Final Project: Build Your Own Labyrinth

University of California, Davis
9 June 2024

Jason Feng and Andrew Hoang
Professor Soheil Ghiasi
UC Davis College of Engineering
Department of Electrical and Computer Engineering
EEC 172: Embedded Systems, CRN 39462

I. Description

Build Your Own Labyrinth is an isometric maze game where the player controls a marble through an obstacle course composed of 25 tiles of various heights. The game has two inputs for controls: the AT&T Universal Remote is used to navigate menus, while the LaunchPad's integrated accelerometer is used to control the marble. SPI is used to display data on the Adafruit OLED, and I2C to stream data from the accelerometer. AWS IoT is used to store level data and top-three leaderboards. There is a static frontend webpage hosted in GitHub Pages which will contain information about the game, such as instructions and a video demo, along with a level editor to submit custom levels and a level and leaderboard viewer. Finally, a Python backend is used to connect the frontend's level editor to the AWS shadow.

There are eight sample levels, along with eight user levels. Levels are encoded as a sequence of 25 letters, with each character representing a single tile in the menu. Level data is stored in AWS, with `GET` requests to retrieve level information to parse into a level. Each level additionally has a leaderboard, which stores the usernames of the top 3 high scorers, along with their scores. This leaderboard is updated when a user finishes a level with a higher score than leaderboard users.

In each level, the player must navigate a maze from beginning to end as fast as possible, tilting the LaunchPad to roll the marble in a direction. Score is based on time to reach the exit, starting at `800` and decrementing every frame to a minimum of `0`. Accelerometer tilt controls the velocity of the marble. The marble bounces off of walls and rolls down slopes, obeying physics, until it reaches the goal.

The frontend integrates a level editor into the project webpage. The frontend is written in Vue.js and Bootstrap for a clean webpage, and the level editor is written in Fabric.js for a responsive and interactive canvas. It can preview sample and user levels using `GET` requests to an intermediate backend, and upload new levels using `POST` requests. In addition to the level editor and viewer, the website contains project information such as gameplay instructions and a video demo.

II. Design

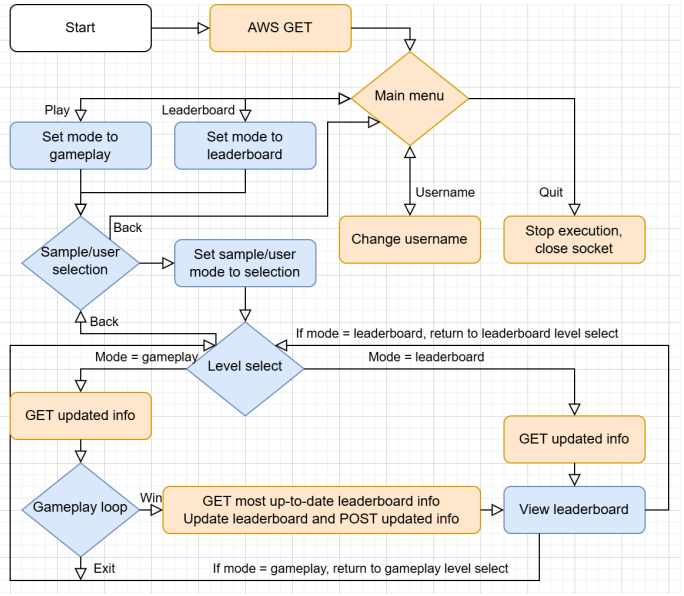

Fig. 1: State diagram of LaunchPad code. Andrew led orange tasks and Jason led blue tasks.

A. Start / Main menu

The LaunchPad application begins by connecting to AWS and using a GET to receive up-to-date level and leaderboard information. It then switches to the main menu, which has three vertical options: change play, change username, leaderboards, or quit. Upon calling the function to enter the main menu, the OLED draws the menu's graphics. Options can be picked by using remote input to move a cursor up or down and pressing `MUTE` to select, with the current selection indicated as a triangle to the left of the option.

During the connection phase, and each access to AWS, a loading indicator is displayed on the bottom of the OLED.

