Technical Report

Prometheus Game Engine

Introduction

In this assignment the task was to build a functional game engine using C++ and utilising the OpenGl graphics library to provide graphics functionality. The key design philosophies behind the game engine are efficiency, stability, portability and extensibility. The features included in the game engine should allow the user to load in resources and design custom behaviours using scripts without having to change code within the main engine. To implement this, an entity-component design pattern was used, whereby game objects are loaded as entities and all functions and behaviour are loaded on within components (Gaul, R., 2020).

Specification

Before development could take place, research was done on the current examples of game engines that use entity component systems (hereafter ECS), and existing frameworks that have already been developed for game engines.

From my research I discovered three main dichotomies found in ECS, these being Inboard vs. Outboard, Static vs. Dynamic Composition and Data on Component Vs. Data on Entity. Deciding on these three options laid the groundwork for all future design decisions (Kylotan. 2010).

Inboard Vs. Outboard

These design patterns decide whether components should be added directly to entities and acted on via calls to the entity (Inboard) or whether to only link them to their entities by ID and call on them via subsystems (Outboard). In this engine an Inboard approach was taken, with components being added to the entity and acted on through the entity which then calls the component functions.

Static Vs. Dynamic Composition

Refers to the structuring of entities. Static composition dictates that all entities should have a known set of components that communicate through the same interfaces with the same data types for all entities. Dynamic composition allows different entities to have different components depending on the entity function, which is more flexible but may be less stable and require more advance lookup systems for components and value. For this project dynamic composition was used for memory and flexibility reasons. The inboard design pattern partly solved the problem of component lookup, but an additional template-based component lookup system was designed for any other cases.

Data in Entity vs. Data in Component

The data in entity patten stores all data and variables in the entity itself and is accessible by all child components, whereas the data in component stores required data within the component to be accessed via functions. This engine uses the in-component design pattern, encapsulating component data within the component.

Design (500)

Game Loop

Core houses the main gameplay loop and it is here that all entities and subsystems are stored. Initialized from the main function, all entities defined by the user are added the system. From these vectors of entities all user defined functions in components are called. This takes place through an inboard system whereby the core object calls the game loop functions (eg. OnTick) for each entity, and each entity then calls the same function for of their components. Each component can have unique behaviour in the game loop functions, and data is encapsulated within the component - following the Data in Component design pattern.

Entities

Entities are created within the Core object, and a shared pointer to the instance is passed back to the user. This system means that if the core is alive, there will be a reference to the entity held in the entity vector, and the entity will not be deleted (Unity., 2019).

Components

Components classes are all derived from the parent class Component, providing polymorphism for child classes, and allowing all components to be stored in a component vector. All game loop update functions are initially defined in Component and overloaded in the respective child classes according to requirement, as all components are updated for each update function this removes the need to pointlessly define unneeded update functions in the child classes. Component also provides interface functions to return pointers to active subsystems in Core, such as Environment and Screen.

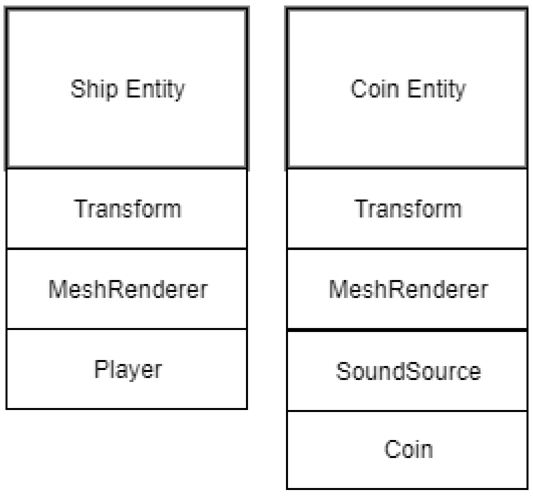

Figure 1. Examples of entity composition.

Transform

Transform component holds information about the rotation and position of the object in the scene, as well as providing access methods to change and retrieve these values. Transform also stores the model matrix for the entity, updating it after a rotation of position change.

Box Collider

Every box collider in scene is checked against the current box collider and the collision return returned and entity transform updated.

Triggers behave the same way but do not change the entity transform, instead raising a collision flag on a per-tick basis denoting whether the trigger has collided with another box collider.

Early Update is used to calculate collisions, allowing the trigger collision to be polled in the on tick function and used by other components without worrying about component update order.

