Jayden Sansom, jksanso2

Homework 3: Game Object Model

Introduction

This assignment allowed for additional functionality to be added to the engine using the SFML and ZeroMQ libraries. The following sections explore the design decisions made in each of the assignment parts.

Game Object Model

The design of my game object model primarily stems from the Component model where I have a GameObject class from which all objects in the game are used within. The GameObject has an id string that is similar to the name that would usually be shown in a hierarchy of an engine, each with an individual name that corresponds to the object. Due to the use of the Component model, I created a has-a relationship where a GameObject will have a collisionable object attached to it. The Collider class is a virtual class from which the object types extend. This is because each of the current objects requires testing on whether or not a player or other object has collided with it. There exists no such objects that do not extend this behavior and therefore, no non-collider class was created as of this time. In the future, this will change to cover objects that only render but have no actions relating to collision within them. Ultimately I chose to utilize the Component model such that my engine could be built and easily extended upon in the future, following the structure laid out in many popular game engines.

There currently exist three files that contain the functionality of the required game objects and inherit from the Collider class, “Player.cpp”, “Platforms.cpp”, and “HiddenObjects.cpp”. The Platforms file contains both of the classes for the static and moving platforms, simply to condense them under their shared category. Similarly, the HiddenObjects file contains the SpawnPoint, DeathZone, and SideScrollArea classes but could easily be separated into different files if desired. These objects can be added to the scene within the main file and then a GameObject can be created and added to the list of models sent and received by the server.

There are also separate lists such as the spawnPoints, playerClients, and deathZoneBounds which each contain pointers to objects in relation to their namesake. The playerClients list will contain all references to the player characters and is dynamically added with each new connection and subsequent message from the server. The spawnPoints and deathZoneBounds lists are checked in the main function rather than the Player class itself as these lists can remain simple rather than convoluting how it is passed in and out of the player.

The sideScrollAreas are not created in a list and are instead checked for collision individually as they control the SFML’s View class as a camera which is created and updated in attachment to the window. This was implemented by creating bounds and when the player collides with them, it will move in the direction of movement by the player’s movement so it can move at the same speed. Currently, only two areas are created, on the left and right sides. In the game loop, when the player collides with it, the view and both bounds will move in the direction specified in the if-statement. If an upper or lower scroll area is needed, it can be added by initializing it before the update loop, checking for collision, and moving in the direction of the player's movement.

Ultimately, my game object model is extendable and will allow for future objects to be added as well as additional components to be given to the GameObject class. A user of my engine would be able to add objects by creating a reference to a new object of that type with the desired parameters, explained within the documentation. If they would like to enable collision on their object, they can set that value to true because of the Collider-inherited method. A GameObject should be created with a given object name and added to the objects list if they desire to share the object amongst clients rather than keep it locally. If the object should be drawn, then it should be inserted into the list of drawObjects. Even the hiddenObjects that shouldn't be drawn have colors set to them for debugging purposes and can be added to that list to test where they are. Lastly, the object should be added to any lists that are specific to their object such as the spawnPoints or deathZoneBounds such that they can be seen and used with other objects of that type.

I believe that my implementation of my Game Object Model is efficient, proven by its ability to run on limited-resourced machines. Also, it is easily extendable and user-friendly due to its use of the component model. Future work would include improving these facts, adding more objects, and decreasing the dependency between client and server files that are the same but not shared.

Multithreaded and Networked Scene

For this portion of the assignment, I had already completed most of the requirements in the previous assignment. My clients could connect to the server at arbitrary times and send data to each other via strings passed with a publisher-subscriber connection to send game object data and a requester-replier to send client information, creating a pseudo-pipe. This did not have to change on this iteration of the assignment as I only needed to handle sending new object information and handling client disconnection.

The disconnection, shown in Figures A.1 and A.2, was handled by adding a boolean value to my Client struct, representing whether the client was currently active or not. When the game window closes, the value will be changed to false and sent across the requester socket to the server. This would be read in and deleted from all object lists, sending that information to all other clients and removing them from their client lists as well.

Something that could be improved from my current implementation is my client-server reliance as not only are the object C++ files duplicated and independent of each other, but when initializing the scene, all objects are created within the client and server independently. If the ID strings are not matching or the parameters given to them are not matching, it can cause errors or not perform as intended. This may be fixed in a later iteration, but for now, running independently works and can be paralleled to a client download that would update to change with the server.

