Allen Lee, Roshni Sahoo, Brian Xie, Mary Zhong

6.S08 Final Project: 1P and 2P Tetris

I. System Documentation

▪ Introduction

We implemented one and two player Tetris. Tetris is played on a 10x20 board, and a random sequence of different tetrominoes fall down the board. The objective of the game is to manipulate the blocks using translations, rotations, and swaps to fill horizontal rows on the board. Once one or more rows are completed, the player's score increases, the row is cleared from the board, and the blocks above the completed rows fall. As the game progresses, the blocks fall faster, increasing the difficulty of the game. The game terminates once the blocks stack to and reach the top of the board.

In 1P mode, the player seeks to maximize his/her score. In 2P mode two individuals play Tetris on their own separate devices with similar rules as in 1P mode. However, there is a major difference: when a player clears a row, not only does his score increase, the player also handicaps his opponent by adding a row to the opponent's board. The game terminates once a stack of blocks on one of the player's boards reaches the top.

We also created a viewable database for players to search for their own personal best score (for 1P mode) and ELO ranking (calculated from win/loss rate in 2P mode), as well as see the top ten highest ranked individuals for 1P mode.

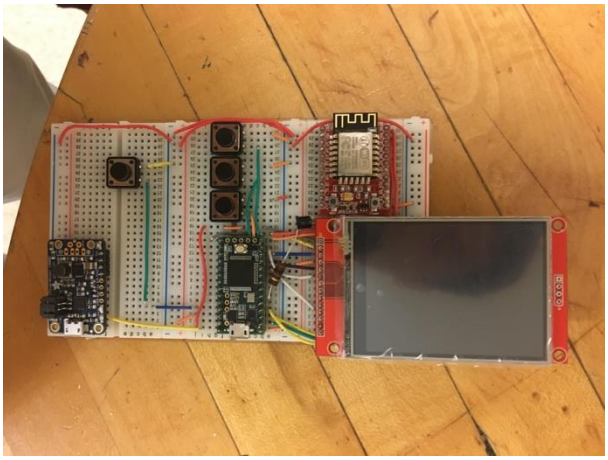
▪ Hardware

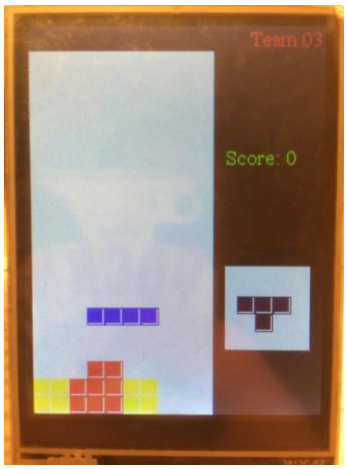
We used a Teensy 3.2 Microcontroller, Color 320x240 TFT Touchscreen, ILI9341 Controller Chip, ESP8266 Wifi Module, and 4 buttons. We connected the Wifi and buttons to the Teensy as we have done in the labs. Every button was connected to GND and a pin on the Teensy (5, 6, 7, 23). For Wifi, TXD, RXD, VIN, GND, were connected to RX1, TX1, VIN, GND, respectively, on the Teensy.

Instead of using the OLED from the lab, we used a Color 320x240 TFT Touchscreen with ILI9341 Controller Chip for our system to have a larger and colored display. We connected all 14 pins to the corresponding pins on the Teensy, including the Touchscreen pins, as according to Reference 1. Additionally, we had a 2n3904 transistor, which was linked to pin 3 on Teensy by a 100 ohm resistor, the VIN pin on Teensy, and the LED pin on the TFT screen.

Display Logistics:

The screen is 320x240 pixels. Recall that our Tetris board is 10x20. Each square on the board is displayed by a 15x15 pixel square. When playing Tetris, the board is displayed at the left, and other information, such as the score and stored block, is printed at the right hand side of the screen.





