CS480 - Senior Project

Joshua Chomicki

December 1st, 2014

Adviser - Dr. Poe

Final Overview - WolfTail™ Game Engine

Introduction

Ever since I received my first video game as a child, I had always been interested in learning the skills required to develop my own. For this reason, the initial requirement that I set for myself when deciding upon a subject for my senior project, was that it must be related to game development. Previously, I have been on teams developing games that never came to fruition, and through that experience I was able to play around with some of the popular, commercial video game engines such as Unity3D and Unreal Engine. They were useful, but I always felt limited by the requirements set forth by the engine's designs. I had to program by the engine's rules and deal with any shortcomings that came up from their design through sickening hacks. It was from this frustration that I decided to create the WolfTail™ 3D game engine for my senior project.

There were a number of outcomes I hoped to achieve by working on this project. Firstly, up until this point, I had never developed any piece of software of substantial size; each project was something small for either a class or work assignment, able to be completed in a week's time at most. I set out to change that with this project. I wanted the experience of developing and designing a large, cohesive software product that would challenge my development skills and make use of the knowledge I had gained in all of my internships and 2 years' worth of college. Second, as mentioned previously, I was annoyed by the lack of control that most major game engines displayed and sought to either fix these shortcomings or learn why they were necessary. I wanted to understand how game objects were handled, how models were rendered on the screen, and how all the subsystems interacted together. What better way is there to learn how an engine works than to be the one who built it? Finally, I wanted to force myself to set an obtainable goal that would forward my desires to eventually release a commercial video game. A significant amount of developers on the internet are in agreement that if a person wants to make video games, then they should do just that and not worry about creating the base engine. While I do see this as realistic advice for the majority of new game developers, I believe starting from the ground up allows more freedom over the content created. If there is an underlying issue that is hampering development, control of the entire source allows the problem to be addressed instead of hacked around. I want complete freedom to create what I imagine, and by developing the engine, I am only limited by my own ability as a developer.

In order to further test my abilities beyond the fact that I started with no idea how a game engine is structured or created, I wanted to try and use as few external libraries as possible so that I was purely the only one developing everything in the engine. I knew that I would be developing the engine with C++, the Standard Template Library (STL), and DirectX. My goal was to try and use no other libraries, and by the time of presentation I will have completed that goal. My decision to use DirectX instead of the alternative, OpenGL, was made from the fact that I would be developing on Windows and did not want to deal with Microsoft's cold shoulder to OpenGL support. To specify even further, I chose to develop on the old June 2010 version of the DirectX SDK. The newer version is built directly into the windows libraries, but has multiple changes to the call structure to account for Windows 8, rendering older tutorials useless. Since I would be learning everything from the bottom up, I chose the option that contained the most resources for when I encountered issues and will convert to the newer library in the future.

Architecture

Deciding on the overall architecture of the engine was arguably the most difficult part of developing the game engine. There are multiple ways to implement every subsystem and tie them together. It did not help that initially I had no idea how any of it was accomplished, and everyone on the internet could not agree on whether one design was better than another. I quickly learned this would be a central theme in game engine development. Designing a massive software system is very difficult without previous knowledge and experience.

I decided to start by following a series of tutorials to get something rudimentary up and running. The result of this work was a single file that contained just enough code to show a rotating cube on the screen. This provided me with enough information on how rendering works that I was able to start creating my engine from the ground up. I saw that there were very distinct parts to the process that I was going to be able to split up into their own classes, with an overarching object that made them all work together. After much more designing, coding, researching, and iteration I developed the design below.

