Step 0: App.js

```
7    import Chatbook from "./pages/Chatbook.js";
8    import Game from "./pages/Game.js";
9
```

```
61          {/* TODO (Step 0.1): add Game page ("/game/") to our app, and pass in userId as a prop (1 line) */}
62          {/* Your code goes here */}
63          <Game path="/game/" userId={userId} />
```

NavBar.js

```
29          <Link to="/game/" className="NavBar-link">
30            Game
31          </Link>
```

Step 1.1: game-logic.js

```
23  /** Game state */
24  // TODO (Step 1.1): Create an object for the initial gameState.
25  const gameState = {
26    winner: null,
27    players: {},
28  };
```

```
42  /** Update the game state. This function is called once per server tick. */
43  // TODO (Step 1.1, pt 2): Create an empty (for now) function for updating the game state.
44  const updateGameState = () => {
45    // This function is currently empty, but we'll add to it later.
46  };
```

```
56  module.exports = {
57    // TODO (Step 1.1, pt 3): Export gameState and the updateGameState function.
58    gameState,
59    updateGameState,
60    spawnPlayer,
61    removePlayer,
62  };
```

Step 1.1: server-socket.js

```
13   /** Send game state to client */
14   // TODO (Step 1.1): Create a function which sends gameState updates to all clients.
15   const sendGameState = () => {
16     io.emit("update", gameLogic.gameState);
17   };
```

```
19   // /** Start running game: game loop emits game states to all clients at 60 frames per second */
20   // TODO (Step 1.1): Create a function which sets up a game loop for running the game.
21   const startRunningGame = () => {
22     setInterval(() => {
23       gameLogic.updateGameState();
24       sendGameState();
25     }, 1000 / 60); // 60 frames per second
26   };
```

```
28   // TODO (Step 1.1): Call startRunningGame to start the game.
29   startRunningGame();
```

Step 1.2: game-logic.js

```
32  /** Adds a player to the game state, initialized with a random location */
33  const spawnPlayer = (id) => {
34    // TODO (Step 1.2): Initialize a new player indexed by id with "position" and "radius" properties.
35    //    The new player should have a random position on the map.
36    //    You can use getRandomPosition() which has been implemented above for you.
37    //    The radius should be the initial radius (defined at the top of this file).
38    //    Bonus challenge: If you want, you can try to also give a "color" property to each player, and
39    //    initialize it to a random color from the colors array at the top of this file.
40    gameState.players[id] = {
41      position: getRandomPosition(),
42      radius: INITIAL_RADIUS,
43      color: colors[Math.floor(Math.random() * colors.length)],
44    };
45  };
```

Step 1.3: game-logic.js

```
53  /** Remove a player from the game state if they disconnect or if they get eaten */
54  const removePlayer = (id) => {
55    if (gameState.players[id] != undefined) {
56      // TODO (Step 1.3): remove the player from the game state
57      // Your code goes here
58      delete gameState.players[id];
59    }
60  };
```

Step 1.4: server-socket.js

```
34  // TODO (Step 1.4): call spawnPlayer on the user id (1 line)
35  // Hint: spawnPlayer takes a user id as an input, which is given here by `user._id`
36  // Hint 2: spawnPlayer is a function from our gameLogic module, which we've already imported
37  // Your code goes here!
38  gameLogic.spawnPlayer(user._id);
```

Step 1.5: Game.js

```
19  const processUpdate = (update) => {
20    // TODO (Step 1.5): call drawCanvas using the `update` transmitted to the socket
21    // (`update` is the current `gameState`)
22    drawCanvas(update, canvasRef);
23  };
```

```
12    // TODO (Step 1.5, pt 2): update game periodically using useEffect hook
13    useEffect(() => {
14      socket.on("update", (update) => {
15        processUpdate(update);
16      });
17      return () => {
18        socket.off("update");
19      }
20    }, []);
```

Step 1.6: canvasManager.js

```
24  const drawPlayer = (context, x, y, radius, color) => {
25    const { drawX, drawY } = convertCoord(x, y);
26    // TODO (Step 1.6): call fillCircle to draw a circle as the player (1 line)
27    // Your code goes here
28    fillCircle(context, drawX, drawY, radius, color);
29  };
```

```
42    // draw all the players
43    Object.values(drawState.players).forEach((p) => {
44      // TODO (Step 1.6, pt 2): call drawPlayer to draw each player (1 line)
45      // Hint: each player `p` has a `position` field, and this `position` field
46      //   has an `x` field and `y` field.
47      // Your code goes here
48      drawPlayer(context, p.position.x, p.position.y, p.radius, p.color);
49    });
50  };
```

