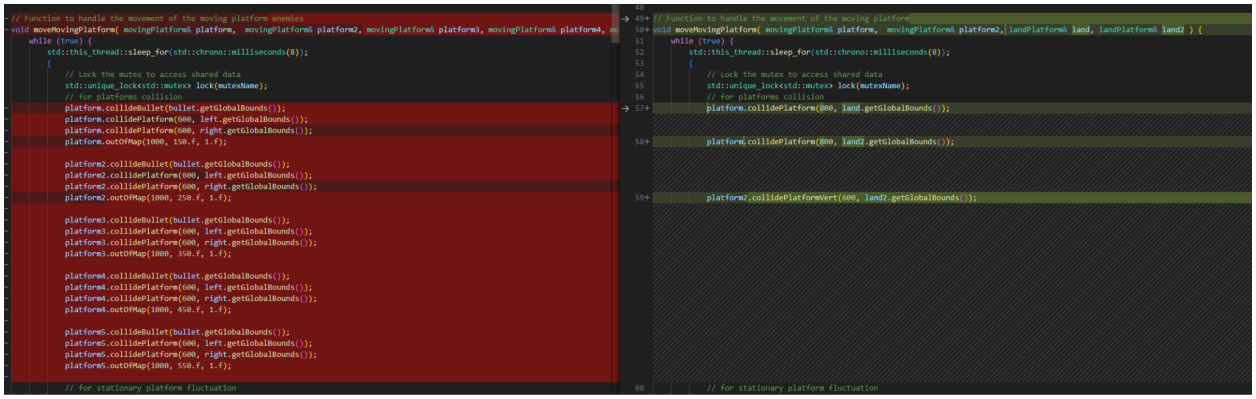
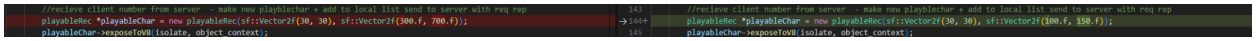
For this write up, I will start with the code diff, which are in 4 files, gameClient.cpp, movingPlatform.cpp, Platforms.h and playableRec.cpp
The diff in gameClient are as follow:



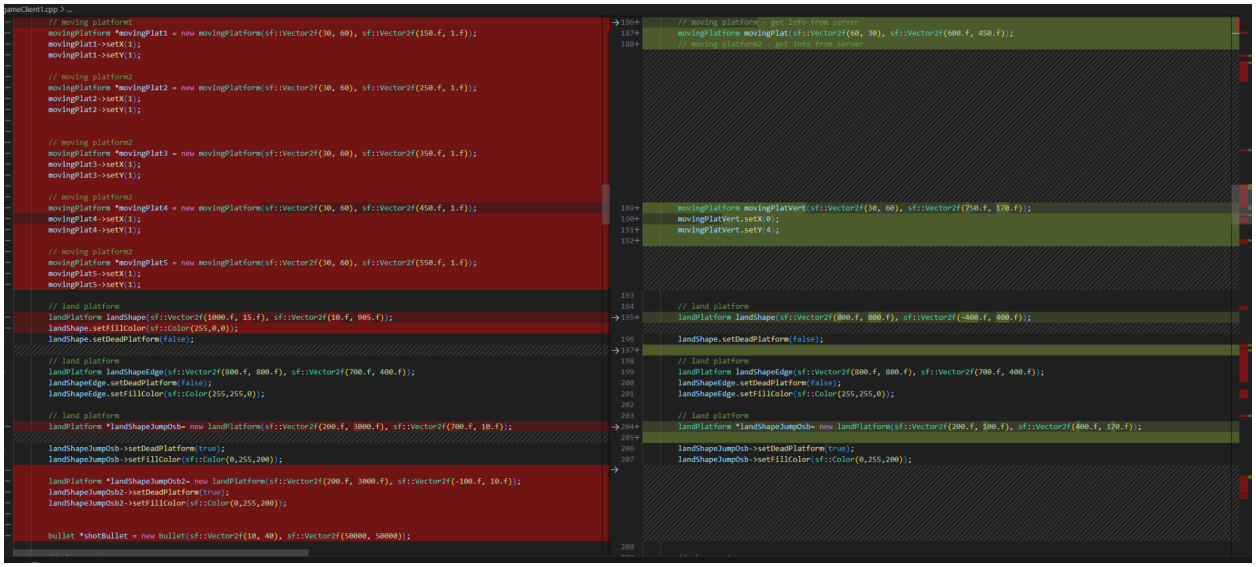
```
// Function to handle the movement of the moving platform enemies
void moveMovingPlatform( movingPlatform& platform,  movingPlatform& platform2, movingPlatform& platform3, movingPlatform& platform4, m
    while (true) {
        std::this_thread::sleep_for(std::chrono::milliseconds(8));
        {
            // Lock the mutex to access shared data
            std::unique_lock<std::mutex> lock(mutexName);
            // for platforms collision
            platform.collideBullet(bullet.getGlobalBounds());
            platform.collidePlatform(600, left.getGlobalBounds());
            platform.collidePlatform(600, right.getGlobalBounds());
            platform.outOfMap(1000, 150.f, 1.f);

            platform2.collideBullet(bullet.getGlobalBounds());
            platform2.collidePlatform(600, left.getGlobalBounds());
            platform2.collidePlatform(600, right.getGlobalBounds());
            platform2.outOfMap(1000, 250.f, 1.f);

            platform3.collideBullet(bullet.getGlobalBounds());
            platform3.collidePlatform(600, left.getGlobalBounds());
            platform3.collidePlatform(600, right.getGlobalBounds());
            platform3.outOfMap(1000, 350.f, 1.f);

            platform4.collideBullet(bullet.getGlobalBounds());
            platform4.collidePlatform(600, left.getGlobalBounds());
            platform4.collidePlatform(600, right.getGlobalBounds());
            platform4.outOfMap(1000, 450.f, 1.f);

            platform5.collideBullet(bullet.getGlobalBounds());
            platform5.collidePlatform(600, left.getGlobalBounds());
            platform5.collidePlatform(600, right.getGlobalBounds());
            platform5.outOfMap(1000, 550.f, 1.f);

            // for stationary platform fluctuation
```

```
// Function to handle the movement of the moving platform
void moveMovingPlatform( movingPlatform& platform,  movingPlatform& platform2, landPlatform& land, landPlatform& land2 ) {
    while (true) {
        std::this_thread::sleep_for(std::chrono::milliseconds(8));
        {
            // Lock the mutex to access shared data
            std::unique_lock<std::mutex> lock(mutexName);
            // for platforms collision
            platform.collidePlatform(800, land.getGlobalBounds());


            platform.collidePlatform(800, land2.getGlobalBounds());




            platform2.collidePlatformVert(600, land2.getGlobalBounds());

            // for stationary platform fluctuation
```



```
    //recieve client number from server  - make new playblechar + add to local list send to server with req rep
    playableRec *playableChar = new playableRec(sf::Vector2f(30, 30), sf::Vector2f(300.f, 700.f));
    playableChar->exposeToV8(isolate, object_context);
```

```
    //recieve client number from server  - make new playblechar + add to local list send to server with req rep
    playableRec *playableChar = new playableRec(sf::Vector2f(30, 30), sf::Vector2f(100.f, 150.f));
    playableChar->exposeToV8(isolate, object_context);
```

