Section 1: I have refactored a lot of my code to follow a hybrid between a generic component model and a strict component model. A lot of the behaviors within the game objects have been shifted into components that can be added to the game object upon creation. The few behaviors I have kept within the game objects are for special behaviors such as respawning a player and generating server update information. The reason for this is because a component model gives me an easier way to introduce new behavior to objects.

For all game objects, I decided to keep the following data inside since I believe it is likely most game objects will have them: objectId (currently serving as an identifier of the object), speed (the speed of the object), xTravel (x velocity), yTravel (y velocity), untranslatedXY coordinates, xyTranslation coordinates, framePeriod (delta time), focused (window focus), canJump (indicator for whether object can jump), collisionToggle (whether the object can utilize collide component). Although a few of these data variables can be shifted into new components, I do not believe the game engine has a need for many objects that would not use these variables.

In the process of refactoring a lot of my game objects into components, I improved the collision handling significantly. Although there are still some hic-ups (like if you are colliding with two objects, you can get ejected downwards/upwards), the hic-ups are minor in comparison to the smooth collisions. In my components I have only the Animation component moving the model using values from xTravel and yTravel – all the other components update xTravel and yTravel to adjust the travel. This is one of the big reasons my collision system has improved. In my refactoring, I decided to create a component for player control – which allows for the player to have control of an object (be it platform, player, etc). I've also made a platform cycle component (which can be used by anything that wants to have a movement cycle) that makes the platform cycle a movement.

Movement of platforms can be customized with my constructors – allowing platforms to remain static, move horizontally, move vertically, or in both x-y directions. They can also

be customized to have different colors and sizes (more constructors to come for more efficient component add-ons). There are currently four constructors that enable customizations. The lengthiest one allows for customization of all its attributes, while the simplest one only spawns the platform at a default location (the simplest one is used primarily by clients to generate a new object).

When creating game objects with my engine, you will need to provide all objects four pointers: one for frame period, one for window focus, one for a list of all other interactable objects, and finally one for x y translation (to set up relative coordinates). Once all these pointers have been set up, all generated game objects have to be pushed onto the object list for thread updates (careful here though, the ALL zone objects should be pushed on AFTER the other objects, otherwise there will be strange collision behavior).

In order to improve multithreading for my game, I have separated the run function for my game objects into to functions – one for what I believe are thread-safe components, and another for what I believe are thread-unsafe components. If the components are accessing shared information from other objects, then in general those are what I would consider unsafe thread components. There are some components that have to execute after a different component (such as animation) so I push them onto their lists in a specific order and have run call them by the order they're in the list.

Section 2: I have not changed the core way I handled multi-threaded networking from the previous project. I did, however, do some bug fixing and moved the server update information generation from the thread into the game objects (a half-way to full serialization). I also made the server act as a sort of client too alongside the other clients. The game object will generate a JSON containing all information necessary for updating on the server and clients. Previously my client join handler thread had a bug where it releases the mutex too soon, allowing for a premature access to the objectList

by the main thread – which causes a read failure. This was fixed by encapsulating more parts of the thread by the mutex.

Unfortunately, I have not yet implemented a way to fully remove a client who has disconnected. Once a client disconnects, they will leave a ghost wherever they last were on the server (others can still collide with it). If the server ends, all other clients will be left hanging frozen.

Because of the way I implemented this from the previous project, my server can handle an arbitrary number of clients joining at any time – even before the server starts. The server, though still synced with clients, will begin to show some lag when the clients reach a high enough number (say around six or more). I intend to make the JSON information more compact to reduce the lag in the future.

Whenever a new object is introduced by the server, the client will generate a game object that will have a reduced set of components since the server is handling the positions of everything. The client will have their own player object with its components to check for collisions with other entities.

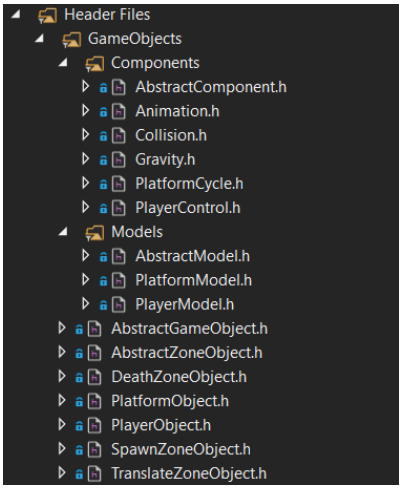
Section 3: I introduced a gravity component that adjusts the xTravel and yTravel of a game object to allow for objects to fall. There is a default velocity and maximum speed that an object can fall at defined within the component. I intend this component to be added to any game object to enable gravity for many objects.