Sound Source

Sound Source is a component to attach sound resources to specific entities. A buffer from a Sound resource is passed to the Sound Source and then attached to entity.

Mesh Renderer

The Mesh Renderer component handles the mesh, shader and texture loading for each individual entity as well as rendering the entity on each update. Mesh renderer retrieves the updated model matrix from the transform component every frame and passes it to the loaded shader.

Resource Management

Resources are inherited from the Resource class to provide child classes with polymorphism. Resources are loaded into a resources instance defined in core and stored in a vector of type resource. This ensures that there will always be a reference to the resource and prevent automatic clean up of the reference in the engine.

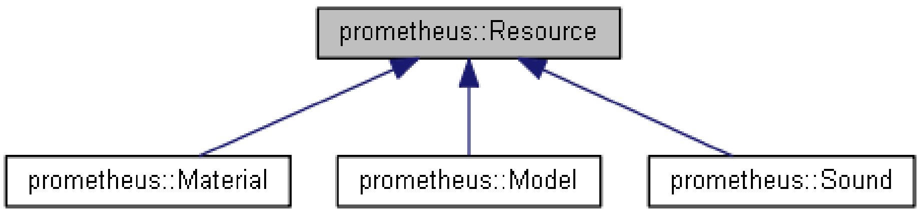

Figure 2. Resource Hierarchy

Model

Handles loading of mesh objects into the engine, taking a path string. Passes the file data to the rend mesh parse function and stores the result.

Material

Handles loading of shader files and texture images from paths and stores as rend::Shader and rend::Texture in class.

Sound

Handles loading of sound files via path. Sound files are loaded in from a file path and data is saved in a buffer for access by a sound source component.

Interfaces

Interfaces are singleton instances that live in the core object and provide access to input devices or helper functions such as timers. Interface instances are initialised in the core and stored as a pointer, and can be accessed through functions in core, one of the reasons that entities must have a reference to the core instance.

Environment

Handles standardisation framerate standardisation and calculates delta time per tick. Ideal delta time is calculated from a goal FPS, and remaining time is slept off each tick if running too fast. Provides access functions for components, where delta time is used to standardise the rate of physics calculations and movement.

Keyboard & Mouse

Polls keyboard and mouse on a per tick basis and provides access functions to the components to check if certain input has happened.

Data Management

Encapsulating input devices within instances and using smart pointers to refer to them handles many problems with memory that might occur. On exiting core loop all references to the interface objects go out of scope, and if the reference count drops to zero the instances will be deconstructed. The same goes for components and resources, occurring once the main loop exists.

Conclusion

Using a component-entity system has created a flexible and extensible framework with which to build games, and by avoiding the issues of complex inheritance hierarchies the user experience is improved. Dynamic composition means the user can build lightweight entities using only necessary components, while saving on memory and performance. Storing the data in component neatly encapsulates scripts, while component lookup and access functions provides flexible component interfacing.

Bibliography

Kylotan., 2010. *Are there existing FOSS component-based frameworks?* [online]. Game Development Stack Exchange. Available from: https://gamedev.stackexchange.com/questions/4898/are-there-existing-foss-component-based-frameworks/4966#4966 [Accessed 17 Jan 2020].

Anon., 2014. *Building a Data-Oriented Entity System (part 1)* [online]. Bitsquid.blogspot.com. Available from: http://bitsquid.blogspot.com/2014/08/building-data-oriented-entity-system.html [Accessed 17 Jan 2020].

Anon., 2014. *Component · Decoupling Patterns · Game Programming Patterns* [online]. Gameprogrammingpatterns.com. Available from: http://gameprogrammingpatterns.com/component.html [Accessed 17 Jan 2020].

Anon., 2014. *What's an Entity System? - Entity Systems Wiki* [online]. Entity-systems.wikidot.com. Available from: http://entity-systems.wikidot.com/ [Accessed 17 Jan 2020].

Unity., 2019. *Entity Component System | Package Manager UI website* [online]. Docs.unity3d.com. Available from: https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/index.html [Accessed 17 Jan 2020].

Gaul, R., 2020. *Component Based Engine Design | Randy Gaul's Game Programming Blog* [online]. Randygaul.net. Available from: https://www.randygaul.net/2013/05/20/component-based-engine-design/ [Accessed 17 Jan 2020].