CSC591 (Game Engine Foundations) Homework 1: Network Foundations

Introduction

In this assignment, I had an introduction to the basic concepts of a game engine. Though we were building a multi-threaded server/client architecture, I paid enough attention to make other parts of my engine as generic as possible. I made my engine as a library which can be used and extended as and when necessary. I will be describing the individual parts of my engine as I go forward explaining each section of this assignment. Also, I will mention all my engine components in a separate list with a brief description of each of them.

Running my program

There are two ways to run my multiplayer game / engine demo:

1. **JAR**:
    i. Find the JAR file for this project in `[root_dir]\out\artifacts\CSC591_GE_HW1.jar\CSC591_GE_HW1.jar`.
    ii. Open a command line and type `java -jar CSC591_GE_HW1.jar s` (for server).
    iii. For running clients, open other command lines, type and execute `java -jar CSC591_GE_HW1.jar c` as many times you want for any number of clients.
    iv. Remember that you need the run the server first and then the clients, otherwise this might throw some exception. This should be normal, as for most multiplayer games, the headless server generally is always running. Also currently, the client searches for a running server in `localhost`, so running the server and client in different computers will not work. If you still want to run it in different computers, follow my second way of running the program and before building it, open `Constants.java` and assign the server's IP to the `SERVER_ADDRESS` String variable.
2. **IntelliJ**:
    i. Install IntelliJ Community Edition.
    ii. Import and build my project.
    iii. There should be two run configurations - one for the server and one for the client. Run the "Server" first and then the "Client". The shortcut for running programs in IntelliJ is `Alt + Shift + F10`.
    iv. If you don't find the run configurations, make two yourself. For the sever, give a command line argument of `s` and for the client, give a command line argument of `c` (without the quotes).

When you run the server, you should see a small square which you can control using `A`, `D` and `SPACEBAR`. You will also see some rectangles (stairs) coming from the top which you can jump on and jump from there to other stairs. When you start the client(s), you will see the same stairs (color and position) coming on their screen as well. These are sent from the server. On the client screen, you can move the client square around and play the game as usual, as if the rectangles were generated from the client side. On the server side, you can see the clients' squares and how they are playing the game. This data is sent from the client(s) to the server. Note that the transfer is a little sluggish, and this is most apparent on the server screen. This is a known issue and I have not put in any effort to optimize it for this homework.

Processing

As suggested by the homework specification, I have used Processing for building my engine and the game. Professor advised to make simple rectangles draw on the screen and to allow the user to interact with them. I have followed that idea and build a simple playground to demonstrate the components of my engine. There are two main components of this playground:

1. **Stairs**: These are rectangles of fixed width and height, but of random colors, which keep coming from above the top of the Processing window with a constant downwards velocity and disappear below the window.

2. **Player**: The player controls a square of random color (of different dimension than the rectangles). He can move the player left and right using the `A` and `D` keyboard keys and jump using the `SPACEBAR`. His left or right motion gets carried on when he jumps. In this way he can jump on the top of the stairs and from each of those, jump to other ones and thus reach the top part of the window. The square representing the player checks collision detection with the stairs, from all sides. The collision response however, is to just put them on top of the stair, no matter where they collided with the stair. This helped me keep it simple and not focus too much on the game itself, and instead put in more effort building the engine. Initially, I had thought of keeping a goal to reach in the top, but due to time constraints, I figured to leave it at this. So, there is no real goal for this game and hence, I simply call it a playground instead of a game. But it is important to note that this simple playground fulfills all the required criteria for this section of the homework and allows me to properly demonstrate the data transfers in my server/client architecture, which I will talk about in a bit.

Before going on to describe the later sections of the homework, I will lay out the architecture of my engine which contains the multi-threaded server/client implementation (which is the main goal of this homework) as well as some other important game engine components, which are significant from a design perspective and future modularization.

