Final Project: SammySays

ECE 167: Sensing and Sensor Technologies
Prof. Josephson
Group: Teresa Begley and Daniel Krygsman
UCSC - Winter 2023
Date: 3/17/24
Git ID: dfcbbc611af0a14a19fd83a196a260b1d0221e03

Table of Contents

Part 1: Introduction

This project is a game inspired by the children's game 'Simon Says,' the electronic toy with four buttons. In the original game, the toy flashes a sequence of colors (with accompanying tones), and the goal is to repeat the sequence using the buttons. The sequence gets longer and longer, adding difficulty to the game.

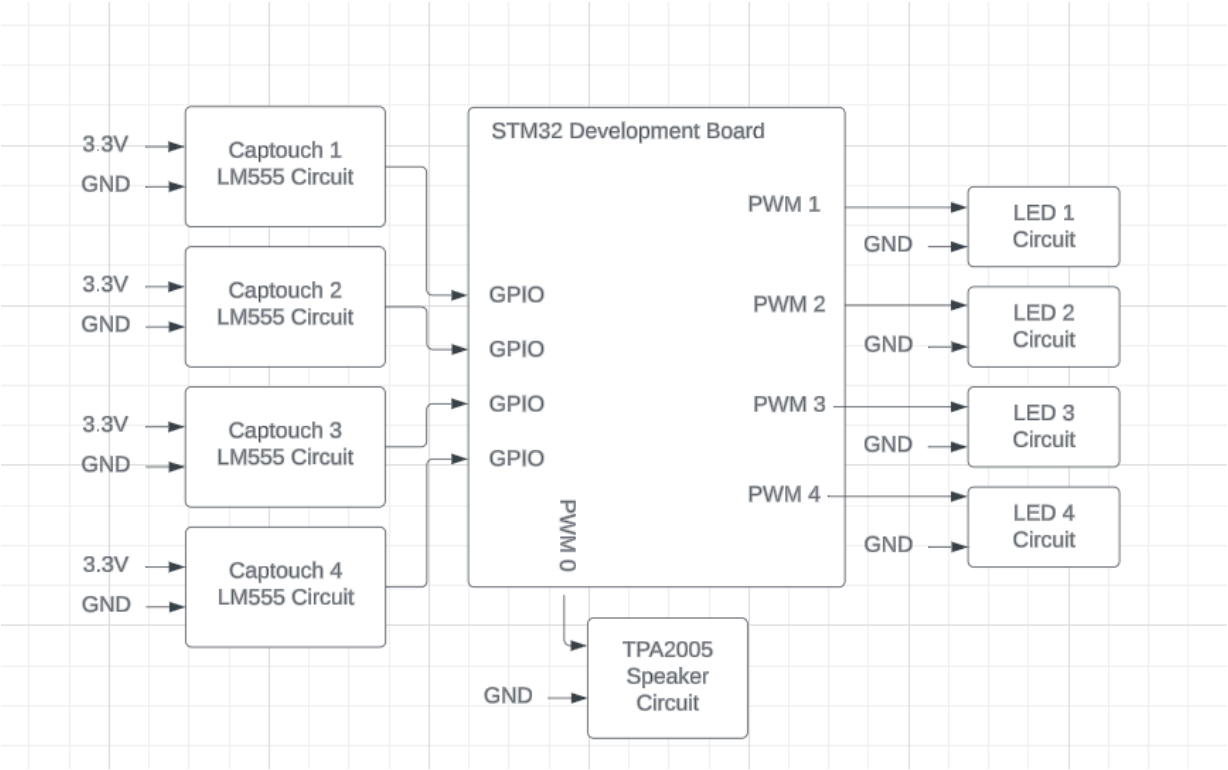

Fig. 1: System block diagram of the SammySays project.

Fig. 2: The game in its case, fully constructed.