B. Change username

The change username function will change what name is saved to the leaderboard, if the player achieves a

high score. This username is PLAYER by default, and can be changed with this function, which utilizes the remote texting code from Lab 3, parsing IR signals as letters, numbers, and certain special characters. This function begins by drawing graphics for username selection, and then enters a loop to scan and parse for user inputs until MUTE is pressed. Numbers 0 and 2 through 9 will allow the user to type in a username with a maximum length of 10, using lowercase and uppercase characters, numbers, and certain special characters. BACK is used to delete one previous character. Pressing MUTE saves the entered username to be written to the leaderboards.

C. Play / Leaderboard, Sample / User, and Level Select

When Play or Leaderboard is pressed, the program sets the mode to the respective button pressed, and enters a menu to select whether to access either a sample level or a user level. When an option is selected, a third menu opens with the choice of eight sample or user levels to select from, depending on the user's choice. Choosing a level opens either the level to play or the leaderboard for that level.

D. Gameplay

During gameplay, the player controls a marble from a starting point to a goal on a 5x5 isometric grid made up of tiles of various heights and slopes. The player uses the LaunchPad's integrated accelerometer to roll the marble around the map, which is displayed on the OLED. During gameplay, any button on the remote can be pressed to exit the level and return to level selection. When the player reaches the goal tile, the OLED displays a victory screen and updates the leaderboards if the user scored high enough. This updated data is sent to AWS through a POST request, and the updated leaderboards are displayed to the user.

E. Leaderboards

The top three scorers of each level are displayed in a level's leaderboard. Users' names and scores are displayed vertically, first place through third place. Pressing any button on the remote returns to the previous screen.

F. GET and POST

Instead of making requests directly to AWS, requests are made to a public intermediate backend that acts as a wrapper around AWS. This backend handles data validation for user data, along with adding the three certificates validating the AWS request. The only information POSTed to the server is updated leaderboard information, which is calculated after a level completes. A GET is made before loading any level or leaderboard, to ensure the LaunchPad has the most up-to-date information. Andrew planned to have asynchronous GETs that constantly ensure data is updated while not degrading input responsiveness- however, it was found that GET requests complete in seconds and do not make a significant impact on gameplay experience. This idea was left as a stretch goal.

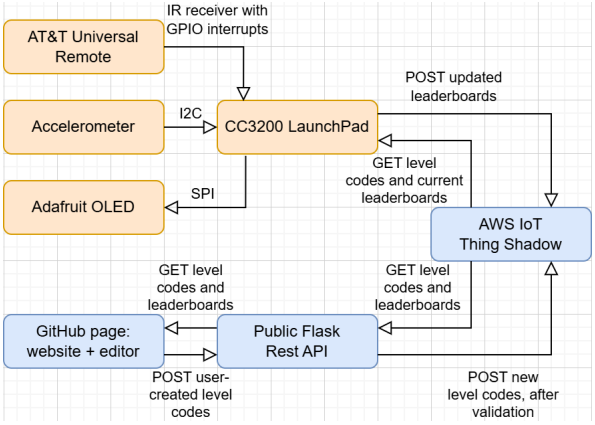

Fig. 2: System architecture of entire project. Andrew led orange tasks and Jason led blue tasks.

G. CC3200 LaunchPad

The CC3200 LaunchPad is used as a main hub to connect all hardware and software together using numerous communication protocols. In order to interact with the AWS server, the CC3200 connects to the internet via a combination of the Simplelink SDK , REST and the libsocket library. To get IR signals from the AT&T remote, the CC3200 utilizes the systick library and GPIO interrupts to translate the signals being sent by the IR receiver into human readable numbers. For interaction with the marble, the board already has a built in accelerometer which utilizes the I2C communication protocol. Finally, in order to display graphics, the GPIO outputs were used along with the SPI communication protocol to send signals to the Adafruit OLED. In terms of programming the Launchpad, the TI configuration tool was used to edit the pin layout of the board, Code Compressor Studio was used to write and flash code onto the volatile memory, and CCS Uniflash was used to flash code into the non-volatile memory.

H. AT&T Universal Remote

In Lab 3, Group 9 received the code Citizen 1041 from Lambert which was used throughout the final project. This code utilizes pulse length encoding to send information. This format encodes a 1 as a long pulse, and a 0 as a short pulse, with a 1 being approximately twice the length as a 0. Since this code was used in previous labs, there was no need to use the Logic Analyzer; however, if given a new code, it would be used to view the waveforms.

NotablyEach press of a number outputs a 32-bit code, and holding down the button outputs a shorter repeat code. The ENTER button did not output a signal, so the group decided to use the MUTE and LAST keys instead of ENTER and DELETE. The up, down, left and right arrow keys also do not output a signal- the group decided to use Channel up and Channel down for the menu screen of the game.