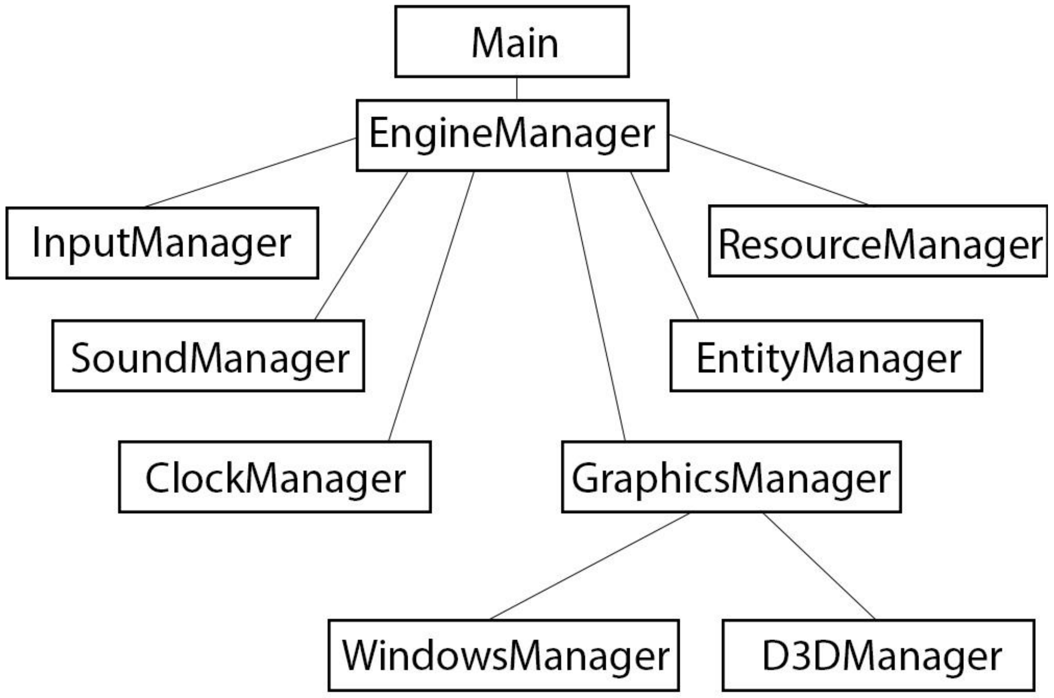

Figure 1. General Initialization Architecture

The EngineManager class is the overarching object that controls interaction between all of the other Managers. In order to start any game, the EngineManager class must be initialized, run, and cleaned up. It handles the rest of the engine by itself. This is the top object that starts the initialization and does the final cleanup. In between initialization and cleanup, this Manager is where the game loop exists. All of the objects are updated and told to render from this Manager.

Keyboard and mouse input is initialized and captured using DirectInput inside of the InputManager class. The InputManager tracks which keys are pressed on the keyboard as well as the screen position of the mouse and its delta since the last 'update.' The mouse positions and key press states are updated by the EngineManager calling this Manager.

Sound support is handled by the SoundManager. It enables control over the primary sound buffer and initializes new buffers that contain the supported audio files for future playback. It is called by the ResourceManager to perform the actual converting to a playable buffer, which is then stored back in the ResourceManager.

The ClockManager handles everything timing related in the engine. As of this moment it is not heavily used other than for creating a consistent time-step using a high performance timer that allows game objects to be updated independently of frame rate. In the future, this would be used for animations, effects, and physics.

The ResourceManager is where all of the assets such as sounds, models, and textures are loaded from the local file system into a useable form by the game engine. All assets are to be stored inside of the Assets folder that is relative to the engine while running. The user decides how the folder structure should be inside, but all assets need to be at minimum in the Assets folder. When a user asks the ResourceManager for an asset to be loaded, it is in the form of a resource ID, which consists of the path to the asset from the Asset folder, delimited by periods. This resource ID is also the key in an unordered map to the actual loaded resource. Every time an asset is loaded, it is stored in the ResourceManager and all future load calls will immediately return the already loaded asset from the unordered map. This allows for asset reuse. The ResourceManager works with the GraphicsManager and SoundManager since they know how to perform the actual conversion from file into useable buffer.

```
//This line will load the model file: Assets\Models\batman.text
ResourceManager::getInstance()->LoadModel("Models.batman", buffers);
```

Figure 2. Example LoadModel call showing the asset path conversion

The GraphicsManager is used for everything rendering related. It creates the buffers from model information, contains the required light shaders, and renders everything to the screen. All rendering is accomplished via Forward Rendering. WindowsManager and D3DManager are both contained inside of the GraphicsManager, as they both contribute to the visuals produced by the engine. The WindowsManager processes everything needed for creating the window and process used for showing the visuals, and D3DManager sets up the DirectX rendering pipeline.

