Breakout-Pong

Bryce DeWitt and Cristian Stransky
Web Development HW6

Introduction and Game Description

In order to create a game that supports multiple rooms and players, we immediately thought to bring back a classic arcade game with a modern spin. In our effort to create an engaging and competitive game, we envisioned something that would have a familiar look and feel to players while also not being a simple game that people have played before. Our team has created a game that is a combination of pong + breakout, allowing each player to control a single paddle and encouraging them to collect points by breaking blocks or scoring on their opponent. In this game, the first player to 50 points will win. Like breakout, we've placed blocks in the middle of the screen with varying levels of "hp", which represents the number of times they must be hit by the player ball before breaking. Additionally, drawing inspiration from the original pong, we've modified our version to allow for another player with a paddle on the other half of the screen to compete in breaking the most blocks and scoring on their opponent by passing their ball past the line on the other side.

When the game is started, we place a single ball that moves on each side of the screen. When a player has destroyed enough blocks to bounce their ball to the other side of the screen, they have the opportunity to "score" on their opponent by passing their own ball past the line on the opponent's side. Players are given 1 point for each block they destroy and 5 points for scoring on their opponent. A player is also awarded 5 points when their opponent has allowed their ball to pass their own goal line. When one of the players scores up to 50 points, the game will end and both players will be returned to the lobby. When back to the lobby, the player list will be re-ordered so that the loser of the previous game is allowed to play the next game against the following person on the list. The winner of the game is sent to the bottom of the lobby to allow the less-experienced players to continue practicing. For players that are in the lobby but not playing, they will be allowed to watch the game between the two participating players, but are not able to interact with the game until their turn to play.

UI Design

On first landing on our game server, a user is prompted to enter the name of a new or existing game they wish to join, as well as their own player name. Upon selecting "join game", the user is be brought to our game lobby, which lists the players currently waiting in the channel. When two players have both joined the lobby, the game will begin when any player selects the "start game" button. Our design uses React to insert an HTML canvas element into the DOM. On each refresh of the canvas (which is currently done every 50ms), a function is called to use the current game state that has been broadcast to the browser by the server to draw each individual element on the canvas. These game state data are stored in a map and are only the values that are required to draw the game on the canvas.

The actual layout and feel of the game was inspired by the original interfaces for pong and breakout, meant to be similar in look and feel to the original breakout and pong games, but

also have an updated and more friendly aesthetic. The player paddles are drawn as simple rectangles, which have also been outlined in black to better distinguish them from the background. The ball for each player is colored to match the paddle of the player that corresponds with them, and indicates the player that will be awarded points for each time the ball scores. The breakout blocks we have placed in the middle of the screen are colored in different hues of green to indicate that lighter colored blocks are easier to break. At the top of each player's half of the screen, their player name is displayed beside their score to indicate the current scoreboard. Though we believe breakout and pong should be familiar to most users, we have added the basic control instructions and win conditions below the game window.

UI to Server Protocol

In order for the user input for the browser client to be transmitted to the server and ultimately the other player, we used websockets to provide continuous two-way communication regarding the game state between our server and the web clients. When a user first joins a game using the given "game name" and "player name" on the initial landing page, their browser is connected to a common websocket channel for all players that have joined that game's lobby. Once connected, each browser is identified in their socket pushes by their unique player name, which determines which user input is attributed to which players.

We configured our game's canvas to fire an onKey event whenever one of the control keys were pressed (in this case up, down, left, or right) which ultimately pass the given key through the socket to the server. All inputs must be passed to the server for validation against the game logic in order to prevent users from circumventing browser-side JavaScript user input checks. While processing all events on the server creates a time and overhead cost, we believe it to be necessary to prevent users from cheating and to ensure the game state stored on the server is always consistent with the user's view. For each onKey event, we pushed a message to the server and returned the modified game state as a response. This game state produced by the server for the client has only the data needed to render the user's display, stripping out all variables necessary to compute game logic but not needed to draw the elements on the canvas.

