# Advanced Tech - DirectX 11 First Person Shooter

**Kieran Bowsher**
**19036291**

*University of the West of England*

April 9, 2022

**The task that was given was to create a game using DirectX 11 and C++. This project uses the rendering pipeline from DirectX 11 to help render and texture objects. A structure of classes and components was used to aid with development of game-play and rendering features for the game.**

## 1 Introduction

There were three themes that consisted of a first-person shooter, a 3D infinite sprint game and a marble rolling game. For this project and the report, the first person shooter was chosen. During the task development, 5 developer logs were produced and uploaded to YouTube to showcase progress made on the game.

## 2 Background and Research

In the games industry, there are three main graphic Application Programmable Interface (API) used: DirectX, OpenGL and Vulkun. This project uses DirectX 11 which is developed by Microsoft. Just like any other graphics API, DirectX uses a pipeline to process information like vertices and indices and produce images that are rendered to the screen.

### 2.1 The Pipeline

As you can see from **Figure 1**, this is the pipeline DirectX 11 uses for rendering (Microsoft, 2022a). Unlike older versions of pipeline, DirectX 11's pipeline is more flexible with optional stages like the Hull, Tessellation and Domain shader stages. The rest of the section will break down the main stages of the pipeline.

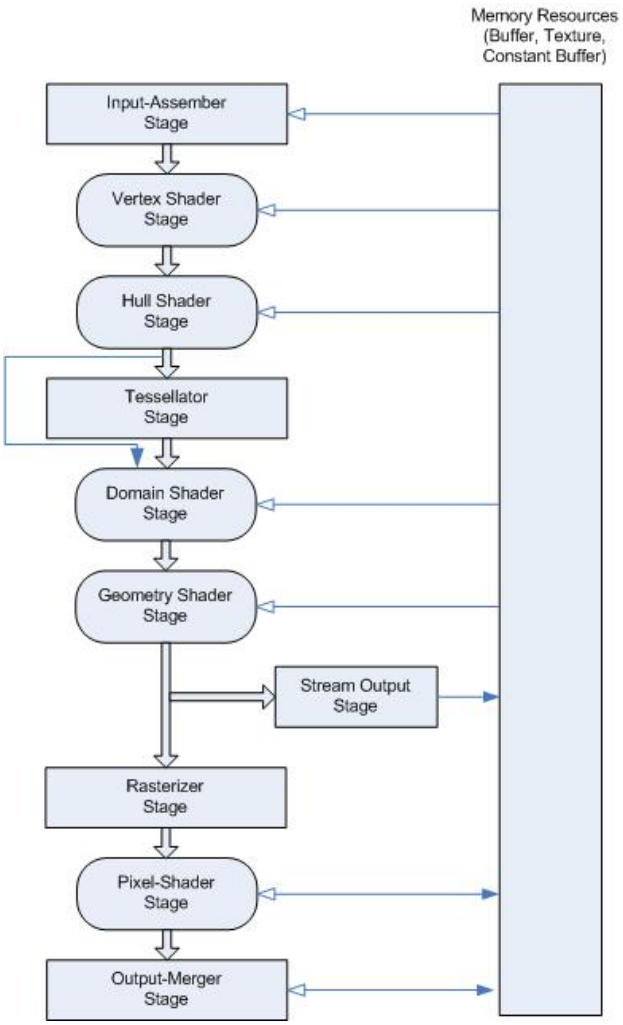


**Figure 1:** *Diagram showcasing the pipeline for DirectX 11*


**Figure 2:** *Example of screen tearing in Half Life 2*

## 2.2 Swap-Chain

An important part of rendering video games is the use of a swap-chain. The idea is to have two buffers: front and back which work together to render and display. They work by having the front buffer display the final render to a monitor/TV and a back buffer which acts as a render target for the game.

When the front buffer has displayed the final render, it gets pushed back to become the back buffer while the back buffer gets pushed to the front to display the final render. Since this happens N number of times per frame, both buffers can only display a certain amount of the image. This produces a visual artifact called screen tearing as shown by **Figure 2** which can be fixed using Vsync, Gsync or FreeSync (Nvidia, 2021).