The last and hardest to design manager would be the EntityManager. The next section will describe the hybrid Entity-Component/Hierarchical Object approach to game object handling that I came up with that is implemented in this EntityManager. Generally, the EntityManager is in charge of keeping track of game objects and the components that make up their appearance and behaviors. This is the manager that has the largest effect on actual production and gameplay, and thus took the longest to design.

Some of the Managers depend on the others for complete functionality, but multiple instances would result in loss of data and bloat. For that reason, each of the Managers is a Singleton whose reference can easily be obtained. Also, a number of the Managers are used by entity components for direct functionality. Entities and components will be discussed in more detail later. While I believe this to be a good design, I am planning on doing a small refactor of the managers so that they contain a smaller Singleton class that components may use to interact with the Managers. Right now, the components have complete access to all of the Manager's functionality and I believe this control should be tightened.

Entity-Component/Hierarchical Object System

By far the largest controversy about game engine development seems to be how the game specific objects should be stored and controlled. The two outspoken rivals are those who prefer a data oriented, Entity-Component System or an object oriented Hierarchical Object System. Both sides have great points, both are useful in different situations, and both have their downsides. After researching different implementations using one or the other, I came to the conclusion that neither was the *best* answer and it depended on what made the developer happier to use. For this reason, I came up with a hybrid of the two and while it could use some more ironing out, I think the first iteration is a success.

The Entity-Component System works by assigning each object in the game world an ID. By itself the ID is nothing more than an identifier. This identifier is then used as a key in a map or array-like data structure that holds a specific bit of data. Whenever the data is needed for that object, it is pulled easily out of its storage location using the ID. The power lies in the ability to just add or remove a set of data from the data structure to change the behavior of the entity.

On the other hand, the Hierarchical Object System works by creating a base object that contains all of the data and functions that make up that game object. This object can then have children that expand upon the parent's data and behavior, creating a tree of objects with behaviors built upon others. This system is very easy to visualize over the data central entity-component system.

My hybrid system combines what I believe to be the best qualities of both the Entity-Component System and the Hierarchical Object System. In my System, each game object must be a child of a GameObject class which is assigned an ID from the EntityManager. The traits and behavior of each GameObject is then defined by children of the Component class. Components are classes that contain some type of data relevant to the containing object and/or some type of behavior. Since it is an object, the Component can really do anything the user wants. Some built in examples of Component children would be PositionComponent, ModelComponent, and CameraComponent. PositionComponent handles the Entities location in the game world and has functions to translate and rotate that position. ModelComponent holds onto the buffer that holds all of the vertex information for rendering the assigned model. CameraComponent has a function for returning the required information that the renderer needs from the cameras position in the world.

