Matthew Warren - CSC 481 HW3 Writeup

For my game model, I decided to stick with an object-centric design. There was a point as I was starting this project that I was thinking of switching to property centric, as my HW2 was a hybrid of both and could have pivoted to focus on either one. I liked the idea of having the positions of each object stored in a list in the client and server, as it would be easy to access and send around positions. However, as I thought about it more and messed around with implementation, object centric made the most sense as that was how my objects were already structured, and it would not be much harder to just have vectors/maps of pointers to objects around my client and server and access their fields through that. The first problem I had to tackle was sending entire objects across the server. I decided to have the server create the objects, and send their current state to the client whenever they connect. From that point onward, the client and server would only send around the needed information between each other (mainly positions).Both client and server would update their objects with the passed information. Doing it this way reduces overhead of sending unneeded information, and made me less concerned with how efficient sending entire objects had to be. In HW2 I was just sending space separated strings and parsing them for the positions, so since I had that code already I just extended it to work with whole objects. I gave each object a method to convert all its fields into a space separated string. Things like Vector2f's would just have its 2 values placed one after another, and since the string fields will be in the same order every time, the client will know how to parse the string. Here is an example of a sent representation of one of my platforms:

"PL 8 50.000000 50.000000 160.000000 535.000000 160.000000 535.000000 n/a"

The first item there is an identifier I gave to each of my object types. That way when a client receives one of these strings, it knows which object to construct. In this case, PL stands for a static platform. I created a file to hold these constant identifiers to reduce room for error when using them. The second item is a unique integer identifier for that object. This was necessary since I was only sending minimal information after this, I needed a way for the client and server to know which object it was talking about. Every message relating to a specific object after this will include that unique identifier in it. The next 2 are the x and y vector 2f fields for the size, then x and y for current position, x and y for starting position, and a path to the texture. (It knows how to handle n/a if there is not one). By creating all the game objects on the server, then just sending them out to clients to connect like this, each client is able to get exact copies of each object and I only have to create them once. This makes changing around the environment really easy as well, only having to update them on the server. The server keeps track of when objects move, so any client connecting after the first will get the current positions for all objects in these strings. The object model also allowed me to take advantage of polymorphism. All of my game objects extend from 1 base class. So both the client and server have a map that holds each game object id mapped to a pointer to that object in memory. Anytime I needed to do something to all objects (ex. A lateral translation due to a player reaching a side boundary). All I had to do was loop over that map and perform whatever operation. There are also a few vectors holding pointers to specific object types (ex. Moving platforms) that I looped over for functionality only

relating to those. I did not want to have to loop over the whole object map to only look for specific object types when trying to do something, as that would really slow things down as more objects entered the game.

After not using a timeline in part 4 of hw2, I re-implemented it into my client and server. One thing about my design is that the server is responsible for moving and publishing the moving platform positions, and the client is responsible for moving everything else. So the moving platforms and players had to be on different timeline objects. I kept allowing players to change their tic rate, but only had it affect that player. The moving platforms and other players stay on what they are currently using. It did not want players to be able to speed up and slow each other down. However, I did implement pausing to pause it for everyone. I did not see any other way that pausing would make sense in a multiplayer game.

Lateral translations for side scrolling was a bit of a challenge at first. I realized quickly that when the character hits the side border, it needs to move the objects at the same speed the player was moving or it would look weird. At first I was concerned this would be difficult with different tic rates of players, but I realized that I could just pass the frameDelta, which is what is used to determine how far to move things. The next challenge I had here was figuring out how to manage players moving off screen. I had my side boundaries set up with collision to not allow a player to pass them, moving the positions instead. I wanted to keep player movement independent, so they would not get pulled with a player moving to the side. However, then players would be dragged to the other edge of the screen and mess things up, as then they are colliding with the side boundary and still being dragged. To manage this, I decided to implement a way to keep all the players on the screen. If a player is at one end trying to move the screen, it checks the other one to check if a player is there and decide if it should allow movement. This made for a more enjoyable gameplay experience, as players were not getting lost on the screen and it forces players to work together. I think it will also benefit my design in the future, as my side boundaries now have a way to check what is colliding with them, which could open up more functionality in the future. To laterally translate objects, the message sent to the server looks something like this:

“TR 9 4”

Where TR is another keyword in my constants list that stands for Translate Right, 9 is the id of the object that sent it, and 4 is the frameDelta so the other objects know how much to move. I gave my base game object class a method called translate that takes in a framedelta, moving the object that much. On the moving platforms, I had to override it to also adjust where the starting and ending point of its movement are. Just being able to loop over my gameObjects list and call translate, doing different things for the objects solidified my decision is sticking with the object centric design, it works so well for things like this. This message is sent to the server, then the server publishes it out to all clients. The id of the player that caused it is needed so that player is not moved when translating everything.

Spawn Points were relatively simple to implement. Originally I was just going to make it a Vector2f variable, but the more I implemented I realized it needed to be a class. I added functionality for it to be laterally translated with the side scrolling as this was something I wanted in my game. It will drop new players in from the top in the middle of the screen, so mine needed to be able to be shifted. The spawn points are sent with all the other game objects when the server starts, but they do not extend the game object base class, as they are not something that needs collision or need any other properties, so they are sent after the game objects, and the client knows to read for them as well. They hold the spawn point, where it originally started, and their id.

Client disconnects were managed with a heartbeat implementation. Every 3 seconds, each client will send a message to the server. If the server goes 5 seconds without hearing from a client, then it will sent out to remove that player from the game. This was done in its own thread on the client and server, so it should not impact performance at all, and the waiting does not affect anything else.

Death zones were the last thing I implemented. I started with just having any player that collides with it go back to their spawn point, but realized in my game, since I want players to work together, I made it so everyone goes back to spawn and the level resets. This is why each object needs their original starting point as well. This decision also made it so I did not have to worry about having to move the spawn point to keep it on screen, and it potentially moved right over a death zone or some object that could interfere with it. Looking back, I realize that forcing the players to stay on the screen may have caused more problems than it was worth. If I were to do this again I would probably try giving each client a different view, with their player in the center but I have not thought too much about what that would entail. I think that it worked out ok for this part of the project. There is currently 1 death zone in my game, and I colored it red so the players would know what to try and avoid. I imagined they were like a spike or obstacle. I know the project description said these should be hidden objects, but I did not like the idea of hiding an obstacle in my game, and my code could easily be adjusted to not draw death zones if needed.