## 2.3 Input Assembler stage

The Input Assembler stage or the IA stage works by taking primitive information like points and triangle and assemble them into shape primitives for the rest of the pipeline. User defined semantic like "POSITION", "TEXCOORD" and "NORMALS" are used to allow the vertex shader to gain access to the data. In order to get data like points and triangles, buffers like the vertex and index are used (Microsoft, 2020a).

### 2.3.1 Vertex Buffer

The vertex buffer is a buffer that contains all the data about an objects vertices. Each vertex will hold information like position, normal, colour, UV coordinate etc.

### 2.3.2 Index Buffer

The index buffer is similar to the vertex buffer but it only contains an index about which duplicated vertices should be merged when rendering. This will only work if DrawIndex() is called.

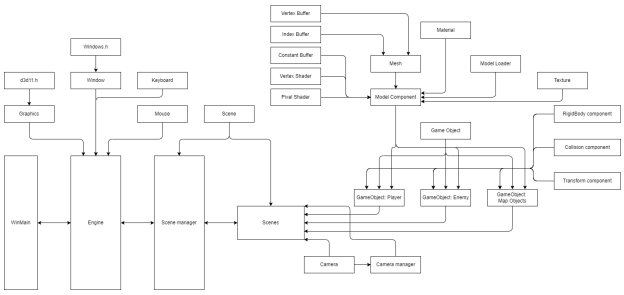


**Figure 3:** *Basic UML diagram of the systems used for the game*

## 2.4 Vertex Shader stage

This is the first programmable shader stage. The vertex shader takes information from the IA stage using user defined semantics and performs per-vertex operations like transformations and pre-vertex lighting. This stage is integral to the rendering pipeline as without a vertex shader, geometry cannot be processed and rendered (Microsoft, 2022b).

## 2.5 Geometry Shader stage

The Geometry shader is similar to a vertex shader with the only difference being their inputs. Unlike how the vertex shader operates on a single vertex, the geometry shader can operate on a full list of vertices that are connected with lines or triangles (Microsoft, 2021a).

## 2.6 Rasterizer Stage

This stage uses a rasterizer operation which takes all the vector information produced by the other stages and creates array of pixels used for displaying onto the monitor/TV (Microsoft, 2020c).

## 2.7 Pixel Shader stage

Similar to the vertex shader. The pixel shader processes any pixels from the Rasterizer stage which cover primitives or render-targets. This stages allows for per-pixel lighting, texture manipulation, post-processing and interpolated per-vertex data from the vertex shader (Microsoft, 2020b).

## 2.8 Output merger stage

This is the final stage of the pipeline. The output merger generates the final render using all the states from the pipeline in combination with a pixel shader, depth buffers and render targets. This stage is also responsible for rendering only visible pixels using depth testing Microsoft, 2021b.

# 3 Method

This section of the report will discuss and explain some the elements and methods used to build the game using DirectX 11.
Here you can see from **Figure 3** the high level view of all the systems in the game. ‘

## 3.1 Engine class

The engine class handles all the rendering, inputs, window messages and updates for the game. Each class like Graphics, Keyboard, Mouse, Window and Scene manager are included using composition and only one instance of each class exists in the engine. The main functions like Draw(), Update() and Input() are called and executed inside a while loop in win-Main.cpp.

## 3.2 Graphics class

The graphics class is responsible for setting up the swap-chain, render pipeline and presenting the final render. When the instance of the class is create, it creates the whole pipeline which includes elements like the swap-chain, depth buffer, rasterizer state, view-port and Input element description.
The class is abstracted to allow objects to handle their own creation since each object will control when the buffers like the vertex and index are created and bound to the pipeline. This also give more freedom to each object in how it want to create itself and what resources and shaders it will use.

### 3.2.1 Physically Based Rendering

**Figure 4:** *Example of a mix roughness and metallic model*

Physically Based Rendering is an approximation on how light reacts to different material properties like dialectic and conductive materials. It uses the micro-facet model which can be describe as small little reflective mirrors which are aligned to a surface (OpenGl, 2017a) and control the specular/reflection lighting.

