CSC 591
Homework 1
Aurora Tiffany-Davis
(attiffan)

1. Section 1: Processing

1.1. Extending Processing Classes

The program uses Java classes that extend the base Processing classes, enabling development in Eclipse. This sped up development, as I am at least somewhat comfortable in Eclipse already. In the future, this could make the project more maintainable, as there are likely more software engineers comfortable working in Eclipse than in Processing sketches. There was one obstacle, in that I had a compilation error related to accessing the Processing core.jar file, but that was quickly overcome.

1.2. Game Object Inheritance Hierarchy

The program uses a monolithic hierarchy. This is not the most flexible approach, as discussed in a recent lecture. This decision was taken to speed up development in the short term. That phrase will reappear several times in this report. The decisions we make as engineers cannot always solely be influenced by what we feel is the best product, but sometimes by what product is good enough to make it to the market in time. In this case there was a very clear deadline for “making it to market”.

1.3. Error Handling

The program incorporates some simple error handling. Almost every function has a try/catch block, with any exceptions being handed off to an error handling function. The same error handler is used on both the server and client, to provide a consistent look and feel for the users and maintainers of the game. At this beginning stage of game engine development, any error causes the program to terminate immediately, to speed up debugging. While error handling was not specifically required in this homework assignment, it is good practice to always include some explicit error handling.

1.4. Collections

The program uses CopyOnWriteArrayList collections, because they are thread safe. They are not the most efficient collections for lookup or update of elements. This decision was taken to speed up development in the short term, as array lists are quite simple to understand.

1.5. User Tips

The program prints out game play tips for the user on the client terminal. The program prints out admin tips (well, one tip - how to quit) for the admin on the server terminal.

2. Section 2: Network Basics

2.1. GUIDs

Every game object has a GUID. Clients are identified by their player object’s GUID. This decision enables some common code for finding and handling objects by their GUID. One downside to this design decision is that you see things on the terminal like “Client 4 has joined the game!” for the first client to join. Player objects are already sorted by the order in which they joined. In order to refer to Clients 1, 2, 3, etc., we need only refer to them by their index within the player object collection (plus one) when communicating to the user. This was not implemented in this version of the game, but could easily be added in a future version.

2.2. Data Passed Between Client and Server

For this assignment, we may send whatever data we choose, however in future assignments we will be expected to send game objects. To prepare for future assignments, The program sends game objects. There were two obstacles caused by this design decision, but both were overcome quickly due to very helpful class forum posts, one by the instructor, and one by a fellow student. First, game objects were serializable at one point during execution, and not serializable at another point. This was due to a difference in what objects had a PApplet property actually assigned, and when. Making the PApplet property transient solved the issue. Second, objects were, at one point during execution, seemingly sent properly by the server, but what was received at the client was not as expected. This was resolved by resetting the output stream prior to sending.

2.3. Who Does What

Server: Creates all game objects. Tells newly connected clients about all game objects (including other player objects, and the client's own player object). Updates clients whenever another client's player object has been updated (added, removed, or moved around on the screen).

Client: Keeps local copies of all game objects. Watches for keyboard commands. Tells server about changes to the client's own player object. Renders all game objects to the screen.

Rationale: This was an intuitive decision. To me, the client is the player, and the player looks at the screen - the server is just there to keep everyone coordinated.

2.4. Processing draw() Function as Game Loop

The draw() function, called repeatedly by Processing, is used as the main game loop on both the client and the server. On both the client and the server, this enables easily acting upon keyboard commands, and provides a convenient place to perform synchronous communications. On the client, this is actually used to render game objects to the screen. On the server, no game objects are rendered to the screen, but that option is there if it should be needed in the future. The only obstacle that presented itself in this design was some initial confusion about how to work with noLoop() to facilitate a graceful shutdown.

2.5. Procedural Content Generation

The program performs a very simple kind of procedural content generation, in that it randomly sizes and places into the game world a few obstacles. While not required for this homework assignment, it is interesting, and it aided in testing collision detection.

2.6. Graceful Shutdown

The program, on both the client and server side, allows the user to explicitly shut down. The program ensures (at least to the level that I tested it) that clients respond appropriately to a server shutdown, and that the server responds appropriately to client shutdowns. Shutting down the server is a normal activity, and should be handled gracefully by both the client and the server. Event more so, a shutting down a client is a normal activity, and should be handled gracefully by both the client and the server. This should be especially helpful if future assignments involve more clients or more servers.

3. Section 3: Putting it all Together

3.1. Distinguishing Among Arbitrary Number of Clients (User)

In order to help the player distinguish their avatar from an arbitrary number of other players' avatars, the client colors its own avatar in a unique color. This was necessary for testing and debugging. This could be expanded upon in the future, to allow such things as avatar customization, halo effect, etc.

