Jeremiah Knizley

Homework three writeup

Dr. Card

October 29, 2023

Part One:

For part one, my goal was to be able to create a design that made it easy to make new objects and add them to my game. To accomplish this, I decided I would refactor what I currently had, and then test how easy it was to add new things by doing just that with the remaining requirements (spawn points, death zones, side boundaries). On a simple level, my design looks like this: A1.

Breaking it down one at a time, all of my objects inherit from GameObject. GameObject is an abstract class that is supposed to be extensible enough that almost any object you could think of can be extended from it. Because if objects have common behavior, it's much easier to write code for them. So, for each GameObject, there are three essential fields: Drawable, Collidable, and Stationary. Drawable means that (in some form or another), the object inherits from sf::Drawable. This way your RenderWindow (Or GameWindow in my case), knows that it can safely cast the GameObject to an sf::Drawable (A2). Collidable is similar; it means that your object inherits from either sf::Sprite or sf::Shape. That way, when checking for collisions, you can safely cast to one of those. Stationary is a bit different and not entirely necessary. Stationary (Or static) refers to objects that don't move, or at the very least, aren't a part of multiplayer environments. These are objects that need to be updated every time the window draws.

These fields are primarily used to make life easier and so that you have a general idea what the object is for. On the other hand, you can ignore these fields and make your object something else. For example, Events will likely be a GameObject, but will not be collidable, static, or drawable.

More importantly are the abstract functions that each GameObject must implement. I'll cover them in order and explain why they are important. First is toString. I got the idea when I was experimenting with getting JSON to work. Part of the process is implementing the two methods "to_json" and "from_json". I took this idea (partly because I couldn't json to work), and decided that it would be a very good thing to implement into my GameObject, particularly for non-static objects that get sent over the network. Essentially, when you want to transfer your GameObject over the network, you use your toString function to get the essential elements of your object and transfer it over. I say essential, and I mean that. Because each object must inherit this abstract function, they can define whatever toString method they like to suit their needs. For example, I have no need to send my character's color or texture because those don't change and are set in the constructor. But later on, if I start changing the color or texture of my character mid-game, I can add it into the toString method easily. The same applies to base GameObject fields like "drawable". Characters are always drawable, so I have no need to send that in my string.

toString pairs with constructSelf, which is a method that converts a string (Probably from toString) into a new object of that type. The use case for this is simple; you need to send your Character, or any other game object, over the server. You convert it to a string, send it through, and the server distributes the string to each client. Then those clients take the string, call constructSelf, and display the created object (A3). When you want new things to be added to the

toString function, it's very easy to update both. You just tack on whatever else you need to the end of your string and update constructSelf to match.

The next abstract function is makeTemplate. This, I'll admit, is not really needed. All it does is create a blank object and return a pointer to it. Essentially it does the same as the new keyword. This function is mostly there to make it clear that that object is not actually in use. More on that later.

The last function is getType. If you need a more specific understanding of what object you are working with, you can use getType to determine what type your object is. In my implementation, each class has a static const integer field labeled "objectType", with a unique integer ID representing the class. Essentially, each GameObject has two modes of identification; the drawable/collidable/static fields, and the objectType. Both of these have very common use cases. In your GameWindow, all you care about is whether or not something is collidable or drawable, so that you can draw it or check bounds as needed. But, when actually handling those collisions, it can be very helpful to know what type of object you are working with. For example, when colliding with a SideBound, you don't want to stop movement, but instead pass through it and change the view.

All of these combined make it extremely easy to use pointers to abstract your objects into GameObjects, and then convert them back into what you need when you do something with them (A4). Now, your GameWindow no longer needs to know every type of object that is going to be added to it, and make lists for each type. They can all just be GameObjects, and it'll figure out the other details it needs later using the common functions.

Now for some lightning rounds for each object. Platforms (at the moment) are literally just rectangle shapes. They have one field (which I don't use yet), but other than that they are no

different. They mainly serve as boundaries for the player. However, unlike my character, I do need to send color over the network, so my toString and constructSelf methods contain color, position, and size.

MovingPlatforms are slightly more interesting. They differ from platforms in that they are nonstatic GameObjects, meaning they get sent to the server, and need to be updated using toString and constructSelf. They also have boundaries, which the server uses to determine their pattern of movement. Additionally they have speed and the type of movement (vertical or horizontal). When sending these over the server, they contain everything that Platform contains plus speed.