We implemented the board by creating a coordinate system such that we can access the square in a particular row and column using only the *coordinate number*, as shown. To calculate the coordinate number, we compute *coordinate number* = 10**row* + *column*, and for the other way around, *row* = *coordinate* / 10, *column* = *coordinate* % 10.

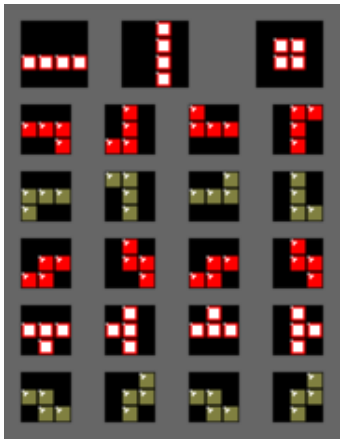
	Column 0	Column 1	…	Colunn 9
Row 19	190	191	…	199
Row 18	180	181	…	189
…				
Row 1	10	11	…	19
Row 0	0	1	…	9

C Libraries:

We created Block and Button classes that contain methods and attributes that aid in implementing the game.

Block library:
In Tetris, there are seven different types of shapes, which we refer to as Square, Stick, S, Z, L, J, and T. Each block has an inherent shape and number of possible rotations about a pivot point, and we distinguish them from each other by color. See the table and image below for details:

Name	Color	# of Rotations
Square (O)	1 (Yellow)	0
Stick (I)	2 (Blue)	2
S	3 (Red)	2
Z	4 (Green)	2
L	5 (Orange)	4
J	6 (Pink)	4
T	7 (Purple)	4



We will discuss specific implementation details for our Board class in part VI.

Button Library:
The button library contains the same Button class we implemented in homework exercises and labs. We also added some additional functionality that we will discuss in part VI.

Server Side:

1P Mode: We POST to the database once a game terminates. We post the player's Kerberos and their score to the database.

2P Mode: Before the game starts, we need to establish a connection with another player. We have a separate database to which players post their kerberos and status (ready or terminated), and the number of rows cleared (we'll use this later). Once both players are ready, the game starts. During the battle, every once in a while we'll POST an update and GET an update from the opponent, and use this information to appropriately add rows and terminate the game. Once the game is terminated, we GET the players' ELOs, and post our updated ELOs.

Rankings: There are two general types of GET requests – the first is high score and ELO search by kerberos. The user inputs the desired player's kerberos, in which case the GET request takes in one parameter, the kerberos. The second type of request simply GETS the ten top ranked players by high score, and displays their kerboroi, high scores, and ELOS. This request is called whenever the user enters the rankings and top scores page, and has no parameters.

We will discuss specific functions and implementation in part VI.