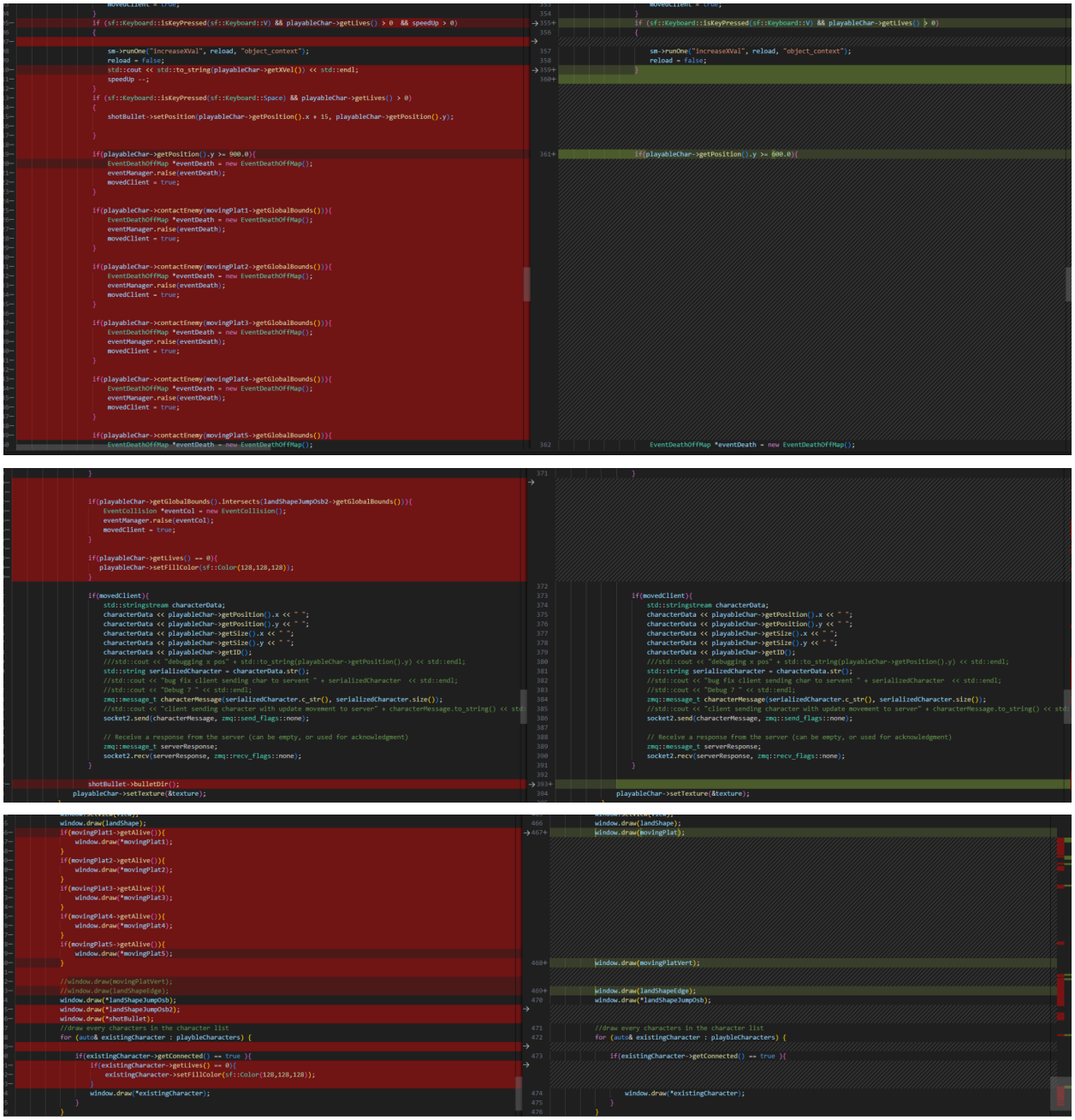


```
// moving platform1
movingPlatform *movingPlat1 = new movingPlatform(sf::Vector2f(30, 60), sf::Vector2f(150.f, 1.f));
movingPlat1->setX(1);
movingPlat1->setY(1);

// moving platform2
movingPlatform *movingPlat2 = new movingPlatform(sf::Vector2f(30, 60), sf::Vector2f(250.f, 1.f));
movingPlat2->setX(1);
movingPlat2->setY(1);


// moving platform2
movingPlatform *movingPlat3 = new movingPlatform(sf::Vector2f(30, 60), sf::Vector2f(350.f, 1.f));
movingPlat3->setX(1);
movingPlat3->setY(1);

// moving platform2
movingPlatform *movingPlat4 = new movingPlatform(sf::Vector2f(30, 60), sf::Vector2f(450.f, 1.f));
movingPlat4->setX(1);
movingPlat4->setY(1);

// moving platform2
movingPlatform *movingPlat5 = new movingPlatform(sf::Vector2f(30, 60), sf::Vector2f(550.f, 1.f));
movingPlat5->setX(1);
movingPlat5->setY(1);

// land platform
landPlatform landShape(sf::Vector2f(1000.f, 15.f), sf::Vector2f(10.f, 905.f));
landShape.setFillColor(sf::Color(255,0,0));
landShape.setDeadPlatform(false);

// land platform
landPlatform landShapeEdge(sf::Vector2f(800.f, 800.f), sf::Vector2f(700.f, 400.f));
landShapeEdge.setDeadPlatform(false);
landShapeEdge.setFillColor(sf::Color(255,255,0));

// land platform
landPlatform *landShapeJumpOsb= new landPlatform(sf::Vector2f(200.f, 3000.f), sf::Vector2f(700.f, 10.f));

landShapeJumpOsb->setDeadPlatform(true);
landShapeJumpOsb->setFillColor(sf::Color(0,255,200));

landPlatform *landShapeJumpOsb2= new landPlatform(sf::Vector2f(200.f, 3000.f), sf::Vector2f(-100.f, 10.f));
landShapeJumpOsb2->setDeadPlatform(true);
landShapeJumpOsb2->setFillColor(sf::Color(0,255,200));


bullet *shotBullet = new bullet(sf::Vector2f(10, 40), sf::Vector2f(50000, 50000));
```