For each message delivered through the socket, we also used an Elixir BackupAgent to save the modified game state delivered as the response to the browser in a central place where each browser client can refer to it as a single source of truth. This also allows us to hold a separate state for each game based on their given "name", which can then be referred to in the future to recollect the game state for a game after user exit. This allows users to reconnect to an initialized game and have the running game provided to them via the socket.

Data structures on server

To store the entire game state on our server, we devoted a significant amount of thought to create a data structure that was intuitive for our use in development, as well as able to fully preserve the state of the game at a single point in time, which would allow us to support starting and stopping the game. The primary data structure of our game is a single map created within our elixir game logic, within which we have 2 additional maps containing the

data for each player individually and one more map the contains the data regarding the breakout blocks. After much consideration and revision of this structure over the development of our game, we think this structure provides us with a good balance between ease of access to data and a logical grouping. Furthermore, this structure allows us to easily create a client view of the game state that contains only the data required for rendering the game canvas on the browser side.

Another data structure we use within our server to help with the game logic is a map of the constants we use for creating and computing the game logic within our server-side functions. Since these values are only editable from the game source files and should not change over the course of a single game, we didn't need them to be within the game state being captured in the backup agent. However, in order to prevent unnamed constants from starting to build up within our code, we factored them out to our single constants map.

Implementation of game rules

While the client view of the game is sent to the browser for rendering in front of the user, the game logic controlling the elements of the game are all restricted and controlled within our server-side elixir logic. When a player moves a paddle up or down (the only input our game accepts while active), the request to move the paddle is passed directly to the server; if the move is valid, the server logic will edit the game state to reflect the change and send it back, otherwise it will send back an unedited state of the game to be rendered again.

While it is necessary to restrict the movement of the paddles since they're directly under the control of a user, our game must also control the movement of the balls to allow them to interact with other elements of the game. We use a GenServer to call a function to move the ball at a regular interval, which delegates to other functions to determine if there are any elements (blocks, paddles, walls, scoring areas) blocking the movement of the ball for that tick. Upon "hitting" an element, other than just affecting the movement of the ball, there may also be other changes made to the game state. The game state returns at the end of our function to move the ball will always include changes to the location of the ball, but may also include changes to the ball speeds, block HPs, and score.

Challenges and Solutions

The largest initial challenge we faced was due to the nature of our game and our need to constantly modify the game state and render the new view to each user's web client. Our first solution to this issue was to only communicate with the server when any user provided input to the game. Because our initial concerns largely were largely with respect to the latency in server-browser communication, we thought this method would limit the total number of messages being sent, requiring the server to only compute and return the game assuming no additional user input would be made. At each additional call through the socket, the server would precompute the full outcome of another game with the new modification. However, with the requirements to have all game logic reside within the server, as well as the complexity of rendering many game animation frames that may never be used (since there would be another user input before they all were displayed), we decided to move all game logic to the server and rely on a message broadcasted from the socket to all players for each time the game state had

been changed. This ended up having negligible impact on UI latency and allowed us to simplify the game logic and data structures.

Another challenge that took a significant amount of time and effort to overcome was the method we would use to render the ball movement within the game at a consistent interval. Again, our initial inclination was to have a function on the web client that would repetitively call the server to provide the next game state, but we did not want to rely on the client-side scripts to run our game. In order to create a separate process on the server that could interact with and call functions within our game, we utilized a GenServer that sends a message at regular intervals to refresh the game state within the server and store it in our BackupAgent. Solving the problem of relying on the web client to "tick" our world, we are also able to have it continue running within the server even after all clients have disconnected from the channel. While this is an interesting side effect that might be useful for some other games, we decided a breakout+pong game should stop when there is no player connected, so our GenServer also allows us to cancel our send_after function when no longer necessary.

Additionally, when creating the GenServer to interact with our game module, we faced difficulty extracting the name of the game from the web socket when starting a GenServer that needed to reference it. After spending time combing through the documentation and consulting with TAs about the possibility of extracting the name of our game from the socket itself, we were finally able to work around the issue by adding a name within the game data structure itself and extracting that when passed through the socket. While this solution works well enough to fit our purposes and still be fairly efficient, we are still working to create a way to have our games be identified only by their references from the sockets.