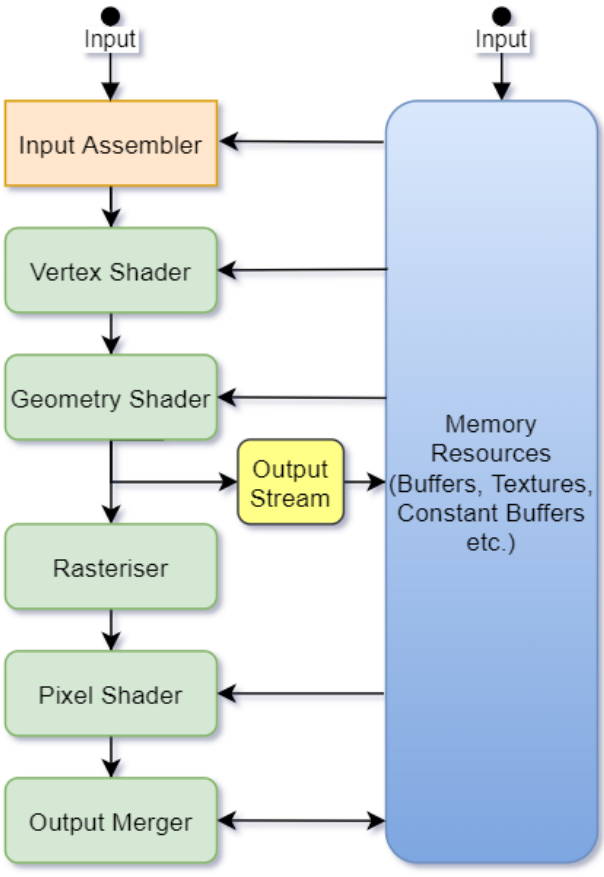

Figure 1: *A flowchart representation of the DirectX pipeline*

3 Direct X 11

3.1 Direct X 11 Pipeline

Figure 1 displays a flowchart representation of the DirectX API working as an interface between the operating system and the graphical hardware, instructing it on how to render objects. The sections shaded green feature common shader cores are programmable by using High Level Shading Language (HLSL) which make the pipeline extremely flexible and adaptable (Carter, 2007).

3.2 Swap Chain

A graphics adapter holds a pointer to a surface that represents the display on a monitor, called a front buffer. As the monitor display is refreshed, the GPU will send the contents of the front buffer to the monitor to be displayed. However, when rendering real-time graphics this method could cause a problem. Monitor refresh rates are very slow in comparison to the time it takes for hardware to compute the next frame to be displayed, and if the application updates the front buffer while the monitor is halfway through a refresh, the image displayed will be cut in half with a combination of old and new frame in the same display. Back buffering solves this issue by drawing another frame in an off-screen surface called a back buffer. The application then has the freedom to prepare the next display for the monitor whenever the system is idle without having to consider the monitors refresh rate. A process known as 'surface flipping' will then move the back buffer to the front buffer by flipping the surface pointers, creating a swap chain and preventing screen tearing (Microsoft, 2020).

3.3 Vertex, Index and Constant Buffers

The Input Assembler (Figure 1), collects data from the video memory and prepares it for use in the pipeline. It requires data from a collection of buffers: the vertex buffer, the index buffer, and the constant buffer.

The vertex buffer contains the data used to define geometry within the scene, which could include position, colour, texture data etc. Each vertex in the vertex buffer is identified by an index in the index buffer, which can be used to render primitives more efficiently (Microsoft, 2020). A constant buffer then allows the transition of data from the GPU to the CPU.

The vertex shader is where the values held in the buffer can be manipulated, with some examples of this being a coordinate translation, or a normal calculation. The pixel shader is used to manipulate the faces of the drawn polygons, which are run by the GPU for every visible pixel drawn to screen. They take in many effects such as colour, texture and lighting to produce an output value at the end (Rastertek, 2010).

