Advanced Technologies Task 1: DirectX 11

Alex Feetham
17016942)

University of the West of England

May 11, 2021

This report outlines the principles behind the DirectX 11 API and how it can be used to create a game.

1 Introduction

The first version of DirectX was released in September 1995 as the Windows Games SDK. The technology inspired the release of the original Xbox console in 2001, getting its name from the X in DirectX (Edge Staff, 2013), and has been included within the Windows Operating System since Windows 98. DirectX is an amalgamation of several its application programming interfaces (APIs) developed by Microsoft with the X used a shorthand term for each of the included APIs which are:

- Direct3D
- DirectDraw
- DirectMusic
- DirectPlay
- DirectSound

Direct3D (D3D) 11 is the rendering library within DirectX 11 which is used for writing high performance 3-dimensional graphics applications using modern graphics hardware on the Windows operating system. Direct3D is a low-level library in the sense that its API closely models the underlying graphics hardware it controls (Luna, 2012). This report focuses on the Direct3D API and where the term DirectX is used henceforth, it is in relation to the Direct3D API.

2 Related Work

Countless games have been created using DirectX 11; (PC Gaming Wiki, 2021) estimates that over 2350 games utilise DirectX 11. An extremely notable user of the DirectX 11 is the Call of Duty franchise. Infinity Ward use the DirectX 11 API in their game engine (IW Engine) and titles that use this include:

- Call of Duty: Ghosts (2013)
- Call of Duty: Infinite Warfare (2016)
- Call of Duty: Modern Warfare Remastered (2016)
- Call of Duty: Modern Warfare 2 Campaign Remastered (2020)

3 Method

3.1 Direct3D Devices and Swap Chain

A Direct3D device allocates and destroys objects, renders primitives, and communicates with a graphics driver and the hardware. In Direct3D 11, a device is separated into a device object for creating resources and a device-context object, which performs rendering (White et al., 2018)).

A swap chain is a collection of buffers that are used for displaying frames. Each time an application presents a new frame for display, the first buffer in the swap chain takes the place of the displayed buffer (White et al., 2018). The front buffer is what sends the image/frame from the graphics card to the monitor however, the front buffer is often updated faster than the monitor can refresh its image. This leads to a problem called screen tearing. Screen tearing occurs when an application updates the front buffer while the monitor is in the process of refreshing. This results in the

Figure 1: *Example of Screen Tearing*

upper half of the display showing the old frame and the lower half showing the new frame. Figure 1 illustrates this effect and it can be clearly seen that the canoe and paddle are misaligned. The swap chain solves this problem by swapping the pointers to the back buffer and front buffer. The back buffer is used to prepare the next frame while the front buffer is displaying the current frame. Direct3D can be synced to present the front buffer as the monitor is refreshed to avoid screen tearing. A D3D device and a swap chain can be created at the same time they work in unison via the function call `D3D11CreateDeviceAndSwapChain`.

3.2 Graphics Pipeline

Figure 2 shows illustrates the different stages of rendering pipeline for DirectX 11. Graphics hardware that supports Direct3D 11 can be designed with shared shader cores. The GPU can program these shader cores and schedule them across the functional blocks that make up the rendering pipeline (Hollasch, Coulter, and Bazan, 2017).

The input assembler uses fixed functions to read vertices out of memory to form geometry primitives and create pipeline work items. Next, the vertex shader processes the vertices from the input assembler and performs operations such as transformations, skinning, and lighting. From here, information is passed to the geometry shader. While the vertex shader only operates on a single vertex at once, the geometry shader works on the vertices of a full primitive plus the vertex data for the edge-adjacent primitives (Figure 3). The stream output is linked to the geometry shader and must be created together (although it can be turned off). The stream output acts as a tap in the pipeline to concatenate primitives from the geometry shader to the output buffers which can be recirculated on subsequent passes as pipeline inputs. This is useful for pipeline data that will be reused. The rasterizer converts vector information into a raster image to display real-time 3D graphics. During rasterization, each primitive is converted into pixels while interpolating per-vertex values across each primitive. Rasterization includes clipping vertices to the view frustum, performing a

