COMPUTER GAMES STUDIO 2 – SEMESTER 1 - REPORT

Luke Dicken - Jones – U1752044

DECEMBER 13, 2018

UNIVERSITY OF HUDDERSFIELD

Contents

Introduction

Initial idea

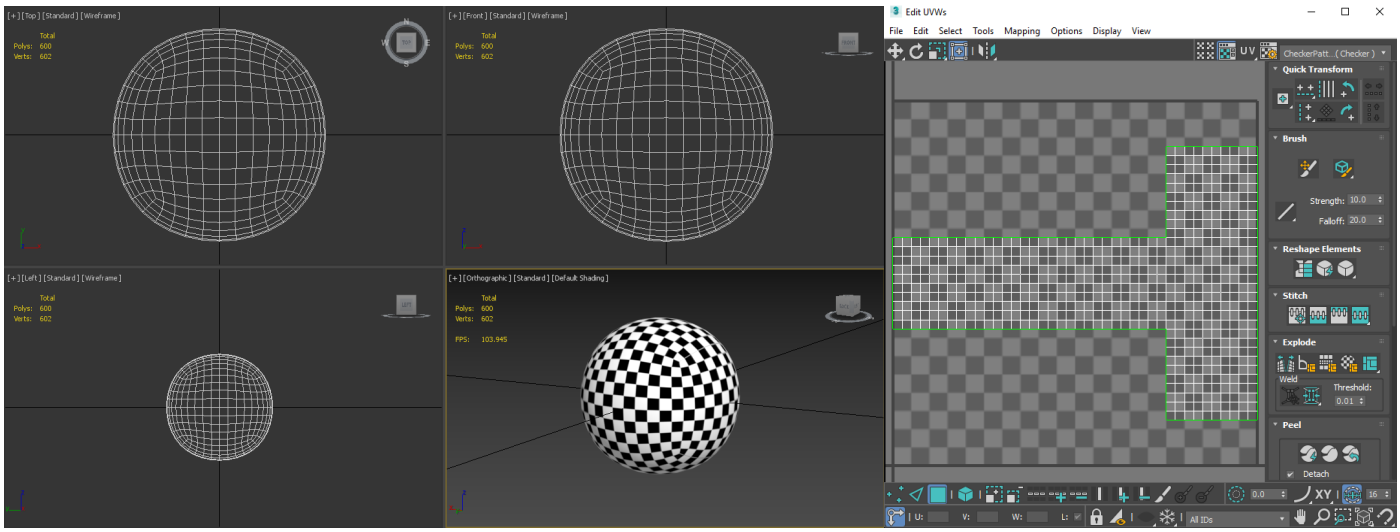
An example of a previous student's work (Andrew, 2015) was shared with the class before we had begun to assemble our scenes, which consisted of a terrain containing expanses of water. This inspired my scene, which is a sandy island. This would allow me to experiment with the rendering of various object types (e.g. a specular beach ball, transparent leaves on trees/foliage), with the size limitations of a small island providing a sufficient limit to the quantity of 3D models required.

Features

Skysphere/Quadsphere

I wanted the sky to be represented by a sphere, rather than a cube, to eliminate the possibility of visible seams. However, I discovered that most available sky textures were designed for cube models, and therefore couldn't be applied to a standard sphere model. When researching the possibility of mapping a cube texture to a sphere, I discovered a geometric design known as the 'quadsphere' (quadrilateralized spherical cube) (RubyDoc, n.d.). This is essentially a sphere that has been divided into six equal regions corresponding with the faces of a cube. Once UV unwrapped, I would be able to apply one of the many skybox textures to this quadsphere model, creating my skysphere.

I discovered that an unwrapped quadsphere model could be found in the free 'Procedural Examples' asset pack on the Unity Asset Store (Unity Technologies, 2016). I imported this into 3ds Max and modified the unwrap so that it was compatible with my chosen skybox texture.



In the graphics class, I have set its position to the camera's current position, so that it will stay centred around the camera. Its scale is set to large values so that it will encompass all objects in the scene. The values are negative so that the textured faces of the skysphere are visible from the inside.

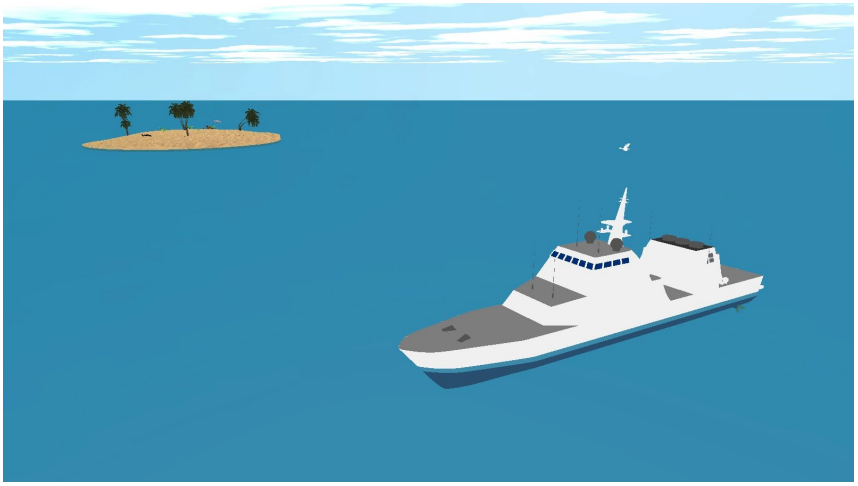
Object-oriented structure

My implementation of an object-oriented structure required the most development time out of all of this project's features. I deemed it necessary to dedicate a significant portion of development time to this, as it would increase the organisation and scalability of the project.