3.4 Transform, Rotations, and Scales

Transforms, rotations, and scale calculations are computed using matrices in DirectX. These matrices are shown in the equations below, along with their attached syntax from the DirectX library.

$$[x_1 y_1 z_1 1] = [xyz1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix} \quad (1)$$

The transformation matrix (1), which is called by $XMMatrixTranslation$

$$[x_1 y_1 z_1 1] = [xyz1] \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2)$$

The scale matrix (2), which is called by $XMMatrixScaling$

$$[x_1 y_1 z_1 1] = [xyz1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & sin\theta & 0 \\ 0 & -sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3)$$

The rotation matrix (3), which rotates around the x-axis, is called by $XMMatrixRotationX$

$$[x_1 y_1 z_1 1] = [xyz1] \begin{bmatrix} cos\theta & 0 & -sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4)$$

The rotation matrix (4), which rotates around the y-axis, is called by $XMMatrixRotationY$

$$[x_1 y_1 z_1 1] = [xyz1] \begin{bmatrix} cos\theta & sin\theta & 0 & 0 \\ -sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5)$$

The rotation matrix (5), which rotates around the z-axis, is called by $XMMatrixRotationZ$

One advantage of using matrices to describe these effects is that you can multiply them together to create one composite matrix that contains the effects of all products but only needs one call in the script in order to be applied. This process is known as matrix concatenation, and is defined by the formula: $C = M_1 \cdot M_2 \cdot M_{n-1} \cdot M_n$ where C is the composite matrix, and the M components are the individual matrices. Microsoft documentation (2020) states that in most cases only 2 or 3 matrices are likely to be concatenated but there is no inherent limit.

3.5 Render States

In DirectX, there are three render states that encapsulate settings that can be used to configure the API. The

three states are the $RasteriserState$, the $BlendState$ and the $DepthStencilState$.

The RasteriserState is used to customise the rasteriser stage of the pipeline (Iedoc, 2020), which is used to convert vector information in the form of shapes or primitives into raster images (pixels), for the purpose of displaying real-time 3D graphics. The BlendState can be used to make transparent primitives and have their textures blend together by calculating the colour of the object's pixels behind the transparent primitive with the transparent object itself. The DepthStencilState tells the output merger how to perform the depth-stencil test, which determines whether or not a given pixel should be drawn.

4 Implementation

4.1 Camera

In this implementation, there is a 3rd person vector camera behind the player controlled character. The camera is initialised as such:

```
1     camPosition = XMVectorSet( 0.0f,
      10.0f, 8.0f, 0.0f );
2     camTarget = XMVectorSet( 0.0f,
     3.0f, 0.0f, 0.0f );
3     camUp = XMVectorSet( 0.0f, 1.0f,
      0.0f, 0.0f );
```

It can freely look around the player with mouse movements, as well as being connected to the character movement in a way that means that the forward input is always in the direction the camera is facing, rather than where the character is facing. There are also clamps to stop it from going under or over the player in a reflex angle.

```
1    if((mouseCurrState.lX !=
     mouseLastState.lX) || (
     mouseCurrState.lY !=
     mouseLastState.lY))
2     {
3          camYaw += mouseLastState.lX
     * 0.002f;
4          camPitch += mouseCurrState.
     lY * 0.002f;
5          if(camPitch > 0.85f)
6               camPitch = 0.85f;
7          if(camPitch < -0.85f)
8               camPitch = -0.85f;
9          mouseLastState =
     mouseCurrState;
10    }
```

4.2 Creation of Objects

All 3D objects and scenes are composed of arrangements of triangles to form meshes (Iedoc, 2020). To draw a triangle to screen, one must define vertex's in the vertex buffer to describe the co-ordinates of each corner.

