6.835 MiniProject 3:
Multimodal Battleship

Due: 5:00 PM Monday, March 12, 2018

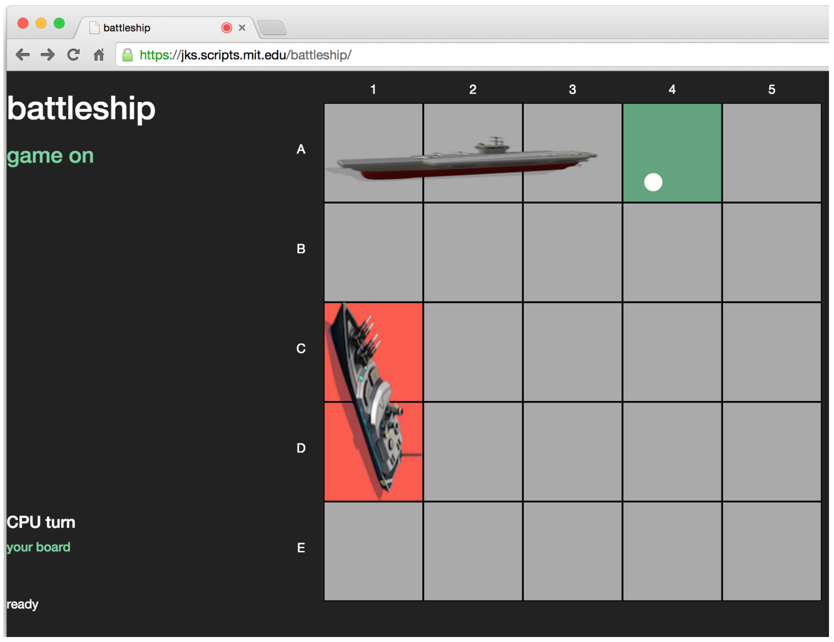

Figure 1: The multimodal version of Battleship that you will be building.

1 Introduction

The goal of this project is to build a multimodal version of the classic board game, Battleship, and to extend on the game in a creative way. The game you build will support both multimodal and symmetric interaction. It will be *multimodal* in the sense that the player can use both hand gestures and speech, and *symmetric* in the sense that the computer will respond to the player's moves with its own speech and visual feedback. You will do this in the form of a web application, using your laptop's microphone and a Leap Motion sensor for capturing speech and gesture input.

The spirit of the project is different from prior ones, where you focused on optimizing performance for a recognition task. Here, the idea is for you to do what you can to make the Battleship UI as functional and fun to play as possible. You'll be graded based on functionality, creativity, and how far you stretch in the version you produce.

2 Getting Started

If you have one of the Leap controllers in a box, please be careful to preserve all the packaging. If there is a plastic strip on the Leap device, please peel it off carefully and keep it in the box. When you turn in your device, please put the strip back on (to keep the device from being scratched) and put everything back in the box as you found it. This will help ensure the device is usable by the next class.

1. Install the Leap Motion SDK for your OS. The Leap Motion SDK can be found online or on Stellar. Position the Leap sensor in front of your keyboard and run through a couple of the pre-packaged demos to make sure things are working properly.

2. Setup your web scripts directory on Athena, if you haven't already. Login to Athena and follow the instructions, and you'll end up with a `web_scripts` directory. (See here about connecting to Athena from your local machine.)

3. Download the packaged `.zip` file from Stellar and unzip it in your `web_scripts` directory. You should now be able to find the starter Battleship app at `https://YOUR_KERBEROS.scripts.mit.edu/battleship/`. Note the `https` instead of http, which is important so you aren't asked to approve use of your microphone every time you open the app. (See here for information about copying files to Athena from your local machine.)

3 Battleship

Battleship is a 1v1 board game, where both players place their ships on a grid, then take turns blindly firing shots at each other's boards by naming grid cells. A more detailed description can be found in the link above.