Characters are nonstatic, drawable, and not collidable. This might seem strange since characters very much do collide with platforms and such. But that is not what collidable means in this context. Here, collidable means “Do they collide with other Character objects”. And in my implementation, they do not; characters pass right through each other as if they weren’t there. Characters have some notable features such as a texture, but not much else. And, since they all have the same texture (for now), there is no need to send information about the texture or color for that matter over the network. Hence my toString and constructSelf methods only contain position and size. Though, again, this can be updated at any time very easily. Perhaps an enumeration could be used to determine what texture to give the character. Characters also include an additional field “Connecting” which helps with detecting disconnects on the server.

DeathBounds are rectangleShapes, and are basically the exact same thing as platforms. The only reason that they are a different class here is to tell the program “You died if you touched this.” Though, it is worth noting that they are not drawable, unlike platforms. Death

Bounds only send information about their position and size, since they are not drawn color is not needed. Having said that, these are client-side objects and are not sent over the network at all.

SpawnPoints are one of the more interesting objects along with Side Bounds. Spawn Points definitely could have been redone and not been made sf::RectangleShapes. The only information they would have contained is their position. However, in the future, these objects could be very easily turned into check points. Basically when you reach a spawn point, the character's spawn point would be updated to the one it reached, much like the flags in New Super Mario bros. And to do that, collision is necessary, which is why they are RectangleShapes. It's also worth noting that Characters have a composition relationship with SpawnPoints. Characters contain a respawn method that sets their position to their spawnpoint's position (which is a SpawnPoint field in Character). Then, when the player hits a Death Bound, the handler will call respawn() (A5). SpawnPoints are not sent over the network, and are not included in Character's toString method. In fact, they aren't even put into my GameWindow. After all, they already reside in my Character, and are only used by the client. Having said that, they have the same toString method as you'd expect any other RectangleShape to have; size and position.

Side Bounds are another interesting object. They are RectangleShapes that are collidable, static, and not drawable. They contain two additional fields: A pointer to the window, and an sf::View. The view stored in the object is the view that the window will change to when the character collides with the object. OnCollision() should be called when that happens, which uses its window field to call window->setView(). While this isn't needed, it makes your collision handling code much cleaner, and realistically you aren't going to be swapping your Side Bounds between windows or anything crazy like that. As for implementing them, I recommend a

somewhat thin window-height invisible wall that the player will hit when it gets near the end of the screen. You'll likely need to use two of them for any one boundary; one for changing to the next scene and one for reverting back to the original. As a side note it's very important that the character can't hit both of them at the same time, or the view won't change (or something MUCH worse could happen).

To demonstrate the capabilities of this GameObject model, I refactored my GameWindow class. It is now fully engaged in abstraction, using only GameObject pointers (and some help from SFML, it is an sf::RenderWindow after all). When you want to add a new object to your GameWindow, you only have to do one or two things. First, add the object using "addGameObject". To improve runtime, the objects are stored into different lists based on their properties (collidable, drawable, static); that way when updating visuals, we don't loop through the entire list of objects, but just the collidable ones. If it is a static object, you are done! If not, you need to add a template too. As mentioned before, templates are blank objects treated as GameObject pointers, and the only reason GameWindow needs them is so it can access abstract methods from the nonstatic classes. For example, the only nonstatic objects I have are Moving Platforms. The client receives these moving platforms in the form of a big, comma-separated string. It gives this string to GameWindow, which separates it into individual Moving Platforms. But, GameWindow doesn't know which class's toString method to use. That's why the first identifier in each object's toString is the objectType. Using this, and a map of templates (sorted by objectType), the GameWindow finds the template for MovingPlatform, and uses its constructSelf method to make the object from the string (A6). It does all of this without even knowing that MovingPlatforms exist. In fact, the only object types GameWindow knows about

are sf::Drawables, sf::Sprites (For the character), and sf::Shape (for collidables), and GameObjects of course. This functionality is what makes it so incredibly easy to add new custom objects to the game environment.

Other than in gameWindow, the only other thing you need to do to add your new object is to create the logic of collision HANDLING (GameWindow already does detection automatically). Collision handling can be a method (Like onCollision in SideBound), or it can just be a sequence of operations that happen when you collide with that object somewhere in a thread. Likely in the next homework it will be using Events, which is fine because Events will be able to be transferred to and from the server as well using GameObject methods like toString and the like.