Our implementation used four capacitive touch sensors in place of the buttons, built into relaxation oscillators to capture the touches in the form of changes in frequency. These relaxation

oscillators were constructed using the LM555 timer chip. In order to control the game and read from the sensors, the STM32 microcontroller was used alongside a proprietary UCSC I/O shield.

Since the STM32 has far more than four general purpose input/output pins and sixteen external interrupt lines, four of these pins were chosen to act as inputs for the capacitive touch sensors. Corresponding interrupt service routines were constructed to trigger on a rising edge from each sensor.

Finally, using a state machine, the sequence which must be followed is randomly generated and displayed on the toy by lighting LEDs around the buttons and playing specific tones. Each game is timed, and the user is required to hit the correct sequence of buttons within a specific time window, determined after some experimentation. The high score and current level is displayed on the microcontroller's OLED display. Lastly, a 3D printed case was constructed to hold the game.

Necessary Parts

- 4 capacitive touch sensors
- 4 LM555 ICs
- 22 pF and 0.01uF capacitors
- STM32 microcontroller with OLED display and I/O shield
- Resistors (680k to 1M ohms)
- 4 LEDs
- Audio amp and speaker
- 3D printing filament

Part 2: Background

As mentioned above, although the ordinary children's version of this game uses buttons, our version relies on capacitive touch sensors. These sensors are widely used, and potentially offer greater durability than buttons due to the fact that they have fewer (in fact, no) moving parts.

The idea behind a capacitive touch sensor is that it is a capacitor which, on being touched, will change its capacitance by some measurable amount. This change in capacitance is typically measured by observing the change in the associated time constant.

The particular capacitive touch sensors used in this project were fabricated specifically for ECE167. They are a small pad with a grounded grid on the underside; when touched, an unstable capacitance change in the picofarad range is generated. When used in concert with a relaxation oscillator, this change can be measured by the STM32 microcontroller in the form of a change in oscillation frequency, the details of which will be further discussed below.

Part 3: Implementation

3.1: State Machine and User Interface

3.1.1: Overview

The first step towards implementing the game was to create a state machine and judge the timing which would set the correct balance between comfortable and challenging for a human user. Some of the basics of the user interface were also decided and implemented during this stage, such as the timing for each state and the display on the OLED.

3.1.2: C implementation

The implementation of the state machine is broken down into six states: chill, prompt, delay, readying, correct input, and incorrect input. The chill state is the initial state of the game which only relies on whether any of the capacitive sensors have been touched. This serves as a waiting state, waiting for a “start game” signal. The chill state also initializes an array of size 100 int variables, the initialization using the rand() function and modulus operator to only initialize indexes with values 1-4. If this signal is received we move into the prompt state which prompts the player with one of the four pwm led outputs, and setting the speaker pwm pin to a frequency which corresponds to the array’s, 1-4 value, at index “n”. Then the state machine automatically moves to the delay state, which turns off the prompted led pwm output, and turns off the speaker's pwm output. The state machine then returns to the prompt state. Both the prompt and delay states use the timers get milli functions and if statements as time thresholds to hold the prompt for one second and the delay for half a second. When returned to the prompt state, the game's current level against the current index, and if the index indicates that they are equal then we move to the reading state. The reading state waits for a capacitive touch input and compares it with the value stored in the array at that index, if it is correct we move to the correct input state, if incorrect then it moves to the incorrect input state. Logic for “timing our” or waiting too long for an input is added to the incorrect logic. The correct input state outputs the tone of the correct cap touch sensor and serves as a delay so multiple inputs would not be read with one touch. After a correct input is detected and delays occur we move back to reading to check the next indexed value so long as it is within the current level. If an incorrect input is read, we move to the incorrect input state and this state turns on all of the on board leds and returns the player to the chill state. This state machine keeps track of values such as index value “n”, the value stored at array[n], level, capacitive touch inputs, and time. A conceptual way to think about this is two separate loops, the prompt/delay loop, and the reading/correct loop. The player moves to the reading/correct loop if all indexes for the given level are prompted, and the reading/correct loop goes back to the promt/delay loop if all input for the number of indexes per level were correct.