Step 2.1: client-socket.js

```
9   /** send a message to the server with the move you made in game */
10  // TODO (Step 2.1): Create a function that will be the client's way of sending move data
11  //    to the server.
12  export const move = (dir) => {
13    socket.emit("move", dir);
14  };
```

input.js

```
1   // TODO (Step 2.1): Import the move function we wrote in client-socket.js
2   import { move } from "./client-socket";
```

Step 2.2: input.js

```
4   // TODO (Step 2.2): This function is where the client will handle user inputs from mouse and keyboard.
5   //    One of the directions is done for you. Complete the rest of the inputs to emit "down", "left", and "right".
6   //    Check https://developer.mozilla.org/en-US/docs/Web/API/UI_Events/Keyboard_event_key_values for names of keys.
7   /** Callback function that calls correct movement from key */
8   export const handleInput = (e) => {
9     if (e.key === "ArrowUp") {
10      move("up");
11    } else if (e.key === "ArrowDown") {
12      move("down");
13    } else if (e.key === "ArrowLeft") {
14      move("left");
15    } else if (e.key === "ArrowRight") {
16      move("right");
17    }
18  };
```

Step 2.3: Game.js

```
14  useEffect(() => {
15    // TODO (Step 2.3): add event listener when the page is loaded (1 line)
16    // Hint: `window` is a global variable on which you should call `addEventListener`
17    // The type of event listener is "keydown", and the listener is the `handleInput` function
18    // we imported from input.js. Refer to documentation for `addEventListener` here:
19    // https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener
20    // Your code goes here!
21    window.addEventListener("keydown", handleInput);
22
23    // remove event listener on unmount
24    return () => {
25      // TODO (Step 2.3, pt 2): remove event listener when the page unmounts (1 line)
26      // This return statement allows us to run code when the user leaves the page.
27      // Hint: `window` also has a `removeEventListener` method
28      // Your code goes here!
29      window.removeEventListener("keydown", handleInput);
30    };
31  }, []);
```

Step 2.4: server-socket.js

```
61        // TODO (Step 2.5): Turn on a socket that listens for the 'move' event
62        // and calls gameLogic.movePlayer
63        socket.on("move", (dir) => {
64          // Listen for moves from client and move player accordingly
65          const user = getUserFromSocketID(socket.id);
66          if (user) gameLogic.movePlayer(user._id, dir);
67        });
68      });
69    },
```

Step 2.5: game-logic.js

```
47  // TODO (Step 2.4): given the player id (id) and keyboard input (dir), move the player 10px
48  //   in the direction given by `dir` (8-10 lines).
49  // `dir` is a string that can take on 4 directions: "up", "down", "left", "right".
50  // Remember that x controls left-right, and y controls up-down.
51  // Hint: Players are stored in gameState.players, and each player is indexed by its `id`. Each player
52  //       has a `position` field, and this `position` field has an `x` field and a `y` field.
53  // Your code goes here!
54  // Move player (unbounded)
55  if (dir === "up") {
56    gameState.players[id].position.y += 10;
57  } else if (dir === "down") {
58    gameState.players[id].position.y -= 10;
59  } else if (dir === "left") {
60    gameState.players[id].position.x -= 10;
61  } else if (dir === "right") {
62    gameState.players[id].position.x += 10;
63  }
```

Step 3.1: game-logic.js, `gameState`

```
55  /** Game state */
56  const gameState = {
57    winner: null,
58    players: {},
59    // TODO (Step 3.1): Add "food" array to gameState here. (1 line)
60    food: [],
61  };
```

Step 3.1: game-logic.js, `spawnFood()`

```
68  /** Adds a food to the game state, initialized with a random location */
69  // TODO (Step 3.1): Implement the following function which spawns in food at ran
70  const spawnFood = () => {
71    gameState.food.push({
72      position: getRandomPosition(),
73      radius: FOOD_SIZE,
74      color: colors[Math.floor(Math.random() * colors.length)],
75    });
76  };
```

Step 3.1: canvasManager.js, line 51

```
50      // draw all the foods
51      // TODO (Step 3.1): Draw foods on canvas.
52      Object.values(drawState.food).forEach((f) => {
53        drawCircle(context, f.position.x, f.position.y, f.radius, f.color);
54      });
55    };
56
```

Step 3.2: game-logic.js, `computePlayersEatFoods()`

```
45  // TODO (Step 3.2): attempt all pairwise eating between each player and all foods
46  // Implement the following function:
47  const computePlayersEatFoods = () => {
48    Object.keys(gameState.players).forEach((pid1) => {
49      gameState.food.forEach((f) => {
50        playerAttemptEatFood(pid1, f);
51      });
52    });
53  };
```

