Assignment 1 – Write up

CSC591 – Game Engine Foundation
Rachit Shrivastava

Section 1: Processing

- Tried hands-on the processing programming language, drew some rectangles, circles on screen.
- Considered an object which moves from one side to another at a certain pace by itself, by pressing left and right arrows you increase its speed.
- But the object will be stationery at first, you'll have to press either left or right arrow for its movement, after that it moves to the direction which the player wants to.
- When the object hits the left or right boundary, it automatically appears from right or left side of the screen respectively.
- If the object touches any object placed on the platform, it stops. Collision with the sun is not implemented, as it is not supposed to be in same z-index as the platform and player. For collision detection I've converted my Processing Rect to awt.Rectangle and then used .intersects() to check for collision. Now you have two things that you could do to make it move,
    - Firstly, change the direction of it movement, if it got stuck while moving in right direction, press keyboard-left button
    - Secondly, you can still go in right direction, but you'll have to keep on pressing right many times, as the default speed of the object becomes zero; and you can only force it to pass through some wall – can be considered as some powerup feature for a game.
- In order to jump, hit Spacebar. You can jump multiple times which means when you're in air you can again jump higher, it's not the earth gravity that I'm using. But however higher you jump (there's no top limit value or sky-limit) it eventually comes back to the platform it was moving to.
- Also while jumping you can use the arrow keys to specify the direction you want to jump to and the jump becomes direction specific.

Section 2: Multithreading Basics

- **ForkExample1.java**
    - It has got 3 threads of which t2 and t3 depends on t1 to finish.
    - As soon as t1 is done, it sends notify(), so of the two threads t2, t3 only one gets executed as notify() is send to any one waiting threads.
    - Thus, the third thread is left hanging and the code never completes execution.
- **ForkExample2.java**
    - It has got the same scenario as ForkExample1, just that now when t1 is done it sends a notifyAll() and notifies all the threads that are waiting for it to complete.

    - As t1 is done, t2 and t3 starts execution simultaneously, any of the two threads can start first.
    - But both run fine this time, and there is no deadlock. Code executes successfully.
- **ForkExample3.java**
    - Here we have only two threads, but I have removed the second synchronized block from the code, but marked busy = false outside of it.
    - This is to show importance of synchronized block, and that t2 would not be notified of t1's progress in this case
    - The code will be stuck forever, and t2 will never complete its task
- **ForkExample4.java**
    - In this example there are three threads, t1, t2 and t3. And they have to be executed in particular order which goes like t1 -> t2 -> t3.
    - Their individual dependency has been defined initially, also now I have set busy to be true when (i==1 || i==2), so that t3 knows that even t2 is busy waiting.
    - The code executes perfectly fine and the thread execution happens in required order.

Section 3: Networking Basics

- In this program, there's a server class and a client class, the server calls accept 3 times and waits for three clients to send it some input data.
- Even though all the 3 clients have been connected to it, it will speak to each client one after other, ie synchronously.
- It just reads in the data from client 1, responds it by sending back the input string into its equivalent uppercase format. Then it terminates the connection to client 1.
- Then it performs the same process for client 2, after that client 3.
- Even if client 3 has send it some data string, it won't read it until it is done with client 2.
- **Note –** you will have to run the server first and then 3 clients separately, after the server is up. This example only deals with 3 clients.

Section 4: Putting it all Together

- For this scenario, I have considered the case that client and server will be communicating with each other over a long duration of time, consider till the server is down or the client closes its connection.
- Also, in order to add n number of clients I've a while loop, which just keeps on listening to serverSocket's accept() call indefinitely.
- The CopyOnWriteArrayList from java.util.cocurrent package maintains a thread-safe list of the number of clients connected and is updated as soon as a client Is connected to server.
- We have a thread per client which does both read and write messaging from server to client, thus this code will have n- threads for communication.
- This will help us achieve asynchronous client connection to server.

- This will also provide a better version for Section 5, where we have to provide asynchronous communication with the use of multiple threads.

Section 5: Asynchronicity!

- For this section I've modified each of my client to have two threads, one to read and one to write into the buffer.
- Same goes for the server, server now will be having two separate threads to handle write and read operations and gain synchronous communication with client.
- This design is taken into consideration keeping in mind the number of communication a server would be doing with multiple clients connected to it. At any instance, m-number of client can send or receive information from the server
- Thus designing the server to have 2 threads for read/write per client gives us a leverage of communicating with any client without any struggle.
- Consider a scenario where 3 different clients are trying to submit their number of kills in the Total-Kills table onto the server at the same time, and the server if has got only 1 thread for read and write, this would create an issue here. Having 2 threads per client for read and write helps us simplifying problems like these.
- I wanted to develop same thought I had in my mind for communication, that it will end when the client terminates the connection or server shuts down (same as I did for Section-4), but somehow this couldn't work in this scenario and as soon as a client receives server's data back, it shuts itself.
- Will have to look into this a bit more, then I would say that would be a perfect architecture for a single server communicating with multiple clients synchronously.

As of now, I'm planning to use one thread per client for communication on the server as my communication model, but I am still looking onto some methods of keeping two threads (one to read and one to write). I find the approach for section 5 very safe approach. Considering every task has got its own thread. Bringing the code to work for multiple communication iterations will be a challenge for me. But my code for Section 4 brings out the same asynchronous behavior for the communication.