This cycle is only broken when an incorrect input is read. The state machine with its state transition logic is featured below.

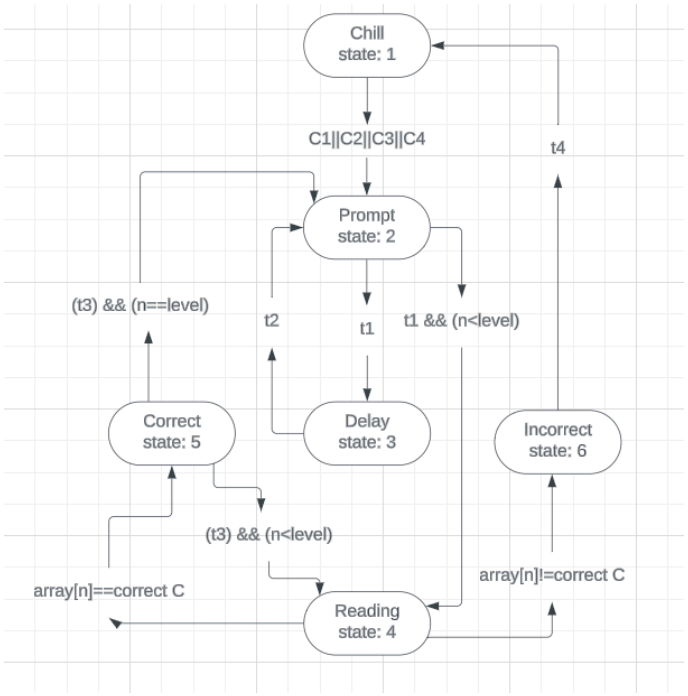

Fig. 3: State Machine Diagram.