Appendix A

Multithreaded and Networked Scene

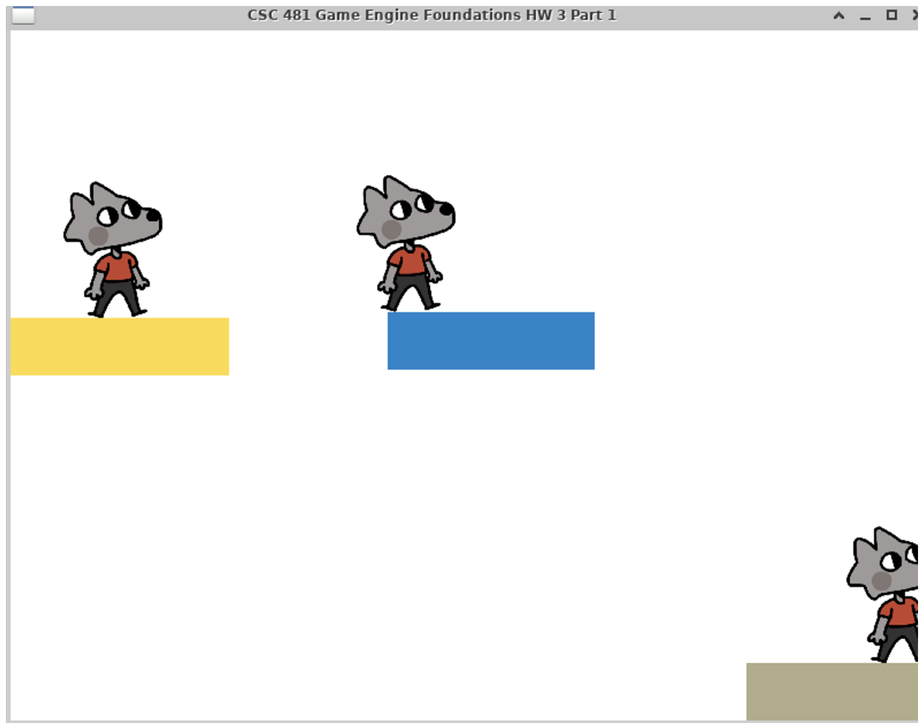

Figure B.1: Clients 1, 2, and 3 connected, receiving platform data and sending player data to and from the Server.

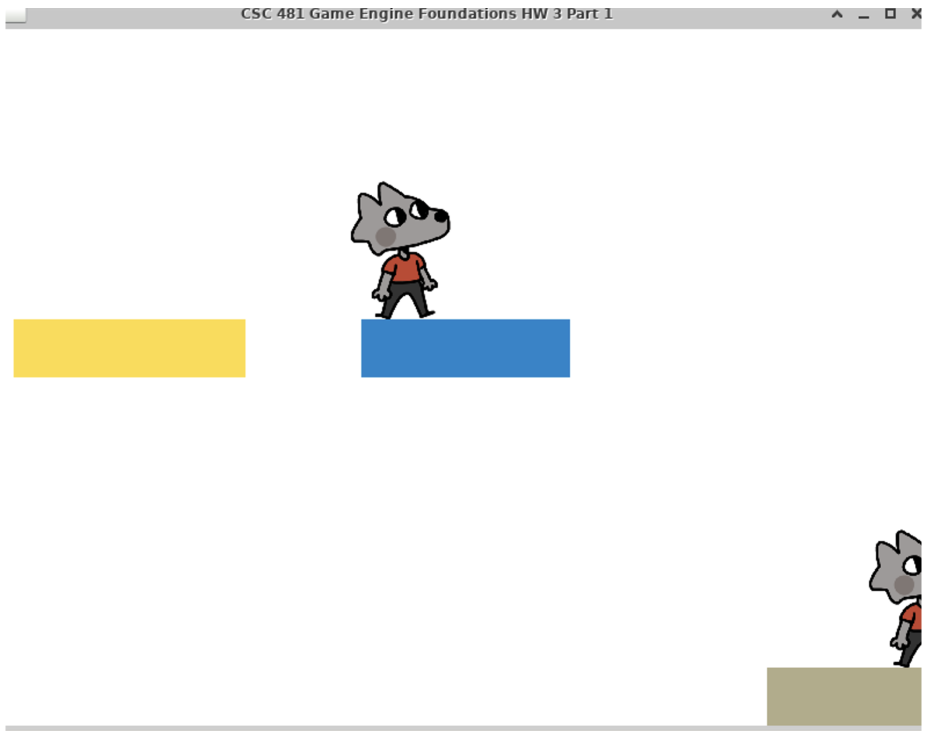

Figure B.2: Client 1 disconnected, no longer showing their player Sprite on other Clients.