CSC 591
Homework 2

Aurora Tiffany-Davis
(attiffan)

1. Part 1 - Game Object Model

1.1. Game Object Inheritance Hierarchy

The program uses a monolithic game object hierarchy. The reasons for this are discussed in the Homework 1 report. This is not the most flexible approach, as discussed in lectures. There were no requirements of Homework 2 that could not be fairly easily accommodated in this game object model, and so it was not changed (although it was of course extended to include new game object types). The object model will be revisited if needed in future assignments. Aside from creating object classes that were conceptually "obvious", a few additional design decisions were made:

- Although all currently extant game objects are located (have x and y coordinates), HW02_ObjectLocated class is distinct from HW02_ObjectGame in order to support possible future game objects that are not located.
- Each game object includes an attribute which explicitly saves the object type as a string. This is included to facilitate easy distinction of objects during de-serialization as objects are communicated between the server and the client(s).

1.2. Collection of game objects

In Homework 1, there were multiple collections of game objects, and these collections were not protected within a class. In Homework 2, this is improved upon. The generic game object class maintains a single collection of game objects. Both the client and the server use the generic game object class to add game objects, get the collection, add or remove or replace game objects by GUID, and cast game objects to more specific types. This change is intended to better control access to game objects, both for thread safety and for code modularity / maintainability. The generic game object class maintains the collection as a ConcurrentHashMap, to get the benefits of thread safety as well as the efficient lookup of a hash map.

1.3. Platforms

Static and moving platforms have more in common than is different between them - for this reason, they both inherit from a generic platform class. A static platform is much the same as a moving platform, except that it moves at a speed of zero.

Platforms are given a randomized shade of a particular color (which color is used depends upon the platform type). Platforms are given a randomized width and height prior to creation. This width and height are randomized within an acceptable range, based on window size and total number of platforms. Platforms are given a randomized location prior to creation. After creation, collision detection is used to move them around randomly until they find a non-colliding spot to start the game in.

1.3.1. Static Platforms - Each of these platforms is a randomized shade of gray.

1.3.2. Moving Platforms

Each of these platforms is given a randomized speed and direction when it is created. It continues to travel at this speed and in this direction until it collides with another collidable object. What happens next depends upon the platform type.

1.3.2.1. Horizontal Motion Platforms - Each of these platforms is a randomized shade of yellow. Upon collision, a horizontal platform stops and reverses course.

1.3.2.2. Vertical Motion Platforms - Each of these platforms is a randomized shade of green. Upon collision, a vertical platform stops and reverses course if it is not carrying a character object, or if it is not carrying that character object up, or if it is carrying a character object up but continuing up would "squish" that character object.

The randomization in platform generation, and in other areas of the code, is intended to improve gameplay. It has an additional advantage, which is that it aids in testing and debugging - it helps to find edge cases that the developer might not have thought of.

2. Part 2 - Multithreaded, Networked, Processing Sketch

2.1. Multithreaded Server and Client

The server runs 1 thread for the game loop, 1 thread to accept new client connections, and 2 threads per connected client (write and read). The client runs 1 thread for the game loop, 1 thread to write to the server, and 1 thread to read from the server.

2.2. Data Sent Between Server and Client

The data sent is always a collection of game objects (or, as discussed later in Part 4, game object proxies). Specifically, the collection represents those game objects that have changed since the previous game loop iteration.

The client sends a collection of one object if the client's character object has changed in some meaningful way. Otherwise the client sends no data. This minimizes traffic and minimizes work performed by the server (and downstream work performed by other clients). The server, if it receives anything from a client, always treats this as a changed object, and so replaces its local representation of this object.

The server usually sends a collection of game objects representing moving platforms. When a client first connects to the server, the server sends a collection of all game objects. When a client disconnects, the server sends that client's character object to all other clients with a removal flag set. When the client receives a collection from the server, it needs to make a decision about each object it receives - does it represent a new object (not already present in client's collection), a changed object (already present in client's collection and removal flag not set), or a removed object (removal flag is set)?

2.3. Graceful Shutdown

The program, on both the client and server side, allows the user to explicitly shut down. As discussed in Part 4, the server may also automatically shut down after performance testing.

If the server shuts down, the clients detect this via errors returned from communication attempts. The client then shuts down as it would if shut down by the user (stops game loop, closes streams, exits Processing sketch, exits program).