PBR uses a simplified version of the rendering equation called the reflectance equation which also uses a function called The Bidirectional Reflective Distribution Function (BRDF) (OpenGl, 2017a). The reflectance equation can be expressed as:

$$L_o(p,\omega_o) = \int_\Omega f_r(p,\omega_i,w_o)L_i(p,\omega_i)n\cdot\omega_i d\omega_i \quad (1)$$

The BRDF that this project uses is a common one used a lot in real-time application called the Cook-Torrance BRDF. This can be shown as:

$$f_r = \frac{FDG}{4\cdot(N\cdot L)(N\cdot V)} \quad (2)$$

where $FDG$ are the micro-facet functions. This project mainly uses micro-facet function from (OpenGl, 2017a) and (Karis, 2013) which are: The distribution function $D$ uses Disney’s GGX/Trowbridge-Reitz. This can be described as:

$$D(h) = \frac{\alpha^2}{\pi((n\cdot h)^2(\alpha^2-1)+1)^2} \quad (3)$$

where $\alpha$ is the value of the surface roughness.
The geometry function $G$ uses GGX and Schlick-GGX. These can be expressed as:

$$G(n,v,\alpha) = \frac{n\cdot v}{(n\cdot v)(1-k)+k} \quad (4)$$

$$k = \frac{(\alpha+1)^2}{8} \quad (5)$$

where $k$ is the remapped value of $\alpha$ and $(\alpha+1)$ is a modification done by (Burely, 2012) to reduce roughness "hotness". Smith’s geometry function is made up of two two Schlick-GGX function and this is shown as.

$$G_{Smith}(n,v,l,a) = G(n,v,a)G(n,l,a) \quad (6)$$

Finally the Fresnel function $F$ uses FresnelSchlick and is shown as:

$$F(\theta) = F_0 + (1-F_0)(1-N\cdot V)^5 \quad (7)$$

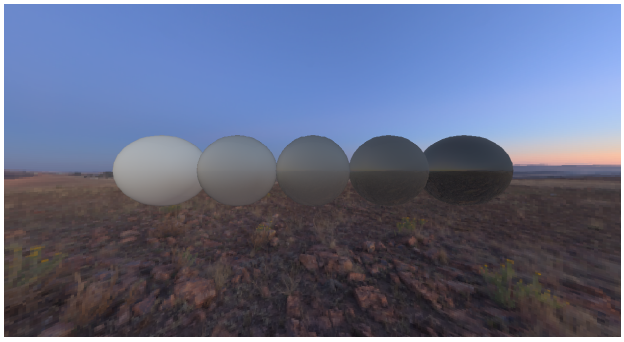

**Figure 5:** *Gradient of different metallic values ranging from 0 to 1*

For image Based Lighting (IBL), the implementation follows (OpenGl, 2017b) and its effects can be shown in **figure 5**. Currently, the project uses two PBR shaders with one using textures and another using a material struct.

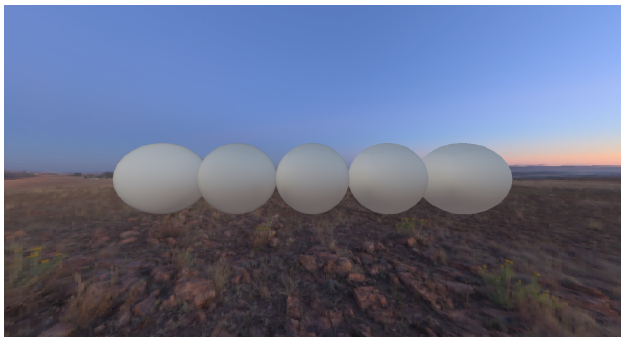

**Figure 6:** *Gradient of different roughness values ranging from 0 to 1*

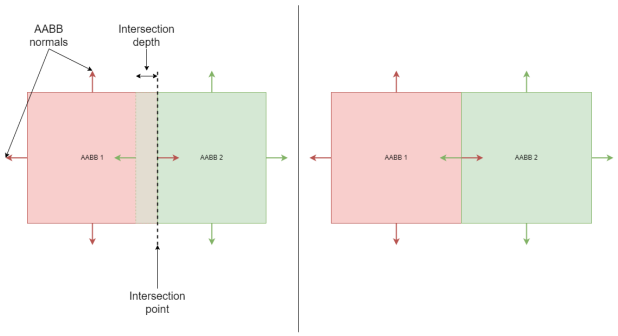


**Figure 7:** *Diagram showing how collision is resolved using the intersection depth and normal from an AABB. Left: AABB 2 intersecting with AABB 1. Right: AABB 2 moved using $p' = \vec{p} + (\vec{n} \times \alpha)$ to re-position AABB 2*

## 3.3 Scene Manager

The scene manager is responsible for each scene behaviour like creation, destroying, activating, deactivating, updating, drawing, input handling and scene switching. Scene are stored inside the scene managers unordered map with an ID before the game is loaded. To swap a scene or delete a scene, the scene manager has functions that can do these operation and these functions need a scene ID.
All the scene are handled using a Finite State Machine (FSM) and this means that only one scene can be active at any time.

