Game Engine Design and Implementation

Game Engine Programming.

Crisolle Infante.

BSc Games Programming.

Initial Specification

This assignment involves the development of a 3D game engine, showing the architecture behind a game engine which can produce a game or tech demo. The engine being described here, the Emdemnn Engine, is made for ensuring a user can easily create and load objects, create shaders and give textures, handle user input such as with a keyboard, and being able to render all of it into a window. Everything being rendered inside this screen or window can then be viewed by allowing a user to create their own Player style class, which allows for movement, by using the existing keyboard class as well as using the transform component which is automatically attached to each created entity.

Table of Contents

Core

One of the first classes that needed to be created for the engine was the Core class. This class is what effectively represents the internals or 'core' of the engine. This is the first thing a user initializes when they want to start using the engine. This is because on initialization, it creates an object that represents the engine, and within the function, sets the references for various other parts of the engine's functionality, allowing access to them.

On initialization, it is what handles the creation of an SDL_Window, initializing the keyboard, resources and creating what is known as a context for OpenGL. When the engine runs, it manages SDL and keyboard events as well as starting the process to iterate through the function calls of all the contained entities and components. This is because it is core that contains the list for all entities that a user creates, and it is here where their tick() and display() functions are called, which is generally what is processed and displayed to the screen.

Going further into the various references that are set, pretty much every private variable has a getter to access them from somewhere else in the engine if needed, as they can only be accesses from core.

Component-Entity System

Entities are the containers for everything inside the game engine. These contain a reference to core, so that it can access the engine, as well a reference to itself, so that components are able to fine themselves. There is also a refence to a transform, and this is because when an entity is added to the core's list of entities, it automatically adds a transform component to them, as each entity would naturally have to exist somewhere in the world once they are created.

When it comes to adding components, making a different function for taking a different parameter type would make it very tedious and inefficient, so instead templates are used. The reason for this is so that when a user wants to add a component with no parameters, or with 1 or 2 or more of different types, there would be no need to create a function that both takes two parameters, but different types. Instead, templates are what helped in this scenario. This allows a user of the engine to add any component type with any number of different parameters, up to a limit of 3, and this is done easily with just 4 functions, making it easy to manage and update should the engine be expanded on. There is also a function that gets a component, allowing a user to find a component they are looking for if it exists.

When a component is added to an entity, not only is this component added to a list of components that each entity keeps, as each entity can hold a number of components, the weak pointer inside each component is given the reference to each entity so they know where they are. After being added, an onInit() function that passes into it the parameters if any, is called automatically when a component is added. This is so that anything that needs to be initialized before anything else is used on that component is done prior, or simply as to not require the user to manually do it themselves, such as setting a scale to a non-zero value on each added entity to their world.

As component is a base class that each derived component inherits from, it is here where every generic access for references or getters is created, allowing each derived component to up the hierarchy, such as finding which entity it is contained in so it can then access core, or so that another component may find and access the functionalities of other present exiting components contained and referenced within the entity, such as the transform.

Resource Loading and Management System

Getting into providing a form of resource loading, after it has been initialized in core, the resource management system follows a similar design thematic to the component-entity system. The resources class is given a reference to core after it has been initialized, similar to how each new added entity is given a reference to core so that it may access the functionalities provided there, and just like entity, resources has a reference to itself.

This is so that when a user loads a resource, just like when a component is added, the resource knows where it is and has the appropriate pointers so it can go up to core if needed. This resource is then added to the list in resources and an onLoad() function is called with argument passed into the load() function.

From here we have the resource base class, where just like component, all general use functionalities such as travelling up the system to get to core is implemented so all derivative resources also have access to these functions. Currently, what enables the user to render all these loaded resources is a mesh renderer class. This derivative class of component has references to derivatives of the base resource, a model and material class which uses the functionalities of rend, that allows these resources to be used and accessed by the containing component of an entity in the world.

Handling Input

As the game engine is designed to help users in creating a game or tech demo, input from the keyboard is very good to include. This class manages all the keyboard events and allows a user to create their own player class which inherits from component, and calls functions such as translate() to modify the transform components that each entity has. They can do this passing in an integer or char, though for the integers using the macro definitions, they may be limited.

Conclusion

Using the component-entity system with inheritance works really well for this game engine, as each object isn't too large or largely complex and the engine only has classes that inherits from one base class and those derivative classes are not further inherited from, avoiding an evil tree problem. It allows for good object-oriented programming, and maintenance for future work should not lead to very inflexible designs of hierarchies as each component type is basically what contains the data, with the entities just being containers, rather than having large inheritance hierarchies that adding further too becomes too complex and needs a work around.

Bibliography

Anon., 2018. *A beginner's look at smart pointers in modern C++* [online]. Internal Pointers. Available from: https://www.internalpointers.com/post/beginner-s-look-smart-pointers-modern-c [Accessed 12 Jan 2020].

Jordan, M., 2018. *Entities, components and systems* [online]. Medium. Available from: https://medium.com/ingeniouslysimple/entities-components-and-systems-89c31464240d [Accessed 12 Jan 2020].

Stein, T., 2017. *The Entity-Component-System - An awesome game-design pattern in C++ (Part 1)* [online]. Gamasutra.com. Available from: https://www.gamasutra.com/blogs/TobiasStein/20171122/310172/The_EntityComponentSystem__An_awesome_gamedesign_pattern_in_C_Part_1.php [Accessed 14 Jan 2020].