3.2. Handling Arbitrary Number of Clients (Client)

3.2.1. Any client participating in the game must react appropriately to all of the following situations:

Another client joining the game
Another client leaving the game
Another client's player object moving around within the game world.

3.2.2. The client handles this in a very simplistic way.

When it receives a batch update from the server indicating the position of all player objects within the game, it essentially forgets everything it knows, and accepts the new information as a whole. This is not the most efficient strategy from a memory allocation / copying perspective. This decision was taken to speed up development in the short term.

3.3. Handling Arbitrary Number of Clients (Server)

As noted in the previous section, the server accepts connections from new clients, and communicates to existing clients about the addition. The server does this in a separate thread, so as to minimize the effect on game loop timing.

4. Section 4: Asynchronicity

4.1. Synchronization Objects

Synchronization was done at two levels, on both the client and the server.

Write threads waited to be woken, and were notified, by synchronizing on the applicable output stream object. This was a convenient and understandable object to use, since each write thread communicates on exactly one stream.

Larger blocks of code were synchronized on the client or server class itself. This was done to make the code very explicit to the reader (as was noted in the sample server code we reviewed in lecture), and to ensure that items used by multiple threads were used in an orderly fashion. There was a debugging process with this, as the first pass at the code had threads waiting inappropriately in some circumstances. As the game engine is refined in future assignments, it is possible that what belongs inside a synchronized block could be fine-tuned (reduced slightly), however for this assignment, robustness was the main goal.

4.2. Processing Versus Multi-Threading

For this phase of the assignment, both the client and the server need to have threads that do unique things (accept, read, write). For this purpose, the class constructor was modified to accept arguments, as in the Fork Example provided by the instructor. A second class constructor with no argument was needed, to satisfy the Processing method call that starts a PApplet application - PApplet.main().

4.3. Graceful Shutdown

As mentioned in a previous section of this report, shutting down is a normal activity and should be handled gracefully. In this phase of the assignment, there were multiple threads, on both the client and the server, which could detect that something else out there had shut down. On the client, if the server shuts down, then so must the client. Additionally, both the client and the server detect a keypress locally to shut down. Coordinating this required more careful detection code, and more flags, than were required in the synchronous version. The flags were used to indicate which of these scenarios was already being handled by some other thread, to avoid repeated work and confusing messages on the terminal.

4.4. Asynchronous Summary

The development process for the asynchronous version was more "finicky" and required more careful thought and planning. The benefits of the asynchronous version were very obvious once the program was operating correctly. The responsiveness during game play was outstanding when compared to the synchronous version.

5. Screenshots

5.1. Server terminal after two clients join the game

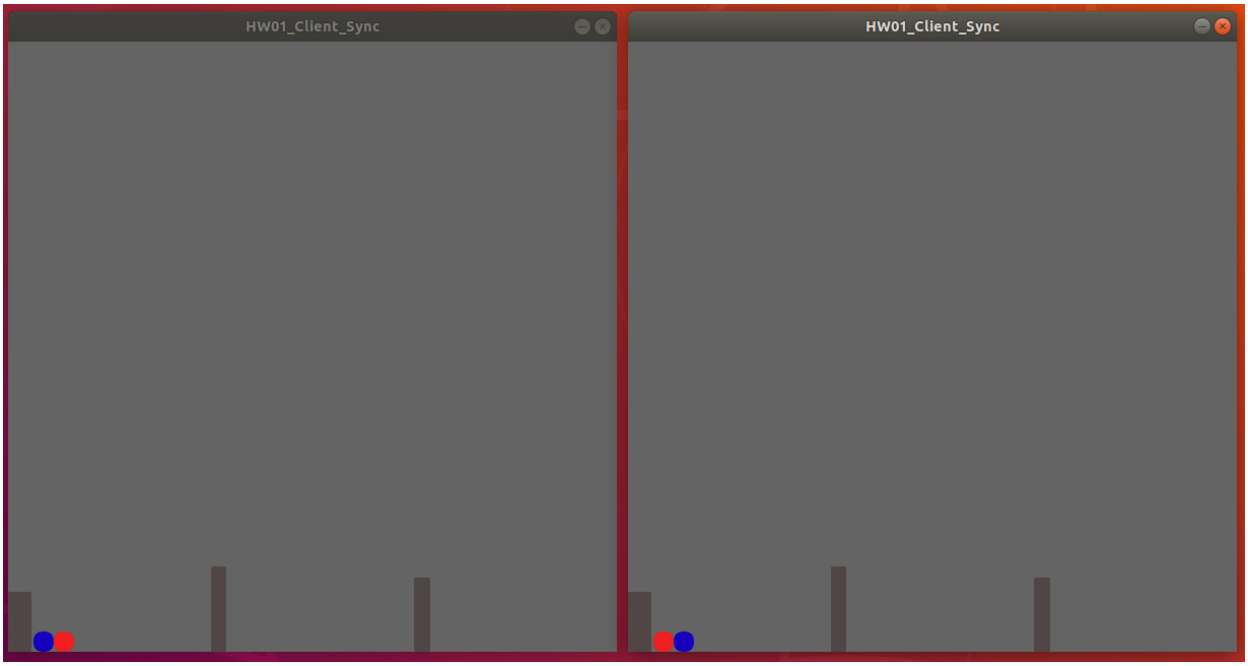
```
Game server is running!
Options (click the window to give it focus first):
        'Q': Quit
Client 4 has joined the game!
Client 5 has joined the game!
```