The state machine is in its own c function, and calls the captouch() function when needed which returns unique values based on which cap touch sensor is detected, this will be explained more in depth in the interrupts, STM pins, and ISR section later.

```
      void statemachine_CAPTOUCHES(void){
          if (state == CHILL){ // first state: CHILL, where it waits for a game to
start
              if (captouch() != 0){ // if any button is pressed:
                  int i;
                  for (i = 0; i < 100; i++){ // generate full 100-value sequence
                      array[i] = rand()%4 + 1; // of random numbers from 1 to 4
                      printf("%d ", array[i]); // print them through serial terminal
for testing
                  }
                  printf("\n");
                  timeprev = TIMERS_GetMilliSeconds(); // reset prev time

                  state = PROMPT; // move to state PROMPT
              }
          } else if (state == PROMPT){

              timecurrent = TIMERS_GetMilliSeconds(); // get time
```

```
            // printf("%d\n", timecurrent);
            if (timecurrent - timeprev >= 500 && n < level){ // go through the
sequence up to the current
                timeprev = TIMERS_GetMilliSeconds();            // level and then
display on PWM/leds
                printf("Target: %d\n", array[n]); // makes testing easier
                if (array[n] == 1){ // this bit turns on the leds in order to show
the prompt
                    PWM_SetDutyCycle(PWM_1, 100);
                } else if (array[n] == 2){
                    PWM_SetDutyCycle(PWM_2, 100);
                } else if (array[n] == 3){
                    PWM_SetDutyCycle(PWM_3, 100);
                } else {
                    PWM_SetDutyCycle(PWM_4, 100);
                }
                set_leds(pow(2, (array[n]-1))); // Set STM LEDS according to the
current n
                PWM_SetDutyCycle(PWM_0, 20); // this makes the speaker blare a tone
according to the prompt
                PWM_SetFrequency(300*array[n]);
                n++;
                state = DELAY; // go to DELAY
            } else if (timecurrent - timeprev >= 500){ // once the sequence has
finished
                set_leds(0); // turn off the STM leds
                PWM_SetDutyCycle(PWM_0, 0); // turn off the tone
                PWM_SetDutyCycle(PWM_1, 0); // turn off the big leds
                PWM_SetDutyCycle(PWM_2, 0);
                PWM_SetDutyCycle(PWM_3, 0);
                PWM_SetDutyCycle(PWM_4, 0);

                n = 0; // reset n to zero for reading purposes
                timeprev = TIMERS_GetMilliSeconds(); // reset time
                state = READING; // go to the reading state
            }
        } else if (state == DELAY){ // this state implements a brief delay between
tones for prompt-reading clarity
            timecurrent = TIMERS_GetMilliSeconds();
            if (timecurrent - timeprev >= 1000){
                PWM_SetDutyCycle(PWM_0, 0); // turn everything off so that the
tones are separated
                PWM_SetDutyCycle(PWM_1, 0);
                PWM_SetDutyCycle(PWM_2, 0);
                PWM_SetDutyCycle(PWM_3, 0);
                PWM_SetDutyCycle(PWM_4, 0);
                set_leds(0);
```

```
                state = PROMPT;
            }
        } else if (state == READING){ // in this state, the state machine looks for
the correct touch


            timecurrent = TIMERS_GetMilliSeconds();

            if (captouch() == array[n]){ // if the current n == the button press
                printf("Pressed: %d\n", captouch());
                printf("CORRECT\n");
                timeprev = TIMERS_GetMilliSeconds(); // reset time
                n++; // increment n and go to correct state
                state = CORRECT;
            } else if ((captouch() != 0 && captouch() !=array[n]) ||
timecurrent-timeprev >= 2000){
                // if the wrong button is pressed OR the game times out
                printf("Pressed: %d | True %d | Index %d\n", captouch(), array[n],
n);
                printf("INCORRECT\n");
                timeprev = TIMERS_GetMilliSeconds(); // reset time
                state = INCORRECT; // go to the incorrect state
            }
        } else if (state == CORRECT){ // state that shows the leds when you get it
right
            set_leds(15); // light up half the STM leds to distinguish from
incorrect state
            //PWM_Start(PWM_0);
            PWM_SetDutyCycle(PWM_0, 20);
            PWM_SetFrequency(300*array[n-1]); // tone associated with the button
you pressed

            if (array[n - 1] == 1){ // led associated with the button you pressed
                    PWM_SetDutyCycle(PWM_1, 100);
                } else if (array[n - 1] == 2){
                    PWM_SetDutyCycle(PWM_2, 100);
                } else if (array[n - 1] == 3){
                    PWM_SetDutyCycle(PWM_3, 100);
                } else {
                    PWM_SetDutyCycle(PWM_4, 100);
            }

            timecurrent = TIMERS_GetMilliSeconds();
            if (timecurrent-timeprev >= 500 && captouch() == 0 && n < level){ //
only go here when the button is no longer pressed
                timeprev = TIMERS_GetMilliSeconds();
                PWM_SetDutyCycle(PWM_0, 0);  // turn everything off
```

```
                    PWM_SetDutyCycle(PWM_1, 0);
                    PWM_SetDutyCycle(PWM_2, 0);
                    PWM_SetDutyCycle(PWM_3, 0);
                    PWM_SetDutyCycle(PWM_4, 0);
                    set_leds(0);
                    state = READING; // assuming you haven't beat the level yet, go
back to READING
                } else if (timecurrent-timeprev >= 500 && captouch() == 0){ // if you
HAVE beat the level
                    set_leds(0);
                    PWM_SetDutyCycle(PWM_0, 0); // turn everything off
                    PWM_SetDutyCycle(PWM_1, 0);
                    PWM_SetDutyCycle(PWM_2, 0);
                    PWM_SetDutyCycle(PWM_3, 0);
                    PWM_SetDutyCycle(PWM_4, 0);
                    timeprev = TIMERS_GetMilliSeconds(); // reset time
                    level++; // increment level
                    if (level > highscore){ // check for new highscore
                        highscore = level;
                    }
                    n = 0; // reset n
                    printf("LEVEL %d\n", level);
                    state = PROMPT; // go to prompt to continue
                }
            } else if (state == INCORRECT){ // incorrect state for when you get it
wrong
                set_leds(255);
                timecurrent = TIMERS_GetMilliSeconds();
                if (timecurrent - timeprev >= 1500){


                    level = 1; // reset level back to 1
                    n = 0; // reset n
                    state = CHILL; // go back to the chill state where you can start a
new game
                }
            }
        }
```

Fig. 4: C code for State Machine.

