O.L.Sheard U1664298

U1664298@unimail.hud.ac.uk

Computer Games Programming 2: Assignment Submission for Semester 1

Contents:

Page 1:

* Introduction
* Modifications to the Engine

Page 2:

* Implementations of existing code
* Scene rendered by Engine

Page 3:

* Current Bugs
* Refrences

Introduction

The software being present was created to the assignment specifications for the module Computer Games Programming 2 using the given template “New Assignment Template” for a Route 2 (Object Orientated) approach to rendering a scene using an engine.

With that in mind the following document will describe the components of the rendered scene and engine in detail, with breakdowns of the coding techniques used and the implementation of the base engine code.

Modifications to the Engine code

With the existing code covering a substantial number of requirements to render a full scene, few changes were made to the engine in order to create the scene. The individual changes have been grouped into categories based on the similarities of effects to the engine.

**Inputs - Mouse**

To allow the mouse to be used for rotation input on the camera, three of the classes were modified to varying degrees. Firstly the ProcessInput(int&, int&) in inputclass.cpp was edited such that the mouse position was no longer constrained to the limits of the screen width or height, ideally the cursor would be constantly reset to the centre of the screen upon running the function but SetCursorPos did not seem to have any effect on this.

Secondly, a new function was created under the positionclass.cpp as MouseInput(int x, int y) this would take the values generated by the ProcessInput function and set the rotation variable stored in the class accordingly, a -ve x value would rotate the camera left, +ve y value rotated the camera to look up etc. However, without the final edit this would leave the camera controls unstable as not moving the mouse would simply continue to constantly add to the rotation of the camera.

Finally, to amend the constant rotation (Unless the mouse was at position (0,0)) in the graphicsclass class, two variables were created to store the X and Y values. During runtime the ProcessInput function would generate the current mouse position, these X and Y values would then have the stored variables subtracted from them before being passed into the MouseInput function, after this the X and Y variables were set to match the values generated by the ProcessInput function. Doing so effectively created a delta X and delta Y values that rotated the camera only if the mouse had moved since the last frame.

**Inputs – Keyboard**

As the provided code already retrieved the keyboard state and process the buttons being pressed as Boolean values, it was effectively a copy-paste job with taking that existing code and applying it such that more buttons on the keyboard could be used to replace, or create new functions, this included the use of Q and E for triggering events and converting the arrow based movement to one of a standard WASD input, allowing for strafing by taking the existing up and down movement codes and applying them appropriately to the +/- X direction for the camera rendering the scene.

**Inputs – Constraining the camera**

Constraining the camera was simply done through taking the magnitude of the vector made between the camera position and the origin, if the magnitude was greater than the limit then a unit vector was generated and multiplied by the bounds so the camera could be set to a location within the sphere of movement.

**GraphicsClass variables – trigger and pressed**

Much like the generated rotation variable stored, the trigger and pressed variables also increase with each frame under specified conditions. For pressed it only increases while IsQPressed() returns true and trigger will increase while a stored Boolean is true. This bool is set to true when IsEPressed() returns true but will then proceed to set itself to false at the start of the next update if the tan((trigger/2) -150) is approximately 0. This allows for single loops of the two ships in the background as their X position follows roughly the same value.

Implementations of existing code

**Trigonometry**

Most notably, the rotation value allows for the creation of orbital paths as well as the initial use of rotating an object that is being rendered. In order to convert the ever increasing rotation value, it is constrained via one of the three main trigonometry functions; Sine, Cosine and Tangent. From this, multiplying the constrained value will increase the amplitude about the axis affected and multiplying the input rotation value will increase frequency in which a full cycle is completed. By comparison, adding to the input rotation will change the starting phase of the cycle in relation to other objects using the same rotation value and adding to the constrained value will simply translate the object about the axis as normal.

Combining this about two or more axes changes the path the object will move on from an oscillation to an orbit about the scene, moving in a clockwise or anticlockwise path depending on the input.

**Skybox**

The standard approach of texturing an object through the pipeline only applies the given texture to the outside of the object under the presumption that the object will be viewed from the outside. However by translating the object by a negative factor, the renderer would place the texture on the inside of the object. By doing so this allowed the generation of a sky sphere that scaled precisely, encompasses the entire scene and cannot be exited by the constraints applied to the camera.

Scene Rendered by the Engine

The rendered scene can be broken down into groups of models as stated below:

**Model 1 – DeathStar**

This object rotates about the Y axis while oscillating vertically by a sine function. As the rotation is based on the pressed and rotation values, the rotation speed can be doubled by pressing Q.

**Model 2 and 3 – Orbiting “Viper” ships**

These two objects orbit around the Death Star object, this orbit also requires that the models are rotated about the X and Z axes to appear as though they are in a stable orbit with the nose of the model following the constantly changing angular velocity.

**Model 4, 5 and 6 – Background Planet and Moons**

The three objects here all use a main path of orbit to keep in the same effective area as the planet. Meanwhile, each moon has an additional orbit to its position translation such that it can orbit the planet and the second moon also contains a third addition to orbit the first moon as well as the planet.

**Model 7 – Diagonal Planet**

This object is simply a demonstration of applying a trigonometry function to all the axes in the renderer. By shifting the initial phase of the X and Z axes, this object will never collide with the previous set of objects even though they have the same radius from the origin.

**Model 8 and 9 – Warping X Wings**

These two models have a translation applied to them to move them out of the way of the main cluster of objects, a rotation about the Y and Z axes and a small oscillation that gives the impression of hovering. However more interestingly their X position is bound to the trigger and a tan function. This allows the objects to leave the scene moving left to right and re-enter the scene with the same position as the model technically gets sent from +ve infinity to -ve infinity before returning to their original position. A slight inaccuracy in the calculation of tan((trigger/2)-150) means that on re-entry the models do jump slightly.

**Model 10 Sky Sphere**

The final model in the scene is effectively a large inverted sphere created so that the rendered texture appears on the inside of the object. As it is scaled to be larger than the movement of the camera it effectively acts as a background for the scene.

Current Bugs

Currently the main two bugs are on the first frame being rendered, the camera is rotated to look roughly 30° to the left and 45°, I presume this is related to the initialisation of the mouse and can be amended. Secondly, the aforementioned X-Wing models jump on re-entry to within the sky sphere due to the limited accuracy of calculated numbers using trigonometry. Applying a function to lerp the value of trigger to an appropriate value will amend this.

References

**Textures**

The texture for the background sky sphere was sourced from:

<https://exoplanets.nasa.gov/assets/stars6.jpg>

The texture for the Death Star model in the centre of the scene is sourced from:

<http://tira.ifrn.edu.br/alunos/joao/Jogo/deathstar.png>

The texture applied to the background planet is from:

<https://tile.gbif.org/3857/omt/0/0/0@1x.png?style=gbif-classic>

**Models**

The X-Wing ships that are rendered in the immediate background are acquired from:

<https://sketchfab.com/models/32c242b812e549b2aa632373fd994e95>

The “Viper” ships (Orbiting the Death Star) are sourced from:

<https://www.turbosquid.com/3d-models/free-space-fighter-3d-model/612994>

All other content was sourced from the engine template provided by the University of Huddersfield under the name “New Assignment Template”.