If a client shuts down, the server detects this via errors returned from communication attempts. The server then closes the communication streams to this client, removes the client's character object from the game, and instructs other clients to do the same.

If the server is explicitly shut down by the user, it stops the game loop, closes streams, exists the Processing sketch, and exits the program.

2.4. Distinguishing Among Arbitrary Number of Clients (User)

In order to help the player distinguish their character object from an arbitrary number of other clients' character objects, the client colors its own character object in a unique color, and all character objects have their player ID displayed. The player ID is globally unique but is limited to character objects, so is not the same as the object GUID. The player ID exists to avoid user confusion that may otherwise arise with non-sequential or very large ID numbers.

3. Part 3 - A 2D Platformer

For this part of the assignment, the game object model discussed in Part 1 is extended to include several more object types.

3.1. Characters

The character object type was already present from Homework 1. Collision handling was improved to handle collisions with all of the object types supported in Homework 2. A character may not leave the bounds of the Processing window, nor jump or fall through platforms, nor be obscured by other objects, nor fail to spawn, nor fail to react to contacting death or win zones. A character standing on a moving platform must move with the platform. All of these behaviors are fundamentally collision handling behaviors. Getting collision handling correct, with moving non-character objects, and implemented in a maintainable way, was one of the most challenging aspects of Homework 2.

A character may be simultaneously collided with several other objects, and so a preference ranking was implemented:
Platform > Death Zone > Win Zone > Character > Boundary

3.1.1. Platform > Death Zone: If a character is collided with both (e.g. it is falling, and in one game loop iteration moves to the bottom of the window where there is also a platform), then the character is allowed to stand on the platform instead of dying.

3.1.2. Death Zone > Win Zone: This should not be possible as the death zone and win zone are disjoint, but this preference is established in case these zones are not disjoint in some future version of the game. If a character hits both zones in one game loop iteration, they die.

3.1.3. Win Zone > Character: If multiple characters hit each other and the win zone, then they both score a point.

3.1.4. Character > Boundary: If multiple characters hit each other and a boundary, the character objects first must get out of each other's way so that one character object does not obscure another.

3.2. Boundaries

A boundary object type was added to standardize how objects are kept within the bounds of the Processing window. With boundaries drawn on all 4 edges of the window, the task of keeping objects in bounds can be implemented in a standardized way via collision handling. This is considered preferable to having special code for dealing with window edges that needs to be maintained if window size is changed or if new object types are added.

3.3. Spawn Points

In order to keep a character from spawning and immediately falling to their death, a "safe" spawn point is desirable. Therefore, each static platform has an accompanying spawn point just above it. When a character spawns (or re-spawns after winning or losing a point), they are assigned one of the existing spawn points, chosen at random. In Part 1, pseudo-randomization of platform width and height is discussed. With spawn points being associated with static platforms, this pseudo-randomization needs to be guided a bit. If there are very few (defined as < 10) static platforms, then each has a minimum width large enough to accomodate 3 character objects. Without such a minimum, it is much more likely that a character will spawn and end up teetering on the edge of a static platform, only to get immediately "picked up" by a moving platform. This could be confusing for players and so the risk of this is mitigated (although not entirely avoided).

3.4. Death Zone

The death zone is established at the bottom edge of the Processing window, for a "the floor is lava" game mechanic, which many people are familiar with from childhood games.

3.5. Win Zone

A win zone is established just beneath the scoreboard at the top edge of the Processing window, to give players something positive to strive for to keep them engaged in the game.

3.6. Scoreboard

A scoreboard displaying the current score of all players in the game is established at the top edge of the Processing window. This is included to make testing of the death and win zones easier, and to provide incentives to players.

4. Part 4 - Performance Comparison

4.1. Network Protocols

The primary network protocol is to send game objects between the server and the clients. The details of this are discussed in Part 2. Here, a second network protocol is added which uses .writeReplace() and .readResolve().