3.1.3: User interface

Sammy Says relies on a leds prompting the different capacitive touch sensors, and a tone from the TPA2005 amplifier/speaker circuit which is unique depending on the prompted capacitive touch sensor. Both the leds and tone of the circuit are dependent on the value stored in array[n], the tone uses the tone generation set frequency function with frequency being

proportional to array[n]. The current level and high score are printed to the OLED screen using the OLED library OLED draw string and OLED update functions. The delay state of the state machine was also added for a better user interface, as it gives the game a period in between prompts where the lit led and the prompted tone turn off. Multiple iterations of the game with different prompt times and delays were used to narrow down on times which suited the player preference. With all of these details, the player can experience a clear, yet challenging game experience.

3.2: LM555 Relaxation Oscillators and Captouch Circuits

3.2.1: Overview

In order to measure capacitance changes from the capacitive touch sensors, four relaxation oscillators were constructed using LM555 timer chips. This is a fairly simple circuit, and is, in fact, a modified version of the circuit which the Texas Instruments LM555 datasheet offers as an example of the timer in astable mode.

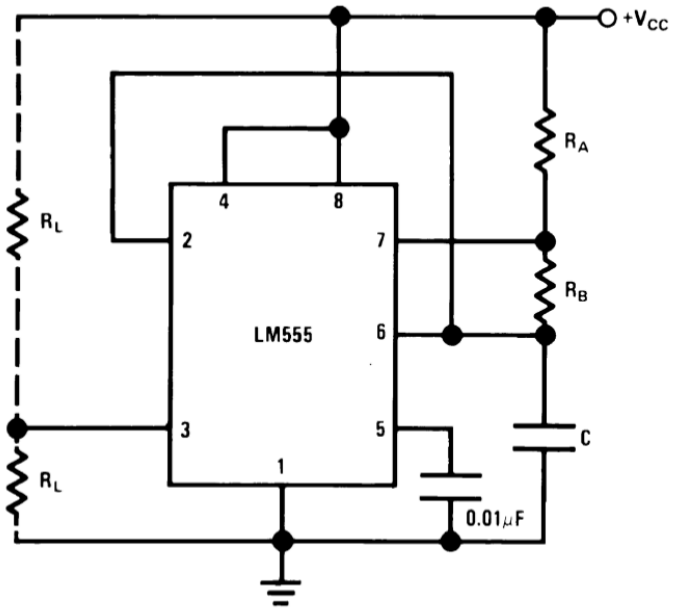

Fig. 5: LM555 timer circuit for astable operation.

In the circuit above, the capacitor *C* is replaced with a 22 pF capacitor and a capacitive touch sensor, in parallel. This allows the period of the square wave, output from pin 3 of the LM555, to change according to whether or not the capacitive touch sensor is being touched. From there, the output from pin 3 was fed into an input pin of the STM32 microcontroller, where an interrupt triggered on every rising edge was used to calculate the period.

3.2.2: Circuit Design

According to the datasheet, R_A and R_B can be chosen in order to deliberately set the period, frequency, and duty cycle of the square wave output by this timer. When choosing resistors, these considerations were taken carefully into account.

