3110 Final Project Design Document

Caleb Mok, Talha Baig, Alan Gao, Newton Ni

November 15, 2017

1 System Description

1.1 Core Vision

We plan to make a Battle Royale style game where players fight each other on a map until only one remains. The game will be in 2D, and players will control their avatar from a bird's eye view.

1.2 Key Features

1. A Lobby where players can choose which game to join
2. Realtime combat between different players
3. A GUI
4. Client/Server architecture

1.3 Narrative Description

When a user opens the main site, they're prompted to create a temporary display name. Then they'll be directed to the main lobby, where they can select a game. Players can only see and join games that haven't started yet (i.e. they don't have 4 people). Once they join, they spawn empty-handed at random locations through the map. Various weapons, ammunition, and health packs will drop initially, that players will have to search for and pick up. Different weapons will have different properties, like rate of fire, bullet velocity, and bullet spread. As the game goes on, a "wall of death" that encompasses the map will shrink, forcing players to congregate in the center and increasing tension.

2 System Design

2.1 Client

The client will be composed of two separate modules: the GUI and Router. The GUI will contain all of the code that displays the interface to the user. None

of the methods in the GUI should be exposed to other modules. The Router module facilitates communication between the GUI and Server. It will be used by the GUI to create and send HTTP requests. Once these requests are fulfilled, the router will parse any returned JSON data and pass it back to the GUI. Each method exposed in the router corresponds to one of the API calls; we expect these methods to be used primarily in the GUI.

2.2 Server

The server module will not have any exposed methods, as communication between the client and server is done through HTTP. We plan on only providing the server with access to the game model. This means that no client-side simulation of the game will be done. This design choice was made due to the nature of multiplayer shooting games - it is nearly impossible for each individual client to simulate other players without help from the server. Thus, to ensure that the game is synced for all players, clients will request the game state approximately 30 times a second from the server, although the exact number is still uncertain.

2.3 HTTP

Our game will be designed as a client/server model, where the client communicates with the server via HTTP requests, using JSON to transfer data. We chose to design our system this way to facilitate multiple players connecting to the same game. As for division of labor, the client is responsible only for rendering the images on the user's screen; the server will handle all changes to the game state/model.

The exact API can be found here: INSERT LINK HERE

2.4 Game Model

The game model is the actual simulation of the game itself. This is where all of the game logic will reside. The exposed methods in the game model will be used by the server only.

2.5 MVC

We will aim to adhere to MVC principles when building our systems. This means a clear division between the different parts of our system:

- The game model will communicate with the server and the server only.
- The server communicates with both the game model and router. Whenever the server receives a request from the router, it will perform any necessary computations with the game model and send the response back to the router.

- The router communicates with both the GUI and server. Whenever the player does something to alter the game state, the GUI notifies the router to send a HTTP request to the server. Once a response to the request is received, the router will return the information back to the GUI.
- The GUI will communicate only with the router.

3 Module Design

See interfaces.zip.

4 Data

In this section, we provide the first iteration of the main data models that we will use. All of the following models are records.

4.1 Player

```
player:
{
  uid: string;
  inventory: string list;     //This will be item ids
  location: int*int;
  health: int;
  direction: int OR variant    //N, E, S, W, NE, NW, SE, SW
}
```

This model represents a player in a game. Each player will have a unique uid, which also corresponds with their username.

4.2 Item

```
item:
{
  iid: string;
  damage: int;
  projectile speed: int
  fire rate: int
}
```

Currently, items only represent firearms, although this may be extended in the future.

4.3 Game State

```
location: {x: int; y: int}

state:
{
  players: player list;
  items: item list;
  elems: string*location list;
  world boundaries: int; //represents radius
}
```

Elems represents a list of all items that are not currently in any player's inventory. Items is a master list of all the items and their attributes currently in the game, and players is a master list of all players in the game. We currently intend on the world being a circle, hence the world boundary is simply an integer representing the radius.

4.4 Data Structures

In the models we've outlined above, we make heavy use of lists. However, in some cases, especially the elems list in game state, we may benefit from using a Map instead. This is something we will decide on during implementation. The reason we don't use a map is two-fold:

1. The number of elements in each list isn't going to be large enough for a map to make a huge difference
2. Converting the state to JSON will be done easier if the data is stored in lists, as opposed to in a map

4.5 Storage

As mentioned above, all of this data will be used in the game model, and ultimately 'stored' on the server.

5 External Dependencies

We plan on using two external libraries:

- For HTTP requests, we will use ocaml-cohttp: https://github.com/mirage/ocaml-cohttp
- For the GUI, we will use lablgtk: http://lablgtk.forge.ocamlcore.org/
- To parse the JSON, we will use good old Yojson.

6 Testing Plan

We will test our systems incrementally as we build them. We divide our work into two parts: the client/GUI, and the server/game model.

6.1 Router/GUI

The client/GUI will be significantly harder to test than the backend. Testing of the GUI will be done first, and testing will occur by actually using the GUI. The router will be tested by simply checking to see whether the HTTP request that was sent is the correct one.

6.2 Server/Game Model

The game model will be completed first, as the server requires the model to work. Testing the game model will involve writing basic OUnit test suites to ensure their validity. We plan on implementing the game model one function at a time, and tests will be written before the completion of each function.

To test the server, we will use an application called Postman, which sends HTTP requests to urls and graphically displays the response. We plan on developing the server one API call at a time, and once an API call is completed, we will test it using Postman.