In .writeReplace(), game objects are replaced by proxy objects. Each proxy object has only one attribute - a string describing the object. Every game object, no matter what kind of object it is, must be replaced with a string. In order to keep the code modular, all of this logic lives in the HW02_ObjectGame class, and none of it trickles down to descendent classes. The string is built by appending only relevant information. The string does not necessarily include information about all attributes of the original game object. For example, all renderable objects have information about color appended to the string. However, nobody other than the client which owns a character object cares whether or not that character is currently jumping - this does not affect how they render the character onscreen or deal with collisions. Therefore information about jumping status is not appended to the string.

In .readResolve(), proxy objects must be turned back into game objects so as not to affect the rest of the existing code.

- Parse the proxy object string, pulling out all useful information
- Create a new game object of the correct type, bypassing the normal auto-GUID to get a specific GUID
- Set any object attributes that may differ from the object creation defaults (e.g. color, speed)

From this point forward, the client and server treat the object returned by deserialization via .readResolve() exactly as they treat the object returned by deserialization in the primary network protocol. This treatment is discussed in Part 2.

4.2. Performance Test Design

A Server game loop performance test comprising 38 scenarios was devised to exercise the game along several dimensions:

- # Clients: 2 or 3
- Network Protocol: send game objects, or use .writeReplace() and .readResolve()
- # Static Platforms: 1, 10, 100, 300, or 500
- # Moving Platforms: 1, 10, 100, 300, or 500

In each test scenario, the server is started, and the server game loop is then started at an extremely high nominal frame rate (10,000 versus the normal 60). If the server game loop were only run at 60 FPS, then performance differences caused by any of the dimensions noted above would only be detectable if they slowed the server down so much that it was not able to run at 60 FPS. The goal here is to run the server as fast as possible, to find out just how fast that is.

After a waiting period, the clients are started. Each client then saves a screenshot. The server starts counting game loop iterations after the first client connects. The server then completes 10,000 game loop iterations. Finally, the server prints out performance measures and exits. This causes the clients to also exit gracefully as discussed in Part 2.

The code was updated such that performance tests can be run by providing arguments to the program when it is started. If no arguments are provided, the code runs using reasonable defaults (primary network protocol, small number of platforms, standard frame rate, no printing of performance results). This flexibility was added to the code for two reasons. First, to allow for easy sanity checks if any problems were encountered during performance testing, by running the exact same code in no-performance mode. Second, to ensure that any improvements made in the code during performance test debugging would apply to no-performance mode as well.

Running performance tests, which involve extreme timing / data structure access scenarios, revealed several issues with the code that had not been revealed in more casual testing - so this was an extremely valuable exercise. Some examples:

- Failure to synchronize some work done on data structures, causing threads to step on each other
- Synchronizing other work too aggressively, or inappropriately, causing deadlock
- Heavy dependence on CopyOnWriteArrayList collections, causing memory issues

These issues were worked through until all performance test scenarios could complete successfully. There is still some dependence on CopyOnWriteArrayList collections, which may be addressed in the future by continuing the shift toward ConcurrentHashMap collections.

4.3. Performance Test Results Summary

Of the 38 test scenarios, 36 were successful and are included in the results. The 2 unsuccessful scenarios were those using 3 clients and 500 moving platforms. At this extreme end of testing, the clients experienced memory issues and shut down - this is why these results were excluded from the Server performance test results.

Server performance with 3 clients was just a bit slower than Server performance with 2 clients. More clients put more load on the server as the server must update all clients every game loop iteration regarding the movement of game objects such as moving platforms, and the movement of character objects. The difference in # clients likely would have been more noticeable if the character objects had actually been moving during performance testing. As noted in Part 2, a client only updates the server if the client's character object has changed in some meaningful way.

There was no clear trend in Server performance based upon # static platforms. Any differences here are likely "noise". This is because the Server does not update clients about unchanging objects (such as static platforms).

There was a very clear trend in Server performance based upon # moving platforms - the higher the #, the slower the game loop. This is because the Server updates clients at the end of every game loop iteration regarding changed objects (such as moving platforms). In fact, aside from updating clients about other clients' character objects, this is the main job of the Server each game loop iteration.

There was only a very slight difference in Server performance based upon network protocol. The secondary network protocol, which uses .writeReplace() and .readResolve() to communicate game object proxies rather than original game objects, is a bit slower. This makes sense, because the object/proxy replacement work takes non-zero time to accomplish. It may be the case that the messages between Server and Client(s) arrive more quickly after they are sent, with the secondary network protocol, but that was not the point of measurement in these performance tests.