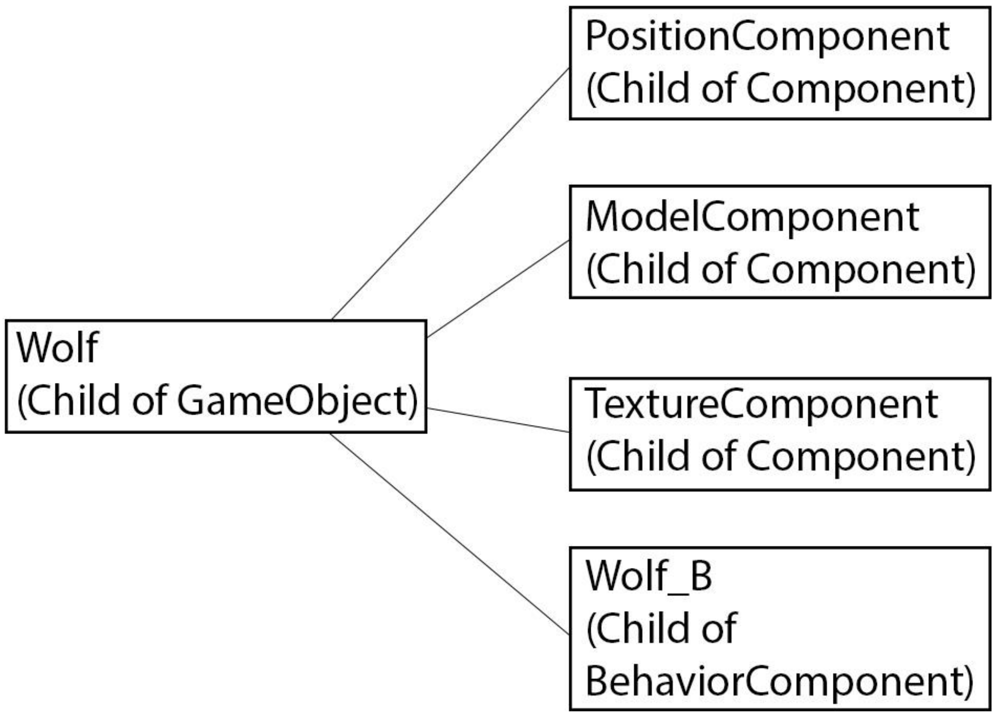
PositionComponent (Child of Component)
ModelComponent (Child of Component)
Wolf (Child of GameObject)
TextureComponent (Child of Component)
Wolf_B (Child of BehaviorComponent)

Figure 3. Example GameObject with attached Components

The power of this hybrid system lies in the ability to use objects instead of purely data. Each GameObject that is supposed to perform some action, no matter how simple, requires a BehaviorComponent. This is a special component that is aware of all the GameObject's other components and actually calls their functions to accomplish their behaviors, as intended by the developer. Do you want a GameObject to float? Attach a custom floating component and call it in the BehaviorComponent. Want to completely change a GameObject's behavior to another? Just attach the new BehaviorComponent. Want to make the GameObject disappear? Remove its ModelComponent. My hybrid system makes swapping behaviors and traits of game objects easy while also making it simpler to visualize the structure of game objects.

The base GameObject class contains functions for attaching and removing components, so that all of the children also have that ability. The EntityManager is the class that actually stores and keeps track of the Components inside of an unordered map of unordered maps. The top layer unordered map has keys for all of the component names. The values are then another unordered map which holds the actual corresponding Component classes keyed by the GameObject ID that they apply too. With this implementation, retrieval is quick using two keys, and it is possible to pull entire maps of Components to iterate over. The EntityManager saves space when running by storing and reusing GameObject ID’s from old, deleted GameObjects. This saves space by not causing the map to grow for a new value when there are old, empty locations available for use.

