CS 428 Final Documentation
VirtualBoard

Kai Huang
Matthew Hoffman
Jonathan Chao
Evan Gryska
Kaushil Patel
Daniel Pugliese
Da Shen
Madeleine Walstad

May 1, 2016

Contents

1 Overview

VirualBoard is a browser based board game simulator. It isn't limited to a single game, or even a set of games, but instead allows the players to perform any action that would be possible with a physical board game. Combine this with VirtualBoard's capability to have new game pieces added to it, and the players can recreate any existing board game - or even make a new one.

Once on the homepage, users can navigate through existing game lobbies or decide to host a new one. Lobbies can be made either to be open to the public, or protected with a password. Within each game, lobby players are divided into teams by the color they choose for their avatar, and they can manipulate the pieces on the board just like real objects.

In addition to a wide array of standard game pieces, there are a number of specialty pieces built into VirtualBoard. These include decks, cards, dice, timers, and notepads. Cards are able to be placed either face up or face down, and can be combined into a deck, which can in turn be shuffled to add an element of randomness to the games. Of course dice also allow for random outcomes, and they can be created with any number of faces up to 24. Players can even make special "User Picker" dice that can randomly choose one player out of everyone in the game. Timers can be used to track turn length in face-paced games, or to set periodic reminders. Notepads allow users to send text to each other in a more permanent manner than chat messages, and can be used for anything from a scorecard to a list of house rules for the current game.

Many classic games are loadable with a single button press, and if players spend time recreating their favorite game in VirtualBoard, they can easily save it and reload instantly in the future. When privacy is important, private zones can be added, in which only the player or players who own the zone may see and interact with piece inside it. The uses for this are endless, whether you're hiding your hand in a game of poker, or passing secret notes to teammates. All-in-all, VirtualBoard gives you all the tools you need to play your favorite games, no matter where you are.

2 Process

When deciding on a process for our team to follow, we used the XP process from 427 as a starting point. Because we are all students, there are some aspects of XP that are particularly challenging to follow. So we chose to follow XP pretty strictly, with two main exception that pair programming and test-driven development are not explicitly required. User stories were assigned in pairs and each developer was expected to be familiar with all code written by their partner. We also decided to let pairs decide when they would write tests. Some pairs did decide to follow test-driven development and write their tests before implementing a feature while others chose to implement first.

We also looked into some alternate development processes such as Feature-Driven Development (FDD). While both considered agile, FDD and XP have some major differences. Unlike XP, FDD values the domain model above the source code itself. In essence, this means that much more of the project is planned up front similar to a waterfall model than is normal for other agile development systems. FDD also has a stricter class ownership system that makes refactorings across many classes more difficult. This difficulty is intentional since refactoring across many classes indicates a problem in the domain model that is at the heart of Feature-Driven Development. Unlike in XP, FDD developers are expected to sit down together and plan out what classes are needed and how they will interact before starting to write source code. Although there is a strong emphasis on testing and unit-tests, fewer and simpler refactorings in FDD means that the process does not need to be test-driven.

While we did not directly employ Feature-Driven Development for our project, the idea of sitting down and planning out our design ahead of time was a large benefit. Being able to avoid the need for large and messy refactorings is a pretty big deal. Therefore, our software development process was rooted firmly in XP with some minor changes and influences from FDD. Outside of that, we required all of our members to attend weekly progress meetings to review code and work together on planning for the next week. Our source code and issue tracking will be managed through GitHub with all updated information posted to the wiki.

3 Requirements and Specifications

- **Story 4 - User is able to send and receive chat messages**
  All users will be able to send instant messages to the whole lobby at any time during a game session. Whenever a new message comes in, all users should be able to see the messages poping up and then disappear.

- **Story 5 - User is able to create, view, and join lobbies**
  When users enter the game list screen, they should be able to see all existing lobbies and create a new lobby if he/she wants to. Also, when there is a lobby, whether protected or unprotected with password, the user can enter the game session(lobby) with certain credentials if needed.

- **Story 6 - User is able to view the current board state**
  After user joins the lobby(aka. game session), he/she will be able to see the current board state hosted on the server.

