Computer Science with Games Programming - Studio 2

Nathaniel Waterworth - U1763480

University of Huddersfield

04.04.2019
Assessed by ZHIJIE XU

Contents

Introduction

Initially, I set out to produce a game that is created using a render engine that incorporates an object oriented structure. This engine would exhibit well documented code whilst demonstrating a correct use of variables between classes, enforcing encapsulation. I decided upon prioritising efficiency and lock my focus on implementing the engines game features such as Mesh Picking, Box Picking, Sphere Collision, Box Collision and Matrix manipulation . My plan was to develop a dynamic office scene that showcases these engine features with the player treated as an office object that interacts with the scene.

By following the tutorials provided by Rastertek (Rastertek. 2016) I implemented various features of a render engine. Also by referring to microsoft's (Microsoft. 2018) documentation on DirectX I managed to convert aspects of the tutorials. Finally tutorials provided by Braynzarsoft provided an insight to certain Game Engine Features (Braynzarsoft 2015).

Initial Design

The plan of action was to develop a game with the following in mind:

- The scene was intended to be an office in order to delegate my focus to rendering features whilst displaying realism.
- Dynamic objects would occupy the office with mouse and keyboard inputs for scene navigation about the office as an office model such as a toy plane..
- Textures that map to a specific object offer detail to the scene.
- The player would fire projectiles at paper planes in order to gain points.

Controls and Direct Input

The executable game can be traversed using keyboard and mouse input; these inputs navigate the player model and camera through the scene. The DirectX API provides a rapid method of obtaining input device information, this Direct input outperforms the regular windows input system.

The following Inputs cause the relative actions:

- WASD - Relative movement forwards, backwards, left and right.
- 1 - Decrease of Height.
- 2 - Increase in Height.
- Arrow Keys - Rotation around X and Y axis.
- Mouse movement - Rotation around X and Y axis (with respect to mouse Delta X and Y position).

These controls were chosen as they function most intuitively, similar to other games movement designs.

The following Inputs set various camera views:

- F1 - FreeRoam (Detach Camera from player model)
- F2 - Shooting sphere colliders at paper-plane sphere colliders game.
- F3 - Mesh Picking.
- F4 - AABB Picking
- F5 - Automated Tracking of another scene object.
- F6 - Frustum Culling.

Scene Models

My models were obtained from Google Poly and TurboSquid. Google Poly offered many low poly models, whilst TurboSquid specialised in more detailed models. Despite my intentions of maintaining a realistic scene, Google Poly was my preference. It proved less difficult to convert their models into text file format as I didn't run into many problems there. However, many of turbo squids models couldn't be directly converted into text file format. Instead, I had to import these models into 3DS Studio Max and make changes.

For certain complex models added to my scene, I also had to convert tiff files to dds to ensure the texture would correctly map to the model.

When adding models to the scene I would clone initialised objects instead of reinitialising new models, this removed unnecessary code. Through the use of matrix transformations I rendered the same model in various locations.

Matrices

Matrices gave me control over the models added to the scene as they were placed using matrix transformations, rotations and scaling. By obtaining the world matrix for each model and multiplying it by a rotation, transformation or scaling matrix I could produce dynamic effects. For instance, the range of paper plane movements are achieved by correctly ordering the matrix functions and developing an offset of models through iteration. I used axes to determine how my model would orient itself and rotate rather than cos and sin functions that proved problematic.

Sound Input

I implemented Direct Sound to add sound to my game. This made the scene more immersive. In my scene I looped a ambient sound to simulate an office environment, played a laser shot sound when firing and a hit sound on player collisions.

The sound class uses a primary and secondary buffer. The primary buffer acts as the main sound memory buffer on the default sound card, the secondary buffer has sounds loaded into it.

As I limited myself to WAV files certain WAV sounds didn't play, so I ensured the

properties of the file were correct in Audacity and then replaced the file back in the data folder of my engine.