Step 3.3: game-logic.js, line 38

```
33      if (dist < gameState.players[pid1].radius - FOOD_SIZE) {
34        // food is within player 1's eat range
35        if (gameState.players[pid1].radius > FOOD_SIZE) {
36          // player 1 is big enough to eat food
37          gameState.players[pid1].radius += FOOD_SIZE;
38          // TODO (Step 3.3, pt 2): call removeFood on a food if the food is in range
39          removeFood(f);
40        }
41      }
42    };
```

Step 3.3: game-logic.js, `removeFood()`

```
119  // TODO (Step 3.3): remove a food from the game state if it gets eaten, given reference to food object
120  // Has some ugly syntax because we're given the _reference_ to the food _object_,
121  // and need to find the corresponding index in gameState.food
122  // Implement the following function:
123  const removeFood = (f) => {
124    let ix = gameState.food.indexOf(f);
125    if (ix !== -1) {
126      gameState.food.splice(ix, 1);
127    }
128  };
```

Step 3.4: game-logic.js, `checkEnoughFoods()`

```
93  // TODO (Step 3.4): spawn a food if there are less than 10 foods
94  // Implement the following function:
95  const checkEnoughFoods = () => {
96    if (gameState.food.length < 10) {
97      spawnFood();
98    }
99  };
```

Step 3.4: game-logic.js, `updateGameState()`

```
 99  /** Update the game state. This function is called once per server tick. */
100  const updateGameState = () => {
101    // TODO (Step 3.4): add computePlayersEatFoods and checkEnoughFoods to game loop
102    // This will compute all pairwise eating between each player and all foods,
103    // and add more food to the game
104    // (Implement two lines):
105    computePlayersEatFoods();
106    checkEnoughFoods();
107  };
```

Step 4.1: game-logic.js, `playerAttemptEatPlayer()`

```
30  /** Helper to compute when player 1 tries to eat player 2 */
31  const playerAttemptEatPlayer = (pid1, pid2) => {
32    //    We can get the players by using gameState.players and indexing by pid1 and pid2 as keys.
33    //    Each player has a position property, which has an x and y property.
34    //    We want to compute the Euclidean distance between the players' positions using the distance formula.
35    //    In order for player1 to eat, we need to check that this distance is less than player1's radius.
36    //    We must also check that player1's radius is bigger than player2's radius.
37    //    If we pass both of these checks, we should add player2's radius to player1's radius.
38    //    Instead of removing player2 immediately, we will just push player2's id to the playersEaten array
39    //    for now, and we will formally delete it delete later.
40    const player1Position = gameState.players[pid1].position;
41    const player2Position = gameState.players[pid2].position;
42    const x1 = player1Position.x;
43    const y1 = player1Position.y;
44    const x2 = player2Position.x;
45    const y2 = player2Position.y;
46    const dist = Math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
47    if (dist < gameState.players[pid1].radius * EDIBLE_RANGE_RATIO) {
48      // player 2 is within player 1's eat range
49      if (gameState.players[pid1].radius * EDIBLE_SIZE_RATIO > gameState.players[pid2].radius) {
50        // player 1 is big enough to eat player 2
51        gameState.players[pid1].radius += gameState.players[pid2].radius;
52        playersEaten.push(pid2);
53      }
54    }
55  };
```

Step 4.2: game-logic.js, `computePlayersEatPlayers()`

```
43  /** Attempts all pairwise eating between players */
44  const computePlayersEatPlayers = () => {
45    if (Object.keys(gameState.players).length >= 2) {
46      Object.keys(gameState.players).forEach((pid1) => {
47        Object.keys(gameState.players).forEach((pid2) => {
48          // TODO (Step 4.2): call playerAttemptEatPlayer helper function (1 line)
49          playerAttemptEatPlayer(pid1, pid2);
50        });
51      });
52    }
```

Step 4.3: game-logic.js, `computePlayersEatPlayers()`

```
52    // Remove players who have been eaten
53    playersEaten.forEach((playerid) => {
54      // TODO (Step 4.3): call removePlayer on each player that has been eaten (1 line)
55      // Note that the playerAttemptEatPlayer helper function has already stored all eaten players in playersEaten
56      removePlayer(playerid);
57    });
58    playersEaten = []; // Reset players that have just been eaten
59  };
```

Step 4.4: game-logic.js, `updateGameState()`

```
141  /** Update the game state. This function is called once per server tick. */
142  const updateGameState = () => {
143    // TODO (Step 4.4): add computePlayersEatPlayers to game loop (1 line)
144    // This will check all pairwise eating between players every loop
145    computePlayersEatPlayers();
146    computePlayersEatFoods();
147    checkEnoughFoods();
148  };
```