- **Story 7 - User is able to manipulate objects on the board**
  The user should be able to perform actions on objects on the board, such as adding objects, removing objects, moving objects by dragging and dropping, and making an object be static in the background. Some of this functionality will be performed by using a context menu when right-clicking on the object. Also, the object should be highlighted with that user's color when they are manipulating it.

- **Story 8 - User is able to go from a lobby to a new game session**
  After selecting a lobby and entering the game password if applicable, the user's game client then loads the data of that session and switches to playing the game.

- **Story 10 - User is able to download the current board configuration and load it later**
  The user will be able to download the current board state configuration at any point during the current game session. After the game configuration file is successfully saved to the local destination, the user will be able to restore the whole board by joining/creating any game session and load the configuration.

- **Story 11 - Pieces are updated for users in real time**
  When one player moves or manipulates a piece or pieces on their own client, the changes should appear on all other players clients in real time.

- **Story 12 - User is able to click and drag multiple objects at once**
  The user should be able to select multiple pieces, either by shift clicking on each piece or by clicking and dragging a box over many pieces. Once the pieces are selected the player should be able to manipulate all of them at once as if manipulating a single piece.

- **Story 13 - A lost connection is handled gracefully, add session cookies to resume**
  When a connection is lost or the page is refreshed the user should be given an option to resume the game they had been in. If they choose to resume their game they will be put into the same game they had been in with the same username and color as before.

- **Story 14 - Implement deck of cards**
  The user can add a deck of cards as a single piece. This deck will initially have 52 standard cards. The user can interact with the deck to add and remove cards, shuffle, and flip. The user should be able to tell how many cards are in the deck.

- **Story 15 - Implement dice**
  The user can add a die as a single piece. This die can have any number of sides between 2 and 24. The user can interact with the die to roll it which cause it to display a randomly decided face.

- **Story 17 - User is able to set the game background from presets**
  The user can use the sidebar menu to select an image to use as the game background, or "tabletop texture".

- **Story 18 - User is able to upload custom game pieces**
  The user is able to upload custom game pieces by providing a link to an image file. The user can also set the size for their desired piece.

- **Story 19 - User is able to configure game objects to snap to a grid**
  The user is able to enable snap to grid. When this is enabled, when a user sets down a piece it immediately moves to the closest position in the grid. There should be a sensitivity measure so that if the new position of the piece is not "close enough" to any grid positions it will not snap.

- **Story 20 - Private zones can be used to hide pieces from other users**
  The user should be able to create a private zone of his/her own color and move any piece into it so that others won't be able to see or select the piece until the user who owns the private zone moves the piece out from it.

- **Story 21 - User is able to select preset game boards such as checkers, and chess**
  The lobby owner can add the full configuration, including the board and all playing pieces, for various games all at once. There should be an button in the menu that brings up a modal window where the user can select from various options. The games we implemented are Parcheesi, Risk, Scrabble, Tic-Tac-Toe, Chess, and Checkers.

- **Story 22 - Housekeeping**
  The user should be able to have a relative easy intuition on every part of the game UI without much help.

- **Story 23 - Notepad**
  The user is able to add a note, similar to a sticky note, on the board. The user enters the text and the font size, and a note appears with the text formatted to fit on it.

- **Story 24 - Beacon**
  The user should be able to indicate a location on the board to the rest of the players in a game by causing a temporary beacon to appear at that location.

- **Story 26 - Use the die as a player picker**
  The user should be able to create a special kind of die piece to be used as a player picker. This die will have one face for each playing in the game, and instead of showing a number when rolled will show a color corresponding to a randomly selected player.

- **Story 27 - Users are able to use timers**
  The user can add a countdown timer to the game board that all other users can see. They can set the start time between 0 and 60 minutes and have the ability to start and stop the timer at will. The server handles the clock updates every second, when counting down.

- **Story 28 - User is able to resize and rotate pieces**
  Once a player has selected a piece they should be able to make it grow or shrink, and to rotate the piece in place.

- **Story 29 - Landing Page**
  When first visiting the site, the user sees a landing page displaying some basic information about VirtualBoard and the developers. Visiting this page should not open a socket and the user should be able to reach the lobby page easily.