Engine

As I started building the engine from scratch, my main goal was to make it as modular and independent as possible. What I mean is that anybody can take my engine and build a game if they want. The graphics portion will obviously be provided by `Processing`, but things like game objects, update & draw loops, networking, etc. will be provided by my engine. This class (`MainEngine`) extends the `PApplet` class in `Processing`. I will try to categorize the components and explain them in detail in the following sub-sections:

1. **Game Objects**: `GameObject` is an abstract class present in my engine, which exposes the update and draw loops to the game. Anybody intending to put any kind of game object which will be drawn to the screen has to extend this class and implement these two methods. In `updatePosition`, they would apply all the game logic and in `drawShape`, they would draw the object to the screen. They can use the `PApplet` class, which will passed to the game by the engine (I'll come to this later). In the playground that I have been talking about, `FallingStair` and `Player` are `GameObject` instances. All these game objects have to be "registered" with the `MainEngine` instance by the `Controller` instance (the class in the game which has extended this class). This can be done with `registerGameObject` or `registerGameObjects`, depending upon if you have one object that you want to register or a list of them. These two functions basically store the game objects in the engine class and they are later used by the `MainEngine` instance to call their `updatePosition` and `drawShape` methods. Now, among these game objects, some might be requiring binding to key presses. This can be done by calling `registerKeypressUsers` in the engine instance (this will be done by the `Controller` instance).

2. **Controller**: `Controller` is an abstract class which should be extended by a manager class in the game. This will be a bridge between the `MainEngine` class and everything else in the game. This is a design decision that I thought will be appropriate for a game developer. This is the class that should have the `main` method. In the `main` method it should ideally call the `startEngine` static method which is present in the `MainEngine` class. The `startEngine` is the one which starts of the `PApplet` `main` method behind the scenes of the game developer. This will consequently call the `settings`, `setup` and finally the `draw` call in `Processing`. In the `setup` method, `MainEngine` calls the `setEngine` method in the `Controller` instance and passes its own reference, which can be used later by the game to call the `PApplet` functions and by the manager itself to call engine specific functions. This method also has the call to a certain abstract `initialize` method in the `Controller` which is basically a way for the `Controller` instance in the game to initialize its components. It could actually do all initializations in the main method, but there is one thing that is changed over the course of time from when it first entered the `main` method to now when the `MainEngine` is making its `initialize` method being called - and that is the `MainEngine` reference itself which the engine set in the `setEngine` method. Now, the game `Controller` can use the engine instance to register game objects, server, client, etc. The class which extends this `Controller` base class in my playground is `SimpleRaceManager` (named such because I had initially thought to make this a racing game).

3. **Server**: `GameServer` is a class which has to be registered from the `Controller` instance using the `registerServer` method in `MainEngine`, passing it the server port (this port is in the discretion of the game itself). Once it gets registered, the engine starts a new thread of the server. This thread opens up a Java `ServerSocket` and listens for incoming connections. Whenever a client connects to this server, the server opens up two new threads for reading and writing data from/to the client. Additionally, the `Controller` instance exposes `sendDataFromServer` and `getDataFromClient` to be used by the game instance which acts as a server to send data from the server and get data from clients. These methods are called by the independent server connection threads when it receives / sends data. The data structures are only partially synchronized from an engine perspective and should be taken care of by the game itself.

4. **Client**: `GameClient` is similar to `GameServer` except that the `Controller` exposes `sendDataFromClient` and `getDataFromServer` for the game instance which runs as a client, to execute. Also, after the `MainEngine` creates the `GameClient` instance, it tries to connect with the server. When it successfully does the same, it spawns two new threads for reading and writing from/to the server. This client behavior can be registered by calling the `registerClient` method in the engine instance. This would ideally be done from `Controller` instance in the game.

Now, I will go over the individual sections and show how the simple playground uses these engine components and builds a multiplayer experience.

Multithreading basics

Before I go to the playground demonstrating my engine, I will cover this part separated-ly. I have kept this part isolated from the entire engine implementation because of its requirements. To run this part:

1. Go to `[root_dir]\fork_example`.
2. Run `javac .\ForkExample[x].java`, where `[x]` is a number from `1` to `4`.

3. Run `java ForkExample[x]`.

I will go through the individual changes one by one:

1. **ForkExample1.java**: First have taken the two thread model and expanded it to an arbitrary number of threads to be spawned. Then I have implemented the producer-consumer model with the skeleton that had been provided. Till now, there is no data structure that the producers and consumers work on really, but there is a producer which sleeps twice and wakes up and notifies other consumer threads. For this change I experimented with the `notify` call. I put an extra sleep before the consumer enters its `wait` such that the producer calls `notify` on any of them. My test was to see whether `notify` calls get stored and are later delivered to anyone entering the monitor. But as it turns out, the first time that the producer notifies the thread there is no one to listen to that `notify` call and hence it goes to waste. The Java API does not deliver that `notify` call at a later stage. Also as the producer exits, there is no one to notify the consumers any more and hence the program goes into a perpetual wait state at that point.