Step 5.1: server-socket.js, `addUserToGame()`

```
28  // TODO (Step 5.1): Fill out the addUserToGame and removeUserFromGame functions to call
29  //    spawnPlayer and removePlayer functions from gameLogic. Make sure you pass in the user's id.
30  const addUserToGame = (user) => {
31    // Your code here (Step 5.1) (1 line)
32    gameLogic.spawnPlayer(user._id);
33  };
34
35  const removeUserFromGame = (user) => {
36    // Your code here (Step 5.1) (1 line)
37    gameLogic.removePlayer(user._id);
38  };
```

Step 5.1: server-socket.js, `addUser()`

```
38  const addUser = (user, socket) => {
39    const oldSocket = userToSocketMap[user._id];
40
41    // TODO (Step 5.1): Remove this call to spawnPlayer, since now we have a spawn button.
42    // gameLogic.spawnPlayer(user._id); DELETE ME!
43    if (oldSocket && oldSocket.id !== socket.id) {
44      // there was an old tab open for this user, force it to disconnect
45      oldSocket.disconnect();
46      delete socketToUserMap[oldSocket.id];
47    }
```

Step 5.2: api.js

```
128  // TODO (Step 5.2): Add an API endpoint to spawn players in, and call it "/spawn".
129  //    Make sure you use a post API endpoint, and don't forget to check that req.user exists.
130  //    You can send and empty object ({}) as a response.
131  //    Hint: You should use the socketManager functions you wrote in Step 5.1.
132
133  router.post("/spawn", (req, res) => {
134    if (req.user) {
135      socketManager.addUserToGame(req.user);
136    }
137    res.send({});
138  });
```

Step 5.2: api.js

```
140  // TODO (Step 5.2): Add an API endpoint to despawn players (if they disconnect), and call it "/despawn".
141  //    Make sure you use a post API endpoint, and don't forget to check that req.user exists.
142  //    You can send and empty object ({}) as a response.
143  //    Hint: You should use the socketManager functions you wrote in Step 5.1.
144
145  router.post("/despawn", (req, res) => {
146    if (req.user) {
147      socketManager.removeUserFromGame(req.user);
148    }
149    res.send({});
150  });
```

Step 5.3: Game.js

```
48  // set a spawn button if the player is not in the game
49  let spawnButton = null;
50  if (props.userId) {
51    spawnButton = (
52      <div>
53        <button
54          onClick={() => {
55            // TODO (Step 5.3): send a post request with user id to spawn api (1 line)
56            post("/api/spawn", { userid: props.userId });
57          }}
58        >
59          Spawn
60        </button>
61      </div>
62    );
63  }
```

Step 5.4: server-socket.js, `removeUser()`

```
54  const removeUser = (user, socket) => {
55    if (user) {
56      delete userToSocketMap[user._id];
57      // TODO (Step 5.4): call removeUserFromGame on disconnect;
58      // a user should disconnect from game if they disconnect from site
59      removeUserFromGame(user);
60    }
61    delete socketToUserMap[socket.id];
62    io.emit("activeUsers", { activeUsers: getAllConnectedUsers() });
63  };
```

Step 5.4: Game.js

```
17    // add event listener on mount
18    useEffect(() => {
19      window.addEventListener("keydown", handleInput);
20
21      // remove event listener on unmount
22      return () => {
23        window.removeEventListener("keydown", handleInput);
24        // TODO (Step 5.4): send a post request with user id to despawn api (1 line)
25        post("/api/despawn", { userid: props.userId });
26      };
27    }, []);
```

Step 5.5: game-logic.js, `updateGameState()`

```
165  /** Update the game state. This function is called once per server tick. */
166  const updateGameState = () => {
167    // TODO (Step 5.5): add checkWin to game loop
168    checkWin();
169    computePlayersEatPlayers();
170    computePlayersEatFoods();
171    checkEnoughFoods();
172  };
```

Step 5.5: Game.js, line 13

```
13    // TODO (Step 5.5): initialize winnerModal state
14    // Implement here (1 line):
15    const [winnerModal, setWinnerModal] = useState(null);
```

Step 5.5: Game.js, `processUpdate()`

```
36  const processUpdate = (update) => {
37    // TODO (Step 5.5): set winnerModal if update has defined winner
38    // Comment in the following code:
39    if (update.winner) {
40      setWinnerModal(
41        <div className="Game-winner">the winner is {update.winner} yay cool cool</div>
42      );
43    } else {
44      setWinnerModal(null);
45    }
46    drawCanvas(update, canvasRef);
47  };
```

Step 5.5: Game.js, return

```
61      return (
62        <>
63          <div>
64            {/* important: canvas needs id to be referenced by canvasManager */}
65            <canvas ref={canvasRef} width="500" height="500" />
66            {loginModal}
67            {/* TODO (Step 5.5): display winnerModal (1 line) */}
68            {winnerModal}
69            {spawnButton}
70          </div>
71        </>
72      );
73    };
```

Congrats on finishing Gamebook!