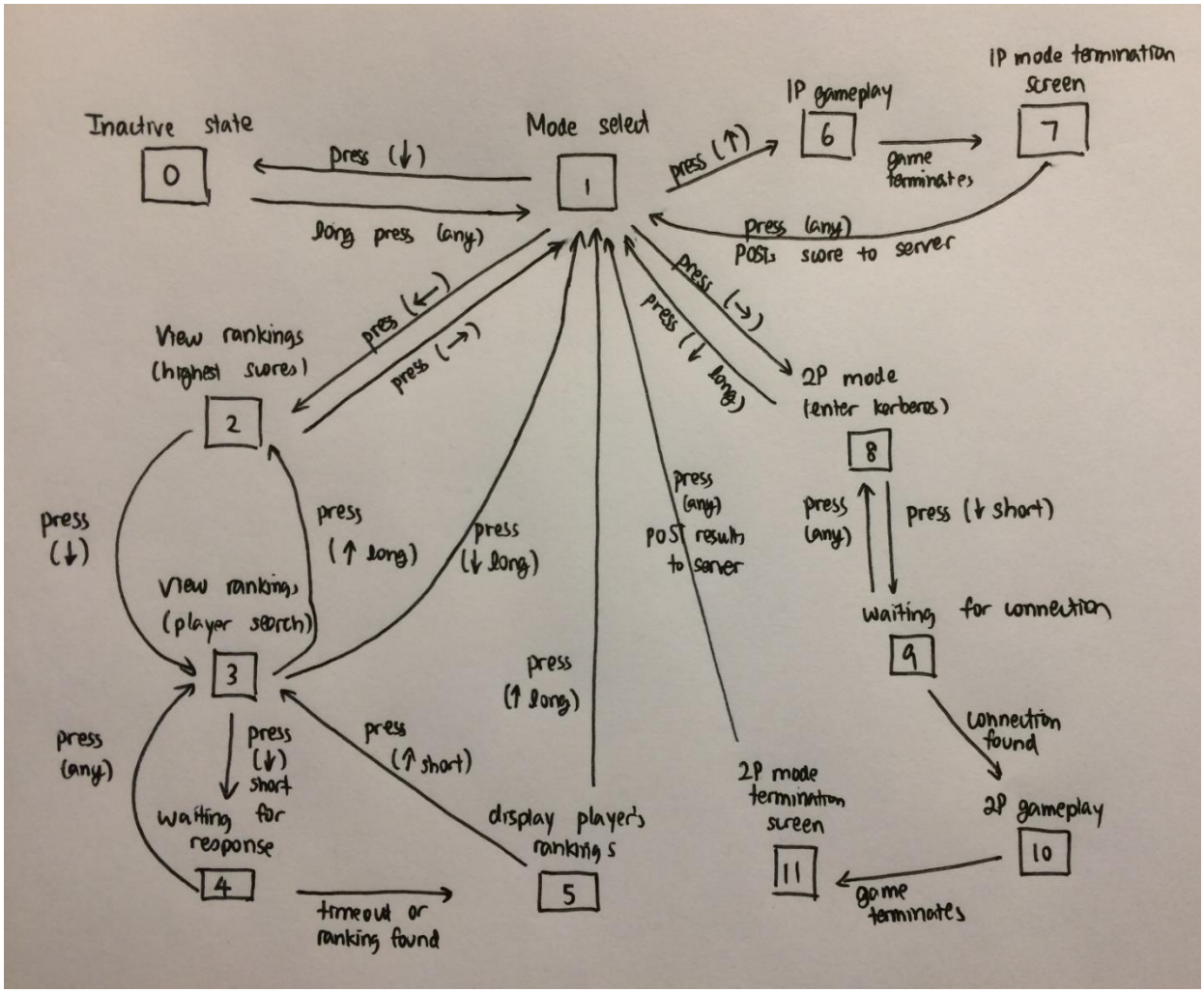
II. Functional Block Diagram

A block diagram is not applicable to our project since we only have four major parts – the Wifi module, the TFT Touchscreen display, four buttons, and the Teensy itself.

Essentially, the Teensy is a central node – all other components interact directly with it. The Teensy activates the Wifi whenever it needs to post scores to the server or get rankings/ELOs. The Teensy updates the display whenever necessary. The touchscreen and buttons send updates to the Teensy.

III. State Machine Block Diagram

Attached below is our state machine diagram. Note that we have four buttons (which are labeled UP, DOWN, ←, and →, like a console controller). States 2-5 are for ranking display, states 6-7 are for 1P mode, and states 8-11 are for 2P mode.



IV. Design Challenges, Fixes, and Rationales

Changes from Project Proposal:

- We used four buttons instead of two, corresponding to up, down, left, and right.
- We ended up using many buttons instead of the IMU, for several reasons:
  - Gameplay is more intuitive with buttons, as it simulates a console controller well.
  - Using the IMU can be very inaccurate, and precision is very important for a game like Tetris.
  - Buttons have an ideal level of sensitivity using them minimizes the amount of lag experienced.
- Instead of having a timer as the game termination state for 2P mode, it made more sense to structure it as a "battle" in which players try to handicap the opponent by sending over rows.
- We didn't use the GPS – instead of challenging nearby players by location, we search for the desired player by kerberos.