In order to receive signals from the remote, a Vishay TSOP31136 infrared receiver was connected to input GPIO peripherals. To protect the receiver from electrical over-stress, a capacitor and resistor were used to make a low-pass passive filter. This prevents high voltages produced by noise from breaking the sensor.

I. Adafruit OLED

The Adafruit OLED was used for displaying the graphics of the game. It receives signals from the CC3200 GPIO outputs using the SPI communication protocol. The students used the provided Adafruit graphics library to set the screen color, print out characters and draw shapes. All of the gameplay graphics utilized the library's ability to draw specific pixels to render the map, and the `fill_circle` function to draw the marble.

J. Accelerometer

The accelerometer was used to manipulate the position of the marble during the main gameplay loop. It is an onboard system that utilizes I2C to communicate the current X, Y and Z orientation of the board. The accelerometer sends information to the CC3200 when the CC3200 requests a read to the accelerometer's address through use of `I2C_IF_Write`.

K. AWS IoT Thing Shadow

The central link of Build Your Own Labyrinth is the AWS IoT Thing Shadow. The LaunchPad and the frontend store data in JSON format in this shadow, and use `GET` and `POST` requests to access or modify it. Each level's information is stored as a JSON object with string `grid` for encoded level data, and object `leaderboard` containing three strings `user1`, `user2`, and `user3` for a leaderboard per level.

The LaunchPad retrieves leaderboard information and level codes using `GET` requests to the shadow, and `POST`s updated leaderboard information to the shadow. It is controlled by the IR remote for menu navigation through GPIO interrupts, and the integrated accelerometer for gameplay using I2C. It outputs to the Adafruit OLED using SPI to display menus and levels.

L. Public Flask Rest API

The frontend indirectly interfaces to the shadow through an intermediate backend, which limits frontend requests to modifying only a subset of data. This intermediate backend makes `POST`s with new level data on behalf of the frontend, validating that level data is in the correct format, along with handling which level this new level should replace. `GET` theoretically does not require this filter, but the frontend lacks the certificates to make `GET` requests, so both requests are handled through the backend.

M. Public GitHub page: website + editor

The project webpage and demo, level editor, and level and leaderboard viewer are hosted on GitHub Pages as a single repository. The level editor is the highlight of the site, allowing users to edit and upload custom levels to be played on the LaunchPad. The level and leaderboard viewer displays levels, along with the top three scorers per level. The website uses Bootstrap 5 and Vue.js, and Fabric.js for the interactive canvases.

III. Implementation

A. IR Remote and GPIO interrupts

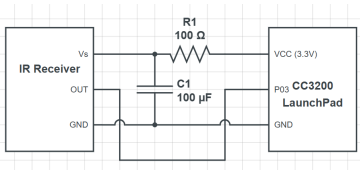

Fig. 3: Circuit diagram for interfacing the IR receiver to GPIO.

The IR remote is used to access the menus. GPIO interrupts triggered by the IR receiver circuit along with the Systick module will be used to parse IR input, and map each 32-bit sequence to a value. Because the arrow keys and OK are not read by the IR receiver, volume up and down are used to navigate vertical menus, and `MUTE` is used to select an option. There is additionally an option to change username, which utilizes the remote texting program from Lab 3 to enter text into the program, overriding the navigation functionality. During typing, `LAST` is used as a backspace character, removing the previous character typed, and `ENTER` is used to save the entered username and navigate back one page.

B. Adafruit OLED

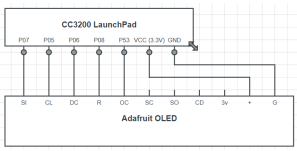

Fig. 4: Circuit diagram for interfacing the Adafruit OLED to SPI.

The pin used to communicate with the OLED was configured using the TI Configuration tool. Pin 7 was used for MOSI, pin 5 used for clock, pin 6 was used for DC, pin 08 reset, and pin 53 for OLEDCS (OC). This is reflected in figure 4. In terms of software, the students used the provided Adafruit graphics library to display various patterns and writing on the OLED. This required students to have the completed `writeCommand()` and `writeData()` functions whose logic goes as follows: Send DC signal via GPIO (data or command), enable SPICS to begin sending data, send a low signal over GPIO to OLED CS, send the data/command with `SPIDataPut`, disable SPICS, send a GPIO signal high to the OLED CS.