2. **ForkExample2.java**: For the second change, I have completed the producer consumer model with a data structure containing random numbers which get added by the producer and consumed (printed on the screen) by the consumer. All the consumers wait unless anybody is notifying them or if the array is empty. The producer on the other hand fills up the array, notifies some consumer and goes to sleep. The consumer wakes up and consumes a number and notifies another thread waiting in the monitor (it may be a producer or a consumer). This cycle continues.

3. **ForkExample3.java**: For this change, I removed all the synchronized blocks around the data structure. This was possible because I have used `ConcurrentLinkedQueue`. I tried the program again and it gave the same output (only the consumer outputs are kept), thus showing me that those data structures are synchronized at an API level. Moreover, there is no wait and notify and instead, all the threads contest for production and consumption at the same time. This test, though not appropriate all the time (too much contention), will still work correctly.

4. **ForkExample4.java**: For this change, I did something interesting and introduced barriers into our producer consumer model using `CyclicBarrier` in Java. If you have noticed in the earlier outputs, there was only one thread which woke up, consumed all the numbers and then went to sleep (because they could not get the lock for the data structure) and other threads went into starvation for that production iteration. Instead I introduced a barrier and tell every thread that it can only consume two numbers. Whenever it has finished consuming two numbers, it has to exit the consumption loop and wait in a barrier. This barrier expects `NUM_THREADS` number of threads to hit it. This gives a chance for other threads to consume it and hit the barrier too. Once they all hit the barrier, the cycle of production and consumption starts again. This is noticeable in the output as one thread consumes only two numbers, and you would be able to see numbers being consumed by other threads.

Networking Basics

As mentioned before, the `GameServer` and `GameClient` are independent components of my engine.
The process described in the homework specification is exactly how I have implemented this and I have explained in the earlier section. In brief, my `GameServer` listens for client connections in a new thread. Each `GameClient` looks for the same server. When a connection is established, the both the server and the client create new threads to read and write from/to each other. There is one minor difference though - the server keeps on looking for accepting connections using the `ServerSocket` instance that it has made. But the client, on the other hand only talks to one server at a time - so it does not need to keep looking for more servers, i.e. it's `run` method (of the `Runnable` interface) exits after connecting to one server. As I will show later, the data transfer will be game objects, so my `GameObject` class also implements the `Serializable` instance. All this is done in the engine components, so the game developer doesn't need to worry about these. He just needs to take care of implementing the functions for getting and sending data to/from server/client(s).

Putting it all Together

In my playground, the server is not headless, i.e. the server also has a screen which renders a player of its own (this is similar to games like Counter Strike where any one of the players in the multiplayer environment becomes the server). This method may not be used by a game made using my engine by simply registering game objects, which draw nothing on the screen (this can be turned on by a boolean flag while registering the game objects).

Anyway, coming back to the point, the job of my server is to generate the falling stairs and send them as game objects across the network. The clients will receive them and render them on the screen. Thus all the heavy-lifting of creating the environment is being done by my server. The clients receive this data and send their own (player) game object instances to the server. The server renders the players (squares) from the client side on the screen. Initially I had done this as a synchronous system, i.e. the client and server have only one thread which is used to read and write. But later I changed this to an asynchronous behavior, so currently my program has no trace of a synchronous behavior, which again is also not the ultimate goal of this assignment.

It's important to note the way I have sent and received the game objects. So whenever my `Controller` instance tells the server (or the client) to send something, it takes the list of game objects and appends a `NetworkStartTag` and a `NetworkEndTag` at the beginning and the end. These are basically dummy game objects (which have a special boolean in them) which are a sign for the other side that a game object stream will start or end. After appending these to the list of game objects that the game has given it, it sends all of the game objects one by one using the `ObjectOutputStream.writeObject` method. This instance is again initialized using the value returned by the `Socket.getOutputStream`. It calls `ObjectOutputStream.reset` after writing all game objects. This has to be done to avoid bugs where the other side keeps receiving stale data. Similarly, the side reading the data uses `ObjectInputStream.readObject`. As soon as it finds a start tag, it starts adding the following game objects in a list. It stops building this list as soon as it finds an end tag. It then calls the respective method from the `Controller` instance (`getDataFromServer` for a client game instance or `getDataFromClient` for a server game instance) and provides them with this list. If the instance of the running game is a server instance, it also sends a `connectionID` in the argument telling which client has sent this data.