For my death zone, spawn point, and side boundary, I created classes that all inherited from a abstract zone class that inherits from my platform object class. I named each of these “zones” of which the player can collide with and trigger a behavior. For now, I have given each of these zones a trigger function that will handle their behavior whenever called. In the collision component, I have a check for object type that will call the trigger function from the zones. Spawn will set a new spawn point and spawn translation for the player, Death will call the kill function on the player, while Translate (side boundary) will set a new translation coordinate for all objects. Currently I have

these zones only trigger upon collision with a player object. I intend to refactor some of these into event handlers for the next project.

Section 1:

Code refactored into this:



Animation component usable by any game object:

```
void Animation::run()
{
    model->getShape()->setPosition(untranslatedXY->x + xyTranslation->x + *xTravel,
        untranslatedXY->y + xyTranslation->y + *yTravel);
    untranslatedXY->x += *xTravel;
    untranslatedXY->y += *yTravel;
    //model->moveModel(*xTravel, *yTravel);
}
```

Heavily customizable constructors (a little loaded, but still):

```
    // Spawns a platform with no details
    PlatformObject(sf::Vector2f* xyTranslation);
    // The most simple constructor for the platform object
    PlatformObject(sf::Vector2f start, sf::Color color, long double* framePeriod, bool* focused, sf::Vector2f* xyTranslation,
        std::vector<AbstractGameObject*>* objectList);
    // Simple constructor for the platform object
    PlatformObject(sf::Vector2f start, sf::Color color, float orientation, long double* framePeriod, bool* focused, sf::Vector2f* xyTranslation,
        std::vector<AbstractGameObject*>* objectList);
    // Constructor for the platform object
    PlatformObject(sf::Vector2f start, sf::Color color, float orientation, sf::Vector2f end, float speed,
        long double* framePeriod, bool* focused, sf::Vector2f* xyTranslation, std::vector<AbstractGameObject*>* objectList);
    // Complicated constructor for the platform object
    PlatformObject(sf::Vector2f start, sf::Color color, float orientation, sf::Vector2f end, float speed,
        float length, float width, long double* framePeriod, bool* focused, sf::Vector2f* xyTranslation, std::vector<AbstractGameObject*>* objectList);
```

Multi-threading purpose run functions:

```
    // Runs all the components within the list
    virtual void runThreadSafeComp();
    // Runs all the components within the list
    virtual void runThreadUnsafeComp();
```

Section 2:

For ClientThread use in updating clients and server information:

```
    // Generates information about game object for server to pass to clients
    virtual json getUpdateInfo();
    // Retrieves information from server update and updates object
    virtual void serverUpdate(json updateInfo);
```

Section 3:

Gravity component usable by any game object:

```
/**
 * Runs the component
 */
void Gravity::run()
{
    // Not a perfect way of handling pause, but works well enough for now (will interrupt mid-jump)
    if (*framePeriod != 0) {
        if (*(this->yTravel) < MAXVELOCITY * (float)*framePeriod && !(*canJump))
        {
            *(this->yTravel) += gravity * (float)*framePeriod;
        }
        else if (*canJump) {
            *(this->yTravel) = MAXVELOCITY * (float)*framePeriod;
        }
    }
    else {
        *(this->yTravel) = 0;
    }
}
```

AbstractZoneObject header:

```
class AbstractZoneObject : public PlatformObject
{
public:
    // Spawns a platform with no details
    AbstractZoneObject(sf::Vector2f* xyTranslation);
    // The most simple constructor for the platform object
    AbstractZoneObject(sf::Vector2f start, sf::Color color, long double* framePeriod, bool* focused, sf::Vector2f* xyTranslation,
        std::vector<AbstractGameObject*>* objectList);
    // Simple constructor for the platform object
    AbstractZoneObject(sf::Vector2f start, sf::Color color, float orientation, long double* framePeriod, bool* focused, sf::Vector2f* xyTranslation,
        std::vector<AbstractGameObject*>* objectList);
    // Constructor for the platform object
    AbstractZoneObject(sf::Vector2f start, sf::Color color, float orientation, sf::Vector2f end, float speed,
        long double* framePeriod, bool* focused, sf::Vector2f* xyTranslation, std::vector<AbstractGameObject*>* objectList);
    // Complicated constructor for the platform object
    AbstractZoneObject(sf::Vector2f start, sf::Color color, float orientation, sf::Vector2f end, float speed,
        float length, float width, long double* framePeriod, bool* focused, sf::Vector2f* xyTranslation, std::vector<AbstractGameObject*>* objectList);
    virtual void trigger(AbstractGameObject* target) = 0;
};
```