Picking

I started out by developing gameplay through picking. By transforming a 2D screen space ray to a 3D view space ray from the crosshair position in screen space (centre of screen). I obtained the crosshair position as it equates to the Z axis of the camera view. Furthermore, by taking both the picking ray and object to be picked into world space I can determine whether they have intersected. To increase the efficiency of this process, I first determined if the ray intersected through sphere colliders of models. If so, mesh colliders were then picked. This reduced many unnecessary checks against the faces that make up models. As each plane model is only 37 triangles and are 60 models in total, the picking checks are reduced from 2220 to around 100 based upon the scene setup. I then proceeded to create an AABB picking check to increase efficiency over mesh picking, though as my models aren't made up of many vertices it didn't have a significant impact on performance.

Collision Detection

I broke this down into parts. Starting with sphere collision between a projectile and paper plane. If the two overlapped score will increase and the plane no longer rendered. Then I moved onto AABB colliders, which I used for each of the models in the scene (except planes). I only completed checks between the player and other objects. Since many objects being checked weren't dynamic this would waste a lot of time checking collision against one another.

Frustum Culling

My final attempt at an increase in efficiency was to cull any objects that were outside the viewing frustum. By creating the viewing frustum from 6 planes. If any part of an objects AABB was within the frustum it would be rendered. This can be seen through pressing F6 as it currently culls the paper plane objects. With V-Sync disabled, I observed an extra 40fps when facing away from the paper planes.

2D Rendering

I embarked upon the rendering of text to the screen as it is a valuable function. It can inform the user of technical information. For instance, Statistics such as the CPU usage, FPS count, number of rendered objects after culling (When F6 is pressed) and score are displayed to screen. Each of these give an indication of the efficiency of the engine and the computer executing the program, especially useful when optimising performance.

I encountered a substantial problem when implementing text rendering into my engine I Came across an issue involving the window being rendered flickering black. I determined the nature of the problem by narrowing the issue down to the most recent changes. As the rendering of text is a 2D feature, the Z buffer has to be disabled temporarily in order for the 2D data to overwrite the pixel location. I had conveyed this in my code. Though I had set the 2D data to be rendered and presented to the screen [using m_D3D->EndScene();]. As well as the 3d data (models) to do the same. This presented the problem as the screen was attempting to present information twice every frame. This was confirmed when I removed one of these lines of code. However due to the position of my 2D rendering code being placed before the rendering of 3D objects, it no longer rendered despite the screen the screen not flickering, whereas before it had rendered.

I then came to the conclusion that as the 2D data wasn't rendered and presented to the screen as soon as it had overwritten the pixel location, the 3D data was then overwriting this data instead and then presented to the screen. Hence only showing 3D objects.

By repositioning the code blocks that caused 2D text to be rendered to after the 3D model rendering I resolved the black screen flicker and managed to complete the rendering of 2D and 3D data.

Rendering to Texture and Reflections

I wanted to further increase the reality of the scene. So I decided upon implementing reflections into my scene. Before I could create a reflection I needed to render my scene to a texture. A problem I faced was the volume of excess code I encountered when rendering to a texture. Though impressive, the render to texture function would be a costly effect that would have to be managed to ensure the engine ran efficiently.

Conclusion

To conclude, I would have liked to have developed a way to determine the best bounding shape for models to perform a more accurate collision test. Also, the use of spot lighting to highlight points in the scene where the player could collect items to gain points encouraging the player to navigate the scene. Furthermore when implementing the rendering to texture function I felt as though I could have given a togglable second perspective which produced a smaller window on the screen imitating security camera view. Despite running into technical issues that hindered my work, I feel I created a sufficient game.

References

Rastertek. (2016). *Rastertek*. Retrieved from http://www.rastertek.com/tutdx11.html.

Microsoft. (2018). *Windows Dev Center*. Retrieved from https://developer.microsoft.com/en-us/windows.

Braynzarsoft. (2015). *DirectX 11 Tutorials*. Retrieved from https://www.braynzarsoft.net/viewtutorial/q16390-braynzar-soft-directx-11-tutorials.