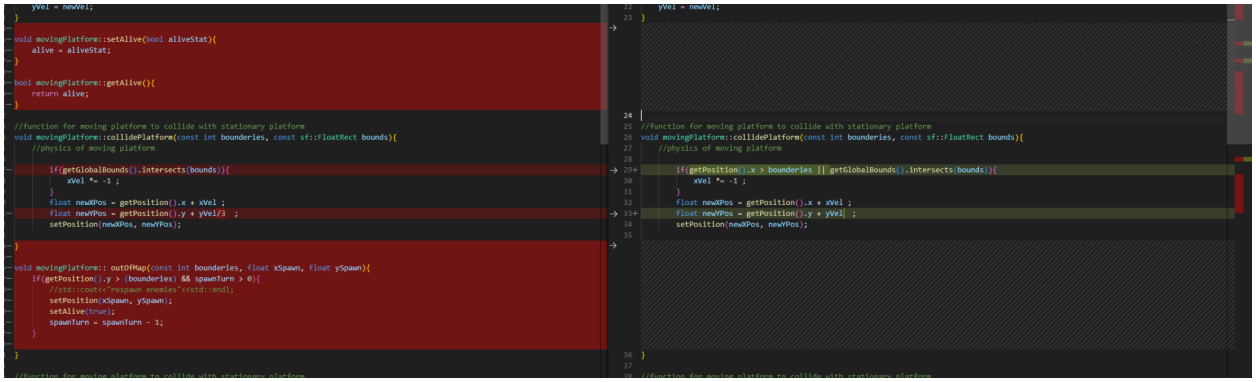
```
    // moving platform1 - get info from server
    movingPlatform movingPlat(sf::Vector2f(60, 30), sf::Vector2f(600.f, 450.f));
    // moving platform2 - get info from server
    movingPlatform movingPlatVert(sf::Vector2f(30, 60), sf::Vector2f(750.f, 170.f));
    movingPlatVert.setX(0);
    movingPlatVert.setY(4);


    // land platform
    landPlatform landShape(sf::Vector2f(800.f, 800.f), sf::Vector2f(-400.f, 800.f));
    landShape.setDeadPlatform(false);

    // land platform
    landPlatform landShapeEdge(sf::Vector2f(800.f, 800.f), sf::Vector2f(700.f, 400.f));
    landShapeEdge.setDeadPlatform(false);
    landShapeEdge.setFillColor(sf::Color(255,255,0));

    // land platform
    landPlatform *landShapeJumpOsb= new landPlatform(sf::Vector2f(200.f, 300.f), sf::Vector2f(400.f, 170.f));

    landShapeJumpOsb->setDeadPlatform(true);
    landShapeJumpOsb->setFillColor(sf::Color(0,255,200));
```

