NSENGINE – A C++ GAME ENGINE

Games Engine Programming – Josh Eyres s5304176

Contents

Introduction

This report aims to give an overview of my game engine I have created for the game engine programming unit, I will go through the initial specification that the engine was based around. I will then move on to go through the design of the engine, explaining how it works in terms of the core loop and the entity component system it uses then I will explain the design choices I made followed by an analysis of the features and functions, any drawbacks the engine might have and justify the choices I made. I will also provide some diagrams throughout this section to further reinforce my explanations. Next, I will analyse the project going through the strengths and weaknesses of my engine and what I think the causes and solutions to those may be. Finally, I will provide a summary and detail what I would improve upon and implement if I were to have more time on this project.

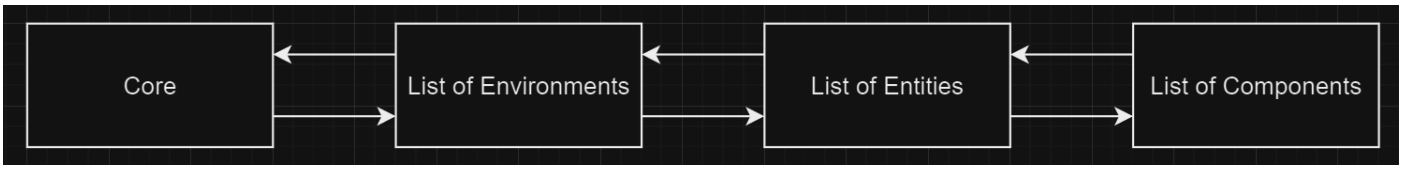
Initial Specification

The specification for this project was to create a 3D game engine and a simple game in C++ using an entity component system, inheritance, or a hybrid of both. The engine needed to implement a good number of features, but also be integrated well with each other to allow for reusability and scalability. Finally, the engine needed to be developed using a version control system, in my case I used git through GitFork and make use of CMake.

Program Design

Overview

My game engine has been built using a hybrid approach of an entity component system and inheritance where there is a base core class that handles initialisation and running all the environments, entities, and components within them. The simple structure is that Core owns all the environments, the environments own the entities within them, and the entities own all their components. This is so that you can trace the path either up or down to get whichever piece of the engine you need to access.



This is then accessed by the user of the engine through functions within these classes to protect the data within and not allow them to directly change required data and variables.

