CSC591 (Game Engine Foundations) Homework 2

Homework Spec

The homework spec can be found in `[root_dir]\docs\HW2.pdf`. Below is a brief excerpt from the same:
Your task for this assignment is to add more functionality to your Processing engine.

Introduction

I built the implementation of this assignment on my last homework assignment code. On a side note, I had already implemented quite a lot of this assignment in the first homework and hence devoted my time to fine tune and optimize my code to remove lags and improve efficiency. Specifically, I extended my Game Object Model from the first assignment a little bit, did not have to do anything for the second part as it was already done, added death zones and spawn points for the third part and impelemented a string network protocol for the fourth part and compared with my existing method of transmitting data over the network. In any case, I would explain each part in the following sections, starting with instructions on how to run my program.

Running my program

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

1. **JAR**:
    1. Find the JAR file for this project in `[root_dir]\out\artifacts\CSC591_GE_HW1.jar\CSC591_GE_HW1.jar`. (ignore the project name as I have been using the same IntelliJ project from the first homework)
    2. Open a command line and type `java -jar CSC591_GE_HW1.jar s` (for server).
    3. 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.
    4. 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**:

    1. Install IntelliJ Community Edition.
    2. Import and build my project.
    3. 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`.
    4. 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 (`FallingStairs`) coming from the top which you can jump on and jump from there to other `FallingStairs`. When you start the client(s), you will see the same `FallingStairs` (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. In this assignment, I have made the server headless, as it should actually be. But as I am using Processing, it is not possible to compeletely make it headless, i.e. to make the display screen invisible. So I have (albeit a little hacky-ly) just reduced the window size for the server process to a mere 10x10, and nothing gets displayed there, thus almost giving it the headless feel. One more thing to note is that if you want to use my string protocol instead of game object protocol for sending data from servers to clients, you need to change the `stringProtocol` boolean in the `com.debalin.engine.util.EngineConstants.java` file. If you remember my last submission, you would notice that this version runs with almost no noticeable lag.

Game Object Model

I looked into all possible cases that I can use for building my Game Object Model and finally decided on using Monolithic Hierarchy for my game engine. It might be a little old in its use cases and there are certain disadvantages like feature creeps and bubble up effects, but if used correctly, monolithic hierarchy can be used to yield pretty optimal results. Furthermore, it should be noted that monolithic hierarchy is the most intuitive among probably other models like pure component or generic component model, which makes it a lot easier to build your game architecture in as a beginning. Also, I have given enough care to make the hierarchy such that it can be extended later with ease. Also, monolithic hierarchy should not be confused

with older C games which have all the functionality in one single big file, but rather just a technique to significantly use inheritance among all other OOP fundamentals to architect your game engine. In the following portion, I will list my inheritance tree with explanations for each node in the tree.

1. `GameObject` : At the top of the inheritance tree is `GameObject` which is a part of my `Engine` and is an abstract class. This contains the basic properties (as fields) needed for any game object - **color**, **size**, **position**, **visible** and **connectionID**. All of these fields are self explanatory except probably **connectionID**. This variable is supposed to keep the connection ID for a client in a multiplayer game. It should also be noted that these variables are specifically suited a 2D game and that's fine for our use case. This abstract class contains abstract methods for `draw` and `updatePosition`. It is inherited by two abstract methods which are also a part of engine:
    1. `StaticGameObject` : This, as the name says is supposed to hold static game objects. As because this does not need any velocity or acceleration this does not have any specific fields, but can be extended later to incorporate other things. It is inherited by one object which is part of the game:
        1. `NonMovingRectangle` : This is supposed to give a base class for my static stairs. This is also an abstract class. This is finally extended and concretized by `StandingStair` and `SpawnPoint`, with respective implementations for the `draw` method and empty implementations for `updatePosition`.
            1. `StandingStair` : A standing stair is a static game object in the game. It gets initialized with the a specific size (from the `Constants` file) and random color.
            2. `SpawnPoint` : This class represents the point from where an individual player would spawn. Every player is passed an instance of this (thus having bits of a composition model on the `Player` class), initialized with a random x position and a fixed y position.
    2. `DynamicGameObject` : This is supposed to hold dynamic game objects. This holds extra variables for `velocity` and `acceleration`. This is extended by an abstract class:
        1. `MovingRectangle` : This forms a base class for all the main components of my game - `Player` and `FallingStair`. They both have velocities and accelerations.
            1. `Player` : This implements the `KeyPressUser` interface provided by the engine. It has a method called `handleKeyPress`. This method is being called for objects which are registered through the `Controller` class. Apart from that `Player` is the most important class in our game. This is the class which is controlled by the user. Obviously it has requisite implementations for the `updatePosition` and `draw` methods. Other

than that, it has methods like `checkDeath` (checking whether the player has collided with a death zone) and so on. It is important to note that my `Player` is running using a state machine with the following states - `ON_GROUND`, `ON_STAIR` and `ON_AIR`. The states are self explanatory.