## 3.4 Camera Manager

The camera manager is stored and used inside each scene. The managers job is to just render the current camera and pass its view and projection matrix to the graphics class. By passing the view and projection matrix to the graphic class, this allows all objects rendered to gain access to these matrices for rendering.
Similar to the scene manager, the camera manager uses a finite state machine (FSM) to render an active cameras in the scene. However, cameras inside a scene can freely update themselves and move around. This means that an object like a player which owns a camera can freely move around and update independently.

## 3.5 Composition based components

The objects in the game use a design method called Component based design. Unlike Unity or Unreal Engine 4 which use Entity Component Based design, this project uses composition-based components for simplicity and quick development. These components can define the behaviour for game objects like player or enemy. Similar to Unity, the project uses components like transform, collision, mesh, rigid body to define each object.

## 3.6 Collisions

To allow game objects to have collision, a collision component is used. This component takes the objects position and scale to produce a bounding box around the object. This can be shown as:

$$min_{xyz} = \vec{p} - \vec{s}$$
$$max_{xyz} = \vec{p} + \vec{s}$$

where $\vec{p}$ is the position of the object with $\vec{s}$ as the object scale. Doing this will produce an Axis Aligned Bounding Boxes (AABB).

For collision detection, a collision handler is used to check if two bounding boxes are colliding. This can be expressed as:

$$Distance = max_{xyz} - min_{xyz}$$
$$Collision = \sum_{i=faces}^{n} distance \leq depthcollision$$

Essentially, to check collision against two AABB, for each frame, a loop is checked over the faces of an AABB to see if any of the faces have had collision. A collision is detected when $distance \leq depthcollision$. When this happens, an index of the face, the index of the faces normal and the depth of the intersection is returned. These values are then used to resolve collision against the two AABB.

To resolve the collision and prevent the objects from intersecting each other, this equation is used.

$$p' = \vec{p} + (\vec{n} \times \alpha) \tag{8}$$

where $p'$ is the new vector produced, $p$ is the current vector with $n$ as the normal of the AABB face that has been collided with and $\alpha$ being the intersection depth. Using this equation will re-position the object to a new point that is away from the object that it intersect with which can be shown in **figure 7**.

```
0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0 0:0
0:0 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2
0:0 2:2 3:0 2:2 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 2:2
0:0 2:2 3:0 2:2 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 2:2
0:0 2:2 3:0 2:2 1:0 4:1 1:0 1:0 4:2 1:0 1:0 4:3 1:0 1:0 1:0 2:2
0:0 2:2 3:0 2:2 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 2:2
0:0 2:2 3:0 3:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 2:2
0:0 2:2 2:2 2:2 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 2:2
0:0 0:0 0:0 2:2 1:0 4:1 1:0 1:0 4:2 1:0 1:0 4:3 1:0 1:0 1:0 2:2
0:0 0:0 0:0 2:2 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 2:2
0:0 0:0 0:0 2:2 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 1:0 2:2
0:0 0:0 0:0 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2 2:2
```

**Figure 8:** *Example of a map file. 0:0 or 2:0 represent an object that the map can load.*

## 3.7 Map layout loading

The positions of all the objects and entities in the level, the type of objects used and the layout of the level is controlled by a text file which is parsed by the map class. An example of this text file can be seen in **figure 8**.

The map class acts as a container and parser where it will read the map file and generate the corresponding object based off a list of ID. This list of ID's is stored inside a $std :: map$ where the key of the map is the ID and the type that the map stores is an enumerator of an object type. An example can look like this $2 : 0, mapObjectID :: Wall1high$.

Since the map stores all the objects, the map also controls all the rendering and updating of each object.