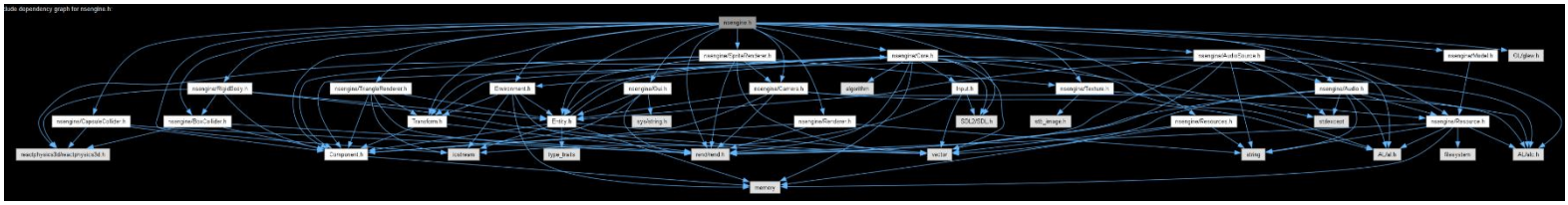


Figure 1 Include dependency of main header file

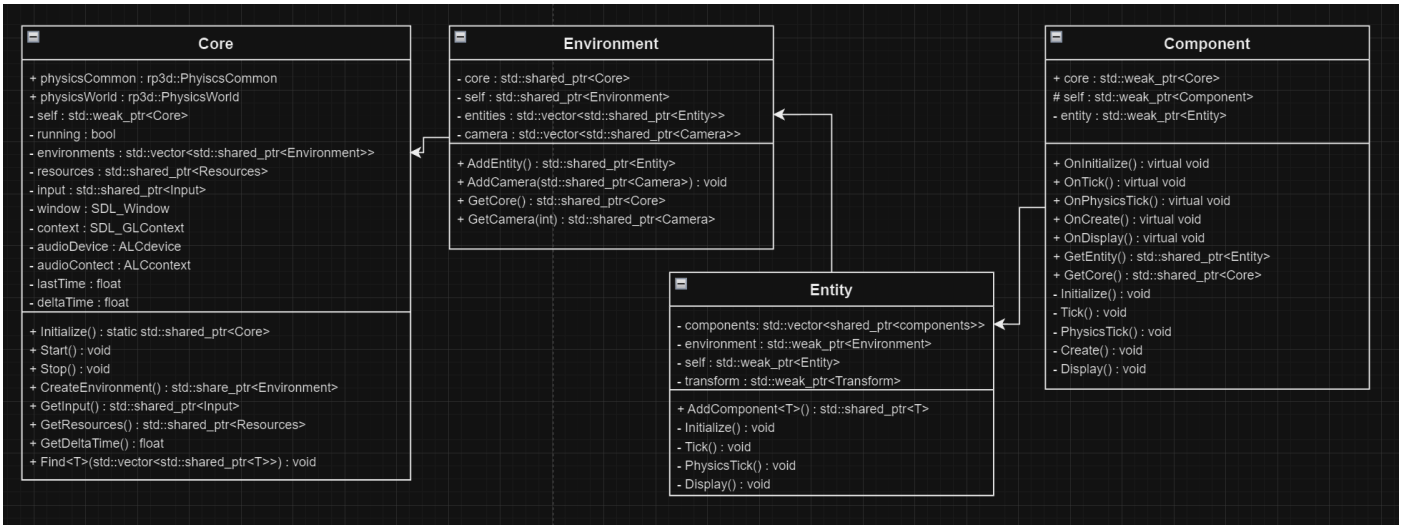
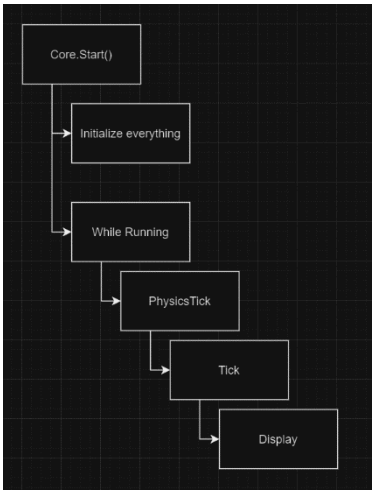


Figure 2 UML Diagram showing the relationship of Core, Environment, Entites and components

How It Works

Engine Loop



Within core there is a start function inside this function it has a running loop that runs through each environment, entity within that environment and calls the associated event functions to allow for sequencing of events. Before this loop there is an initialize function that is ran in the same fashion to allow for setting up components and variables within them. This loop captures and populates the input class with data from the keyboard and mouse. Each entity also calls these event functions for each attached component. This loop allows the engine to properly function.

Events

The component events contained in this engine are:

OnCreate – Called when the component is added to the entity.

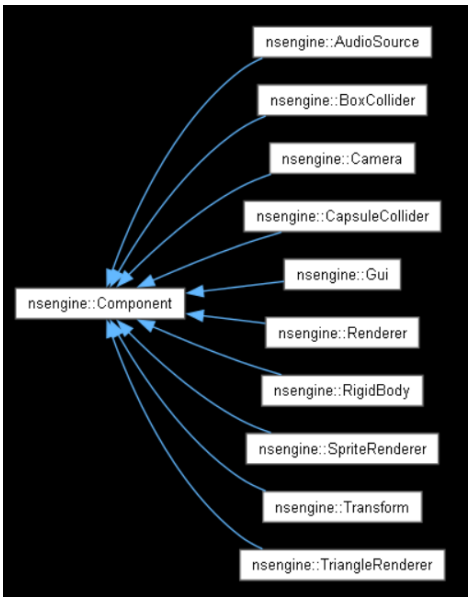
OnInitialize – Called before the running loop.

OnPhysicsTick – Called every frame before the OnTick function used for physics simulation.