2. **`FallingStair`**: This is another important concrete class in the game coming from the line of `DynamicGameObjects`. It has a rather basic implementation with a constant velocity being added to the position with time. Apart from that, it also has the ability to work as a `deathStair` which I will cover in a later section.

3. **`UtilityGameObject`**: This extends `GameObject` and is meant for objects which are not meant to be probably drawn on the screen. Though `GameObject` provides color and size, this abstract class makes all of them transient, thus setting a contract that they are not intended to be used. There are two concrete children of this type of `GameObjects`. These two are used by the network component of the engine itself. When the engine gets a collection of data to send over the network, it encapsulates the stream of data between these two "dummy" `GameObjects`.

    1. **`NetworkStartTag`**: Apart from being the front part of the encapsulation, it also sends some meta information about the connection to the client, like what is the connection ID of their TCP connection. This helps the client(s) to build an efficient `ConcurrentHashMap` of other players in its own implementation.
    2. **`NetworkEndTag`**: This forms the back part of the encapsulation.

These three distinctions will help build any basic type of game objects and later on if needed, they can be extended and diversified to satisfy any kind of requirements. Moreover, the `UtilityGameObject` is in itself a kind of custom `GameObject` which can be used to build anything. Though `Controller` is not a part of the Game Object Model per se, it is still an important part of the architecture. This controller class gives the developers a customizable way to manage their own resources as well as having the affordance to register game objects and forget about calling their `updatePosition` and `draw` methods by themselves. The `Controller` is extended by `SimpleRaceManager` in my playground and is used to take care of sending data to the server and client and receiving data from the same. Other than that it has a `manage` method where the server keeps on spawning stairs and the client keeps on registering the recently updated player values from the server with the engine. This is a brief overview of my Game Object Model and a summary of my `Controller` class. The `Controller` class was detailed in the last assignment README and it has almost remained the same in structuring, so I thought it best not to explain it too expositorily here again.

Multithreaded, Networked, Processing Sketch

I had completely implemented this in the last assignment. I will give a brief overview here one again for clarity. As mentioned in the homework specification, my server sets up the scene for the clients to render. At first, it sends standing stairs which are static game objects to all the clients that it has connected to. The server has two threads for each client - `maintainClientReadConnection` and `maintainClientWriteConnection`. And it has another thread for accepting client connections. As the number of clients grow, the number of threads on the server grow as well. Each client on the other hand has only two network threads - `maintainServerReadConnection` and `maintainServerWriteConnection`.

The server has a "bag-of-tasks" approach in writing game objects into the outward data stream. The main game loop keeps adding new game objects to this bag and the server network thread keeps consuming them. When the bag becomes empty, the server `waits` on this queue. On the other hand, the server main game loop thread `notifies` when it fills up the queue. This is achieved by implementing all of this inside `synchronized` blocks around these queue references. This is an important reason why my game does not lag anymore. Previously, I was running the server thread in a `while true` loop, without any stop, which might not be very beneficial. Now, the server thread stops and sleeps when the bag of tasks is empty.

The server then keeps adding each stair that is spawned in the queue, when they are spawned, and they are never repeated. The client receives them and as they all have constant velocities throughout, it does not need to get the stairs in every while loop. Instead, only the starting positions, velocities and colors would be enough for the clients to render the falling stairs or standing stair throughout their lifetime.

The client on the other hand sends its own location at every `while true` loop iteration because that cannot be rendered on its own without a location update at each instant by the clients. Still there is a `Thread.sleep(1)` to stop the `while true` loop momentarily at each iteration. This helps the OS to schedule efficiently. And it also contributes to the non-laggy performance of my game.

One more thing to note is that whenever a game object is received by the client, it registers that game object in the game object cluster (a list of queues of `GameObjects`) in my game engine. The engine later goes through this cluster one by one and calls `updatePosition` and `draw` for each `GameObject` in each queue in the cluster.