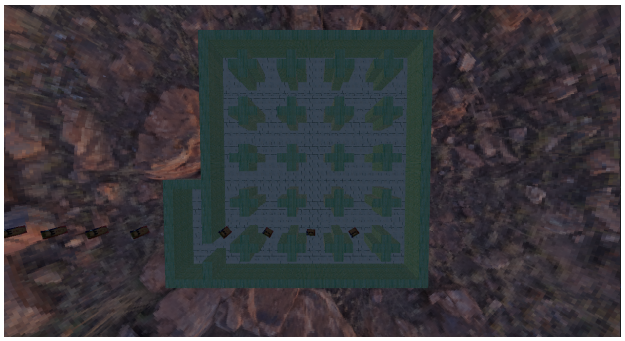

**Figure 9:** *A map from **figure 8 in game***

## 3.8 The game

### 3.8.1 Enemy look at player

Similar to games like "Doom", this game also has enemies that look at the player. In order for the enemies to look at the player, this equation is used.

$$\vec{p} = e\vec{p} - p\vec{p}$$
$$distance = \sqrt{\vec{p}_x \times \vec{p}_x + \vec{p}_z \times \vec{p}_z}$$
$$pitch = \arctan(\frac{\vec{p}_y}{distance})$$
$$yaw = \arctan(\frac{\vec{p}_x}{\vec{p}_z})$$

**Figure 10:** *An example of the enemies looking at the player.*

where $e\vec{p}$ is the position of the enemy and $p\vec{p}$ is the player position. Adding $yaw$ to the y component of the enemies rotation will allow the enemy to always look at the player. However, if the player moves up or down, the enemy will not look up or down with the players movements. Using $pitch$ on the x component for the enemy will allow them to look up or down. Since the game doesn't have vertical movement, $pitch$ is not used.

### 3.8.2 Shooting

To allow bullets to spawned from the player and move in the direction that the player is looking at, as explained by (IFooBar, 2004), the direction can be extracted from the third column of the camera's view matrix. As also explained by (Jeremiah, 2011), this can be shown as:

$$ViewMatrix \begin{bmatrix} right_x & up_x & forward_x & position_x \\ right_y & up_y & forward_y & position_y \\ right_z & up_z & forward_z & position_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (9)$$

Firing the bullet is essentially detecting when a key is pressed and adding velocity to that bullets position using the view matrix direction and this is expressed as:

$$c\vec{a}m = viewMatrix \begin{bmatrix} \cdots & \cdots & forward_x & \cdots \\ \cdots & \cdots & forward_y & \cdots \\ \cdots & \cdots & forward_z & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$
$$b\vec{v} = b\vec{p} + (c\vec{a}m \times 0.05) \times \delta t$$

where $b\vec{v}$ is the bullet's new velocity, $b\vec{p}$ is the bullet position, $c\vec{a}m$ is the camera's view matrix direction and $\delta t$ is delta time. In the equation, $(c\vec{a}m \times 0.05)$ is used to just to lower the velocity of the bullets since just using the $c\vec{a}m$ will produce very fast moving bullets which are hard to see and track during game-play. This can also cause issues for collision since very fast moving bullets may not collide with other objects due to the distance travelled per frame being larger than any objects that it could collide with.

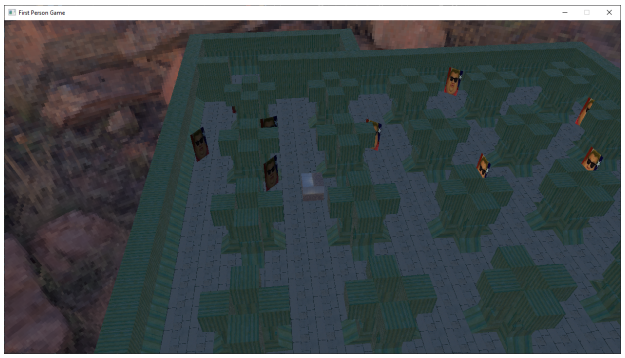

**Figure 11:** *An example of drone mode in the game*

The equation above can be written in a more generic style as expressed by:

$$p\vec{v} = \vec{p} + \vec{v} * \delta t \tag{10}$$

This is mainly used by the ridged body component that any object can use if it's moving like the player or bullets.