- We did not end up using a speaker or LED in situations such as when a player issues you a challenge. It was not a priority because it would not enhance gameplay, but it is something we would implement if we had more time.

Some Challenges and Fixes/Rationales:

1. At first, we wanted to create a nice inheritance structure to implement the Block class. We envisioned that we would create an abstract parent Block class, with various subclasses that implemented functionality of the different blocks. However, we were unable to implement this properly, as the compiler seemed to be unable to determine the type of an instantiated object correctly and the overriding didn't work. We decided to instead just use one Block class, and differentiate among the blocks using the *color* field and doing casework, since each block has a different color.

2. We had trouble displaying the information nicely. At first, we redrew the whole screen every time we updated something, which caused the display to flash. To resolve this, we created the *old_coords* field in the Block class. This allowed us to simply paint white over the old coordinates instead of redrawing the screen.

3. In our original implementation, we had a lot of latency, particularly when we translate or rotate the block. This is because we only redrew the entire board every time the block fell down due to gravity. To resolve this problem, we used the *old_coords* field and separated the board update mechanism from the gravity. This way the board would update immediately once translation or rotation occurred and the awkward lag disappeared.

4. We also encountered many logical problems, mostly pertaining to collisions that would result from rotations and translations. It was relatively easy to check whether a collision would result from a translation, so we simply prevented the translation from occurring if it would create a problem. Coming up with a comprehensive heuristic for rotations seemed infeasible due to the variety of block shapes and possibilities, so instead we adopted the "rotate and revert" method, in which we'd rotate the block, check if it resulted in any issues, and reverted the rotation if it did.

5. When we first set up pulling from the 2P database, we often got ridiculous data (such as 2 million rows cleared!). This happened because we were getting data from ALL of the past posts, not just the ones pertaining to the current game. We resolved this problem by including a *last_id* variable. Every entry in the database that has an ID lower than *last_id* would be ignored. We updated *last_id* each time we pulled from the database. This also elegantly solved the problem of overcounting row clears.

6. We had a lot of trouble with trying to access people who don't exist on the 2P side and initializing *last_id*. While the *last_id* is elegant, it needs to be initialized at first. The way we resolved this was to initialize *last_id* to -1, and whenever we got unreasonable data for row clears (> 9), which only happened initially, we'd set row clears to 0. To solve nonexistent kerberoi, we simply POSTed onto the database as we transitioned to 2P gameplay so that there would always be something POSTed.

V. Parts not in Base System

- Color 320x240 TFT Touchscreen with ILI9341 Controller Chip
- 3 extra buttons (4 total)

VI. Detailed Description of Code

1. Block.cpp/Block.h

Instead of using the exact coordinates of the block to perform rotation/translation/collision detection, we chose to encapsulate the relevant information in the *pivot* and *orientation* fields. Given a *pivot*, *orientation*, and the type of the block (*color*), we can uniquely determine the coordinates of the block. Translation (including gravity) can be implemented by simply changing the value of *pivot*, and rotation can be implemented by simply incrementing *orientation*.

We instantiate blocks using a constructor that takes in an int clr, corresponding to the color (which also indicates the type of block we want). Each block has the following attributes:

- Int *orientation*: denotes which rotation state the block is in (ranges from 0 to one less than the number of possible rotations)
- Int[4] *coords*: stores the current coordinates of the block
- Int[4] *old_coords*: stores the previous coordinates of the block
- Int *color*: value from 1-7, which is mapped to an actual color
- Int *pivot*: a coordinate number – the pivot point at which the piece rotates around

The Block class also contains the following methods. Note that many of these depend on the type of block at hand.