Objects

I aimed to implement a primitive version of something similar to Unity's 'GameObjects' (Unity Technologies, 2018). Objects are capable of holding enough information to allow them to be rendered with a parameter-less call to their 'Render' method. Variables use default values where possible, so when creating a new object only variables relevant to that object require modification. For example, their default translation value is set to the scene's centre, meaning that if a translation value isn't specified, then the object's location will default to the centre of the scene.

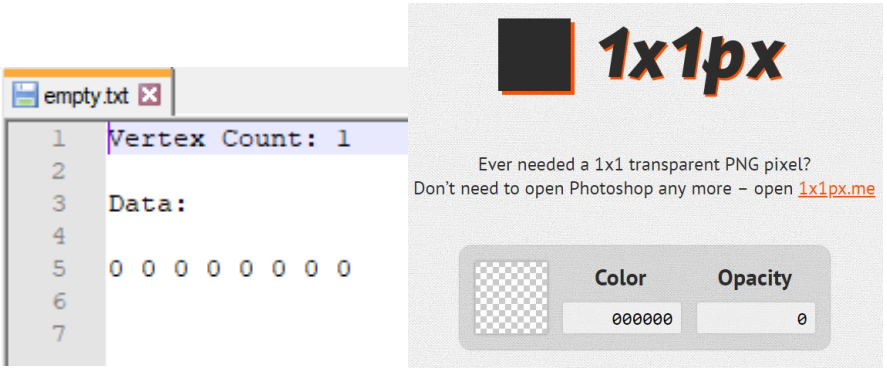
A key feature of my implementation of objects is their hierarchical structure. An object can have a parent, which itself can have a parent, and so on. The parental hierarchy of an object influences its position and rotation in the scene. For example, in my scene I have a seagull, which is the child of a pivot point, which is the child of a boat, which is the child of another pivot point, which is a child of the island. This means that if the island is re-positioned or rotated, all of these children will be re-positioned/rotated relative to the island. A change to the position/rotation of the boat would affect its children (e.g. the seagull), but not any of the objects in its parental hierarchy (e.g. the island).



The most crucial part of this feature is found in the 'SetWorldTransform' method, which first sets the scale, rotation, and translation of the object, and then iterates up the parental hierarchy, applying the rotation and translation of each parent.

Another important feature of this implementation is support for multiple object types. It currently supports standard models, normal-mapped models, and specular models, with it being straightforward to add new model types to this list. An enum is used to hold the names of the different model types. When the class constructor is called, the model type is given in its parameters (defaults to the standard model type if parameters are left empty), which then influences how the rest of the methods in the class behave. For example, normal-mapped models require an extra parameter for the file name of their normal map resource when being initialized, so in the object class's initialize method, initialize parameters are used in accordance to the given model type, in this case using the additional normal map file name parameter.

As with 'GameObjects' in Unity, I've added support for empty objects. My implementation of this involves an empty model and texture file. The model consists of a single vertex, and the texture consists of a single transparent pixel.



When placed in the scene, objects created using this model and texture are completely invisible.

An additional feature I added to the object class was support for 'spinning'. 'SetSpin' takes in values for each axis and applies additional rotation to the object each frame based on these values, simulating spinning. When combined with the parental hierarchy and empty objects, I could cause objects to rotate around invisible points. In one example, I used this technique to make a seagull rotate around an invisible point above the island, simulating flying.

Scenes

My initial justification for creating the scene class was to move the creation of objects from where they previously were in the graphics class to a more suitable location. However, implementing this means that I now have a useful method of creating containers for multiple objects. The scene class holds objects, paired with a string representing their name, in a vector container.

```
vector<pair<string, ObjectClass*>>* worldObjects;
```

Class functionality includes the ability to add/remove objects in the vector container, and the ability to initialize/shut-down/render all objects in the vector container.
I've created an instance of the scene class where the majority of what's visible in my scene is set up, which is initialized in the graphics class.

Camera movement

I was advised by my professor to take inspiration from the implementation in the shared example of a previous student's work (Andrew, 2015) when improving the camera movement.
The camera movement present in my project:

Mouse, Arrow Keys	Determine the direction the camera is facing
W, S	Move the camera towards/away from the direction it is facing
A, D	Move the camera left/right, relative to the direction the camera is facing
PgUp, PgDn	Move the camera up/down
Shift/Ctrl	Increase/decrease the movement speed of the camera

These improvements lead to increased efficiency when examining the models in the scene. For example, the ability to use Shift/Ctrl to increase/decrease the movement speed meant I could reach distant models in a shorter amount of time, and could move with more precision when closer to models.
I have also added support for horizontal and vertical sensitivity, the values of which can be modified in the input class, because I found the un-modified sensitivity to be too high for accurate mouse control.
I have also fixed a bug present in the shared example's implementation, where there would still be a horizontal movement increase/decrease when moving forwards/backwards, despite the camera facing directly up or down.