During Lab 2, which also used this circuit, the recommendation was to choose R_A and R_B such that the frequency of the output wave was between 1 and 5 kHz. However, given the (order of magnitude faster) speed of the STM's clock, it can reasonably be expected to edge-detect much higher frequencies. The equations that determine the frequency and duty cycle of the square wave are as follows:

The charge time (output high) is given by:

$$t_1 = 0.693\ (R_A + R_B)\ C$$

And the discharge time (output low) by:

$$t_2 = 0.693\ (R_B)\ C$$

Thus the total period is:

$$T = t_1 + t_2 = 0.693\ (R_A + 2R_B)\ C$$

The frequency of oscillation is:

$$f = \frac{1}{T} = \frac{1.44}{(R_A + 2R_B)\ C}$$

Fig 6: Frequency, period, and charge/discharge time equations for the LM555 astable mode circuit.

The duty cycle is:

$$D = \frac{R_B}{R_A + 2R_B}$$

Fig. 7: Duty cycle equation for LM555 astable mode circuit.

Recall that C is 22pF and the capacitive touch sensor in parallel. The capacitance of the capacitive touch sensor can be easily measured by constructing a simple RC low-pass filter with a known resistance *R* and inputting a square wave from the signal generator. Next, the oscilloscope can be used to determine how long it takes the capacitor to charge to 63.2% of its maximum voltage. This is the time constant tau, or RC, which can be divided by the known R to determine the capacitance of the sensor. Using a 680k ohm resistor, this method gave an untouched capacitance of about 25pF.

The capacitance C is then 22pF + 25pF when the circuit is untouched. This value can be substituted into the final equation from Fig. ?? in order to choose R_A and R_B and determine the frequency. The resistor values must be quite high for a frequency as low as 1kHz; $R_A = 3M\Omega$ and $R_B = 1M\Omega$ provide a frequency a little over 5 kHz.

Since the 5 kHz values worked decently well in Lab 2, the initial plan was to use them here as well. However, additional considerations forced us to reconsider. Firstly, we judged that a

higher frequency could make the sensors more responsive to touch when used with the interrupts, and secondly, the lab was out of 1MΩ resistors. Thirdly, chaining many resistors in series to get higher resistances would take up space inside the game's case.

Instead, for three of the circuits, R_B was set to 1MΩ and R_A to 1.36 MΩ, using one of the remaining 1MΩ resistors for R_B and two 680kΩ resistors in series for R_A. This results in an expected untouched frequency of roughly 9.119 kHz. Due to resistor availability, one of the four captouches instead uses a 680kΩ resistor and a 1MΩ resistor in series for R_A. The expected frequency of this circuit is 8.325 kHz.

However, the actual frequencies and capacitances cannot be estimated using these methods, because assembling the entire toy, including its case, changes the capacitance. This is due to a variety of factors, likely including wire length and the adhesive used on the back of the capacitive touch sensors to adhere them to the case. The amount to which our initial hypotheses for frequency and capacitance were incorrect will be discussed in the 'evaluation' section of this report. In the meantime, it is possible to make up for those discrepancies in code.

3.2.3: Capacitive Touch Sensor Calibration

In order to make up for deficiencies in our model of the capacitive touch sensors, we assembled the circuits described above, connected them to the microcontroller, and measured the period of the square wave using the microcontroller interrupts. This required the interrupts to be set up successfully, which will be discussed in the next section. The period (time between square wave rising edges) was printed over the serial terminal.

We repeated this procedure twice: once on a breadboard, separate from the case, and again with the project fully assembled. This is because we calibrated them successfully away from the case, and then realized changing the wiring and adhering the touch sensors to the case would probably change the capacitances. It did, but not by much; the cutoff needed to be changed from 'over 150 microseconds' for the circuits with two 680kΩ resistors in R_A and 'over 800 microseconds' for the one that included a 1MΩ resistor to 'over 120 microseconds' and 'over 500 microseconds,' respectively.

3.3: Interrupts, STM Pins, and ISRs

3.3.1 Overview

Initially, the plan for this project was to use a single interrupt and multiplex the LM555 outputs. In this implementation, the multiplexor selector would cycle through [00, 01, 10, 11], sampling from each of the circuits sequentially in order to determine which, if any, were being touched. This might have resulted in missed touches, but since the periods of each capacitive touch sensor circuit are only microseconds long, it could be done reasonably quickly.