C. Start

At the beginning of execution of the program before the main menu, many of the systems require initialization. All of these initializations make up the start phase of the state diagram. First are the communication protocols. In this project we use SPI, I2C and the GPIO interrupts. Next the board is connected to the internet using REST, socket and simplelink.

To enable SPI, the OLED is first reset to get a fresh screen in case it was being used previously.

SPIConfigSetExpClk is then used to configure the special SPI pins. Particular parameters that were edited were the ulBitRate which was set to 100000, the ulMode which was set to SPI_MODE_MASTER and setting the SPI activation to SPI_CS_ACTIVEHIGH for ulConfig. SPI and SPICS were then enabled . SPI is also coupled with the Adafruit OLED which is calibrated using the given Adafruit_init() function.

Setting up the GPIO interrupts is a process that is a bit more involved but still relatively simple. GPIO pin 8 was used to take in the signals received from the IR receiver. During the initialization phase, GPIO pin 8 is configured to interrupt the program on every rising edge and is registered to the risingEdge function. The GPIO interrupts also require use of a systick timer which interrupts every 3000 microseconds and is registered to onExceedTimeInterval to reset the clock when enough time has passed. I2C is simply enabled by using I2C_IF_Open.

At this point, we reach the final portion of the configuration progress which connects the board to AWS services. Firstly, the host and port are configured which are the target website and port number. The CC3200 then connects to the access point and sets the time of when the program is being executed. These are set as macros in the defines, however, they need to be constantly updated for the CC3200 to successfully connect. Before the board can fully connect to the internet, the common.h file SSID_NAME and SECURITY_KEY and SECURITY_TYPE were edited to connect to Jason and Andrews own personal mobile hotspots. This enables the CC3200 to call tls_connect which concludes the starting process.

D. Menus

The menu system in practice is just an infinite while loop with an embedded switch statement that branches out depending on the y position of the cursor. Upon entering the while loop, the function init_mainMenu() which draws the screen and allows the user to press the Channel_up and Channel_down button to change the y position of the cursor which increments by 20 each press. Upon a Channel_up or Channel_down, the menu screen is redrawn with the cursor at the updated position. When the user presses enter, the function returns an enum gamestat which represents which state the game will transition to.

The more straightforward options are the username and quit options. By choosing username, the user is brought to a screen that allows the user to choose their username. Choosing to quit ends the connection between the CC3200 and AWS servers and exits from the program altogether. If the user's cursor lands in the leaderboard or play text, however, the game will transition to another screen which allows the user to select whether they would like to play/view a user made level or a sample level. It also sets the gameplayOrLeaderboards flag which will determine the gamestate later since both states call the same function. The mechanics work exactly like the main menu screen with the only caveat of displaying different texts. Regardless of whether the user chooses user or sample, the game will then transition to a final menu screen where the user can select from user/sample 8 levels. Upon choosing one of the eight levels, the program will then determine the level the user, get the most recent level data using http_getand either transition to the gameplay loop if the gameplayOrLeaderboards is equal to zero. Otherwise it goes to another screen which displays the top scores.

E. Change username

Upon choosing the username option from the main menu, the screen is painted black and a white "USERNAME" appears in the middle of the screen. To get the username from the user, much of the code from lab 3's part 3 was ported over. In brief summary, an infinite while loop constantly checks for new user input using a simple if statement and a function called parseOutputVal() which attempts to decipher the volatile outputVal. If there has been no user input the program keeps checking. After receiving input, the variable nextLetterTimer along with helper functions is used to interpret the variable's value. The timer is decremented in every iteration of the loop, while having a minimum value of 0. When the timer reaches a value of 1, it triggers a switch from looping through values of the current letter to saving the current letter and reading the next letter. Each time the same button is pressed, the timer is reset and the program iterates to the next character. If another button is pressed, the previous character no longer iterates.

Besides the base functionality of typing lowercase letters, caps lock and number mode buttons were implemented using the volume and channel select buttons. Up and down on volume enables and disables caps lock, and up and down on channel enables and disables number mode. When both modes are enabled, the remote treats inputs as holding shift and pressing numbers on a keyboard, resulting in special characters such as !, @, and %.