### 3.8.3 Drone mode

One feature that the game includes is a "drone mode" similar to Assassin Creed Origins "Eagle vision" where the player can control an eagle to have a bird's eye view of an area. However, in this game the player has no control over the bird's eye view.

In the game there are two cameras, a first person camera and a drone camera managed using the camera manager. The first person camera is controlled by the player and the drone camera is controlled by the player's position and an orbit. To allow the drone camera to orbit the player, this equation is used:

$$d = 30 \tag{11}$$

$$\delta = 10 \times \delta t \tag{12}$$

$$\cos\theta = \delta \times (\frac{\pi}{180}) \tag{13}$$

$$\vec{dp}_x = \vec{p}_x + d \times \cos(\cos\theta) \tag{14}$$

$$\vec{dp}_z = \vec{p}_z + d \times \sin(\cos\theta) \tag{15}$$

where $d$ is an arbitrary distance between the drone and the player. $\delta$ is the degrees rotated with $\cos\theta$ as the angle in a 360 degree circle. $\vec{dp}$ is the drone position with $p$ as the players position. To allow the camera to always look at the player, the look at equation used by the enemies is used again and this can be seen in **figure 11**.

# 4 Evaluation

This section will be evaluating some of the elements of the game like map loading duration, scene memory management and the number of objects spawned effects on rendering times and loading times.

## 4.1 Loading times

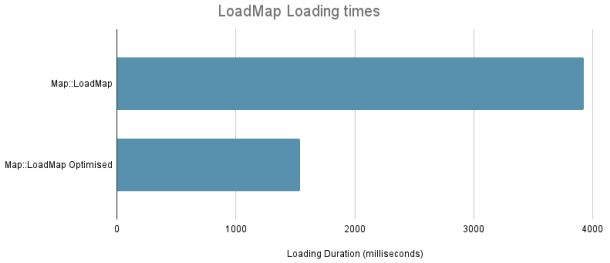


**Figure 12:** *Bar chart comparing two loading time duration for loading the map*

The loading time for the game can vary in duration depending on the size of the map and the number of textures and meshes used. Since the scene manager creates all scenes at once before the game starts, these loading times can quickly increase in duration if more larger maps and levels are used.

For testing, a sample map is used and measured using Visual Studios CPU profiler. The sample map took 3927 milliseconds to load all the objects and textures as shown in **Figure 12**. By changing the code so that the meshes and textures are loaded once and copied every time its being used, the duration of the map loading is reduce to 1540 milliseconds. This is an improvement but for only loading a single map, if more maps were loaded, then this loading duration will increase. Improvements to mesh and texture loading with the usage of mesh instancing (rastertek, n.d.) can improve map loading duration.

## 4.2 Scene memory management

The current implementation of the scene manager in the project does not include proper memory management. The scene manager can load scenes and switch between them. However, the memory of each loaded scene is persistence and never gets unloaded when its not used. If the project was to expand it's scope and include much larger levels with more models, then this current version would not support it well.

## 4.3 Collision performance

The current collision detection in the project doesn't cause any performance issues as shown in **figure 13**. If every object like the map's objects, the player, all the enemies and all the bullets will cause the function to run at 380 samples per 1000 samples/second. The current number of samples the project runs is around 4000 samples per 1000 samples/second which means that the collision detection takes $8\%$ of the total number of samples.

However, as shown by **figure 13**. This is mainly due to the collision between the bullets and every object