In conclusion, the game object model I provided has proven to be very useful. I was able to get side boundaries, death bounds, and spawn points to work almost immediately upon making them using my completed design. Making MovingPlatforms and sending them over the network was simple, as toString and constructSelf provide a uniform and reconstructable mode of serialization. No use in sending the entire object when all I need is position and size.

Part Two:

First, I'll mention that I did part two before part one. Or at least most of it. That is because after looking back on how it operated for homework two, I realized something important. It sucked. So I went back to the drawing board and came up with a design that was better.

I created a goal for my design in order to help me plan out what I needed to do. The following is the design that I came up with after thinking about various execution orders that the server and client could take.

Goal:

For our game, server updates will inevitably cause collision. Therefore, collision must be handled before those updates are displayed. Also, as much time as possible needs to be between when the client sends its information and when the server publishes. That way we can be more sure we have all the info when we receive from the subscriber.

Client:

Thread one:

Process characters -> update visuals -> Process other things -> collisions-> reqrep->

Main:

Event loop -> keyboard input

Server:

Main: Check for new users

Repthread: 0…n threads that process unique client updates.

Pubthread: Publish -> move platforms

The reasoning for this design is as follows. For the logic behind the client, in order to prevent the window from displaying collisions, I split the updates of characters and the updates

of platforms. This is because of the case when the update causes a collision. The collision hasn't been handled yet, so we will only display the character pre-collision, and wait to update the platforms until the next cycle, after the collision handling has been done. That way colliding objects are never displayed.

Second, reqrep happens at the very end of the tic. The reasoning behind this is that, assuming that the client beats the tic rate by even a little bit, this will provide the most time possible between when the client sends its information, and when the server publishes. With this, we give the client its best chance at receiving updated information on the next tic.

On the server side, all main does is wait for new clients. When it receives one, it assigns it a unique port and ID. Then it creates a repthread to receive messages from the client at arbitrary times. Meanwhile, the client receives its new port and connects to it and begins sending and receiving with the repthread. At the same time (Though avoiding critical sections), the pubthread is publishing information about characters and moving platforms on a set tic rate.

With that, the first requirement is satisfied; because it has its own thread and socket attending to it, the client can send messages and immediately the server will begin updating and responding to them. As a side note, "arbitrary" is a bit vague here. Technically you are limited by the number of ports you can use. However, I've been able to use more than ten with no problems.

Input is easily handled by the client sending its updates every tic. Now, keep in mind, due to the constraints on SFML, input handling happens in a separate thread, and isn't synchronized because it's sort of impossible to synchronize with something like "When will the player push the button?". So it's possible that keyinputs will be delayed until the next tic, but unlikely because reqrep happens at the very end of the tic.

The scene is drawn partially due to the static objects on the client side, and partially due to the nonstatic objects on the server side. Every tic, the server sends its MovingPlatforms and characters using toString, as I discussed in part one. It does this by separating each string by a comma which the client then parses to it with constructSelf. As a side note, in order to increase performance, all of the characters are placed first in the string, so that they are easier and quicker to separate from the platforms when the visuals are updated.

My runtime object model is included in both server and the client, though my server only utilizes Characters and Moving Platforms. There's no need to send four sets of Death Bounds when they are all the same after all.

Client disconnects are a little more interesting. When I submitted homework two, I had a rudimentary form of client disconnects, but it wasn't up to par. This time I have fixed it so that the right behavior should happen no matter the cause. First, if the cause is the player closing the window, included in the toString() call for a Character is a field called "isConnecting". When the window is closed, one last message is sent with this field set to false, which tells the server to erase the character from the map. Once that is done, the server is no longer sending that character's information to the clients. And, because the GameWindow update cycle clears all of the non static objects every iteration, the disconnected character is not drawn or even loaded into the window anymore. No tricks involving making the character invisible. It's just gone.

But that is for intentional disconnects. For unintentional disconnects, something different happens. Unintentional disconnects can be from a number of reasons. In our case though, it's probably due to the client being forcefully closed by the terminal. Though this should work for a network error too. To accomplish this, repthread uses a non blocking receive loop, and counts the number of tics that goes by. When it receives a new message, it resets the counter. After 100 tics

of not receiving a message from the client, it erases the client's information from the map just like if it had disconnected normally (A7). This can be set to anything by the way, depending on whether or not you want your client to be able to reconnect after a longer delay. For example in Counter Strike, you have somewhere between 20-30 seconds before you are disconnected from the server.