Initially, the user input would be printed from the left side of the screen and extend to the right as more characters were inputted. This was changed since it did not really match the aesthetic of old Atari's arcade games. Instead, the characters dynamically center as they are inputted. To do this, the entire username must be deleted and reprinted whenever the user inputs or deletes a character. This process requires heavy use of the outStr helper function. This function takes in a starting string position, a message, an Y value and a color. It then prints the string starting from the given position. Whenever the user inputs a new letter, the entire row is erased and the new current username is printed starting from the original start point minus the width of the username divided by two. When the user presses enter, the username is saved and the game state changes back to the main menu screen.

F. Leaderboard

There are two different mechanisms to the leaderboard: displaying and updating. When the user goes through the main menu with the leaderboard state and selects their chosen level, get_leaderboards is called which parses through the receive buffer from http_getto find the given

level using `strstr`. From there, each user along with their score is parsed, using `strstr` again, and copied into global buffers which are then all printed.

Updating the leaderboard only occurs after a user completes a level. First, `http_get` is called to get the most up to date information. Each user leaderboard data within the program is then cleared and the leaderboard for the recently finished level is loaded. From here, the times of the user are compared with the times from the leaderboard. Depending on which leaderboard score the user beat, the username and score will replace the old data and the leaderboard will be updated. If the user did not beat any of the scores, the same old names and scores are used to update the leaderboard. After completing a mission, the user is also forced into viewing the leaderboard screen using the above method of displaying the leaderboard.

G. Gameplay

If the user decides to enter a level instead of viewing leaderboards, the level's data string is parsed into a `5` by `5` array of 8-bit integers. Each value stores the height and slope of each tile. With heights ranging from `0` to `10` and `5` different slope types including a flat tile, the least significant `4` bits are used to store each tile's height and the next `3` bits store the tile's slope type. Next, the program enters a gameplay loop which constantly reads accelerometer data, handles collisions, and detects if the marble falls off the map or touches the goal. If the marble falls off the map, its position is reset to the level's starting position. Pressing any button on the remote in this state returns to the previous menu. The user's score is implemented as a counter decrementing each frame from `800` and reaching a minimum of `0`. Therefore, the player is encouraged to finish the level as quickly as possible.

Level data is stored as a string of `25` characters. `10` nonzero heights and `5` different slopes, plus an empty tile, results in 51 different combinations. This is represented with the 52 lowercase and uppercase alphabetical letters. Each letter `a` through `y` and `A` through `Y` maps to a value which encodes height and slope type. Any other character in the string is interpreted as an empty tile, which is equivalent to a tile with height `0`- the frontend implementation uses the letter `z` for an empty tile. This data is accessed when drawing the level, and when calculating collisions.

All calculations are done via integers to avoid floating-point rounding errors and casting floats to ints. Instead, very large integer values on the scale of `10,000` are used, and divided by some constant to result in `128` by `128` coordinates for rendering on the OLED. Additionally, despite being viewed from a 45° angle, coordinates are internally calculated upon a 2D square grid instead of a rhombus. This adds difficulty to parsing accelerometer input, but greatly simplifies collision detection and slope physics.

Before the gameplay loop is entered, a `128` by `97` grid is generated: the color of each pixel of the stage. This helps with redrawing the marble as it moves across the stage, as only some specific pixels should be redrawn every frame, instead of the hundreds of pixels of a single tile being redrawn every frame. The tiles are drawn in order of decreasing distance to the observer, with the backmost tiles being drawn first: that way, tiles in front overlay those tiles, avoiding complex logic for determining what pixels are shown.

The loop starts by retrieving the accelerometer's tilt on the X and Y axis with an I2C request. Due to the game being viewed from angle, the velocity vector ¡X, Y¿ must be rotated 45° counterclockwise to maintain consistency with the viewer. This is done through three trigonometric calculations: multiplying both values by (sqrt2)/2 (more specifically multiplying by 7,071 and dividing by 10,000), setting the current X velocity to the sum of the new vector's X and Y, and setting the current Y velocity to the new vector's X minus the new vector's Y.

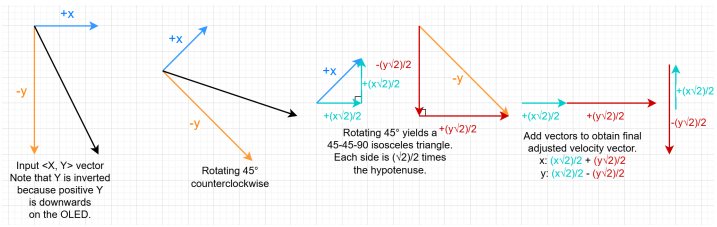

Fig. 5: Calculations for new velocity vectors, calculated every frame.

