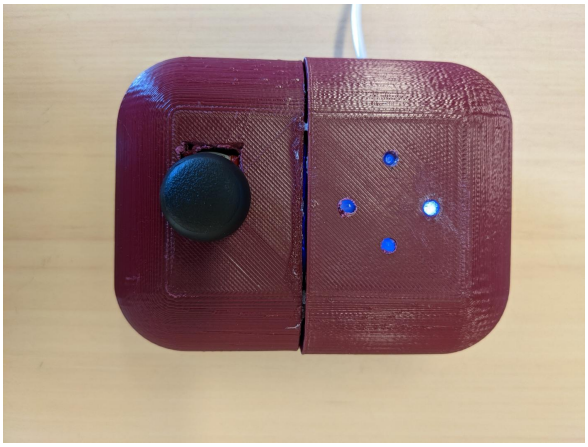
Spooky Blind Maze

By: Aarush Bothra, Isaiah Wildenberg, and Max Lee

Our problem was to create a maze game using only the particle photon and the website. The reason this was difficult was that there was not a simple way to display a maze and update it for a player to follow their progress. So instead, we decided to make a blind maze. This meant that although there would be no way for a player to see the full maze, they would get hints through LEDs and a vibration motor.

To interact with the game, the player would hold a box that had a joystick and 4 LEDs on the top. The LEDs were arranged to show the four directions the player would be able to go in. During the game, the LEDs would be one of four colors: red, green, white, or blue. If the LED showed red, that would mean a wall is located in the LED's corresponding direction. If it showed green, then it would indicate a spot that the player could go to. If the LED was white, it would indicate a spot with a bonus point, and a blue LED would indicate the goal or end of the maze.

To move in the direction of the LEDs, the player would interact with a joystick. Moving the joystick in a direction would be reflected in the LEDs as well, which would change if the player moved to a valid location. If the player moved into a wall/red LED, then the LEDs would not change and the vibration motor would buzz, indicating an invalid position. When a player reaches a blue LED, the LEDs would flash red and white, indicating that the maze had ended.



To start the game, a player would go to the website and enter a difficulty. This interaction would trigger a cloud function that would start the game with a maze of the desired difficulty. At the end of the game, the player can also get their score from the particle and add it to the leaderboard. This is done when the player inputs their name on the website after the end of the game. The website then checks a cloud variable for whether the game has actually ended, and if it has, it checks another cloud variable for the score. The website takes the name and score and puts them onto a google sheet, for storage in case the website is refreshed. The website checks the google sheet and populates a table on the home page that displays the leaderboard for the player, in ascending order.

Website code explanation

The website code is divided into multiple files: Index.js, App.js, Index.html, and App.css. App.css and index.html both handled the display elements for the main page. Index.html mainly contained divs for app.js to put elements into.

Index.js is where the Node.js server lives, and is where the cloud functionality is. When the website is first loaded or any of the buttons on the website are pressed, the (app.get('/data1', async (req, res) =>) function is called. In this function, the website uses the Node.js google sheets api to interact with the google sheet that stored the leaderboard data. This asynchronous function authenticated with the google api servers using a private key file and pulled the leaderboard data from the google sheet. It then sorted the data from lowest to highest time and published it to the /data1 url in a json format, for app.js to grab later.

At any button press or page load, the updateTable() function in app.js is also called. This function clears and repopulates the leaderboard table every time it is called, by pulling data from /data1 as mentioned above. In the for loops in app.js, each of the elements of the webpage are created and shown on the website dynamically, so the table can be generated as the google sheet is updated.

Index.js also handles the interactions with the particle cloud. In the (app.post('/particle', async (req, res) => {) function, the website checks the gameEnd cloud boolean to see whether the game has ended. If so, the website gets the playerScore cloud variable and the player-inputted name from the website and puts them on the google sheet. The function then refreshes the leaderboard. All of this is triggered when the player hits the submit button by the name text input box. If any of these cloud variables return values that can create errors, an error message is displayed on the /particle portion of the website.

C++ Code Explanation

The maze functionality was handled by the C++ code. Through the website the difficulty is updated which sets a variable to indicate which one. The maze is then initialized using the “customInitBoard” function. Knowing the difficulty one of the three different maze maps will be sent into the function. The maps are stored as arrays and initialized at the beginning of the code. The array is run through the “customInitBoard” function which senses where the start and end of the maze is.

