AGE Engine - Design Document

Jayden Nikifork

Project Overview

As a high-level overview, I shall describe how a client programmer would typically use my implementation of the AGE Engine. Firstly, they must create a game. A game owns four objects: the input manager, the output manager, the scene manager, and the game clock. The input and output managers work fairly similarly. The input manager scans for input from the user then sends them to the logic components. The output manager is sent data from the logic components, then processes the data and outputs it. Input controllers and outputs views must be attached to their respective managers by the client. These are what control input and output. The default concrete input controllers and output views that the client will likely use are the keyboard controller and the terminal view (they function as you'd expect). By this point the user is ready to begin actual game development. They are likely to begin by creating a scene and a series of game objects. Scenes store and control the updates of a game's state, and are contained within the mentioned scene manager. Game objects are the different entities within a game like players and enemies. Game objects are abstract, and are meant for the client to inherit from. The client then has full creative freedom to implement various game interactions however they wish. To do so, they will utilise the default components and handlers, or create their own. Components are used to encapsulate different parts of a game object's state (how they look for example), and handlers control how a game object reacts to different game states.With of these building blocks, a plethora of games can be developed with this engine using fairly reasonable client-side code.

Updated UML

https://drive.google.com/file/d/1eRmh9_VVjucdPjPLT2CcpFdZL7zfhung/view?usp=sharing

The UML was too big to fit on the page. Please utilise the above link. If the image provided by the link is no good, please see the UML pdf in the project submission.

Design

Design Philosophy

My main design philosophy for my AGE Engine was modularity and extensibility. The idea was that the client could easily create their own components, detectors, and handlers given a little bit of knowledge of the inner workings of the engine. For example, if a client wanted to create animations for a game object, rather than doing so directly within the game object, they could abstract away animations into an animation component subclass, and simply have their game object inherit from the animation class. The same is true for event detection. More information on what some of these terms mean is provided in the subsections below.

Scenes and Game Objects

A scene is an abstraction that holds a game state–think of it like a level in a video game. The scene manager is used to hold a container of scenes, and determine which one is active (which one the user is to interact with) at a given moment. A scene manager has only one active scene, which is the only one that is updated on each frame. Scenes hold a couple different containers of objects. The first is game objects. Game objects are the different entities that react within a game, for example, the player, enemies, object spawners, visual effects, etc. In my implementation, the game object class is abstract, and it is left up to the client to create their own concrete game object classes. The design choice leaves an open-ended way for the client to easily express their creativity. Plus, implementing the concrete classes is pretty simple. Each game object has a position (called a transform), a virtual initialization method, and a virtual update method. The initialization method is called when an object is spawned into a scene (via the scene's spawn method), and the update method is called on each clock tick, and each is up to the client to implement. Game objects have what are called components and handlers, or at least that's how it may feel when interacting with them as a client programmer. In actuality, game objects inherit from these classes in order to gain their functionality. Handlers will be mentioned in more detail later. Components are classes that inherit from the abstract component class, and are used to abstract away a game object's state. The two that currently exist are transform (as mentioned previously), and sprite. Sprite is a very lightweight abstraction of how the game object looks. As a side note, only game objects that inherit from sprites are sent to the output to be displayed. The amount of encapsulation of a component is left up to its implementation. Other than game objects, scenes hold a container of detectors, as mentioned below.

Detectors and Handlers

Detectors are an abstraction used to detect events within the game state. Each detector is given access to the scene that contains it, meaning it has knowledge of the entire game state. The default detectors are input and collision detectors. When a detector detects an event, it sends a detected object to what are called handlers. Handlers are abstractions that cause game objects to react to detected events. Like components, game objects must inherit from handlers in order to utilise them. Handlers each have a virtual method to be implemented by the game object. This method is called by a handler's respective detector when it detects an event. For example, if an input controller sends an input, the input detector will detect this action, send an input detected object to the input handler, which will then call the virtual method.