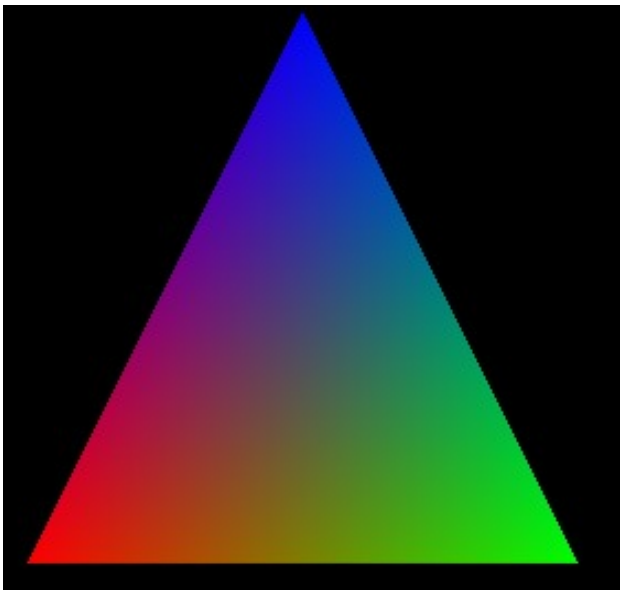

Figure 2: *A coloured triangle rendered with 3 defined vertices*

```
1     Vertex v[] =
2     {
3          Vertex( 0.0f, 0.5f, 0.5f,
     1.0f, 0.0f, 0.0f, 1.0f ),
4          Vertex( 0.5f, -0.5f, 0.5f,
     0.0f, 1.0f, 0.0f, 1.0f ),
5          Vertex( -0.5f, -0.5f, 0.5f,
     0.0f, 0.0f, 1.0f, 1.0f ),
6     };
7
```

This vertex struct contains three values for the x,y,z co-ordinates of the vertex in world space, and it also contains four more variables to determine the RGBA of that vertex, which produces the image in Figure 2.

Instead of redefining 3 new vertices to make another triangle and convert this image into a square, it makes sense to reuse the two vertices that would connect with the new triangle. The two triangles are therefore defined as 0,1,2 and 0,2,3 in the index buffer, and another vertex is added to the vertex struct to complete the square (Figure 3).

```
1   DWORD indices[] =
2   {
3         0, 1, 2,
4         0, 2, 3,
5   };
6
```

4.3 Texturing

In DirectX, there is a 2D (u,v) co-ordinate system to help map a texture onto its target object. These values

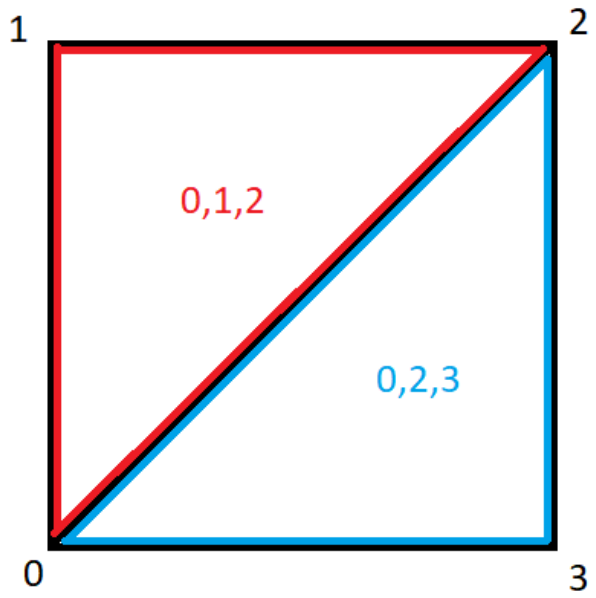

Figure 3: *Each triangle in a face can be described to the index buffer by the vertices that make it up*

are normalised to 0-1, so they are not affected by the size of the texture in pixels. When applying a texture to all sides of a cube, the vertex buffer must hold 24 vertices so that the normals on each face are facing outwards at every corner.