The “loop” function can then begin. If the maze is not finished the “directional” function is called. This checks whether or not the positions adjacent to the player in the maze are walls or empty spaces that would allow the player to move in that direction. If a viable space is in that spot, the LED corresponding to the up, down, left, or right position is lit up green, if not the LED is red. Later in the maze as the player approaches the end, the LED will be lit up blue to signify the end is in that position.

From there, inputs from the joy stick will be taken. The joystick outputs two values corresponding to horizontal or vertical movement, how large or small the value is represents the direction the joystick is being pushed. For example, a value of about 2000 means that the joystick is stationary, over 2000 indicates movement to one side, while under 2000 is movement to the other side. The maximum values are about 4000 and 0. So the code checks for an output of over 3000 or under 1000. This can then tell us which direction the user is trying to move. To convert the numbers into directions, first we test if any movement is being made by seeing if any

of the values are outside of the 1000-3000 range. If it is, the “movement” function is called. The movement function then senses which value, the horizontal or vertical, is being changed and in which direction. A character representing the direction is then returned for the function. A new character is then set to the same character as the function and sent into the “updateGrid” function. This function then uses that character, which represents the desired direction to be moved in, to move the player in that direction. Although if there is a wall in the spot where the player wants to move, the function will not move the player and instead the motor on the joystick will buzz, giving a physical feedback that the movement failed.

```
      if( (joyX < 1000 && prevJoyX > 1000) ||
            (joyX > 3500 && prevJoyX < 3500) ||
            (joyY < 1000 && prevJoyY > 1000) ||
            (joyY > 3500 && prevJoyY < 3500)
            ) {

            char direction = movement(joyX, joyY);
            Serial.println(direction);
            updateGrid(board, xPos, yPos, direction);
        }
```

```
void updateGrid(char board[10][10],int & xPos, int & yPos, char direction) {
   board[xPos][yPos] = BLANK;
   if (direction == 'l' && xPos > 0 && board[xPos - 1][yPos] != 'W') {
     xPos--;
   }
   else if (direction == 'r' && xPos < lengthX - 1 && board[xPos + 1][yPos] != 'W') {
     xPos++;
   }
   else if (direction == 'u' && yPos > 0 && board[xPos][yPos - 1] != 'W') {
     yPos--;
   }
   else if( direction == 'd' && yPos < lengthY - 1 && board[xPos][yPos + 1] != 'W') {
     yPos++;
   }
   else {
     digitalWrite(motorPin, HIGH);
     delay(500);
     digitalWrite(motorPin, LOW);
   }

   board[xPos][yPos] = ROBOT;
}
```

The whole process can then be repeated, allowing the player to move freely inside the maze with the joystick. That is until they reach the goal. To check if the player has, an if statement is put at the end of the loop. This if statement senses if the “hasChar” function is false. What the function does is parse through every spot on the maze to see if the character that represents the end is present. If it is not it will return false. Because when the player reaches the goal, the player character will be placed on top of the goal character, the goal character will no longer exist making the function return false. The maze will then be ended and the time taken at the start of the run will be subtracted from the time at the end, giving the score that is then sent to the website.

The biggest issue we encountered while coding was transferring the code from the normal VSCode to the particle. The code was originally made to work in the VSCode terminal. This made it easier to test and bug fix, along with allowing the code to be tested before all the particle work was done. All the necessary functions besides those using the joystick were coded, and instead of using LEDs the code used “cout” to display whether the player could

move in each direction. The “cout” was eventually replaced with the LED commands and the joystick functions were added among many other changes needed to adjust the code for the particle. Upon compiling the code returned 99+ errors. Every bit of the code was then painstakingly debugged and issues were pervasive in every aspect of the 342 line code. The LED stuff returned a lot of issues in particular with many different lines of code needed to make it function. We also realized that the particle cannot access files. This was a major issue because all the maze maps were stored in files that were accessed using “ifstream” that no longer worked. We instead turned all the maps into arrays and stored them in the code. Far too many issues were encountered to detail them all, but those are just a few.

```
char easy[101] =
"WWWWWWWWWW-P-----OWW-WWW-WWWWW--------WWWWW-WWWPWWW---WWWWWWW-WWWPWWWWW-----WWWWX-WW
PWWWWWWWWWWWWWW";
char medium[101] =
"WWWWWWWWWW-P------WW-WWWWWW-WW----WO--WWWW-WWWW-WWP--W----WWW-WWWWW-WW---W----WWXW--
-WWPWWWWWWWWWW";
char hard[101] =
"WWWWWWWWWWX-------WW-WWWW-WPWW----W---WWWWWWWWW-WW--PWW---WW-WWW--WWWW-----W--WWPW-W
----WWWWWWWWOW";
```

Electrical Schematic