Conclusion:

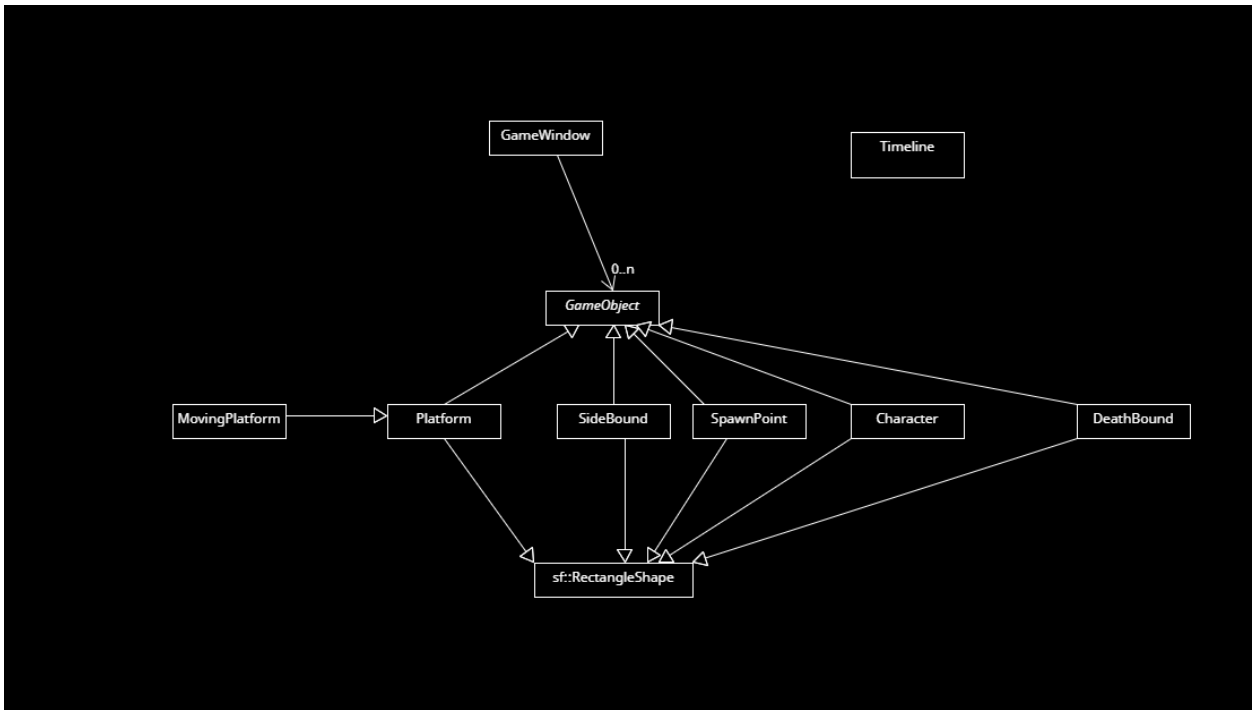
After changing my design of both my object model and my networking, my performance seemed to drastically increase. Before, my program typically struggled to meet an 8 ms tic or even 16 tic requirement. Now it can seemingly do any tic rate and still function properly (though it'll still look bad on anything 32 or higher). I can even go as low as 2 tic and it will easily keep up with it. Not only that, but the game looks and feels more consistent. I changed how my collision works, and gave the game a more structured order of operations, centered around my runtime object model and my networking setup. Using them I limited the amount of information being communicated to its smallest possible degree to improve efficiency while still retaining necessary details. All-in-all, I think after implementing my two designs, my program is much more extensible and more efficient than it used to be. After all, implementing the new objects took a mere fraction of the time that implementing previous objects would take. As a matter of fact, after creating the object, DeathZone took a mere six lines across my client to implement. And that number of lines would have been the exact same even if I was sending it to the server. If you're curious, it took two lines to add it to the window, three to handle collisions, and one to add the template to the window (Which actually isn't necessary for a static object). I consider that to be quite impressive considering it's an object that my window doesn't know exists.