Data Flow

Within the engine, data flows in the following manner:

Model → Controller → Model → View

A loop within the game clock (which is in the model) begins the flow. This loop has no restriction on its tick rate, and is responsible for obtaining data from the controller, and beginning each clock cycle of the game (I'll be referring to these as frames). Inputs are scanned for on each iteration of the loop, but each frame begins 0.05s after the previous one. The reason for this is so multiple inputs can be cast on a singular frame (the ncurses library is otherwise limited). I utilised the observer pattern to implement the data flow. The game clock notifies the scene manager, which notifies the active scene, which notifies the output manager. The only exception is the input manager, which is directly called upon by the game clock to scan for inputs. I would have liked to have reduced the coupling of these two classes, but this seemed like a decent option given the time constraints, and using the observer pattern would have likely required insight into the type of the subject (the clock). Upon each frame, the scene first runs the update methods on each of its game objects, runs the detectors, sends data to the output stream, then finally notifies the output manager. What I have not yet mentioned is my customised input and output streams. My AGE Engine has global input and output streams. I took inspiration from std::cin and std::cout for the global scope. The input stream functions as a typical stream, but the output stream is a little less stream-like. For example, the output stream does not mutate itself on operator>> calls, but rather must be cleared via its clear method. Plus, it has specific methods for getting and setting game statuses. Overall, the flow of data in my engine is fairly natural, but what is most important to note is that game objects update before detectors run. This was carefully thought out, and was chosen because it makes more sense for detectors to react to an updated game object state instead of one that was about to be overwritten by the update method.

Extra Credit Features

Scenes

As mentioned, scenes are an easy way to store reusable game states. The implementation of scenes makes having different levels or screens (i.e. gameplay or win/loss screens) within a game very easy. The tricky part about scenes was deciding how they should interact with detectors and game objects. I was unsure if each scene should have a container of detectors or if the scene manager should have a container of detectors. I decided on scenes each having their own detectors since some scenes may not require certain detectors, and it makes more sense with how detectors themselves are actually implemented. Furthermore, I wanted the scene manager to have functionality such as changing scenes, which could be utilised in the implementation of concrete game objects. I decided on a solution which was to create a method in the game object, which calls a method in the scene, which calls a method in the scene manager. It worked, but I was not so happy with this solution as it felt sort of messy, but in the end felt it was reasonable under the time constraint.

Modularity and Extensibility

Pretty much everything game-play related is modular and expandable in this engine. As mentioned, clients can create their own components, handlers, and detectors. What I did not yet mention is that they can also create their own input controllers and output views, in case they do not want to use a keyboard or terminal. Although it sounds like I'm putting the client to work if they want to use this engine, that is not the case, since the default concrete classes I have implemented work well, are straightforward to work with, and provide enough functionality to implement games with all the specifications of the project. Implementing these capabilities was difficult due to C++'s static typing. I did not want the client to have to utilise any templating when creating the games specifically (not creating handlers and detectors). This required some smart use of templating and polymorphism behind the scenes, but in the end, I came up with a solution that felt very natural to use as a client programmer, which was my goal.

Final Question

If I had the chance to start over, I would have begun some development before delving too deep into my first UML diagram. I found that I spent a lot of time creating my first UML, which did prove to be helpful during development, but I don’t think the time saved outweighed the time spent. For me, development is part of the brainstorming stage. Especially as someone who never really used C++ before CS 246E, I sometimes don’t immediately know which solutions are most feasible for a given problem unless I try implementing them first. With the UML, I did absolutely no development until I submitted it on due date 1, and I found creating it to be a challenge. For example, I expected to use the observer pattern everywhere throughout my implementation, but once I started development I felt like that was not the best solution. Now, I’m not saying I should have developed the whole engine before creating the UML. Rather just a little bit of each at a time in unison. I think with that strategy, I would have been able to work a little more swiftly, and have been able to spend more time cleaning up my solution, documenting code (which to be frank I did not do), and developing my games.

Conclusion

Overall, I believe my implementation of the AGE Engine is fully capable of developing games that include all functionality of the specification, alongside the possibility for the client programmer to extend the engine’s capabilities beyond the specification through its

modularity. This has been one of my favourite projects to work on and I am grateful for the opportunity. I would also like to thank Brad and the ISAs for making CS 246E a fantastic learning experience and an entertaining time.