5. APPENDIX

5.1. Performance Test Results

5.1.1. Example results from one test scenario

PERFORMANCE TEST RESULTS	
Network Protocol:	Use .writeReplace() and .readResolve()
# Static Platforms:	1
# Moving Platforms:	500
# Clients:	2
# Total Game Objects:	511
Requested # measured iterations:	10000
Actual # measured iterations:	10000
Time to Initialize (s):	0.469
Time to Wait for Clients (s):	10.379
Time to Run Game Loop (s):	2306.184
Time per Game Loop Iteration (s):	0.23061840000000003

5.1.2. Results Raw Data

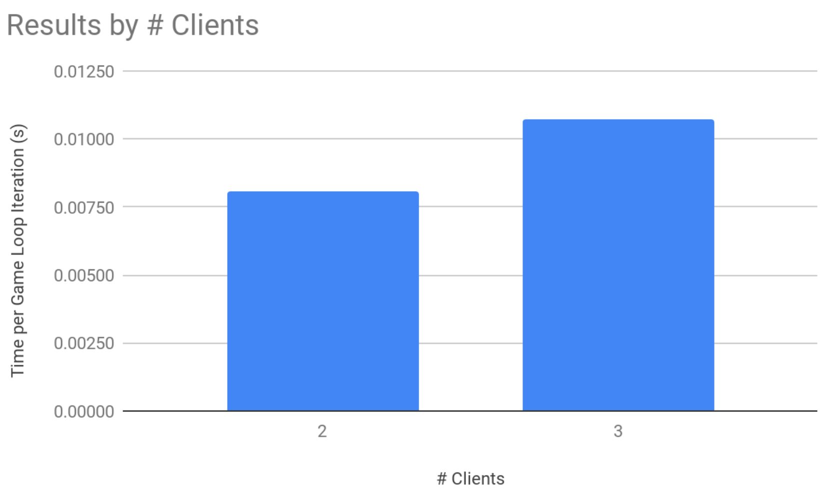
All measurements are in Time per Game Loop Iteration (s).

# Static Platforms	# Moving Platforms	2 Clients Send Game Objects	2 Clients WriteReplace	3 Clients Send Game Objects	3 Clients WriteReplace
1	1	0.00073	0.00083	0.00101	0.00096
1	10	0.00115	0.00126	0.00122	0.00101
1	100	0.00880	0.00513	0.00938	0.00833
1	300	0.05193	0.05243	0.06037	0.08142
1	500	0.22051	0.23062	memory issues	memory issues
10	1	0.00071	0.00097	0.00082	0.00096
100	1	0.00076	0.00080	0.00061	0.00084
300	1	0.00081	0.00075	0.00127	0.00042
500	1	0.00098	0.00109	0.00159	0.00122

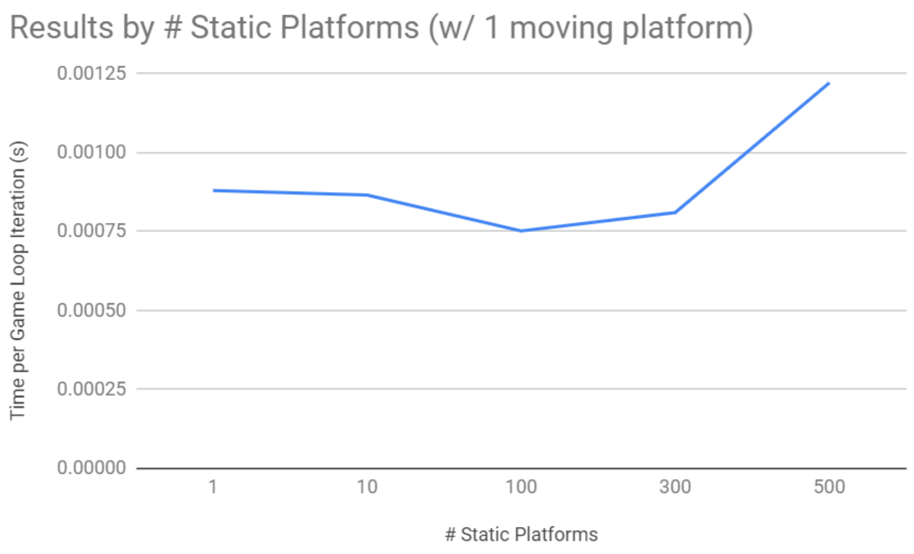
5.1.3. Overall Average Comparisons

Note: because the Server performance results for test scenarios with 3 clients and 500 moving platforms are excluded due to memory issues on the Client, the results for 2 clients and 500 moving platforms are also excluded. The only exception to this is the comparison of Server performance results for # Moving Platforms.

