There are few things that I found challenging when I was working on this assignment:

- My implementation at first experiences serious latency, especially when there are more than one client connected to the server. It normally takes two seconds to see changes on the client #1 and about another two seconds on client #2. It eventually turns out that the this lagging issue is caused by an extra thread in server that constantly reads the list of clients, and then send locations of each character to other clients. Instead of listening to changes on each end as needed, this thread dumbly streams all the time even sometimes when those characters and platforms remain still. Initially I though it would be a good way to keep clients' location updated simultaneously so there will not be any frame or data loss, but on the contrary, due to relatively large data are being sent every frame, there rises frame loss. So I made modification that only when a key is pressed, this client's speed, rather than location, is being passed to server. That is to say, rather than sending out object's current location, the client tells server player's character speed and let calculation happen in each individual client. This change made huge improvement on performance. I was thinking this might have rough similarity on *event*. An event, rather than an updated object, is being sent to minimize the calculation needs to done in the main thread.

-Moving platform is unexpected. Due to there are only one character for each client, previously I merely include an instance of *movableObjet* that represents character in each client. To make characters to interact with platforms, I arranged platforms as static instances in server. However, the problem is, unlike static platform, there is no way for moving platform to be communicated. On the one hand, if moving platforms are set like static class in server to make other class easy to fetch, its location cannot be updated and therefore they will not render dynamically in each player's screen. On the other hand, if movable *sceneObjects* are composed in each player thread, similar to *character*, it will be drawn as movable but their locations cannot be synchronized from client to client. The worse thing is that doing this means each unique client has its own moving platforms being updated, sent, and rendered even though they seem like identical one on screen because they may overlap on each other. In order to fix this problem, I need to make connection with *Player* and *sceneObject* rather than simply use "has-a" relationship, and then let both server and client contain a list of drawable object. This ensures each rendering side gets a non-static instance of moving object.

-It was expected that the object will "fall off" the moving platform when it steps over the edge. In current implementation, I set an instance of movable object to indicate which platform this character is landing on. The nice thing is that collision detection between them become easy to deal with, the character does not bothering going through the whole list of platforms. However, it seems expensive and over-simple. I was wondering if there are more sophisticated way to "mark" one object on one another so they can more easily compare with each other. I though an id-based object data structure may have massive help when facing with a large-scale object list.

My design as, two interfaces *networking* and *renderable*. This *networking* is mainly used for objects that supposed to have capability to iterate over the list of clients and order them to send object through their sockets. In this assignment, this interface works with *networkingObject*, which serves for each *player*. *Renderable* is designed for objects including platforms and zones that can be drawn on *PApplet.* Since in a multi-client game, we expect to all the online players being reflected in each client's applet. This interface includes a *render()* method that accepts a list of renderables and a display applet as parameters .

Player, as the only controllable object in the whole system, is a special case of renderable. More than drawing texture on specified position, it contains collision detection. This time, instead of using a frame counter lock to control rising and falling of a jumping object, I found a simple State-Machine alike enum struct is handy. I have *Walking*, *Rising*, *Falling* and *FallingCollide* defined as object states. Player's character will initially set to be in *Walking* state and once a space key is triggered, this object's vertical velocity is changed along with its state set to be *Rising*. After it reach a certain fixed of distance or hitting on the bottom of someone else, it enters Falling state, in which its vertical velocity is flipped. Also, during that process, if its base overlaps with some other object, it will be considered to be landing on the platform. It will then be in *Walking* state which erasures the object can be moving horizontally like it can on the ground. It will be checked every frame if this object is reaching the edge of the platform. If so, it will be *Falling* again and so on until it reaches the ground.

Besides characters, there are components that considered as environment objects. They are monolithically *SceneComponentPoint*, *SceneComponentObject*, and *SceneComponentMovableObject*. *SceneComponentPoint* is used to represent a point in a scene, and it contains basic location information such as its position on x and y but does not require to implement fields like texture, width, and height that are mandatory in the following two. Static renderable object such as static platform and death zones are based on SceneComponentObject. It requires developers to specify initial locations, initial speed, height and width. Besides, it is also viable to directly pass a *PImage* as texture so this object's height and width will be automatically fill in. In the body, it has location information and rendering method. A more advanced class SceneComponentMovableObject controls a self-moving object. The default sets it to moving from left to right as its routine. This class inherits from SceneComponentObject but also additionally includes a method that constantly updates this object's location back and forth.

This assignment enables a simple two-dimensional action game like *Flappy Bird*. However, I think there could be more to make the game more playable. For example, players are easily getting bored in a fixed scene. Now that we have static object and non-static object, we can generate scene randomly for each player and change the setting of platforms periodically to make this game more unpredictable. Also, more modules can be extended. A bullet-firing trap can be extended from death zone and renderableMovableObject. It fires bullet from distance in short intervals. This trap should be regarded as movable because part of it needs to be draw dynamically also as death zone because once players get to touch the bullet, they will be sent to the spawn point. Additionally, since now every representation on scene can be generated in a non-static way, it is also doable to make them edible by players themselves. Similar to the idea of *Little Planets,* players get the chance to create their own stage. It will make the game more fun.