In Multimodal Battleship, a human player faces off against a computer. Most of the game logic has been given to you, and you are tasked with facilitating the conversation between the player and CPU. Also, MM Battleship takes place on a 5x5 grid with 2 ships-per-player, although you can configure these parameters later if you'd like.

Here is a storyboard illustrating Multimodal Battleship:

1. In `setup` mode, the player can grab, move, and release their ships onto the grid to deploy them. Note that the CPU's ships are automatically deployed.
2. When the player is satisfied with their own deployment, they can start the game with a speech command: *start*
3. In `playing` mode, the player and CPU take turns firing a shot and then responding to the opponent's shot.
4. The player fires a shot by pointing to a grid tile with their hand and triggering the shot with a speech command: *fire*
5. The CPU fires a shot by synthesizing speech for the grid tile (*fire C 5*), and highlighting/blinking the tile as a visual cue.
6. The player / CPU responds to the shot with an appropriate spoken response: *hit, miss, you sunk my* `SHIP_TYPE`*, game over.*
7. Play continues until either the player's or CPU's ships are all sunk.

4 Implementation

All the code you write will be JavaScript, in `app/main.js`. There are a number of helper functions in `app/helpers.js` that you will need to use for moving ships, highlighting/blinking tiles, processing speech, and generating speech. For each step below, some of the substeps have been implemented for you and we've indicated which helper functions might be useful for you.

A few practical notes:

- Use Chrome to play the game, since it supports the various APIs needed. If you are new to web/JS programming, a useful tip is to use the developer console and `console.log(message)` statements heavily.
- The game only moves forward if your hand is being recognized by the Leap. If things seem frozen, make sure your hand is above the Leap.

- Speech recognition will work best if you are playing in a quiet environment, with headphones to prevent any synthesized speech from getting piped back into the recognizer. You can see what is being recognized by setting `SPEECHDEBUG` to `true` in `app/config.js`.
- If you ever want to skip `setup` mode, you can change `SKIPSETUP` to `true` in `app/config.js`. This will auto-deploy both players' boards and take you right into the turn-taking phase of the game.
- On a Mac computer, you may need to change the `VOICEINDEX` in `app/config.js` to 17 so that the speech system uses English. On a Windows machine, change the `VOICEINDEX` to 4.

4.1 Moving the cursor with Leap data

Start by getting a white, circular cursor to move based on the hand position reported by the Leap. Once you know where the cursor is, try highlighting the tile that it is selecting. You can do this in the following way:

- See the Leap SDK's `screenPosition` function for getting the hand position in screen coordinates ([x, y]).
- Move the cursor by calling `cursor.setScreenPosition(cursorPosition)`. You may find that you need to offset the Leap-reported screen position in the Y-dimension, if you are unable to select the bottom row of tiles.
- Determine which tile is being selected by calling `getIntersectingTile(cursorPosition)`. If `cursorPosition` overlaps with a tile, it returns that tile in board coordinates: {`row:r, col:c`}. If the cursor is off-board, the function will return `false`.
- Highlight a tile, if necessary by calling `highlightTile(tile, color)`, where color is a hex code (see the `Colors` object in `app/config.js`).

4.2 Deploying ships

Enable the player to deploy ships in the following way:

- Check if they are hovering over a ship using `getIntersectingShipAndOffset`, which returns both a ship and grab offset, if the player is hovering over a ship.
- Check if they are grabbing using the Leap hand's `grabStrength` or `pinchStrength` values.

- If they've grabbed a ship, move it accordingly using `ship.setScreenPosition([x,y])` and `ship.setScreenRotation(angle)`. You can work with `hand.roll()` to get the hand's rotation. Be sure to take the grab offset into account, or you'll see a jump when the ship starts moving.
- If they've released a ship, try placing it onto the grid using `placeShip(ship)`. If it fits onto the grid, it'll snap into place, and if not, it'll go back to the left side of the screen. Don't forget to release the ship programmatically, by setting `grabbedShip` to `false`.