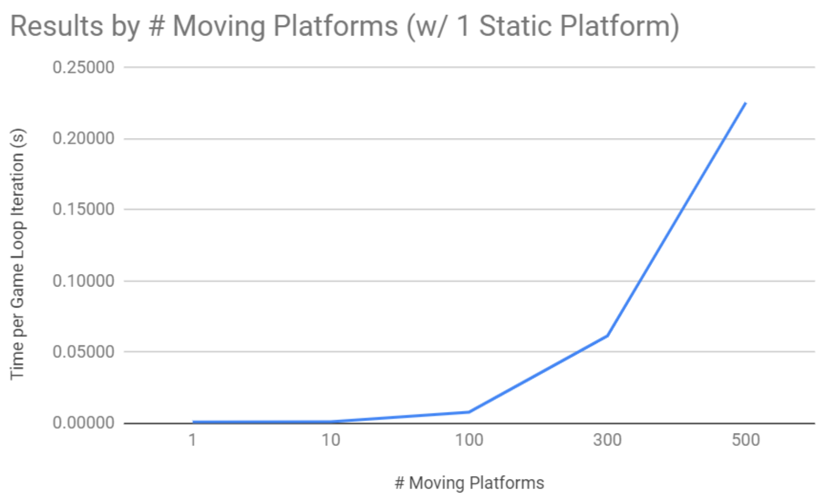
5.1.3.1. # Clients



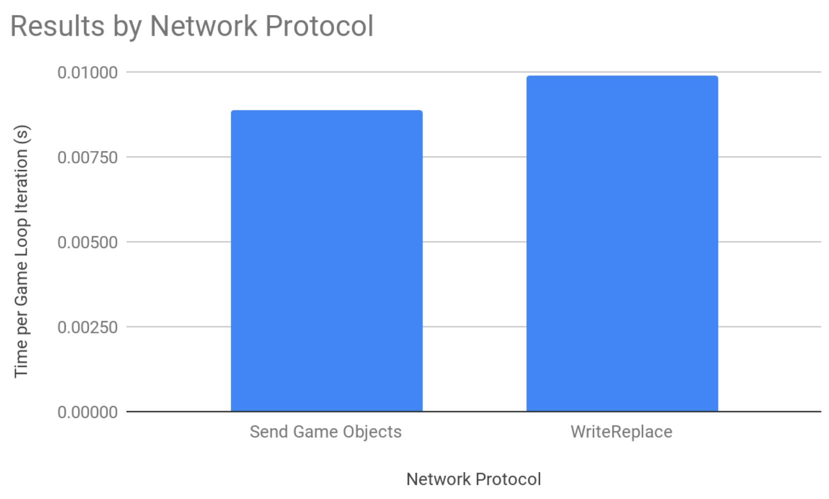
5.1.3.2. # Static Platforms



5.1.3.3. # Moving Platforms

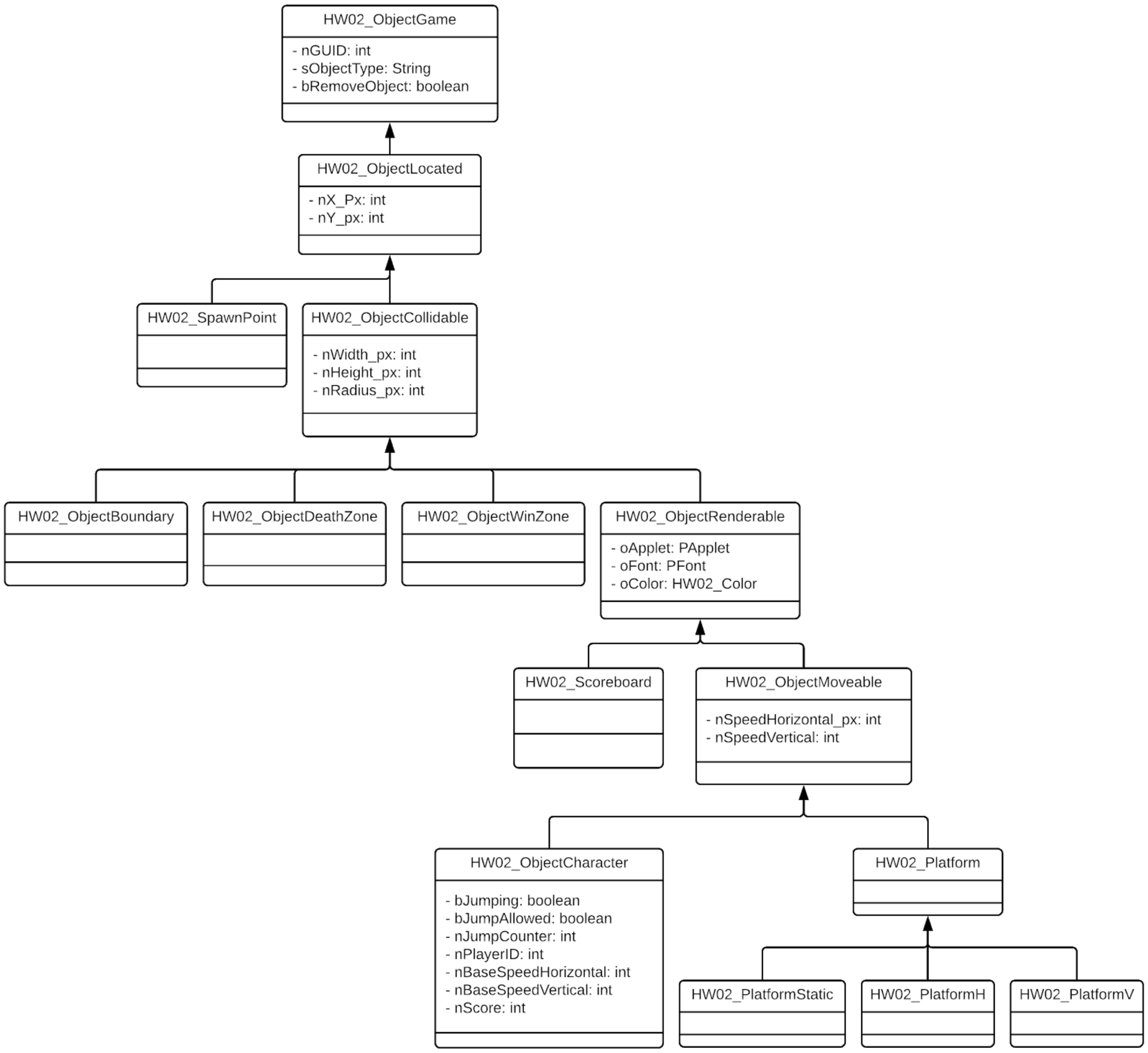


5.1.3.4. Network Protocol



5.2. Class Diagram