Known Bugs/other concerns

1. When running the program on a computer that is very bad (my laptop), inconsistent behavior results. This is largely due to the fact that the tic rate fluctuates constantly because it only has a “Do not exceed” limit and nothing more. Implementing a moving average for this should alleviate this. Having said that, I have noticed that some machines just seem to run this kind of thing better. Nvidia in particular seems to have issues.
2. If this was on a real network, the port selection process would certainly have to be more thought out than “Increase the port number by one for each client”.
3. Nothing else! I genuinely think this is a big step up from homework two.

Appendix

A1:



A2:

```
void GameWindow::update() {
    std::lock_guard<std::mutex> lock(*innerMutex);
    clear();
    //Cycle through the list of platforms and draw them.
    for (GameObject* i : drawables) {
        draw(*(dynamic_cast<sf::Drawable*>(i)));
    }
    for (std::shared_ptr<GameObject> i : nonStaticObjects) {
        if (i->isDrawable()) {
            draw(*(dynamic_cast<sf::Drawable*>(i.get())));
        }
    }
```

A3:

```
std::string Character::toString()
{
    std::stringstream stream;
    char space = ' ';

    stream << getObjectType() << space << isConnecting() << space << getID() << space << getPosition().x << space << getPosition().y;
    std::string line;
    std::getline(stream, line);
    return line;
}

std::shared_ptr<GameObject> Character::constructSelf(std::string self)
{

    Character *c = new Character;
    int type;
    int connecting;
    int id;
    float x;
    float y;

    int matches = sscanf_s(self.data(), "%d %d %d %f %f", &type, &connecting, &id, &x, &y);
    if (matches != 5 || type != getObjectType()) {
        throw std::invalid_argument("Type was not correct for character or string was formatted wrong.");
    }


    c->setPosition(x, y);
    c->setConnecting(connecting);
    c->setID(id);

    std::shared_ptr<GameObject> ptr(c);
    return ptr;
```

A4:

```
else if (collision->getObjectType() == SideBound::objectType) {
    SideBound* sb = (SideBound*)collision;
    sb->onCollision();
}
```

A5:

```
void Character::setSpawnPoint(SpawnPoint spawn)
{
    this->spawn = spawn;
}

void Character::respawn() {
    setPosition(spawn.getPosition());
}
```

A6:

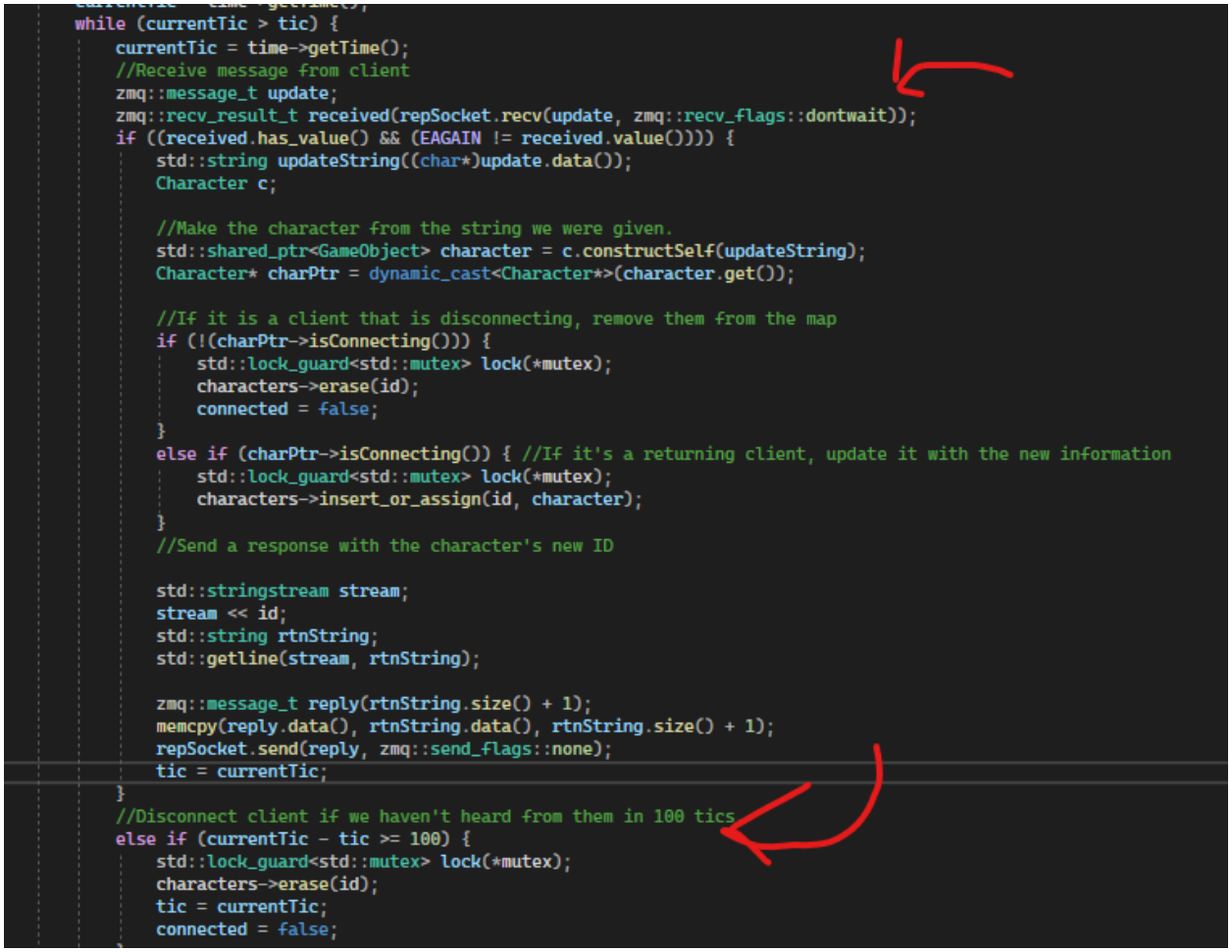
```
void GameWindow::updateNonStatic(std::string updates) {
    std::lock_guard<std::mutex> lock(*innerMutex);

    char* currentObject = (char*)malloc(updates.size() + 1);
    if (currentObject == NULL) {
        throw std::runtime_error("Memory error while updating static objects");
    }
    int pos = 0;
    int newPos = 0;

    //Scan through each object
    while (sscanf_s(updates.data() + pos, "%[^,]%n", currentObject,(unsigned int)(updates.size() + 1), &newPos) == 1) {
        //Get the type of the current object.
        int type;
        int matches = sscanf_s(currentObject, "%d", &type);
        if (matches != 1) {
            free(currentObject);
            throw std::invalid_argument("Failed to read string. Type must be the first value.");
        }
        //Push the newly created object into the array.
        nonStaticObjects.push_back((templates.at(type))->constructSelf(currentObject));
        //update position and skip past comma
        pos += newPos + 1;

    }
    free(currentObject);
}
```

A7:



```
while (currentTic > tic) {
    currentTic = time->getTime();
    //Receive message from client
    zmq::message_t update;
    zmq::recv_result_t received(repSocket.recv(update, zmq::recv_flags::dontwait));
    if ((received.has_value() && (EAGAIN != received.value()))) {
        std::string updateString((char*)update.data());
        Character c;

        //Make the character from the string we were given.
        std::shared_ptr<GameObject> character = c.constructSelf(updateString);
        Character* charPtr = dynamic_cast<Character*>(character.get());

        //If it is a client that is disconnecting, remove them from the map
        if (!(charPtr->isConnecting())) {
            std::lock_guard<std::mutex> lock(*mutex);
            characters->erase(id);
            connected = false;
        }
        else if (charPtr->isConnecting()) { //If it's a returning client, update it with the new information
            std::lock_guard<std::mutex> lock(*mutex);
            characters->insert_or_assign(id, character);
        }
        //Send a response with the character's new ID

        std::stringstream stream;
        stream << id;
        std::string rtnString;
        std::getline(stream, rtnString);

        zmq::message_t reply(rtnString.size() + 1);
        memcpy(reply.data(), rtnString.data(), rtnString.size() + 1);
        repSocket.send(reply, zmq::send_flags::none);
        tic = currentTic;
    }
    //Disconnect client if we haven't heard from them in 100 tics
    else if (currentTic - tic >= 100) {
        std::lock_guard<std::mutex> lock(*mutex);
        characters->erase(id);
        tic = currentTic;
        connected = false;
```