- *getColor()*: returns the color/type of the block
- *getCoordinates()*: returns the coordinates of the block
- *getOldCoordinates()*: returns the previous coordinates of the block
- *rotate()*: increments *orientation*, calls *setCoordinates()* to update *coords*
- *setCoordinates()*: updates the values of *coords* and *old_coords* given the current values of *pivot*, *color*, and *orientation*

2. Button.h/Button.cpp

The Button class is very similar to what we already implemented in class. However, there were several additions we made, so that we could “press and hold” a button and have a continuous effect.

- *getState()*: Gets the *state* (0 = unpressed, 1 = tentative pressed, 2 = short press, 3 = long press, 4 = tentative unpressed) instead of the *flag*.
- *getPress()*: Gets whether or not the button is pressed (essentially whether the *state* is 2 or 3). Used for in-game gravity button.
- *getPressForScrolling()*: Same as *getPress()* but digitally reads from the appropriate pin first. Used for scrolling through letters.

3. Board class

Within the .ino file, we define a Board class, which implements the board functionality. We first define some constants (e.g. screen size, square size, etc), timers for updates, and other relevant fields (e.g. instance of current block, gravity – how fast blocks fall). The most important field is *board*, an occupancy matrix of integers using our coordinate system. 0 indicates the square is empty, and 1-7 indicate that coordinate is occupied by a square of the corresponding color.

The constructor *Board()* and function *reset()* reset the board to all 0s, the default and current gravity to 500 (one drop every 500ms), and initializes other instance variables and timers to zero.

The other relevant functions are:

- *update()*: this is the backbone of the update mechanism. It first checks whether there is a collision by calling *check_collision()*. If there is, it calls *check_row()*, which checks to see whether there are any row clears and handles them appropriately. If there is a row clear, it calls *initialize_block()*. Finally, it calls *pretty_print()* to update the display.
- *fall()*: moves the current block down one square
- *translate()*: moves current block either left or right depending on the direction passed in, and checks to make sure the translation is valid (i.e. doesn't crash into other pieces/walls)
- *lock_piece()*: called whenever there is a collision. Since the current piece is moving downwards before it collides, we don't want to put it into the *board* array until it collides and becomes locked, during which we change the square values the current block covers to the number corresponding to its color.
- *check_collision()*: checks whether the current piece has collided into something below
- *initialize_block()*: initializes a new block using a random number generator at the top of the screen. Within this function we also check whether we've hit the end game condition – whether blocks have built up on the board to the top. Returns true if the game should terminate.
- *rotate_current_piece()*: checks if rotating the current piece is valid (doesn't crash into other pieces/walls), and if it is, rotates the piece by calling Block.rotate()
- *check_row()*: checks if there are rows that are filled, and calls *delete_row()* for each such row
- *delete_row()*: deletes a particular row from a board, and moves every row above it downwards one square
- *pretty_print()*: prints the current state of the board onto the display. In order to prevent our screen from flashing, we only uncolor the old coordinates of the block and color the current coordinates of the block. We do this by repeatedly using *fill_one_square*, passing in either the color or 0, which indicates that we should fill it with white.
- *fill_one_square()*: fills one square corresponding to a square in our coordinate system. Takes in the particular coordinate as well as the color to fill it with as arguments, and draws the appropriate rectangle.
- *holding_print()*: calls *fill_hold_one_square()* four times to print the reserved block at the right of the screen
- *fill_hold_one_square()*: essentially same as *fill_one_square()*

4. Ino (main) file

We first instantiate and set up all the relevant objects, such as the board, wifi, and buttons. We also declare constants such as update times, and initialize the timers. The loop function instantiates the state diagram mentioned earlier. It first gets the readings from the button classes, and then based on the current state updates the status of the system/display and takes appropriate actions. In particular, in the gameplay modes (states 6 and 10):

- If the down (gravity) button is being pressed, it temporarily changes the gravity to 40 (blocks fall one square every 40ms). This is especially useful for 2P mode in which speed is important.
- If the up button is being pressed, it rotates the current piece.
- The left and right buttons translate the block.
- Touching the screen swaps the current piece with the reserved piece. However, a piece cannot be swapped more than once between consecutive collisions.
- The blocks will automatically fall based on a gravity timer.