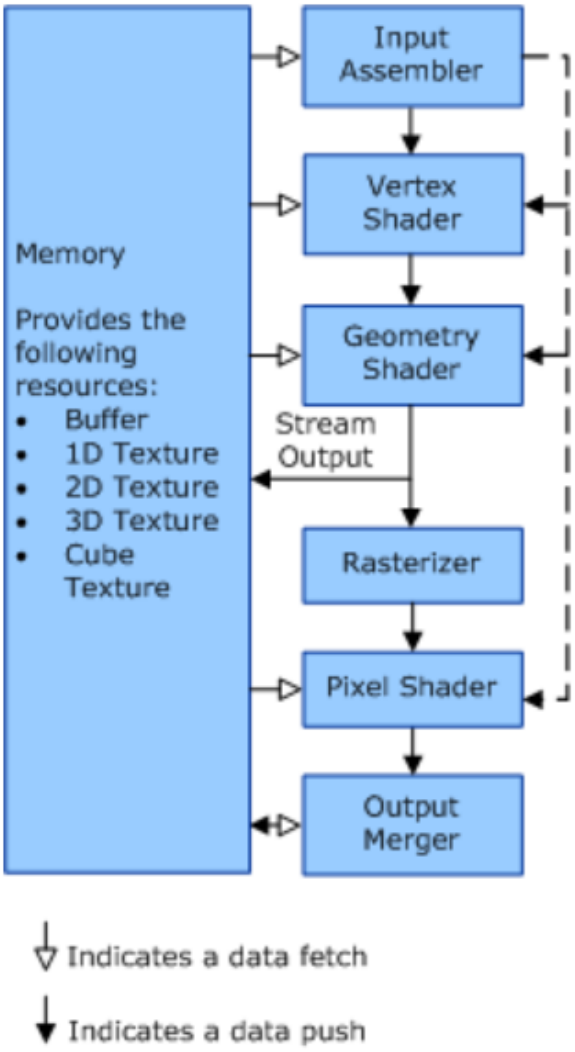

Figure 2: *Direct3D Graphics Pipeline*

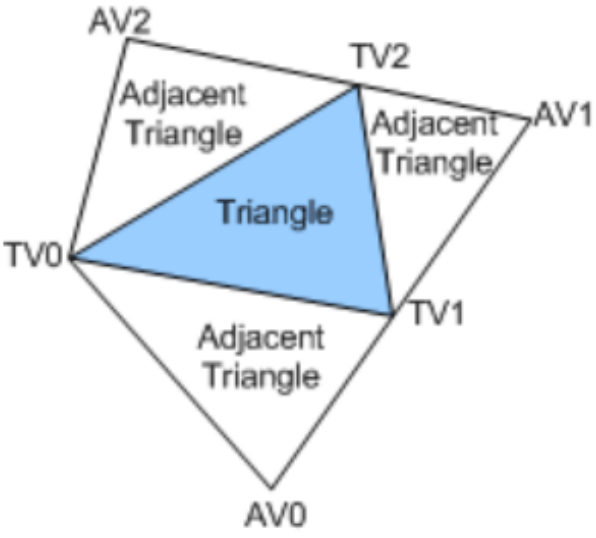

Figure 3: *Direct3D Graphics Pipeline*

divide by z to provide perspective, mapping primitives to a 2D viewport, and determining how to invoke the pixel shader. The pixel shader enables rich shading techniques like per-pixel lighting and post-processing. The pixel shader combines constant variables, texture data, interpolated per-vertex values and other data to produce per-pixel outputs. The output-merger (OM) stage generates the final rendered pixel colour using a combination of pipeline state, the pixel data generated by the pixel shaders, the contents of the render targets, and the contents of the depth/stencil buffers. The OM stage is the final step for determining which pixels are visible (with depth-stencil testing) and blending the final pixel colours. (White et al., 2018) and (Hollasch, Coulter, and Bazan, 2017)

3.3 Vertex, Index, and Constant Buffers

A buffer resource is a collection of fully typed data which is grouped into elements. Buffers can be used to store a wide variety of data such as:

- position vectors
- normal vectors
- texture coordinates
- indexes
- device states

A vertex buffer contains the vertex data used to define geometry. Vertex data includes position coordinates, colour data, texture coordinates, and normal data. Index buffers contain integer offsets into vertex buffers and are used to render primitives more efficiently. Index buffers contain a sequential set of 16-bit or 32-bit indices. Each index is used to identify a vertex in a vertex buffer. Constant buffers efficiently supply shader constants data to the pipeline. They can be used to store the results of the stream output (White et al., 2018).