Audio

The sound class is based on a RasterTek tutorial (RasterTek, 2016), with some modifications. I have added support for audio looping and methods for stopping and resetting the current wave file.

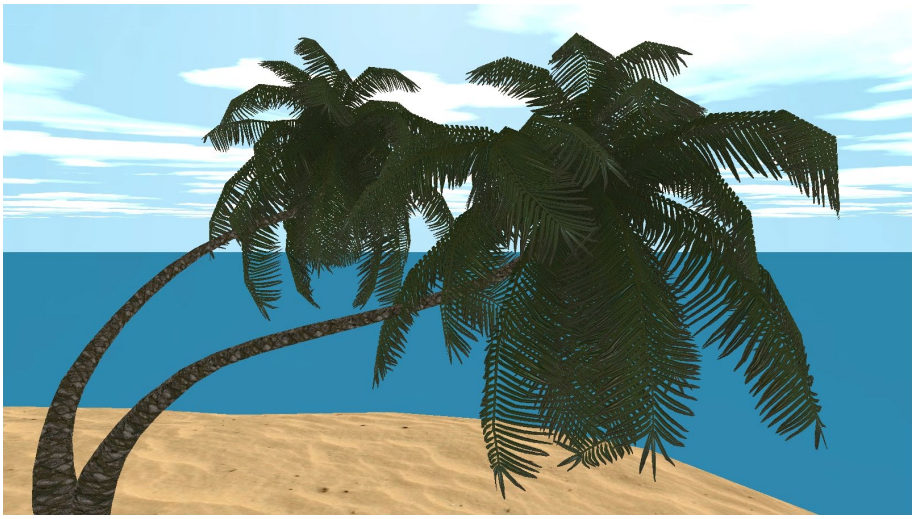
Camera/audio dynamic event

I have added a dynamic event that demonstrates how the camera and audio can be adjusted at runtime. Input from the 'Enter' key is now supported in the input class, which when pressed will switch the current camera type and audio track. A reference to the seagull that's 'flying' above the island is found, which the camera will attach itself to, locking its Y axis rotation to face the island's centre and its X axis rotation to a specified angle. Simultaneously, a new wave file is loaded into the sound class and played. The original camera movement and audio can be reverted to by pressing the 'Enter' key once more.

Alpha blending

Since the trees/foliage models I included use transparent leaf textures, I was required to research alpha blending techniques. An issue I faced was where transparent areas of overlapping leaves would not blend with each other, creating an undesirable effect.

After consulting the documentation of the D3D11_BLEND_DESC structure (Microsoft, 2018), I discovered the 'AlphaToCoverageEnable' member, which when enabled fixed this issue.



Clouds

The sky plane and its shader class are based on RasterTek tutorials (RasterTek, 2016). Along with including the shader class in the existing shader manager, I also modified the sky plane parameters to better suit its appearance to my scene.

Conclusion

Further development

With more time, there are RasterTek tutorials I would like to have followed to improve this project further. One such tutorial describes how to create realistic looking water (RasterTek, 2016), which would improve the appearance of my scene. Similarly, there are tutorials relating to shadows (RasterTek, 2016), which my scene would also benefit from.

Final thoughts

I'm satisfied with the decisions I've made regarding the features I chose to implement. In particular, I feel the implementation of the object-oriented structure provides a strong foundation for future DirectX development.

References

Andrew, E. (2015).

Microsoft. (2018, December 5). *D3D11_BLEND_DESC structure*. Retrieved from Microsoft Docs: https://docs.microsoft.com/en-gb/windows/desktop/api/d3d11/ns-d3d11-d3d11_blend_desc

RasterTek. (2016, January 3). *Tutorial 12: Perturbed Clouds*. Retrieved from RasterTek: http://www.rastertek.com/tertut12.html

RasterTek. (2016, January 3). *Tutorial 14: Direct Sound*. Retrieved from RasterTek: http://www.rastertek.com/dx11tut14.html

RasterTek. (2016, January 3). *Tutorial 16: Small Body Water*. Retrieved from RasterTek: http://www.rastertek.com/tertut16.html

RasterTek. (2016, January 3). *Tutorial 40: Shadow Mapping*. Retrieved from RasterTek: http://www.rastertek.com/dx11tut40.html

RubyDoc. (n.d.). *QuadSphere*. Retrieved from RubyDoc.info: https://www.rubydoc.info/gems/quad_sphere

Unity Technologies. (2016, July 6). *Procedural Examples*. Retrieved from Unity Asset Store: https://assetstore.unity.com/packages/essentials/tutorial-projects/procedural-examples-5141

Unity Technologies. (2018, November 21). *GameObject*. Retrieved from Unity User Manual: https://docs.unity3d.com/Manual/class-GameObject.html