Next, the marble's absolute coordinates are translated into coordinates of the grid by subtracting constants from the marble's coordinates, along with dividing the results by another constant to scale the coordinates to the `5` by `5` grid. If the marble is on a valid coordinate of the grid, collision and slope detection logic takes place. First, the slope of the current tile is determined using the level data array generated before entering the gameplay loop. If the marble is entering the bottom of a slope of the same starting height or a flat tile of the same height, the marble is allowed to enter- otherwise, the marble's position is reverted to the position from the previous frame. If the marble is entering a lower tile, it is able to fall onto the tile. When the marble is on a tile, its vertical velocity is also set to `0`, and if it is airborne, its vertical velocity increases due to gravity's acceleration.

Once the new coordinate of the marble is determined, it is drawn onto the OLED. The marble's position during the previous frame is first overwritten by drawing over it with the pixels at that location. Next, the new marble is drawn. The drawn marble's `x` coordinate is found by subtracting `y` from `x` and dividing by a constant. The `y` coordinate is more complicated, as the marble's height must also be taken into account- the height is subtracted from the sum of `x` and `y`, then divided by a constant.

Once a new marble is drawn, it must be covered by tiles that are in front of it. Each tile closer to the viewer's perspective is queried: if the marble is completely covered by the wall, it is completely not drawn at all. Otherwise, if the right wall is between the horizontal bounds of the wall, the left side of the marble is overridden with tile colors- this same logic is applied to the left wall with the right side of the marble.

Finally, the time counter is updated with the current time, by displaying the hundreds, tens, and ones digit in

succession using division and modulus to isolate one digit at a time. One more check is done to reset the marble to the starting tile if its height becomes less than some threshold- if it falls off the grid. A slight delay is implemented between frames, and the loop begins anew.

When the marble reaches the goal, detected when the marble's grid x and y are both 4 and its height matches that tile's height, the level ends and a victory screen is shown along with the user's current score. A GET is sent to the shadow to retrieve the most up-to-date leaderboards. Next, the user's current score is compared to the three scores in the leaderboards, and the top three scores are selected. The updated leaderboard is displayed to the user, and a POST request is sent to the shadow with the new leaderboard, updating the global leaderboard for all users to see. Pressing any key will return to the corresponding level select.

H. LaunchPad GET and POST

GET and POST from the LaunchPad follow much of the same structure as Lab 4. The program begins with connecting to the Internet and opening a socket. Time is updated on the LaunchPad to enable encryption, the access point defined in common.h is connected to using connectToAccessPoint() from network_utils.c, and the connection to the AWS endpoint is finalized with tls_connect(). After the connection is initialized, GET and POST may be freely utilized. This connection remains over until QUIT is selected on the main menu, which safely closes the connection.

The http_getfunction makes a GET request to AWS. This function begins with an empty buffer of length 512. GET request headers are copied into this buffer to fulfill the request, with information on the base URL, endpoint, and connection. The buffer is then sent to the server, and the response from the server is stored in a global buffer of length 10,000- this response is long due to holding level and leaderboard data for 16 levels, along with AWS automatically inserting a timestamp for each field.

The updateLeaderboards function makes a POST request to AWS. Like http_get, it begins by appending headers to a buffer, with extra headers holding information about the format of the data to be sent, along with its length. The length is calculated by adding the results of multiple calls to strlen() to determine lengths of individual sections of the request body. The actual body of the request is appended next, by calling strcpy to copy level name and leaderboard information into the request body. After the request is sent, the server's response is printed to UART for debugging purposes, but is not used in gameplay.

I. Level and leaderboard data parsing

Level data must be extracted from http_get before it can be parsed. Two functions, get_level and get_leaderboards, accomplish this by using strstr to locate the index of certain JSON keys. Both functions begin with finding the occurrence of the level code: sample1 through sample8 and user1 through user8, depending on the player's selection. From this index, the next instance of "}} is located, signifying the end of the current level's data.