OnTick – Called every frame after the OnPhysicsTick.

OnDisplay – The last thing called.

Entity Component system



This game engine is built around an entity component system. Essentially the engine has a base component class which is derived from to create new types of components, such as colliders or transforms, these components get added to a parent entity which holds a list of all the components attached to it. This allows for very flexible environments because you can add a new entity into the environment and very quickly add previously created components either from the user or components built into the engine itself. This means the user can rapidly prototype and create games with the functionality they have already implemented.

Environment

In this engine there is support to have more than one environment to switch between, like scenes in unity, this is done by the environment owning everything below it. This means that theoretically the user could destroy an environment and none of the entities within it would display. Meaning the user could create functions to load new environments.

Design Choices

I have structured my engine to mimic the method unity for their game engine, albeit a much simpler version to make it faster and easier to use.

React Physics 3D

I chose to integrate a physics engine, React Physics 3D into the engine, this allows for complex physics to be used in the engine whilst also not being overwhelming as I wanted it to be lightweight and simple. It provides the basics of rigid bodies, different types of colliders, collision detection and much more. I then wrapped this in my own classes to make it even simpler for the user so they can add collision and physics by adding a rigid body and collider component to their entities.

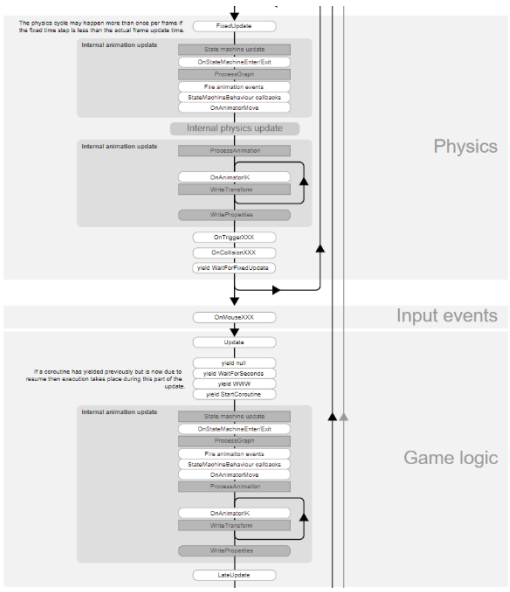


Figure 3 Running loop of unity game engine

Rend

This engine makes use of an open gl wrapper known as Rend, which is a library that I have adapted to achieve the rendering tasks necessary for this game engine, a big reason to use this library is that you do not need to clean up memory because it utilizes RAII and releases its resources properly, basically meaning that any resources are automatically cleaned up when they go out of scope.

Core Components

Within the game engine I have developed a few core components for use within the engine they are as follows:

Audio Source – This component is used to handle any audio within the game, The user can set the audio through the resource system and simply play it from this component.

Collider – There are two colliders; a box collider is simply used to create a collider in the shape of a box for collision, it also houses a simple Axis Aligned Bounding Box (AABB) collision check to check

against other box colliders. There is also a capsule collider which provides the same functionality in terms of the physics engine just in the shape of a capsule but does not house the AABB Collision check.

Camera – The camera component is responsible for what the player sees, it integrates with the Transform component to allow the user to set the camera to be in whatever position or orientation they would like. It also houses the ability to follow a set entity, which is useful for making things like player cameras. Anything that renders to screen should also reference this component by getting the View and Projection matrixes used to move objects into the cameras view.

GUI – This component allows for rendering 2D text onto the screen via orthographic projection, the font for text is set using the resource system, then the component houses things like set position and set text for use in the engine.

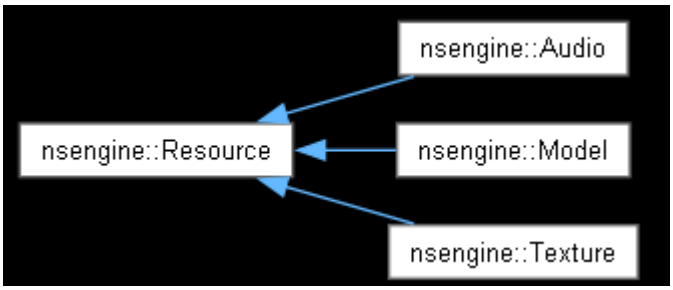
Renderer – This component handles rendering any models the user may want, the user simply sets the path to the model they want using the set path function and the class handles the rest. There is also a sprite renderer used to render textures directly that can also handle 2d animations.

