3110 Final Project Design Document

Caleb Mok, Talha Baig, Alan Gao, Newton Ni

December 7, 2017

1 System Description

1.1 Core Vision

We have to made 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. The camera system remains true to Boxhead zombies where the main player is at the center of the view. There is a caveat though which is a ring of death encompasses and slowly grows as the game progresses as to force the players not to camp.

1.2 Key Features

1. A Lobby where players can choose which game to join
2. Realtime combat between different players (collision detection algorithm)
3. A GUI (event handling and repainting the canvas)
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 (or create) 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 and ammunition will drop initially, that players will have to search for and pick up. Different weapons 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 and Implementation

2.1 FrontEnd

The client is composed of two main components: the GUI and Router. The GUI contain all of the code that displays the interface to the user. None of the methods in the GUI will 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. These methods are implemented by the router.

2.1.1 Router

The router essentially implements the API for the front end. In our implementation we only used GET and POST requests. The router fetches the world state from the backend, and sends post requests to the world whenever the state of the world changes from an event of the GUI. As a design decision, the router, factors out all the code of using cohttp into one function called makeNetworkRequest and from there all the functions use variants to distinguish themselves from another to sort out whether the request is a POST or a GET. The function makeNetworkRequest maps from the response to a Yojson, and then can map again to a type of the choosing as long as a parsing method is provided. To use this response which can be whatever type (according to the map parameter in makeNetworkRequest), a callback parameter is provided for each method of the router.

2.1.2 GUI

We went through several implementations of the GUI. First we started with the Graphics package, but did not like how it was not that flexible and switched over lablgtk. However, we soon realized that not only was lablgtk impossible to download, but also was a library with an extremely steep learning curve. We then switched back over to graphics. However, there was a problem with graphics in that it would not take keystrokes well such that the strokes were very choppy and not smooth. We then thought about polling after every few milliseconds, but the problem with this was that graphics does not deque the key meaning that it would be stuck forever in that key when pressed. Thus, we had a major hack (that we are super proud of). We used the SDL library event handler, however this meant that we had to have an SDL GUI open. Thus, we created an SDL GUI of 1x1 pixel, and used that event handler to handle our key strokes :). The GUI also used a draw module, to execute the drawings with the main invocation continuously redrawing based on whether it got a request or not. We also have a lobby page that allows players to choose which lobby they want to go to.

2.2 Backend

2.2.1 Server

The server module does not have any exposed methods, as communication between the client and server is done through HTTP. Only the server has 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 30 times a second from the server. Implementaton wise, the server had to use the cohttp library so that the client could access the game model through a very restricted API. The server also had to do conversion of types into JSON objects in order to send information over to the client.

2.3 Game Model

The game model is the simulation of the game itself. Many interesting features and implementations are noteworthy in this model. There is the collision detection algorithm which checks the diagonals of the objects and sees if there exists a collision. It was implemented using a difference hash, and a D-Tree. This is vital to see if a bullet hit a player. There also exists somewhat of a bullet language. This allows us to easily engineer which attributes a bullet has, how it accelerates, how its damage changes over range/time, and even if multiple bullets can be spawned.

2.4 HTTP

As seen above, our game was 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: https://apex3110.docs.apiary.io/

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

Our goal was to keep not only the view and model separate, and by extension the client and server, but also keep all the modules completely separate from one another, and try to keep the .mli files short and have a narrow interface. We did this in the frontend, by keeping the types and associated json parsing functions in a separate module, then the router was also a separate module. The lobby page of the GUI and the main game page were also separate modules in of themselves. The drawing of the main GUI page was also separate from the main GUI loop. For the backend, we created the types in the module Stypes, and the server, handling the cohttp, itself was a separate module. The game model was also separated into different models. The State module was the main interface to the game. This module had to rely on the Collision Module for the collision detection. There is also the Ammunition module that has our engineered bullets.

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;     // item ids
  current weapon: string
  location: int*int;
  health: int;
  direction: ivariant    //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; // 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 four external libraries:

- For HTTP requests, we will use ocaml-cohttp
- For the GUI, we will use graphics and sdl
- To parse the JSON, we will use good old yojson

6 Testing Plan

Each of us first implemented our separate modules first. We then tested each of our modules in the respective manner. Newton built unit tests for his collision detection algorithm and and for whether his bullet language would work as well. Alan built the server, so he used a variety of postman requests to test the server according to the API we built on apiary. Then, Talha had the client, and tested each of the individual json parsers on utop with dummy data and then once Alan was finished, both checked if the API was a sufficient replacement to Postman. Caleb then checked the lobby to see if that worked by interfacing it with the game GUI. White box testing was used, as there were only so many ways to interact with the lobby. After we ensured that the server and client could talk to one another, we made the decision to debug the server using the client. This may not have been the best idea, but as a justification, for a game, unit tests are extremely difficult to make and this seemed as the most efficient albeit not most robust methodology of testing the server. The trouble we ran into is that we owould not know if bugs were from the GUI or game model, and this did result in some lsat minute panic, but thankfully everything settled down.

Known bugs: The move request is not working The join request does not work for large amount of inputs.

7 Division of Labor

Newton Ni: Did the entire game model including collision detection, respresenting information, and the engineering bullets model. Also, worked on GUI. 50 hours.

Alan Gao: Built the server and its according types including JSON parsing. Pair Programmed with Talha on majority of the GUI. Instrumental in debugging and getting types to compile. 30 hours.

Talha Baig: Built the client and its according types including JSON parsing. Pair Programmed with Alan on majority of the GUI. Worked on getting code to compile, creating make files, installing packages and sending around http requests for testing: 30 hours.

Caleb Mok: Built the initial lobby and then did the event handling for the GUI. Did a majority of the work in deciphering GUI API, 20 hours.