The $D3DX11CreateShaderResourceViewFromFile$ can be used to read in an image from an external file. A sampler state will need to be defined with all the variables on how the shader will render the texture. In this solution, the ground is textured with a grass texture, with an added normal map (Or bump map) layered over the top to give the appearance of depth without needing to complicate the model of the ground to achieve the same effect, which would increase the number of triangles in the scene and reduce performance.

4.4 Lighting

There are three different kinds of light sources: Point lights, spotlights, and directional lights. Point lights act like lightbulbs in the sense that they send light in all directions from their source but have a max range of attenuation. Spotlights work like flashlights in the sense that they emit in a cone from the source. The light that was implemented in this solution was the directional light, which have no source or falloff, just direction and colour. The vertex structure must be updated to hold normal data, which can be used to define the intensity of a light striking the surface.

4.5 Models and Animation

In this solution, there are two different implementations of importing models from external files. To create the player character, the MD5 format was chosen to implement a model that had the functionality for a skeleton structure to define the vertex positions (Iedoc, 2020).
The prototype for loading a MD5 model in and storing it is:

```
1 bool LoadMD5Model(std::wstring
    filename,
2    Model3D& MD5Model,
3    std::vector<
    ID3D11ShaderResourceView*>&
    shaderResourceViewArray,
4    std::vector<std::wstring>
    texFileNameArray);
```

Once the model is loaded, an MD5ANIM animation can be applied to it. This format is sectioned into 5 different parts. The header stores information about the file and animation, such as frames per second or number of joints. The hierarchy is a list of joints used in the model, which should match with the .md5mesh of the parent model if the animation is to work. Another part handles the bounding box information for each frame of the mesh, for collision detection or picking. The baseframe stores the position and oreintation of each joint in its default position, and the final part is the frames information detailing exactly how the joints are moved and rotated.

Figure 4: *The final render of the implementation, with the instanced trees, and animated MD5 model*

The solution also contains static 3D models of trees and leaves which are loaded into the scene and instanced randomly across the ground. There are 400 trees being created and 1000 leaves per tree, without instancing this would demand 400,000 draw calls per frame which would be far too overwhelming for a system to handle. By saving the geometry on the GPU, it no longer has to call multiple times to render the same

object again, and drastically increases the efficiency of the program. Instancing is discussed by Andersson, 2011, at the Battlefield 3 GDC conference, stating that the reduced draw calls are hugely beneficial for the CPU.

5 Conclusion

Instancing and bounding volumes go a long way to reducing the computational demands of this program, but the task manager report shows that 100% of the GPU is being used in order to run it. This is due in part to having an uncapped frame rate, which is allowing the hardware to push much harder than it needs to, especially when the character is idle.

The final implementation did not resemble a game as such, but provides a solid framework for one to be constructed. Whilst a 3rd person camera, textures, models, animations, and lighting are all implemented, the next step in this prototypes development would be to implement a physics system that allows for collisions and object interactions to facilitate the creation of puzzles and gameplay.

Bibliography

Andersson, Johan (2011). "D3D11 Instancing". In: *Game Developers Conference. Direct X 11 Rendering in Battlefield 3*.

Carter, Chad (2007). *Microsoft® xna™ unleashed: graphics and game programming for xbox 360 and windows*. Sams.

Iedoc (2020). *Directx11 tutorials*. URL: `https://www.braynzarsoft.net/viewtutorial/q16390-braynzar-soft-directx-11-tutorials`.

Luna, Frank (2012). *Introduction to 3D game programming with DirectX 11*. Stylus Publishing, LLC.

Zioma, Renaldas and Simon Green (2012). "Mastering DirectX 11 with Unity". In: *Game Developers Conference*. Citeseer.

Microsoft (2020). Graphics Pipeline. url: https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphicspipeline.

Rastertek (2010). Buffers, Shaders, and HLSL. url: http:// www.rastertek.com/dx11tut04.html