Game Engine Programming

Charlie Lloyd-Buckingham

A star in space

Description automatically generated with medium confidenceProject Overview:

The intent of this project was to create a game engine in which loosely followed the Entity Component System standard. There are several features I wanted to include in the engine, specifically making it as accessible and customisable as possible.

This mainly came to affect the rendering pipeline, with the allowance of post processing and the automatic material property to shader uniform process.

A full list of my implement features is here:

* Resource Loading:
  + FBX files: - Mesh data.
  + OGG files: - Music and SFX.
  + Material files: - Properties files for mesh materials, including shaders and textures.
    - Mesh materials.
    - Skybox materials.
  + GLSL files: - Shader programs containing the code for Vertex and Fragment shaders.
  + PNG files: - Texture files used for the meshes.
    - Singular
    - Cube-Maps
* Model Rendering:
  + Rendering using OpenGL.
  + Multi-Material Rendering.
  + Shaders.
    - PBR.
    - Unlit.
    - Reflection probes
  + Runtime material property editing.
* Audio:
  + 3D audio effects using OpenAL.
  + Audio looping
  + Audio playing on start
* Post Processing
* Debugging:
  + Gizmo rendering for ‘Debug’ mode.
  + Console outputs for ‘Debug’ mode.
  + Framerate calculator.
  + Print Log, Warning, and Error functions.
* ECS
  + Entity creation
  + Entity destruction
  + Component creation
  + Component destruction
  + Transform component-based positioning.
  + Delta-Time
  + Updates
  + Fixed Updates
* Scene Loading
  + Scene switching

Research:

Unity:

A large portion of the research into the engine was spent on looking at how other engines, in particular Unity, solve problems. Specifically how unity organises its update loop – execution order.

{ <https://docs.unity3d.com/Manual/ExecutionOrder.html> }

Though Unity is a much larger Engine, my code follows loosely the same flow as there’s:

Text

Description automatically generated

Loop Execution Order:

* Fixed update
* Input.
* Update.
* Render.
* End of frame.

The main differences from Unity’s execution order and my own is that I am not yet separating rendering into different discrete sections. Right now everything is being rendered at the same time, the only way objects are seen above each other is using a depth buffer. So if a wall is occluding a Gizmo, the Gizmo will not be shown.

PhysX – NVidia:

Though I didn’t end up getting it fully implemented I began working on combining my engine with Nvidia’s PhysX physics engine. The PhysX docs are split between two different pages (both downloadable from GitGub).

{ <https://github.com/NVIDIAGameWorks/PhysX/tree/4.1/physx/documentation> }

* PhysX API: Contains a list of all classes
* PhysX Guide: Contains an in-depth guide to setting the PhysX simulations up.

LearnOpenGL:

A large portion of my rendering process was designed around the tutorials from LearnOpenGL. Specifically focusing on the pages on PBR:

{ <https://learnopengl.com/PBR/Theory> }

Since these shaders where so different to the ones required for simple specular lighting, this was one of the main focuses on why I needed to get the ‘Material properties to shader uniform’ pipeline implemented. And though LearnOpenGL doesn’t have a tutorial for this or mention it much at all. There where other sources around the internet that allowed me to implement it.

OpenAL

OpenAL also has its own documentation, other then the notes from the labs I had a look at this to help set up the 3D audio.

{ https://www.openal.org/documentation/OpenAL\_Programmers\_Guide }

Diagrams:

The engine consists of 38 classes, each in charge of doing wide range of things. The structure of the program is shaped as follows:

Diagram

Description automatically generated

Figure 1: class diagram showing general structure of engine.

Though the engine is big, a lot of the classes inside it aren’t required to do anything in it. The only ones really required to be accessed are Core, Scene, Entity, and the Component classes.

The game I have made using the engine is created by inheriting directly from the Scene class and the Component class. The way scene loading works in the engine is to supply an inherited scene class where the LoadScene() function is overridden and replaced with calls to AddEntity() and AddComponents(). This allows for anything to be spawned, anywhere in the scene with any behaviour the developer might intend.

Diagram

Description automatically generated

Figure 2: Game inheritance for scene loading

This also extends to the component system, since an entity will treat any component the same way, a developer can assign any intended behaviour they would want to an entity by creating a class that inherits from component. This gives them access to frame inputs, the GraphicsManager and the ResourceManager, as well as other things in the scene.

Diagram

Description automatically generated

Figure 3: Game inheritance for component classes

Improvements:

Unfortunately, without the implementation of physics, and mesh animation the engine isn’t finished. With these future features implemented, almost any single player game could be created.

With that said though, there are even more improvements and optimisations the engine should really go through, if it was to be looked as like a realistic alternative to one of the more commercially available engines.

The first examples would be the resource system, so far this can only load very specific file types. The resource manager also never deletes the entities it has access too, this can cause memory to continuously build up over the course of a larger game. Ideally the memory should be dropped if no objects inside the game are taking ownership of a resource using shared pointers.

It would also be worth while to look into simple networking for multiplayer, so far, I haven’t done any research into this so it might be completely impossible with the engine’s current architecture.

References:

* Learn OpenGL, 2016. *PBR / Theory* [online]. Available from: <https://learnopengl.com/PBR/Theory>
* Unity, 2020. *Execution Order* [online]. Available from: <https://docs.unity3d.com/Manual/ExecutionOrder.html>
* Nvidia, 2021. *PhysX Guide* [online]. Available from: <https://github.com/NVIDIAGameWorks/PhysX/tree/4.1/physx/documentation>
* OpenAL. *OpenAL Programmers Guide* [online]. Available from: [www.openal.org/documentation/OpenAL\_Programmers\_Guide](http://www.openal.org/documentation/OpenAL_Programmers_Guide)