Rigid Body – The Rigid Body is the component responsible for the physics within this game engine, there is in built collision so that the entity will not just fall through the world, the user can choose to change the type of rigid body between dynamic, kinematic, and static. There are functions to add a collision shape to handle the collision, a trigger shape for collision events without affecting physics, and the user can move the rigid body by simulated forces for physically accurate movement rather than just forcing to a position.

Transform – This component handles the positional and rotational data for the entity, it is automatically added to each entity when they are created as almost all entities will need it to be moved around the environment.

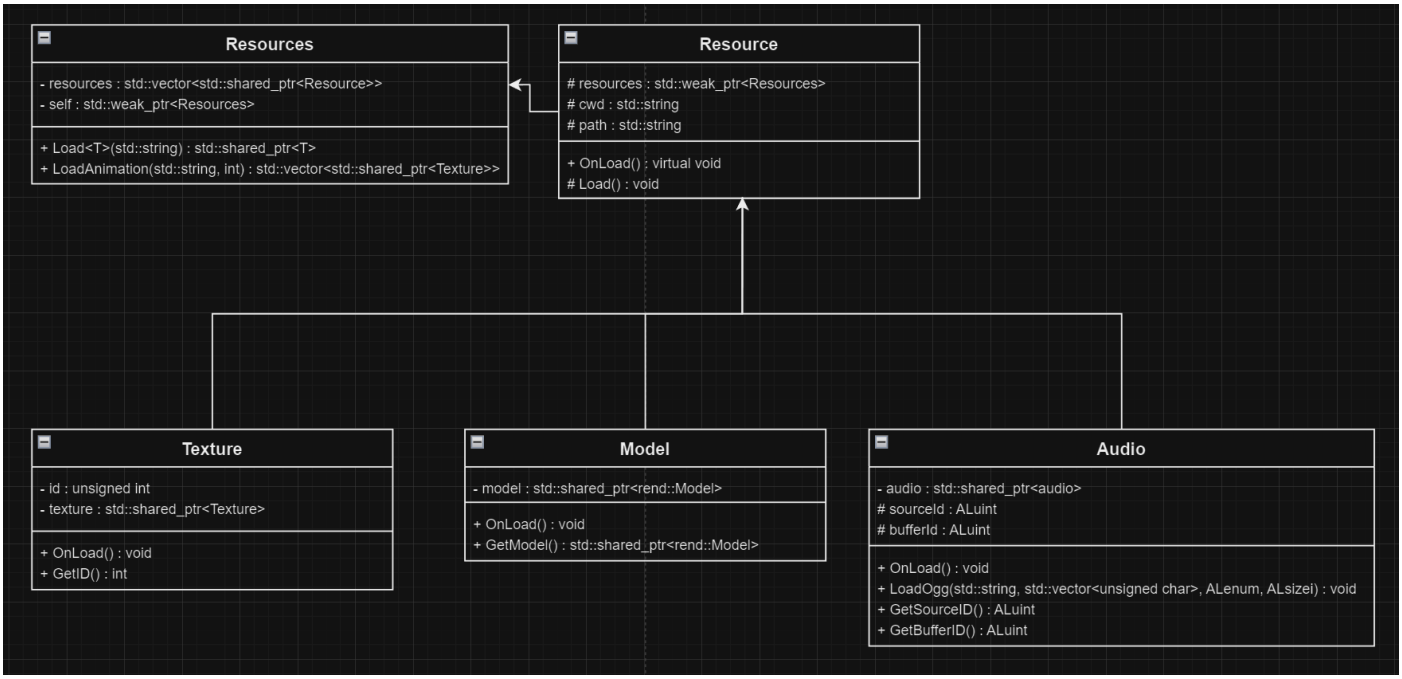
Resource System

The resource system is how the classes that use assets will get their data, currently there is support for Audio, Models and Textures and by extension fonts, as the font system makes use of textures.