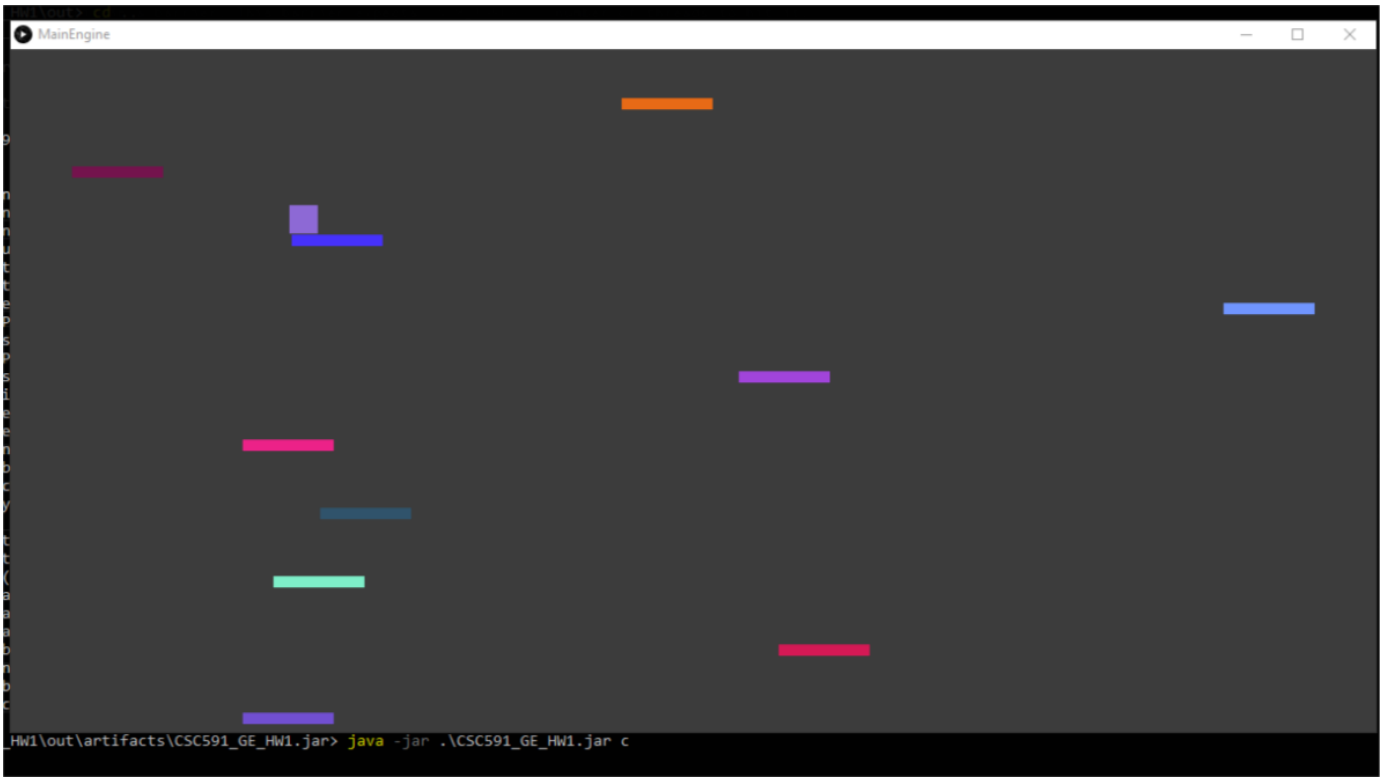
Asynchronicity!

I have already explained how my server and client creates two threads as it accepts connections (one for reading and one for writing). In total there are 3 threads running for the client - (1 for main game loop + 1 for reading data from server + 1 for writing data to server). For server, the number of threads depends of the number of connected clients. It can be represented as `1 + 2 * n`, where `n` is the number of connected clients. The data received on each end is asynchronously made available to the game as there are separate threads for these tasks. The main game loop (which keeps calling the `manage` method in the `Controller` instance) keeps updating the set of stairs on the engine side (by calling `registerGameObjects`) with the one that is on the game application side.