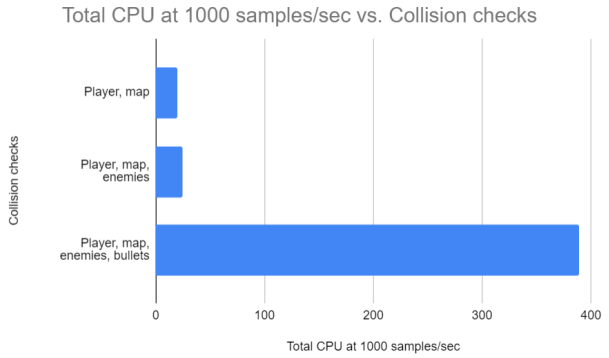


**Figure 13:** *Bar chart showing each type of collision and how many samples are used at 1000 samples/seconds*

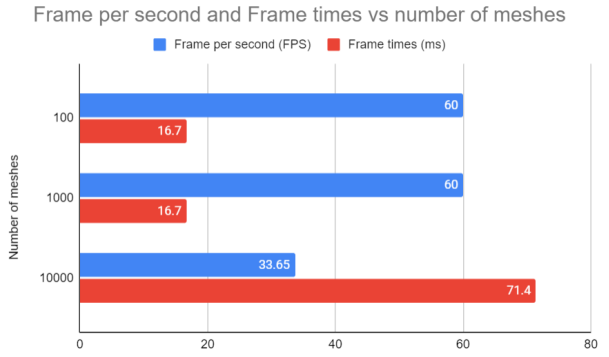


**Figure 14:** *Bar chart showing the number of meshes in the scene vs rendering times*

in the map. If the map had 100 objects and there are 30 bullets, then there will be 3000 collision checks per frame. This will increase exponentially if the map has more objects.

A solution is to lowering the number of collision checks used by using an accelerator structure like a Bounding Volume Hierarchy or OctTree.

### 4.4 Rendering duration

The number of meshes in the scene has a big affect on rendering times. As shown by **figure 14**, having 10000 meshes in the scene will cause the frames per second to drop by $43.91\%$ and increase frame times to $327.54\%$. The main reason for this is mainly due to the fact that each time a mesh is rendered, a draw-call has to be made which means that each frame, 10000 draw-calls are called. One solution would be to use instancing as explained by (rastertek, n.d.), all of these objects will share the same vertex and index buffer and the only difference between them would be their position, rotation and scale and doing this will improve rendering times. Another technique would be the usage of frustum culling where any object outside the camera will not rendered.

## 5 Conclusion

The final product from the project is a very basic first person shooter that somewhat emulates the older first person shooter. However, while the core loop of moving and shooting enemies is there, other features like a menu and win/lose state are current not there. Having an abstraction of classes and components allowed for easier development of game-play features and graphic features like multiple shaders and materials in a scene.

## Bibliography

Burely, Brent (2012). *Physically Based Shading at Disney*. URL: https://media.disneyanimation.com/uploads/production/publication_asset/48/asset/s2012_pbs_disney_brdf_notes_v3.pdf. (accessed: 29.12.2021).

IFooBar (2004). *How to extract direction vector from View Matrix?* URL: https://www.gamedev.net/forums/topic/259231-how-to-extract-direction-vector-from-view-matrix/2578141/. (accessed: 04.04.2022).

Jeremiah (2011). *Understanding the View Matrix*. URL: https://www.3dgep.com/understanding-the-view-matrix/. (accessed: 04.04.2022).

Karis, Brian (2013). *Real Shading in Unreal Engine 4*. URL: https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf. (accessed: 29.12.2021).

Microsoft (2020a). *Input-Assembler Stage*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-input-assembler-stage. (accessed: 05.11.2021).

– (2020b). *Pixel Shader Stage*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/pixel-shader-stage. (accessed: 05.11.2021).

– (2020c). *Rasterizer Stage*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-rasterizer-stage. (accessed: 05.11.2021).

– (2021a). *Geometry Shader Stage*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/geometry-shader-stage. (accessed: 05.11.2021).

– (2021b). *Output-Merger Stage*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-output-merger-stage. (accessed: 05.11.2021).

– (2022a). *Graphics pipeline*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline. (accessed: 05.11.2021).

Microsoft (2022b). *Vertex-Shader Stage*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/vertex-shader-stage. (accessed: 05.11.2021).

Nvidia (2021). *Adaptive VSync*. URL: https://www.nvidia.com/en-gb/geforce/technologies/adaptive-vsync/technology/. (accessed: 11.11.2021).

OpenGl (2017a). *Theory of PBR*. URL: https://learnopengl.com/PBR/Theory. (accessed: 16.11.2021).

– (2017b). *Theory of PBR*. URL: https://learnopengl.com/PBR/IBL/Diffuse-irradiance. (accessed: 16.11.2021).

rastertek (n.d.). *Tutorial 37: Instancing*. URL: http://www.rastertek.com/dx11tut37.html. (accessed: 06.04.2022).