The resource base class holds the on-load functions the entity and components that the derived classes can override to add their own functionality.

The resource is then used in a Resources class which is where the actual load function for the asset is because it is a template function to allow for the user to implement any other resources, without having to re-set this function up. The resources class also contains a function to load a whole directory of textures which is simply a quality-of-life function to load an animation so the user can just enter a path to a directory of frames.



Project Analysis

Strengths

One of the biggest strengths that my engine has is that it is simple and lightweight, I aimed to make this the case to allow for rapid development and production of games without too much code. Another benefit of this is that it is not resource intensive at all, which makes it very scalable in terms of the engine itself not using up too much allowing the game itself to be more complex should the user wish.

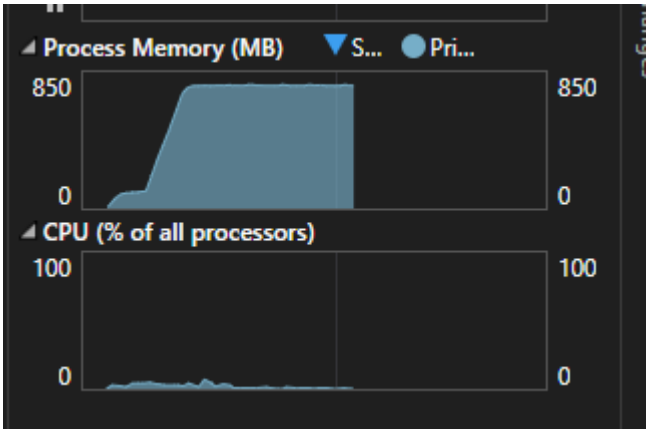


Figure 4 Resource usage on demo game

Another strength I believe this engine has is that it houses a full featured physics engine which handles much of the physics for me, it has support for much more than I have managed to implement that I could implement in the future, such as velocity, contact points, different collision shapes, joints and much more.

This engine also uses smart pointers which add some memory protection inherently by stopping memory leaks compared to normal raw pointers.

Weaknesses

While having a simple and lightweight engine is a strength, I believe that is also a weakness in some regards, for instance because of this it lacks some of the more complex features other engines may offer. One example of this is proper collision testing instead of just AABB testing. This is something the physics engine I am using offers however I ran out of time to implement it properly. A weakness I personally found with this engine is that

it is not always obvious what you are doing, for example when you add a collider without a renderer it is very hard to tell exactly where it is and what it will collide with.

This engine is also restricted to png and ogg files for textures and audio which is something I would like to improve in the future.

Conclusion

In conclusion, I believe that I have succeeded in creating a game engine to the original specifications, it is based on a hybrid ECS-Inheritance system, it has many features that I feel are well integrated with each other however, I feel it could benefit from being more complex. If I had more time, I would implement more features from the physics engine and improve the cleanup of the game engine itself when things are deleted. I would have also liked to create an actual user interface for the engine itself to further mirror other engines, I feel this would make it easier to work with and more obvious what was happening. I would also like to add cross platform compatibility. Finally, I believe that the setup and framework is there for the game engine, while I may have run out of time to implement all the features I wanted too it is a very strong point to continue working on and turn it into something even better.

References

User manual (no date) *ReactPhysics3D*. Available at: https://www.reactphysics3d.com/usermanual.html#x1-20001 (Accessed: 16 January 2024).

librend-byexample (no date) *librend by example*. Available at: https://brightspace.bournemouth.ac.uk/content/enforced/343659-TBC/librend-byexample.pdf (Accessed: 16 January 2024).

Technologies, U. (no date) *Rigidbody*, *Unity*. Available at: https://docs.unity3d.com/ScriptReference/Rigidbody.html (Accessed: 16 January 2024).

Technologies, U. (no date a) *Monobehaviour*, *Unity*. Available at: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html (Accessed: 16 January 2024).

Technologies, U. (no date b) *Order of execution for event functions*, *Unity*. Available at: https://docs.unity3d.com/Manual/ExecutionOrder.html (Accessed: 16 January 2024).