4 Architecture and Design

4.1 Front End

The game client can run in any modern web browser, and is mostly based on the BabylonJS library. This is a WebGL library intended for 3D games that we used for our 2D top-down board game simulator. We also used bootstrap for most of our menus and non-gameplay interface.

We start the user on a welcome screen, which provides basic information about what VirtualBoard is and who made it. From there, the user is brought to a lobby page where they can choose a username and color as well as join a lobby.

4.2 Back End

The server that the clients connect to is created in Python using the Tornado library. Each client maintains a tcp socket to the game server, which mostly just serves as a forwarding switch. Originally, we had planned on using Django to host the static pages, user system, and lobby system, but this was changed to make the server not need to store any dynamic data on disk and use only Tornado.

The server itself maintains the game state and forwards updates between the players. Originally the server didn't do anything but record the game state and forward updates from each player to all other players. However, we later added features such as dice, decks, and private zones which are managed by the server.

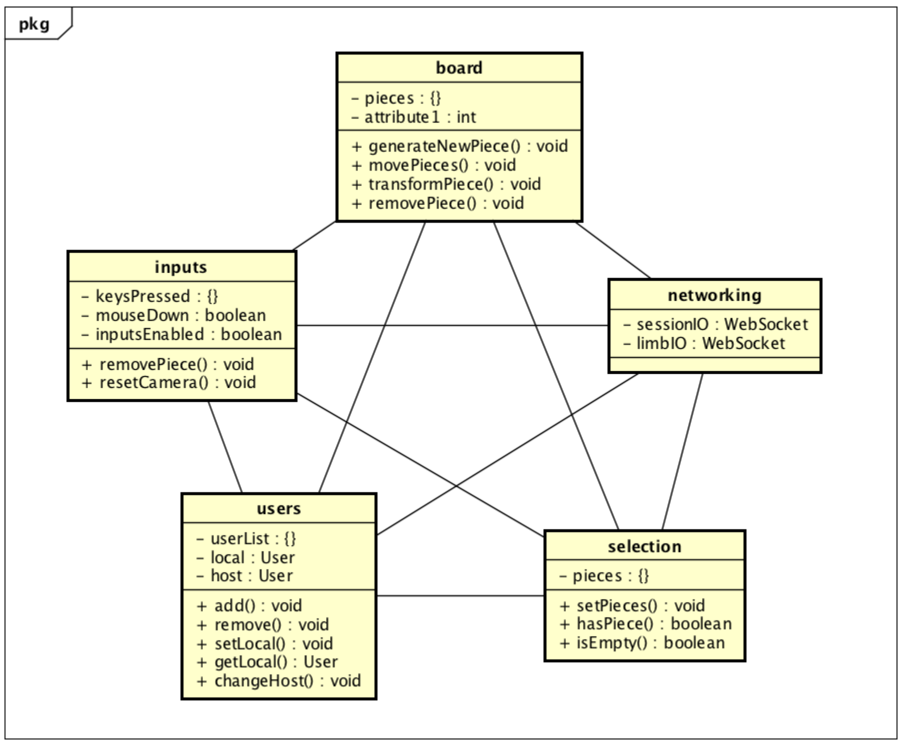

Figure 1: UML Diagram for Front End

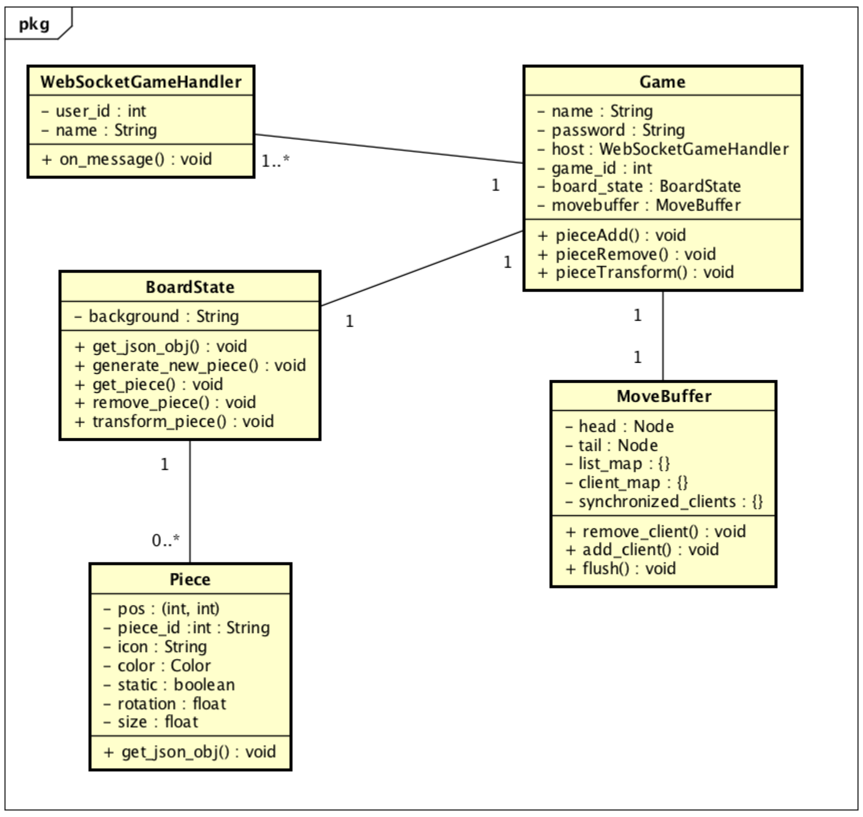

Figure 2: UML Diagram for Back End

4.3 API

All communication between the clients and the game server is done through a tcp socket in the form of JSON formatted messages. The general format for a message from a client to the server is formatted as follows:

```
{
   "type" : "messageType",
   "data" : [
      {
         "piece" : idOfFirstPiece (int),
         "param" : otherParameters (any)
      }, {
         "piece" : idOfSecondPiece (int),
         "param" : otherParameters (any)
      }
   ]
}
```

At the top level, the message has a type and a data component. The data component is formatted as an array to allow the clients to potentially send updates for many pieces at the same time. Each element of the data array is an object with a piece parameter that refers to the id of a piece. Other parameters vary based on the message type. Some message types do not have a piece parameter (such as chat messages) or refer to multiple pieces (such as adding a card to a deck). Some messages don't use the data array format if only the most recent one matters (examples include clearing the board, changing the host, etc). Most host only commands follow that format. Messages from the server to each client are similar, except that there is an additional "user" property to each object in the data array that specifies the user who generated the message.