Note: This simplified class diagram does not show static attributes, and does not show methods

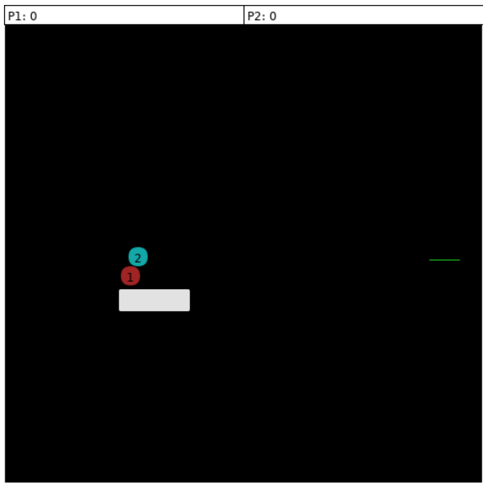


5.3. Screenshots

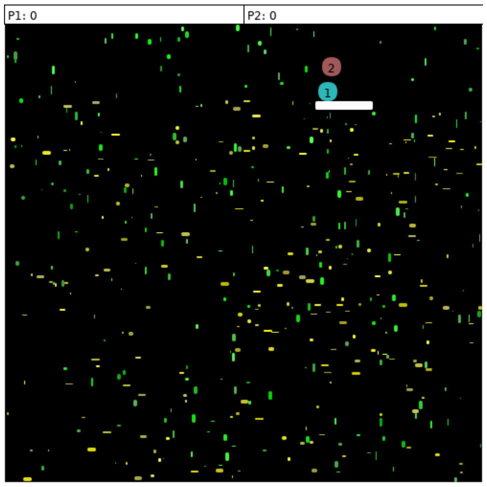
5.3.1. Gameplay

These screenshots represent just a few of the many performance test scenarios.