However, that plan was designed for a scenario in which pins and external interrupts were limited. In reality, the STM32 has a total of sixteen external interrupt lines and (theoretically) 81 input/output ports, rendering the multiplexing unnecessary.

3.3.2: Pins and STM libraries

Although the documentation says that the STM32 comes with 81 I/O ports, obviously not all of them are feasible to use as capacitive touch sensing inputs. Some of them are used by other libraries which turned out to be important to features that this project needed. Others do not appear to be accessible from the I/O shield.

There are three groups of general purpose input/output (GPIO) pins on the STM32: A, B, and C. Initially, we wanted to use only pins from one of these groups for ease of initialization. We settled on GPIOB, since pins PB4 and PB5 were used by the QEI.h library and therefore easy to find out how to initialize. However, most other GPIOB pins accessible from the I/O shield are already used by other libraries. We instead used pins from both GPIOB and GPIOA: PA6, PA7, PB4, and PB5, all of which are usable as inputs and capable of being used to generate interrupts.

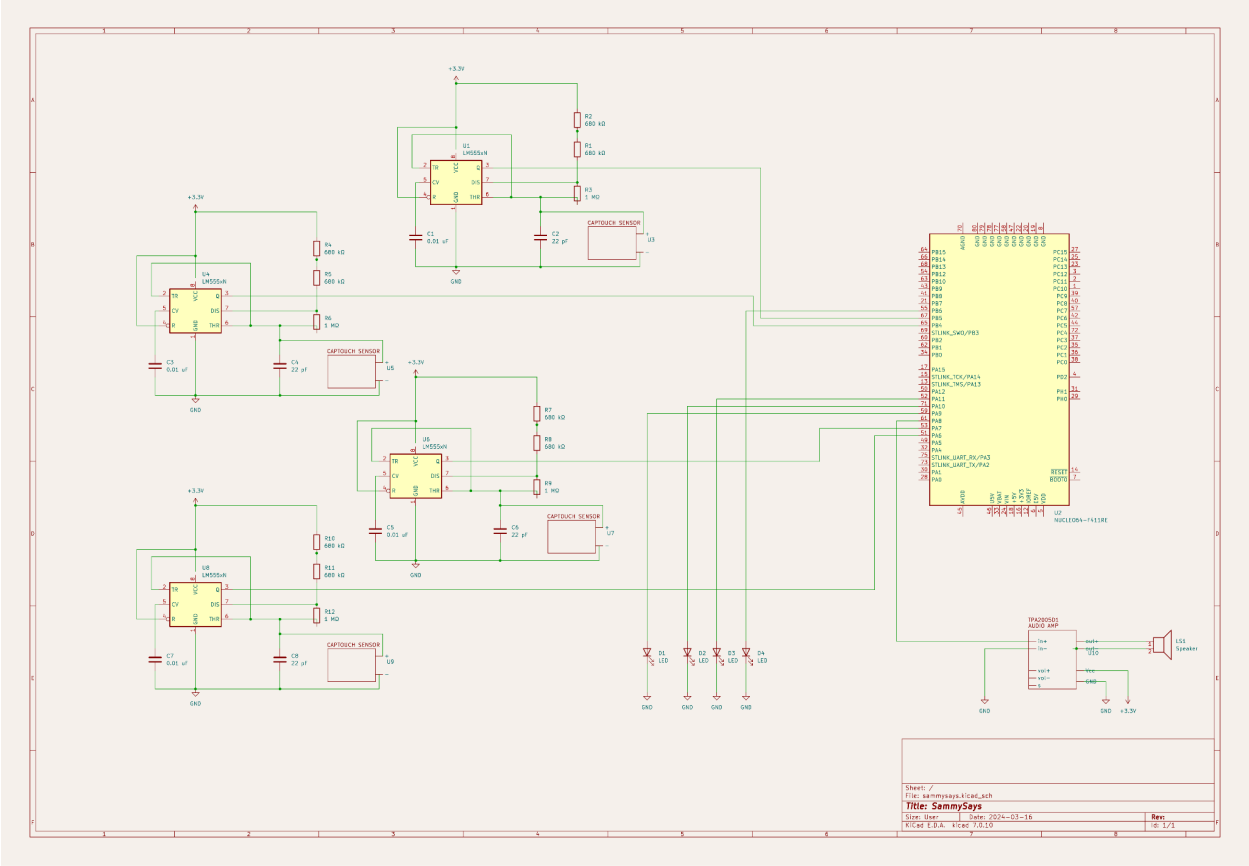

Fig. 8: Full circuit diagram, including STM pin usage. This diagram is included in better quality as an appendix.

As for other pins, the PWM pins and PWM library were used for both the LEDs and the speaker, as they required minimal additional setup in order to be used to light the LEDs. As such,

they were quickly dismissed as possibilities for sensor circuit inputs. PWM_0 is the input to the audio amplifier which supplies the speaker with a signal, and PWM 1 through 4 powers the four LEDs.

3.3.3: STM32CubeIDE and Interrupts

In order to set up an interrupt triggering based on the input from the pins, the following steps were followed in code.

On initialization:

1. Create a GPIO initialization struct for each GPIO group you plan to use. Fill it with the pins, or'ed together, and configure mode (to decide when the interrupt should be triggered, in our case on rising edges) and pull.
2. Set the interrupt priorities and enable the interrupts for your chosen external interrupt lines.

Within the ISR:

1. Check which pin caused the interrupt.
2. Clear the interrupt flag.