4.4 Data Pipeline

Most actions in the server follow a general pipeline. An action is generally triggered in one of two ways.

1. The user triggers an input handler, which is located in `src/static/inputs.js`
2. The user changes the board by selecting something from a menu, which is located in `src/static/menu.js`

The `VBoard.inputs` object is responsible for handling mouse and keyboard input from the user and dispatching them. It also contains most of the handlers for things such as resizing pieces, adjusting the camera, and other buttons.

The `VBoard.menu` object is responsible for opening and closing menus, and contains most of the handlers triggered by pressing any menu button.

From either of these two objects, a corresponding handler in the `VBoard.sessionIO` object is called. `VBoard.sessionIO` is located in `src/static/networking.js` and contains methods that create JSON formatted messages and sends them through the tcp socket to the server.

Once the server receives a message, it is handled in the `WebSocketGameHandler` class. The `on_message` method reads the "type" parameter and dispatches the message to the corresponding handler in the `Game` class.

The methods in the `Game` class updates the `BoardState` class, and broadcasts the appropriate JSON formatted response to other players in the server through the TCP socket. When a client receives a message, it is handled by the `messageHandler` method in the `VBoard.sessionIO` object. This handler then reads the "type" parameter and dispatches the message to the corresponding method in the `VBoard.board object`.

The `VBoard.board` object is located in the `src/static/board.js` file, and contains the logic for updating and rendering the current board state.

There are some exceptions to this pipeline, most notably moving pieces has a custom pipeline due to client sided prediction and move buffering. Instead of the `VBoard.inputs` object calling the corresponding `VBoard.sessionIO` function, it sends the mouse movements to the `VBoard.selection` object. This object handles the set of pieces that the user is currently manipulating, as well as the logic for selecting and deselecting pieces. Most of the functions that affect all currently selected pieces also have a handler in `VBoard.selection`, which is called by a corresponding function in the `VBoard.inputs` object.