From this point, the functions diverge. get_level saves the level's grid data by finding the occurrence of substring grid and copying the 25 letters afterwards into a buffer. get_leaderboards does something similar, but with three separate buffers: it finds the occurrences of user1, user2, and user3 and extracts the next 14 characters. This data is the user's 10-letter username, one space, and the user's score as a three-digit integer. Although user1, user2, and user3 seemingly conflict with user1 through user8, the only data parsed at this point is the data for a single level, forcing the program to only output the leaderboard and not the level data.

J. Frontend

The project's webpage with information about the project and a demonstration doubles as a level editor and viewer for users to create and submit custom levels. The site is hosted on GitHub pages. It is structured with Bootstrap 5 and Vue.js for basic interactivity, plus custom CSS backgrounds. The level editor and viewer are Fabric.js canvases- Fabric.js was chosen for its simplicity in implementation and ease of use.

The editor's canvas is rendered as a 5 by 5 2D array of tiles, where each tile consists of 15 parallelograms: the top and two sides of the tile, for each of the five flat and sloped tiles. The tops of tiles turn light gray when hovered over with the mouse, and turn white when selected by clicking. When a tile is selected, number keys 1 through 5 are used to switch the tile from one slope to another, by making the current slope's three parallelograms invisible and making the desired slope's parallelograms visible. Tiles can also be dragged up and down for a total of 11 distinct heights, snapping to the grid by the process of dividing by a constant, rounding the result, and multiplying that result by the same constant. A tile dragged to the lowest height turns a dark gray, and is treated as a missing tile- in gameplay, a pit will appear in place of the tile, as an obstacle. Saving the created level first translates each tile's data into a string of 25 letters, mapping the 2D tile grid to a 1D string in row-major order, then POSTs the data to the backend, along with data on which user level to save the data to.

The level viewer reuses much of the editor's tile code to display existing levels, with the main difference that levels cannot be edited. Instead, when the webpage loads, it sends a GET request to the backend to retrieve updated data of all levels and leaderboards. It then processes data into JSON objects which can be parsed by the canvas to display a level. When a button is clicked, the corresponding level data is loaded into the canvas, changing tile heights and appearances to match the level data. Leaderboard information is additionally output at the bottom of the page, underneath the canvas.

K. Public Flask Rest API

The intermediate backend is written in Python's Flask framework, and accepts GET and POST from the frontend, and makes GET and POST requests to the LaunchPad's

AWS IoT shadow. This intermediate backend is written for security purposes- frontend users may be able to use inspect element to change the contents of `POST` requests. As a remedy, this backend acts as an intermediate step between the frontend and the device shadow, and will only `POST` to the LaunchPad if requests have valid data. The frontend will therefore not have the certificates required to `POST` and `GET` to the LaunchPad, and instead make requests purely through the backend, which does have credentials. This backend is hosted on PythonAnywhere.

This backend acts as a wrapper around frontend requests. When a `GET` is received from the frontend, a `GET` is sent to AWS, along with the three device certificates. The result of that `GET` is returned to the frontend. A `POST` is slightly more complex- the backend must ensure that the data is valid before sending it to AWS. One constraint it adds to the input data is that all slopes must have a height of `9` and under, as slopes with height `10` cannot be properly rendered by the OLED, due to being too high. Additionally, input data is checked to ensure the string has length `25` and consists of valid characters. The level to be saved is checked as well, to ensure its value is between `1` and `8`. If checks fail, an error code and message are returned to the frontend- otherwise, it formats the data in the AWS format and sends a `POST` to AWS, relaying the returned information to the frontend.

IV. Challenges

For the main menu, most of the functionality already existed within the previous labs and did not require significant debugging. The main challenges of getting the menu system to work included getting letters to dynamically center in username mode. Initially, when inputting the username, the letters would emerge from left to right. This, however, made the screen empty and not as pleasant to look at. To implement the dynamic centering, the main issue was knowing where to erase and update characters since any user input would change the position of every character. We solved this problem by erasing the line entirely and redrawing the entire char array.

Another issue that occurred was the main menu incorrectly changing game states when attempting to play. Whenever a user would attempt to choose a level regardless of whether it was a user or sample level, an additional level select menu would appear urging the user to select a user level. Additionally, levels selected would not be the levels that actually displayed on screen. The cause of our additional menu screen was due to a flag variable that controls the game state changing from gameplay to leaderboard containing a garbage value specifically when attempting to play. The solution to this problem was to initialize the flag variable to zero during setup. The issue of the wrong levels being loaded was due to incorrectly parsing through the AWS shadow data. Not only would the level data be loaded but also the leaderboard data. This was fixed by using a separate variable whose purpose is solely to store level data using `memcpy()`.