This part of the project can be tricky. You may want to play around with grabStrength thresholds, roll values, and potentially even smoothing functions to arrive at a smooth(er) user experience.

4.3 Starting the game with speech

Take a look at the `processSpeech` function, which is called periodically by the system whenever speech has been recognized. As mentioned before, you can see what the system is recognizing by setting `SPEECHDEBUG` to `true` in `app/config.js`. Use the included `userSaid` function to detect the "start" command in the speech transcript, and use `gameState.startGame()` to start the game.

4.4 Player's turn

Enable the player to *fire* at a tile, then have the CPU respond using speech, based on the shot's result. Do this by specifying another case to the `processSpeech` function to process the *fire* command. Next, implement `registerPlayerShot`, which should:

- Determine which tile the player is shooting at (`selectedTile`).
- Fire the shot at the CPU's board using `cpuBoard.fireShot(shot)`. The result will tell you whether the shot hit, missed, sunk a ship, or ended the game.
- Generate the CPU's response by synthesizing speech using `generateSpeech(message)`. The CPU should respond appropriately with one of the following: *hit, miss, you sunk my SHIP_TYPE, game over.*
- If the game is over, you can end it by calling `gameState.endGame("player")` to indicate that the player won.

Once you are satisifed with this behavior, uncomment `nextTurn()` to proceed to the next step.

4.5 Machine's turn

During the CPU's turn, have it dictate a shot to the player, then enable the player to respond with *hit/miss/sunk/game over*. First, take a look at `generateCpuShot`, in which you should:

- Use `gameState.getCpuShot()` to generate a new, random shot.
- Synthesize the CPU's speech for the shot using `generateSpeech(message)`. For example, *fire A 5*.
- Provide a visual cue for the player by calling `blinkTile(tile)`.

Next, add another case to `processSpeech` to detect the *hit/miss/sunk/game over* responses from the player, and then call `registerCpuShot`.

Finally, take a look at `registerCpuShot(playerResponse)` – very similar to `registerPlayerShot` – where you should:

- Test the shot against the player's board.
- Have the CPU respond appropriately to the shot's result using `generateSpeech(message)`. For example, *Awesome!* if their shot was a hit.
- If the game is over, you can end it by calling `gameState.endGame("cpu")` to indicate that the CPU won.

For now, you can ignore the truthfulness of the player's response, and just check the shot status directly. Once you are satisified with this behavior, uncomment `nextTurn()` to hand it back over to the player.

4.6 Let's make this interesting

Now that the basic implementation of Battleship is working, extend your game in one or more different ways in order to create a more interesting, natural, and fun game for the user. You will be graded based on how creative you are and how far you stretch in your extensions, in addition to the functionality of the extensions. Some suggestions are:

- Give the CPU some personality. It could generate a wider variety of speech in different situations. For example, not just *hit / miss* but also a clue about how far off the player was. Or family-friendly curse words when the CPU has missed a few times in a row.

- Decide what the CPU should do if it suspects that the player is *lying*! You can determine if this is the case by comparing the computed shot result with the player's spoken response to the CPU's shot. Maybe it could make the board disappear - see GLaDOS for inspiration.
- Try different strategies for deploying ships. For example, instead of dragging them onto the board, the player could point at a tile and say "battleship here".
- Design a more attractive UI. Engage the user with sound effects and visual stimulation.
- Any other ideas you have!

5 Submission

Please submit the following to the course website in the Homework section:

1. Your source code. Please submit your entire `battleship` directory, zipped up. Make sure your code is also running on `https://YOUR_KERBEROS.scripts.mit.edu/battleship/`.
2. What were some of the challenges you faced in implementing (a) ship deployment, (b) the player's turn, (c) the CPU's turn, and what did you do to solve these problems?
3. What kinds of extensions did you make to your implementation? Give details about what you did. Include how to use the extensions in-game, how you implemented the extensions, and what interaction/experience the extensions add to the game for the user.