For these functions, the `VBoard.selection` object, located in src/static/selection.js, then triggers the corresponding call to `VBoard.sessionIO`. For everything except moving pieces, the pipeline is identical to the process previously described. When a piece is moved, the client's board state is updated immediately, instead of waiting on getting a confirmation from the server. This client sided prediction allows gameplay to appear smooth despite server latency. After that, the move data can then be sent through the network.

However, when `VBoard.sessionIO.movePiece()` is called, it buffers the data in a `VBoard.inputs.moveBuffer` object before sending data through the tcp socket to the server. This is because we would otherwise flood dozens messages every second about piece positions, each of which makes the previous message obsolete. By buffering the movement of pieces, we can send only the newest piece position at the end of some timeout, saving bandwidth. On the server end, a similar move buffer is also maintained for the same reason.

Once piece movement information is received from the server by a client, it is processed in the `VBoard.board.transformPiece()` function. This function determines whether or not to actually render the change, or if this movement might be something that was already predicted. Predictions that are not met by the server will reset after a given timeout.

5 Reflections and Lessons Learned

- Matt
  This was much more of an actual group project than anything I have worked on in the past. Most group projects that I have worked on either resulted in me and maybe one other person doing all of the work, or everyone doing completely isolated components that did not really need to integrate together. I was worried that the former might be the case with this project, but I realized a big part of the problem was my own need to dictate how every part of the project should work. I became personally invested in the project and would not accept anything that I deemed as less than ideal. Programmers take great pride in their code, so simply going in and changing things to work the way I wanted could lead to issues within the team. After running into a couple of problems in the first iterations of the project, I instead tried to talk with team members about how they were going to go about their user stories at the beginning of each iteration and provide some suggestions. I think this helped us run into fewer problems as the project drew to a close.

- Kai
  Working on the project is a challenging but rewarding experience. We had to tackle numerous implementation problems as well as familiarize ourselves with new concepts and techniques. Implementing long polling web server with Python Tornado framework and rendering graphics with Babylon WebGL are completely new to most of our team. And so it is overall a great learning experience.

- Jonathan
  The project taught me a lot not only about web app design and tools, but also about working on a team and the challenges that come with it. It was the first time I had worked on a Tornado app from scratch, having played around with Tornado a little bit in the past.

- Evan
  Overall the project was a good learning experience working on a unique problem. The process we used was rocky at the start, but went smoother after issues were worked out, especially regarding how we handled code ownership. This project taught me a lot about how to use Git properly and how to work with different types of people. I also learned a lot about socket programming that I did not know before.

- Kaushil
  This project served as a valuable experience. Being a physics major, I didn't have much experience working on a CS related project of this size. I not only got to learn more about GIT, socket programming, Javascript, web and game development but also learned more about working in a group. As a group we faced some challenges but were able to resolve the issues to a reasonable level. Personally, I had a few hiccups during the first few iterations and realized I was not communicating well. I tried to be more communicative through the rest of the project but still need to further improve upon it.

- Daniel
  This project has been a very good learning experience. I had not worked on project before which involved using different languages for different parts of the system, and so learning about how this is done and actually implementing features that involved both the Python and JavaScript side of our project was a valuable challenge. I also

learned a lot about using git, especially how to work with branches. Finally, this was only the second time I have worked on a team of this size, and so dealing with the challenges that having so many people entails was a big part of this project for me, and while by the end of the project we were on the same page, I definitely can improve on how I communicate with teammates.

- Da
  This project has been a great experience for me so far - I have definitely learned a lot too from what went wrong. We first chose a less efficient way to implement our system but switched back to the right road after few weeks of struggling. Choosing the right way (right framework, right language, etc.) of implementing a new system has a far-reaching impact on the development process. We eventually have the whole project working in the way we wanted it to be initially.

- Madeleine
  This project was overall a success. I haven't had much experience with socket programming or with javascript graphics libraries, so I learned quite a bit. In the first half of the project, we had some challenges as a team. I think this came from different understandings of the process we were following and some strong opinions about how our code should look. We were able to resolve this with some good communication and we were able to move forward and continue working without any conflict.