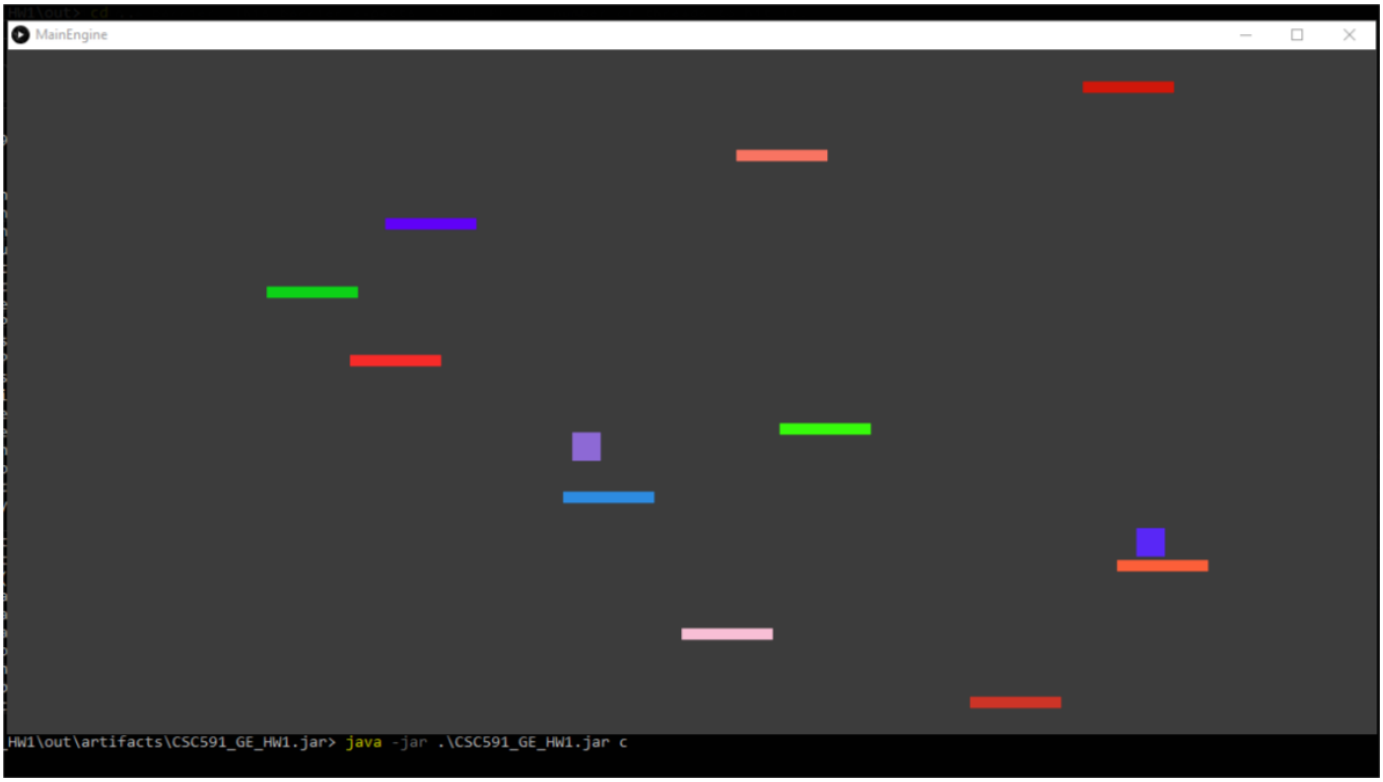
For synchronizing the data received from the game perspective, I use a `ConcurrentLinkedQueue` of `GameObject` instances. This is because all my game objects are never accessed randomly. They are mostly used by my engine to call their `updatePosition` and `drawShape` methods one by one. Making a `ConcurrentHashMap` does not make sense in that respect. Moreover, when my game objects are registered with the engine, the `Controller` has to pass an integer referencing the `gameObjectListID`. The engine internally maintains an `List` of `List` of `GameObject` instances. This allows the `Controller` to set a list of `GameObject` instances at a certain position of this game object cluster at once (like the client setting the list of stairs that it receives from the server, in the engine). It should be also noted that all the game objects that are sent are of `Serializable` nature. It any game specific class extends the `Gameobject` class and adds its own variables, the developer (like me in my playground) should make sure that the ones which are not required to be sent over the network, be marked `transient`, as I have done with certain class variables in `FallingStair` and `Player`.

Appendix

Client screen purple square represents player



Server screen with 1 client (blue -> server, purple -> client)



Server screen with 2 clients (blue -> server, purple, green -> client)