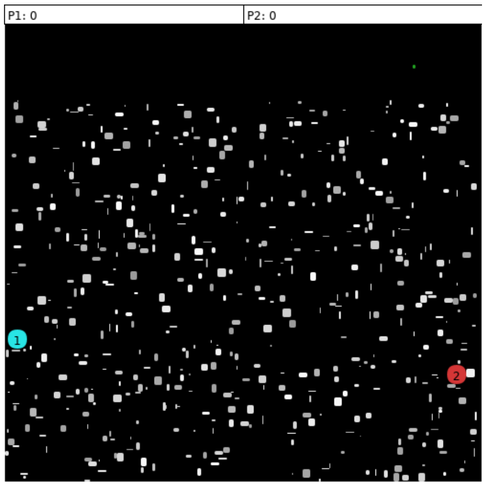
5.3.1.1. 2 Clients, 1 Static Platform, 1 Moving Platform



5.3.1.2. 2 clients, 1 static platform, 500 moving platforms

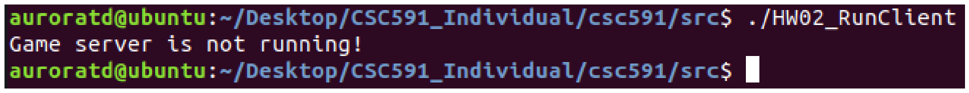


5.3.1.3. 2 Clients, 500 static platforms, 1 moving platform



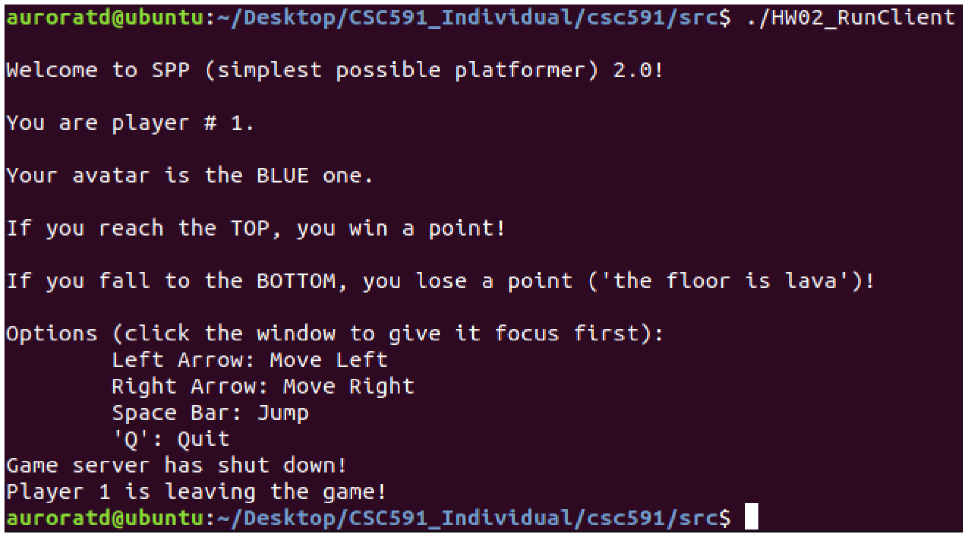
5.3.2. Shutdown Scenarios

5.3.2.1. Client Terminal (Start Client Before Server)



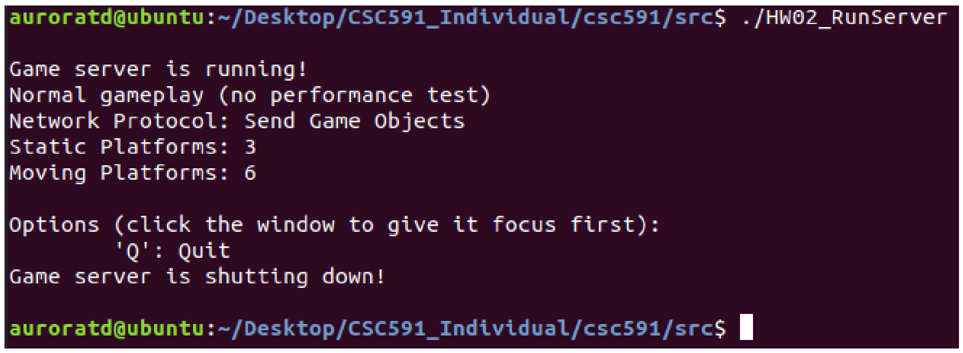
```
auroratd@ubuntu:~/Desktop/CSC591_Individual/csc591/src$ ./HW02_RunClient
Game server is not running!
auroratd@ubuntu:~/Desktop/CSC591_Individual/csc591/src$ 
```

5.3.2.2. Client Terminal (Start Server, Start Client, Stop Server)



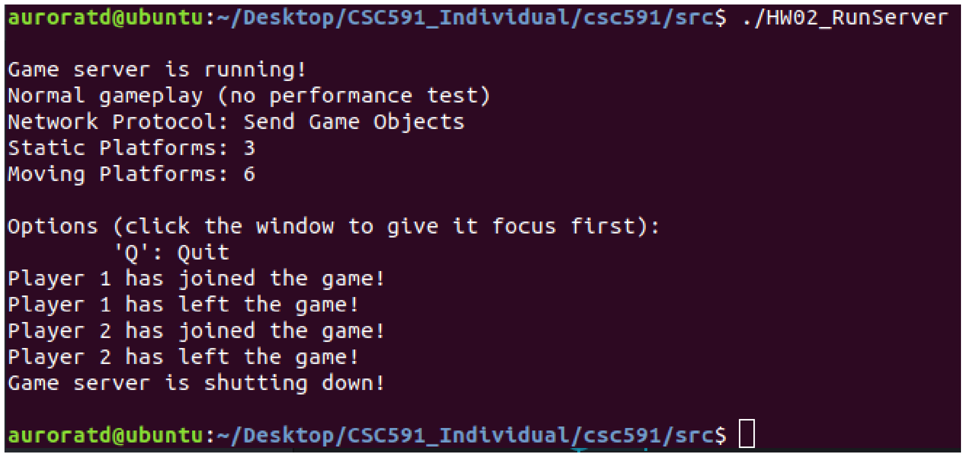
```
auroratd@ubuntu:~/Desktop/CSC591_Individual/csc591/src$ ./HW02_RunClient

Welcome to SPP (simplest possible platformer) 2.0!

You are player # 1.

Your avatar is the BLUE one.

If you reach the TOP, you win a point!

If you fall to the BOTTOM, you lose a point ('the floor is lava')!

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!
Player 1 is leaving the game!
auroratd@ubuntu:~/Desktop/CSC591_Individual/csc591/src$ 
```

5.3.2.3. Server Terminal (Start Server, Stop Server)



```
auroratd@ubuntu:~/Desktop/CSC591_Individual/csc591/src$ ./HW02_RunServer

Game server is running!
Normal gameplay (no performance test)
Network Protocol: Send Game Objects
Static Platforms: 3
Moving Platforms: 6

Options (click the window to give it focus first):
        'Q': Quit
Game server is shutting down!

auroratd@ubuntu:~/Desktop/CSC591_Individual/csc591/src$ 
```

5.3.2.4. Server Terminal (Start Server, Start Client, Stop Client, Start Client, Stop Client, Stop Server)



```
auroratd@ubuntu:~/Desktop/CSC591_Individual/csc591/src$ ./HW02_RunServer

Game server is running!
Normal gameplay (no performance test)
Network Protocol: Send Game Objects
Static Platforms: 3
Moving Platforms: 6

Options (click the window to give it focus first):
        'Q': Quit
Player 1 has joined the game!
Player 1 has left the game!
Player 2 has joined the game!
Player 2 has left the game!
Game server is shutting down!

auroratd@ubuntu:~/Desktop/CSC591_Individual/csc591/src$ 
```