The states relevant to GET requests first get information using a server side utility function. It then prints the appropriate information to the screen. The states relevant to POST requests construct the string that should be passed into the request, and calls a server side utility function to perform the POST.

The other states are well documented in the UI of the system – the display will print directions whenever you enter a particular state, so we will omit the discussion here.

5. Server side code

Within the .ino file, we also have several utility functions. Note that most of the important processing happens in the python server code.

- *post_2p*: Posts to the 2P database. Takes three arguments: Kerberos, status, and number of rows cleared.
- *get_2p*: Pulls from the 2P database a status update for a particular player. Takes two arguments: Kerberos and last_id to consider.
- *send_data*: Used for rankings database – posts score, ELO, and corresponding Kerberos.
- *pull_top_score*: Gets the top score for an individual from the rankings database.
- *pull_elo*: Gets the most recent ELO for an individual from the rankings database.
- *get_rankings*: Implements state 2 – gets the top 10 individuals by top score.

In addition, we have two python files that handle the server side processing:

- *my_ranking_server.py*: Handles communication with the rankings database. If it is a GET request, we either get the top score and ELO of the particular player if a kerberos is specified or just get the kerberoi, high scores, and ELOs of the top ranked individuals if no kerberos is specified. We always order by decreasing order of highest score, or by time for ELO. POST requests insert the relevant information into the database.
- *my_2P_server.py*: Handles communication with the 2P database. When we do a GET request we always order by time to get the most recent updates, and return a string that contains the number of rows opponent has cleared, opponent status, and most recent ID considered. POST requests are similar but the other way around.

VII. Power Management

Power management is impossible to implement in any meaningful way while the device is being used, since Tetris requires very low latency updates. Here are some changes we made to reduce power consumption during other settings:

1. *Remove all unnecessary devices*: The first obvious thing we did was remove all unnecessary devices that could consume power, including the GPS module, the microphone, and the IMU.

2. *Sleep the display in inactive mode*: Whenever we enter inactive mode (state 0), we would sleep the display as it is unnecessary during this state. We wake it up when we enter mode select (state 1) via a button press.

In this system, we use a 1200 mAH battery that has voltage 3.7V. Below are estimates for how long this battery will last given different cases of usage:

1. *The system is in sleep mode (state 0).* In this state, the Teensy is in deep sleep and consumes 0.0086 mA of current at 5V. The color screen will be fed an input of 0V in the LOW mode, so it won't be consuming any current. The wifi module stays in modem sleep mode, running at 15 mA and 5V. The Teensy sleeps for 49 ms, then wakes up for 1ms (and runs with 34 mA while awake) to check if there is a button press that would wake it up. With these numbers, the system in state 0 would require 79.833 mJ of energy per second, meaning that the battery would last about *56.6 hours*.

2. *The system is in 1P mode.* In this mode, the Teensy is always on, operating at a current of 34 mA. The wifi module is in modem sleep mode (15 mA), except when its sending data to the server (posting scores), and operates at ~145 mA. The color screen operates at about 100 mA. If we suppose transmitting data through the wifi module takes about 100 ms, and each game is around 3 minutes, the system would require about 575.361 mJ per 1 second, which means that the battery would last approximately *7.71689 hours*.

3. *The system is in 2P mode.* Once again, the Teensy is operating at 34 mA and the wifi module is operating at 15 mA, and when it transmits the current goes up to 145 mA. In this mode, however, we also receive data from the database, and when this happens, the wifi module runs at ~55 mA. Finally, the color screen still operates at about 100 mA. Once again, we assume each game is 3 minutes long, but this time, we receive and send row clear and status data every 5 seconds. The system would require about 592 mJ per 1 second, which means the battery would last approximately *7.5 hours*.

References:

1. **https://www.pjrc.com/store/display_ili9341_touch.html**