A 2D Platformer

My 2D platformer is complete with static and dynamic platforms, controllable player, spawn points, death zones and scoring.

1. **Characters**: This is represented by the `Player` class. And as I had mentioned before, it is controlled by the user himself/herself by pressing `A`, `D` and `SPACEBAR`.
2. **Static Platforms**: This is represented by the `StandingStair` class. The player can jump on this and move on it. Collision detection detects whether the player has collided with this kind of platform or not. The collision response is to offset the player in the y direction to sit over the stair and keep the x position the same.
3. **Moving Platforms**: This is represented by the `FallingStair` class. The collision response and detection is same here like `StandingStair`. Other than that they keep coming from the top and disappear after they cross the bottom of the screen. All of the stairs have random colors and random velocities generated from the server adn sent to the clients. One thing to note is that when you start the program and see that on one client there are certain stairs which are not showing up on the other client, then this is because I rely on eventual consistency to extract as much efficiency as possible. Hence, my stairs are only sent when they are newly created. Thus if a stair was created before and sent to a certain client, it might so happen that the stair hasn't reached the bottom, but another client joined in, then it will not get the stair which is probably midway on the other client. This is how the game (playground) is architected.
4. **Spawn points**: These are hidden are their positions are randomly decided on the x axis by the `Controller` which spawns all of this and have a fixed y position, written in the `Constants.java` file. For each client there is only one spawn point.
5. **Death zones**: These are basically falling stairs which are not rendered on the screen. When a user steps on one, their outline is shown in red and that denotes that the user has fallen into a death zone. After this collision, the player is regenerated from a spawn point.
6. **Scoring**: The scoring is done for each player and is shown on each client on the top right portion of the screen. The score is calculated based on the idea that hwo long can you stay on the top portion of the screen. Additionally, if you step on a platform you get points, so its better to keep jumping on a platform even if you can't reach the top. It's a minimalistic game, but with this scoring method, it becomes difficult to score and hence brings in challenge.

Performance Comparison

I have implemented the network code using two protocols - one where I send `GameObjects` after serializing them (I was doing this before only) and another where I convert the `GameObjects` to custom strings, send them across and parse them to create `GameObjects` again. To choose which protocol you want to choose,

change the `stringProtocol` boolean value in the `EngineConstants.java` file. The rest will be taken care of by my code.

In the first method, I take care in tagging the unnecessary objects as `transient` so that they don't get sent over the server. So when Java serializes them, it excludes all of those variables and everything works as expected. I did not need to use `writeReplace` or `readResolve` because of the way I have architected the code. In case of strings, I take the `GameObject` just before sending over the stream and put the name of the class delimited with ":" along with all the important variables that I would need on the other side to inflate it back. On the other side, I split the string, and extract all the information and use that information to form the `GameObjects` back to normalcy.

My metric calculation code is present on `SimpleRaceManager` and `GameServer` classes, so I have taken them separately and put them in the `tests` folder in the root directory. This is because metric calculation does not make sense in the main branch and will be better if organized into a different folder. To run them, just copy and replace these three files in the actual code and build it again.

In the specification it asks to calculate the time taken for 1000 game loop iterations. But having separate threads for networking and game loop, it made sense to change the metric a little but to have more effect on the actual networking protocol used. So what I did was use three configurations:

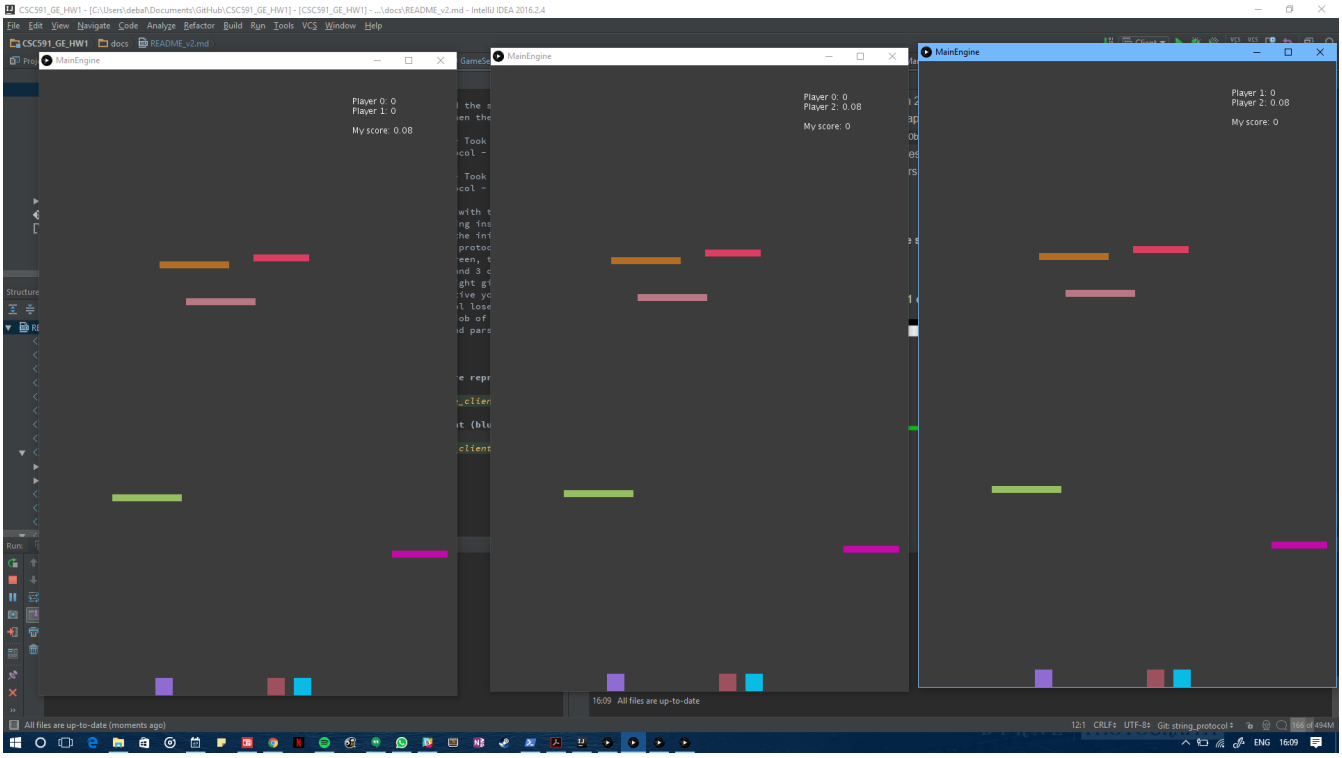
1. Send 5000 game objects from the server to the clients - velocity of stairs range from 0.5 to 1.3 - averages a maximum of 10 objects moving on the screen at one instant:
    1. 2 clients: I measured the start time from when the network starts sending the objects and the end time when the clients have completely rendered all of those objects.
        1. String protocol - Took 60.174 seconds.
        2. Game Object protocol - Took 66.76 seconds.
    2. 3 clients:
        1. String protocol - Took 49.331 seconds.
        2. Game Object protocol - Took 48.51 seconds.
2. Send 15000 game objects from the server to the clients - velocity of stairs range from 0.1 to 0.3 - averages a maximum of 100 objects moving on the screen at one instant:
    1. 2 clients: I measured the start time from when the network starts sending the objects and the end time when the clients have completely rendered all of those objects.
        1. String protocol - Took 246.89 seconds.
        2. Game Object protocol - Took 270.01 seconds.
    2. 3 clients:

        1. String protocol - Took 212.48 seconds.
        2. Game Object protocol - Took 201.252 seconds.
    3. Send 15000 game objects from the server to the clients - velocity of stairs range from 0.1 to 0.3 - averages a maximum of 500 objects moving on the screen at one instant:
        1. 2 clients: I measured the start time from when the network starts sending the objects and the end time when the clients have completely rendered all of those objects.
            1. String protocol - Took 265.98 seconds.
            2. Game Object protocol - Took 259.223 seconds.
        2. 3 clients:
            1. String protocol - Took 231.8 seconds.
            2. Game Object protocol - Took 228.83 seconds.

If you want the raw results with timestamps, they are in the `tests` folder. These results give some interesting insights. String protocol would seem cheaper on the network and that shows in the initial results. But as the number of clients increases, the game object protocol seems to do a much better job. In the end, for the 500 game objects on screen, the game object protocol completely outperforms the string protocol in both 2 and 3 client cases. This probably is due to the fact that string protocol probably might give you a cheaper network overload, but ultimately from a game engine perspective you need to convert them to `GameObjects` and that is where the string protocol loses, because apparently, from the results, Java itself does a much better job of serializing and de-serializing that me makings string from game objects and parsing and creating game objects from strings.

Appendix

Three clients



Death stair shown in red outline