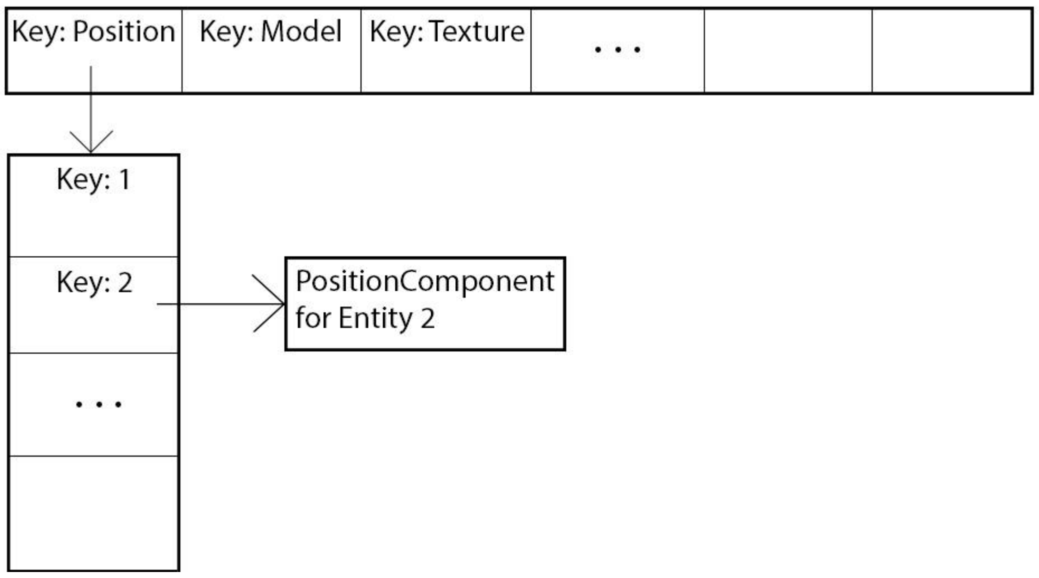

Figure 4. EntityManager unordered map of unordered maps for storing Components

<u>Difficulties</u>

There were many challenges that I knew I was going to have to face before starting this project. As mentioned earlier, I headed into this project without any clue on how a game engine is designed. I had only previously worked on the top level of other game engines, without any knowledge of how the lower levels functioned. I had to perform extensive research in order to purely draw out a design. The books and websites that I read through gave hints on how specific sections of game engines work, but I could not find anywhere a distinct indication of how everything works *together*. Not only did I not have the background knowledge to make this project previously, I had never designed a large, cohesive product and trying to do so without any indication how any of it would work proved quite difficult. I relied on the ability to write small sections that I knew would be needed, and then broke them down into separate parts that, after many revisions, would eventually form the Managers that I ended up with.

An issue that I did not account for was my inexperience with 3D Math. I started the project thinking that the math would be the easier part, but once I started actually writing the PositionComponent, which handles most positional 3D math of the entities, I realized that once again there were many different ways to perform the operations, and there were not any good tutorials or descriptions that fit the type of work I needed to complete. It was hard to find help when most developers do not have to worry about the barebones 3D math that is performed in the background of most game engines. My largest issue was eventually solved through many hand-drawn graphs and conceptualization. I was trying to make rotation independent of the world's axis, but all that occurred was orbit-like rotation around the world's central axis. This was solved by performing the rotation and translation separately from the final result and then just adding it on to the previous position and rotation. This was not a solution simply found online.

Along the lines of math, I initially was looking forward to learning how to write custom shaders, but ended up having little time to focus on something very specific to one small part of the engine. I spent just enough time to understand how shaders work and to create a couple for the included light types, but I still do not understand them to a full extent. They ended up being significantly more difficult to develop than I previously thought.

The final difficulty, which affected this project the most, was time. I knew this project was going to be relatively complex and time consuming, but I did not expect developing a 3D game engine to be as complicated as it ended up becoming. I realized this was a much larger project than could be completed in the span of a semester and I am glad I was able to start during the previous summer. Every single system is very intricate and has many little details that need to be accounted for. This required much time to be spent in small areas of the engine in order to fix weird bugs that popped up, or to design around a specific quirk that appeared. There were also multiple extra problems I had to account for, such as converting model files into a format that my game engine could read. I had to set aside time to write a small helper program that I could run model files through in order to perform the converting. I decided fairly quickly to focus on the engine as a whole, instead of very specific parts. This made it possible to complete the most features in the time allowed. Even after I turn this in, a lot of work still needs to be completed before it can be used to develop commercially viable video games.

Conclusion

Overall, I feel great with the amount of work that I put into the project and the state of completeness that I achieved given the difficulty and timeframe. I learned a lot about how game engines are developed and function, as well as much more about coding with C++. It seemed that there was never a right way to do anything, only a way that the developer felt most comfortable with, and that mindset was both nerve-racking and a creative reprieve. Even though I am turning this project in, I am definitely not finished. There is so much more to do that I just didn’t have the time to implement. After developing the WolfTail™ game engine for so long, I have come to the realization that it is a development black hole. There are always things to add and improve, and it will never be truly finished. Game engines are like fractals: they are just as complex the deeper you delve into them. There were multiple time while working on the project that I started to second guess my abilities and the decision to develop an entire game engine for my senior project. The shear amount of functionality required to have a minimum working engine was large, and most designing had to be completed with limited knowledge. It was a great test of my abilities as a developer and I am glad that I pushed through. Looking back, I do not believe that I would have done anything different. I feel like I spent my time well, and I am happy with the result.