4 Method and Evaluation

To implement DirectX 11 to create a game scene I used Visual Studio 2019. As I had no prior knowledge about DirectX, I followed an online tutorial by (ChiliTomatoNoodle, 2018). The tutorial covered all of the fundamentals of D3D 11 and was very informative in the breakdown of all aspects of D3D 11. Using the tutorial series, I was able to implement and understand much of the core framework (device and swap chain, the graphics pipeline, buffers, and shaders) of DirectX 11. Additionally, I implemented texturing, a camera, and a point light. Texturing was implemented by inheriting the base properties from a class which created a cube. This new class created a variant of a cube which has a wooden crate texture applied and can be seen in Figure 4. This was done by creating a new surface texture for the cube:

```
1 AddStaticBind(std::make_unique<
    Texture>(gfx, Surface::FromFile("
    Images\\box.png")));
```

The external library ImGUI was implemented to create a camera and a point light, the control panels for which can be seen in Figure 4. ImGUI is an open-source C++ graphical user interface (GUI) library and is designed to simplify the creation of GUIs for low-level programs such as the one created in this project.

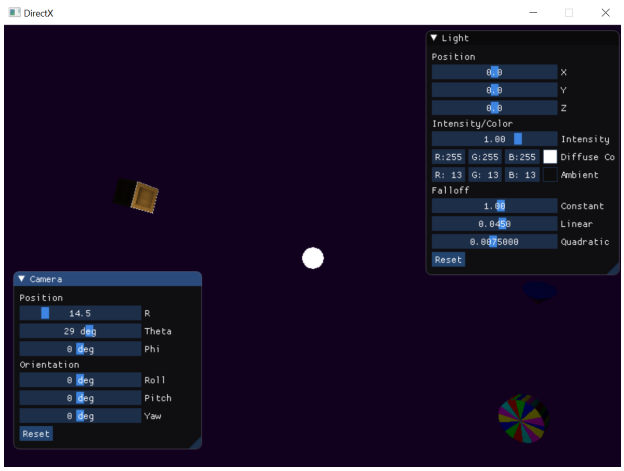

Figure 4: *Final Prototype*

5 Conclusion and Limitations

While the prototype produced did not meet the requirement of the task to create a first-person shooter game using DirectX 11, I was able to implement a great deal of the fundamental elements of DirectX 11 and learnt a great deal in the process. Progress of this task was severely impacted by the Covid-19 pandemic; a week into the main development time for this task I contracted the virus. This left me unable to complete any work on the project for several weeks while I recovered from it which caused progress to majorly fall behind. I worked on the project when I could whilst working on the other tasks however this is as far as I managed to progress this task. Additionally, Appendix A only contains one link to the first and only progress log because of my contraction of Coivd-19.

Bibliography

ChiliTomatoNoodle (2018). *Hardware 3D (C++ DirectX Graphics)*. URL: https://tinyurl.com/WIKIplanetchili.

Edge Staff, (2013). *11-X, WEP, Midway, CyberPlayGround, FACE – the rejected names for Microsoft's first console*. Edge Magazine. Accessed: 09/02/2021. URL: https://web.archive.org/web/20130921053639/http://www.edge-online.com/features/11-x-cyberplayground-ehq-

the-rejected-names-for-microsofts-first-console/.

Hollasch, L.W., D. Coulter, and N. Bazan (2017). *Rendering Pipeline*. Windows Developer Documentation. Accessed: 09/02/2021. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/display/rendering-pipeline.

Luna, Frank (2012). *Introduction to 3D Game Programming with DirectX 11*. Accessed: 09/02/2021. Stylus Publishing LLC.

PC Gaming Wiki, (2021). *List of DirectX 11 Games*. Accessed: 09/02/2021. URL: https://www.pcgamingwiki.com/wiki/List_of_DirectX_11_games.

White, S. et al. (2018). *Direct3D 11 Graphics*. Windows Developer Documentation. Accessed: 09/02/2021. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11.

Appendix A

Video Link - Week 1