**Game Engine Programming Assignment Report**

Introduction

This report will cover the implementation of this project, in which the aim was to design and create a simple and functioning game engine, along with a tech demo to demonstrate some of these features. The project began with gathering information and inspiration from existing game engines (e.g. Unity), and taking their pros and cons into consideration before deciding on how they could be improved. Before starting the implementation of the engine, there had to be a clear design structure that permitted any expansion once a simple structure was created. The engine needed to begin with a basic Component Entity System (CES), that worked alongside a ‘core’ that was used to control the engine, and withhold the standard game loop. The CES consisted of a system that held game objects and components that can interact with one another. This gave the ability to inherit from the basic component class to create necessary components for the game objects, which allowed for control over certain features when used outside of the engine library and within a game created using the engine. Extra required features, such as a class to hold the screen’s data, were also specified to allow the core to pass this information to the components. With the endless possible expansion of the engine, time limited the implementation of certain features that could have been included, but a functioning engine and tech demo was still able to be developed to the initial specification.

Research and Development

During the development of the engine, research was required when designing certain features and implementing them. Unity was used as reference point when designing a lot of the game engine’s features; an example of this is the idea of using game objects which ‘are the fundamental objects’ that are used by the engine and can ‘represent characters, props and scenery’ (Unity 2018). This also gave aid when designing the functionality of the basic component. Unity’s components have the ability to communicate with one another, allowing for advanced functionality and efficiency. They can also be easily interacted with by the user on the game development side, so all of these abilities were deemed necessary to include in the engine design. Issues arose during the evolution of some the features; one example of this was the creation of the game loop and calculating time. In order to fix the framerate of the game, the game loop must be controlled by time and to only be repeated after a particular amount of time had passed. The amount of time passed between each frame must therefore be calculated (deltaTime). This is calculated by taking the number of ticks (milliseconds) that have passed (which can be gathered using SDL’s getTicks() function), and subtracting the number of ticks that had passed by the previous frame. This gives you the difference in time between the last and current frame, which you can then use to delay the program at the end of the loop and fix the framerate of the game. When first implemented, the deltaTime was only calculated when required, rather than during every frame. Once it was calculated during every iteration of the game loop, objects’ movements were much smoother.

High Level Description

The program’s design was created with the notion of allowing future expansion, which meant that the initial design needed to be simple, clear and well structured. This meant that the first thing that was designed was the core, which holds pointers to all the game objects, along with pointers to all the necessary classes that the core will utilise. These consist of:

* Input – This holds the data of all the current inputs made by the user.
* Context – This holds all any information relating to real life, such as deltaTime.
* Screen – This holds the information relating to the screen that outputs to the user.
* Camera – This holds the information of the current camera in use.

The core is also home to the game loop, and it is where the update and display functions are called. These functions are called on all components within a game, and each will perform a different task during runtime relating to said component. Input is also gathered within the game loop, as well as fixing the framerate. The core could be expanded at any time by adding new classes or extending the game loop, but these classes were a good starting point.

Game objects are used in every game, they are the objects that the player can see or interact with. The basic game object within this engine simply stores all the components that are added to it, and act as a pathway between components and the core or other components. Template functions are used to add or get components, meaning that a component of any type can be added to any game object, increasing the expandability. The game object class is inherited from when creating any new game object within a game that is created using the engine, meaning that you can design a game object with a set of components with respective values, and copy that throughout the game (i.e. enemies or bullets).

The component class is a basic class that is intended to be inherited from. It acts as a template for any component, and allows the passing of data between components by attaching get functions to retrieve a component of a certain type. This allows data to be passed from one component to another, so that values for one game object can be stored in respective components, and no irrelevant details needs to be stored within a game object. An example of unnecessary components would be a Player component on an object that is intended to be an enemy; the user should not be able to control an enemy and the data required for a Player component would therefore be irrelevant to an enemy game object.

Drawing to the screen is a fundamental process for all game engines. This engine utilises both OpenGL and SDL to carry out this process, where SDL handles the window and OpenGL carries out all the necessary functions relating to graphics. The Renderer class is used as a wrapper to simplify all the OpenGL code. This means that the complex code for OpenGL is abstracted behind the Renderer class, which uses simplified functions to carry out any process relating to the graphics. The ShaderProgram, VertexArray, and VertexBuffer are all wrapped within the Renderer class. The ShaderProgram handles all the shader code and carries out the drawing of the vertex array objects. The VertexArray class stores and manages all the vertex array objects, which hold vertex buffers that store information about vertices. The VertexBuffer class assigns and stores information to be sent to through the graphics pipeline. Any game object that is to be drawn to the screen requires a Renderer component as that is what transfers all the information of each game object to the graphics card to be drawn.

To generate a controllable game object, it would require a Player component. This controls how the input from the user is managed, and how it manipulates the player game object. This can be inherited from to create a new component, that can manipulate the player in any game to do whatever is needed. In order to move a game object, it also requires a transform component. This stores any data relating to its position within the game world, along with its rotation and scale. The transform component is also used to adjust this information, and it is used to retrieve the information when it comes to drawing or collisions.

Unfortunately, time didn’t permit the completion of the collision component for this project. This would have stored the game objects collision box, which would be used in comparison to all the other game objects that have collision boxes, to calculate if there has been a collision in world space. This could have then been used to perform a task defined by the game itself, for example a bullet colliding with a breakable object, leading to that object being broken, and the bullet destroyed.

The reason for not creating the Input, Context, Screen, as components is because they will always be static and will never need to be manipulated by the user of the game engine. If the engine was fully finished, the Input class would include all of the possible inputs for the keyboard and mouse, while also accommodating for extra devices such as gamepads or joysticks. The Context class would be extended to account for cross-platform support by defining the location of the engine’s data differently for each platform. Finally, the Screen class could be extended to allow for multiple resolutions and screen sizes.

The Camera class was designed not to be a game object, even though it is in the world space. This is because the game objects will never need to interact with it, and would not require any components to be attached to it. If it was necessary to manipulate the camera more than the Camera class allows, this could be done by inheriting from the class and the user adding more functionality. To extend the Camera’s current functionality, a vector of cameras would be added to allow for multiple cameras in once game. This could be used for either multiple angles, using render textures to create mirrors, or for split-screen multiplayer games.

Analysis - 250 Words

References

https://docs.unity3d.com/Manual/class-GameObject.html

https://docs.unity3d.com/ScriptReference/GameObject.html

https://docs.unity3d.com/ScriptReference/Component.html