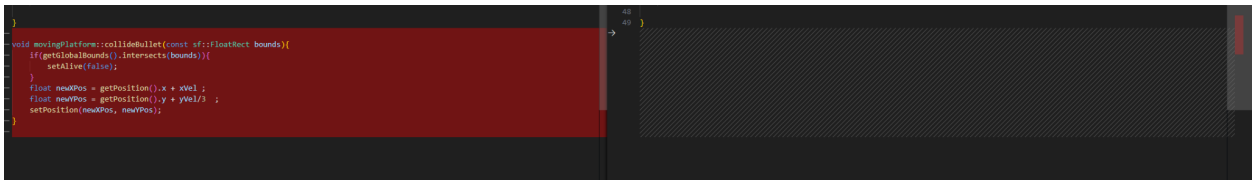
```
// Create a thread for moving platform logic
std::thread platformThread(moveMovingPlatform, std::ref(*movingPlat1),  std::ref(*movingPlat2),  std::ref(*movingPlat3),   std:
int64_t last_time = gameTime.getTime();
int i = 0;
int speedUp = 1;
while (window.isOpen()) {
//recieve when new playablerecs from server with req rep + don't wait flag
//loop through the current playbleCharacters vector, if same id isn't found
    //make new character with that information and push it back
```

```
    // Create a thread for moving platform logic
    std::thread platformThread(moveMovingPlatform, std::ref(movingPlat), std::ref(movingPlatVert), std::ref(landShape),std::ref(lan
    int64_t last_time = gameTime.getTime();
    int i = 0;

    while (window.isOpen()) {
    //recieve when new playablerecs from server with req rep + don't wait flag
    //loop through the current playbleCharacters vector, if same id isn't found
        //make new character with that information and push it back
```

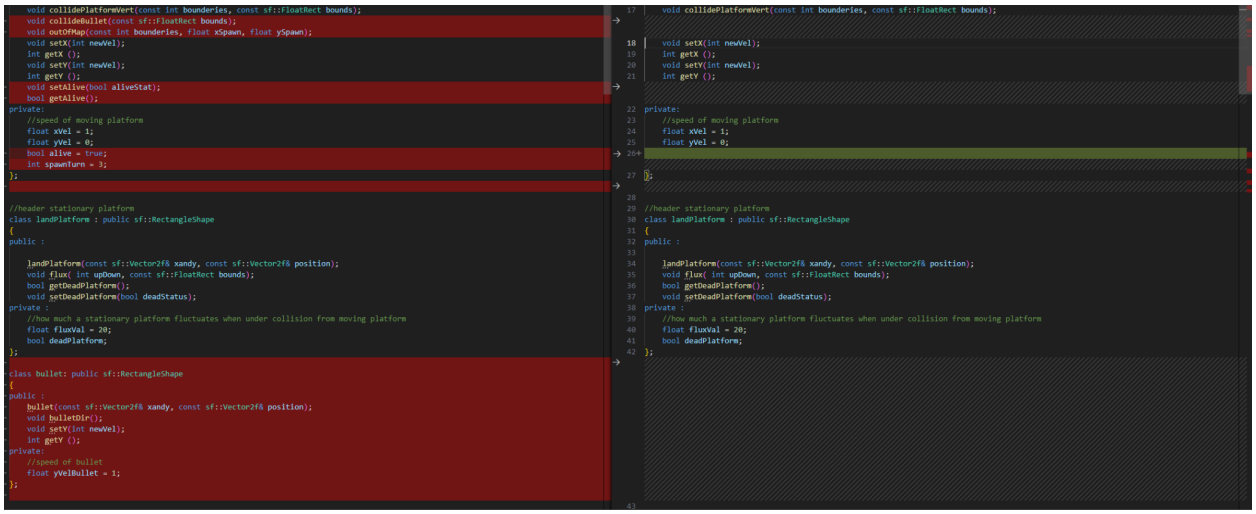


Here’s the diff in movingPlatform:

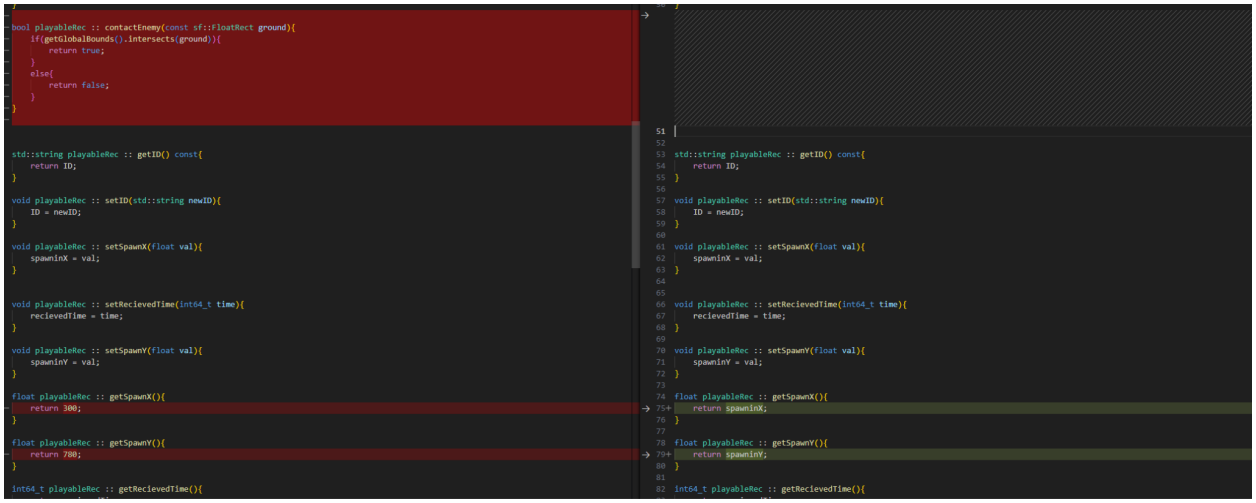


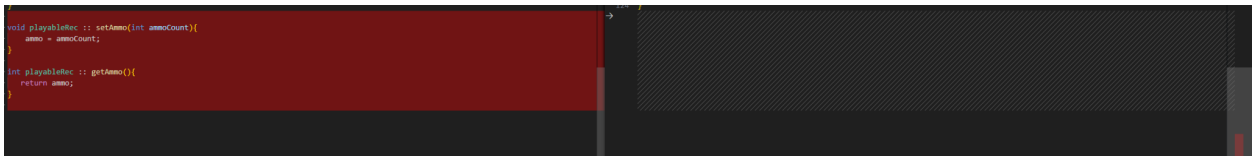


Here’s the diff in Platforms.h



Here's the diff in playableRec:





Next, I will write about how I handle scripting as well as how the I did the 2nd game
For scripting, I decided to script a new function for the playable Character that speeds them up by 30 units (they go from 3000 to 3030), to do this, I had to first edit my make file to be able to run V8, I also got the scriptManager as instructed on the example code, as well as the code for v8helper.cc. In this part, I edited my playeableRec class to expose it to V8 , and then making a version of get and set xspeed to be compatible with V8

```
v8::Local<v8::Object> playableRec::exposeToV8(v8::Isolate *isolate, v8::Local<v8::Context> &context, std::string context_name)
{
    std::vector<v8helpers::ParamContainer<v8::AccessorGetterCallback, v8::AccessorSetterCallback>> v;
    v.push_back(v8helpers::ParamContainer<v8::AccessorGetterCallback, v8::AccessorSetterCallback>("xSpeed", getXSpeed, setXSpeed));
    return v8helpers::exposeToV8("playableRec0", this, v, isolate, context, context_name);
}


void playableRec::getXSpeed(v8::Local<v8::String> property, const v8::PropertyCallbackInfo<v8::Value>& info)
{
    v8::Local<v8::Object> self = info.Holder();
    v8::Local<v8::External> wrap = v8::Local<v8::External>::Cast(self->GetInternalField(0));
    void* ptr = wrap->Value();
    int x_val = static_cast<playableRec*>(ptr)-> xCharVel;
    info.GetReturnValue().Set(x_val);
}



void playableRec::setXSpeed(v8::Local<v8::String> property, v8::Local<v8::Value> value, const v8::PropertyCallbackInfo<void>& info)
{
    v8::Local<v8::Object> self = info.Holder();
    v8::Local<v8::External> wrap = v8::Local<v8::External>::Cast(self->GetInternalField(0));
```

I decided to op the context like how they did it in the example as well as initializing v8 the same way, (I did this in my client) . As for the script itself, I used a function called change speed to add 30 to the xspeed of the player and then call that function is the script.

```
JS changeSpeed.js > ...
1   // This script moves the GameObject with handle "gameobject0" (it's guid)
2   function changeSpeed(x) {
3       var tx = playableRec0.xSpeed + x;
4       playableRec0.xSpeed = tx ;
5   }
6   changeSpeed(30);
7
```

As for the 2nd game, Firstly, I added a few fields and functions in the movingPlatform class, this includes a collideBullet function (this takes in the bounds of a bullet to detect collision, I made a bullet class, I will elaborate on this further later in this write up), this function detects if the movingPlatform objects comes in contact with a bullet and then set it's alive status (this is a new bool alive field I added)to false if it gets it, then the main game loop, when drawing objects, will check if this movingPlatform is alive or not. A second function I added is the outOfMap function, which takes in an int boundary and 2 floats as spawn in coordinates, how this function works is that when a movingPlatform function moves past a certain y value, it will decrease the spawnTurn field by 1 (a new int private field I added in this class, original it is set to 3). The idea is that I will spawn 5 vertical moving platform moving vertically downwards, from up to down, these are the enemies, when the player shoot at them, they die and disappear, if they move pass a 1000 in y coordinate, they will respawn back at the top as the next wave of enemy.

Next, let's talk about the player character and the bullet, as for the character, I added a function called contactEnemy that takes in the bounds of a rectangle object, this function returns a bool true or false, in the main game loop, when a player comes in contact with one of the enemy rectangles moving downwards, a death event from the previous project will be called, decreasing the lives field of the player by 1 every time this happen. As for the bullet, it is a new class that extends off of a rectangle and function close to how a the previously mentioned moving platform functions, it is spawn in at an arbitrary location, upon pressing space, it will be spawned in front of the player character and then be shot upwards, if it comes to contact with an enemy moving platform, that platform will die for it's specific wave. Furthermore, I also implemented a check for speed up script, I added an integer check with value of 1, upon the first time the player presses v to speed up, this value will decrease to 0, as I intent for the player to only have 1 speed up per game.I also made the player turns grey when they die. One more thing is the spawn point as well as the "side bar" I decided to add two side bars on each vertical edge of the screen, when a player bump into these bars, they will be resetted into sort of the middle of the screen (I changed the spawn point of the player to fit this style of game better). Also, I previously made it so that when a player drops below the 600 Y coordinates, they will lose a live, I redid to so the "bar" will be at 900 as well as I made a stationary platform at 905 to indicate that this they will lose a life at that exact point.

As for multiplayer, I use my server and client to show each players how the other player/client is doing in their game and weather they are dead or not. In the future, I want to include a points field in each playableChar objects, this points field will be printed by each clients, keeping track of this will be the same implementation as other fields.