5.2. Client terminal after joining the game

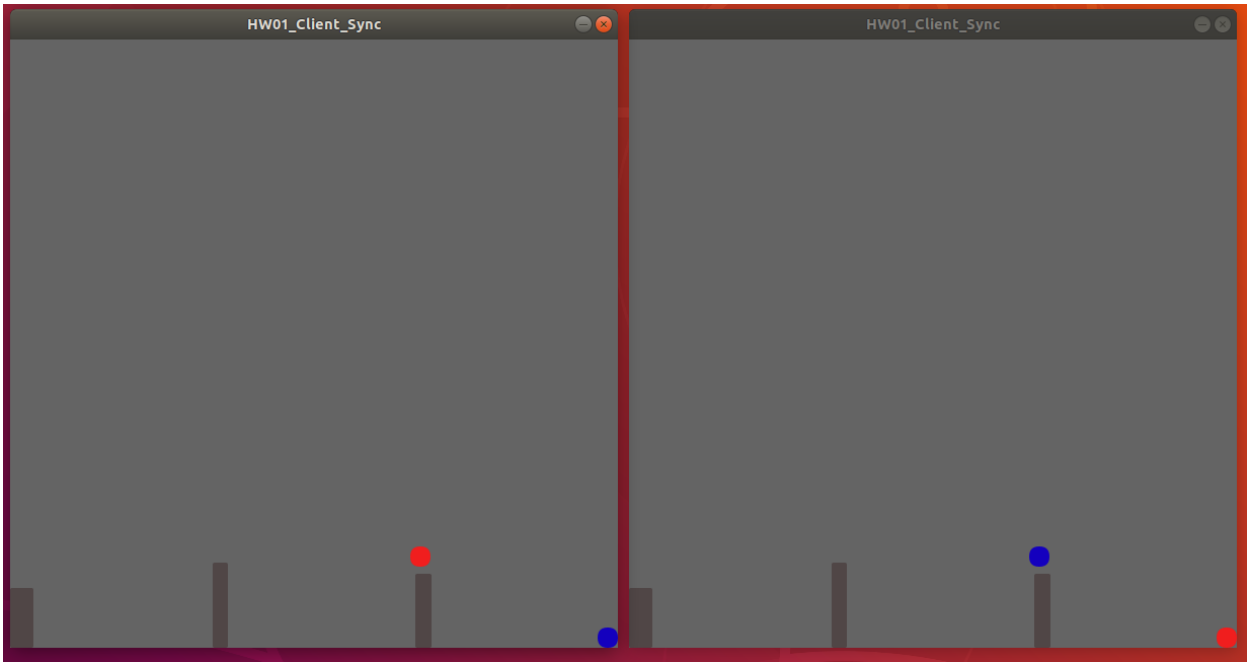
```
Game client is running!

Your avatar is the BLUE one.
Options (click the window to give it focus first):
        Left Arrow: Move Left
        Right Arrow: Move Right
        Space Bar: Jump
        'Q': Quit
```

5.3. Client views after two clients join the game



5.4. Client views after one client jumps onto an obstacle, and the other into the wall



5.5. Server terminal after one client leaves the game

```
Game server is running!
Options (click the window to give it focus first):
        'Q': Quit
Client 4 has joined the game!
Client 5 has joined the game!
Client 5 has left the game!
```

5.6. Client terminal after game server is shut down

```
Game client is running!

Your avatar is the BLUE one.
Options (click the window to give it focus first):
        Left Arrow: Move Left
        Right Arrow: Move Right
        Space Bar: Jump
        'Q': Quit
Game server has shut down!
Game client 4 is leaving the game!
```