One problem encountered regarding gameplay is that there is not enough memory in the LaunchPad to fit a full `128` x `128` grid of pixel colors. The team's initial plan was to maintain an entire array of pixels for O(1) access, removing the need to recompute each tile's pixels each frame. Although using a full 128 x 128 grid was not possible, a `128` wide by `97` tall grid was used instead, which was enough to hold every pixel while not exceeding the memory limit. This grid maps to the bottom `97` pixels of the OLED display, and the upper `31` pixels are simply mapped to black.

Regarding communication protocols and sensors, there were no real issues in implementation. The one issue which was swiftly solved (after a day) was Andrew's I2C accelerometer not working. The issue was resolved by changing the onboard jumper configuration- one misplaced jumper was causing the accelerometer to be unable to return data.

By far the most pressing challenge was the collision and rendering engine during gameplay. The team spent 20 hours over a weekend mapping out physics logic, and implemented rendering after many mathematical calculations to determine where items should be rendered on the OLED. This task was made even more difficult with the addition of slopes, which required 10 more hours of calculations to ensure the ball can roll up and down slopes. The team successfully finished the engine to achieve a smooth physics engine and an impressive isomorphic point-of-view.

V. Future Work

There are three main features that we would have liked to implement if we had additional time. This included additional gameplay features, a way to successfully interrupt the AWS push at the end of the level if the user loses connection, and a way to automatically update the user levels without degrading the responsiveness of the UI/gameplay.

Currently, the gameplay mainly only features being able to move a marble on the screen. Ideally there would be other obstacles apart from falling off the edge of the map. These would be things such as trampolines, fire, and moving entities. These things would make the game a bit more engaging.

One possibility not accounted for is the LaunchPad becoming disconnected from the internet. As of now, the program simply hangs and becomes unresponsive when disconnected before finishing a `GET` or `POST`. One possible fix for this is to exit the request if it takes longer than some time value, and roll back level and leaderboard data to the previous level and leaderboard data snapshot.

User level automatic updating can be done through a separate LaunchPad connected to the original LaunchPad through UART or similar data transfer protocol. The current implementation makes a `GET` request before every level selection to ensure that level and leaderboard information is up-to-date. A secondary LaunchPad could enter a loop of constantly sending `GET` requests, and updating the main board's level data asynchronously. This would allow the original board to continue executing gameplay without requiring a `GET` immediately before the level begins. In the end, this idea was given low priority, as `GET` and `POST`

requests already execute in mere seconds.

VI. Bill of Materials

This final project is software oriented and does not require any material that hasn't already been provided in class. Apart from the already required CC3200 Texas Instrument Microcontroller and AWS account, this project will require the AT&T S10-S3 remote, a Vishay TSOP 31336 Infrared receiver, a 100 ohm resistor, a 100 microfarad capacitor, an Adafruit OLED display board, and various circuit components to connect modules together. No other materials are used, and the project can be completely recreated using only provided laboratory materials.

Item	Description	Manufacturer	Cost
1.5" SSD1351 128x128 RGB OLED	OLED display for menus and gameplay	Adafruit	$39.95
CC3200-LAUNCHXL	LaunchPad microcontroller	Texas Instruments	$66.00
S10-S3 Universal Remote	Controller for menu navigation	AT&T	$56.77
AA battery (2pc)	Powering remote	Generic	$1.20
400 Point Solderless Breadboard	Connecting LaunchPad, OLED, and IR circuit	Generic	$0.99
Assorted breadboard jumper wires (10pc)	Connecting LaunchPad, OLED, and IR circuit	Generic	$0.67
TSOP31336	IR receiver circuit	Vishay	$1.40
100Ω resistor	IR receiver circuit	Generic	$0.06
100μF capacitor	IR receiver circuit	Generic	$0.40

References

[1] *CSS Gradient - Generater, Maker, and Background*, https://cssgradient.io. Accessed 30 May 2024.

[2] *EEC172 Final Project Assignment*, https://ucd-eec172.github.io/labs/project.html. Accessed 27 May 2024.

[3] *Fabric.js Javascript Canvas Library*, http://fabricjs.com. Accessed 30 May 2024.

[4] *PythonAnywhere*, https://www.pythonanywhere.com. Accessed 4 June 2024.