Graphics Report

Gabriel Faes
5711834
https://github.com/Gabrio05/DirectX12Game

University of Warwick

December 22, 2025

The project consists of a short semi-rhythm game where the objective is to avoid vases being thrown at a stationary player by leaning left or right and rotating.

1 Introduction

The objective of the project was to create a small fully-functional game using the DirectX 12 API directly. A number of graphical techniques were implemented and will now be presented.

2 Core Pipeline

The core of the graphical pipeline is handled in the Core file which sets up the required code to generate graphics with the GPU. This includes choosing the adapter (GPU on the computer), setting up the queues (although only the graphics queue is ultimately used in this project, a potential major improvement), setting up the swap chains (2 in this case; although more is possible, 2 was more than reasonable for a small project like this), setting up the render target view (to actually see the graphics), depth stencil view (to correctly render objects based on their distance from the camera), and shader resource view (for shaders to work properly and correctly read buffer information).

The Core also creates the Root Signature which will allow shaders to read from buffers with the correct data.

3 Models

The level is loaded from a text file which takes in a number of commands and then requests the code to load them all in.It resembles the following:

```
sphere,20,50,1000,Models/Textures/qwantani_moon_noon_puresky.jpg,Models/Textures/black-image-8192x4096-rectangle.png,

static,Models/acacia_003.gem,0.01,0.01,0.01,5,0,0,

staticRepeat,10,0,0,

staticRepeat,20,0,0,

instance,line,100,10,0,0,3,0,0,10,10,10,Models/Takeout_Food_01a.gem,Models/Textures/TX_Takeout_Food_01a_ALB.png,Models/Textures/TX_Takeout_Food_01a_NH.png,
```

each line representing a different way of loading, with information such as position, scale, and texture locations also represented. Once loaded in, models will be drawn every frame based on that stored information.

Models can contain textures with alpha values, any alpha value below a certain value (0.5 in these pixel shaders) is discarded so that a further object can be drawn in its place instead.

Meshes can be either CPU-instanced (by repeatedly calling the above "staticRepeat" function in the levelData) or GPU-instanced (by calling the "instance" command instead). The instances are either drawn in a straight line or a grid (the grid adds a randomness factor so that objects are initially placed randomly in that square, they are then stationary). This instancing requires a separate shader to read the values from the "VS_INPUT" object instead of the static buffer.

Additionally, the first GPU-instanced object (in this project) has vertices move in the wind with the following shader:

```
float4 worldPos = mul(input.Pos, input.World);
float phase = dot(worldPos, phaseVary);
float windFactor = sin(TIME * windSpeed + phase) + sin(TIME / 2 * windSpeed + phase * 2);
```

```
4 float3 windOffset = windDirection * windFactor *
      windStrength * input.Pos.y * input.Pos.y;
5
6 worldPos.xyz += windOffset;
```

where windDirection, Speed, Factor, and Strength are hard coded (but could be added in a static buffer to dynamically change).

4 Animation

There is one animated model (the T-Rex) which has its animation change based on when a vase is thrown (from idle to roar) and whether the level is finished (plays death once). Animation is handled by the Animation file which manages bone positions and sending those corresponding world matrices to the vertex shader, which will then handle the final stage of calculating the actual vertex position based on which bone that vertex is attached to.

5 Lighting

The game features lighting from a global source, which illuminates all models in the same way (including parts which would be obscured by other models or even the same model). Normal textures are read from, translated to world coordinates by TBN transformation, and are then dot multiplied with the global light vector to obtain the final lighting for that pixel (a flat 0.1 is added to all lighting to avoid pure black pixels from normals facing away from the light source).

```
float4 the_normal = normalMap.Sample(samplerLinear, input.TexCoords);
 if (the_normal.r != 0 || the_normal.g != 0 || the_normal.b != 0) {
  the_normal.rgb = sqrt(the_normal.rgb);
  the_normal.rgb = the_normal.rgb * 2.0 - 1.0;
  float3 N = normalize(input.Normal);
  float3 T = normalize(input.Tangent);
  float3 B = normalize(cross(N, T));
  float3x3 TBN = float3x3(T, B, N);
  the_normal.rgb = normalize(mul(the_normal.rgb, TBN));
 }

 float3 incoming_light = normalize(float3(-1.0, 1.0, -1.0));
 colour.rgb = (colour / 3.14159265).xyz * 10 * (max(dot(incoming_light, the_normal.rgb), 0
 colour.rgb = sqrt(colour.rgb);
```

Texture values are read as sRGB, and so both the normal and eventually the albedo, are gamma corrected (this is not full gamma correction, as that would require the exponent to be 1/2.2, but 1/2 is close enough). The If statement is there for the sky not to have any lighting (this was faster than making a new PSO for it).

Drop shadows were attempted by making a second draw call on all objects (except the sky and ground) and modifying their world matrices to align with the light and project onto the ground. However this resulted in awful flickering (most likely due to an incorrect shader) and time constraints did not allow for debugging.

6 Conclusion

Although this report is sadly rushed due to time constraints, the overall project was still very successful, with a much greater understanding and introduction on how modern games handle the graphical pipeline from beginning to end. The game is surprisingly smooth (having done CPU ray-tracing previously) and could (with major refactoring) serve as the base of a game engine for a bigger game.