3. Use the timers library to determine the time since the previous interrupt (period of the input wave).
4. Save the period to the array holding 300 samples from which to calculate the moving average.
5. Exit the ISR.

The main challenge in all of this was to determine which ISRs corresponded to which pins. This is not listed clearly anywhere in the available documentation that we could see. The solution was to use the STM32's dedicated IDE, STM32CubeIDE, in order to find out which external interrupt lines are associated with which pins. This revealed that PB4 is connected to EXTI4, and PB5, PA6, and PA7 are connected to EXTI9_5. This made defining the ISRs relatively very easy.

The initialization steps look like this.

```
GPIO_InitTypeDef GPIO_InitStruct2 = {0};
GPIO_InitStruct2.Pin = GPIO_PIN_6|GPIO_PIN_7;
GPIO_InitStruct2.Mode = GPIO_MODE_IT_RISING;
GPIO_InitStruct2.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct2);
// EXTI interrupts init
HAL_NVIC_SetPriority(EXTI4_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(EXTI4_IRQn);
```

Fig. 9: C code to initialize input pins PA6 and PA7 and EXTI interrupt 4.

The next steps are as follows. EXTI4 is used as an example for the sake of being concise, because the other ISR manages two additional pins.

```
// external interrupt ISR for rising edge of pin PB4
void EXTI4_IRQHandler(void){
```

```
  // EXTI Line interrupt detected
  if(__HAL_GPIO_EXTI_GET_IT(GPIO_PIN_4) != RESET){
      // clear interrupt flag
      __HAL_GPIO_EXTI_CLEAR_FLAG(GPIO_PIN_4);

      // update current/prev times in us
      prev1 = current1;
      current1 = TIMERS_GetMicroSeconds();
      // subtract to get the period
      period1 = current1 - prev1;

      // fill this captouch's period readings array
      periodReadings1[num1] = period1;
      // update array index/roll over if there are already 300 readings
      num1++;
      if (num1 >= SAMPLESIZE){
          num1 = 0;
      }

  }
}
```

Fig. 10: C code for an example ISR..

This procedure can be repeated as needed. It is possible to utilize several more pins, although that was beyond the scope of this project.

3.4: Case and Assembly

Detecting individual capacitive touches forced us to create four separate LM555 circuits, one for each sensor. With these circuits and the TPA2005 Speaker Circuit, the obvious approach was to create a 3d printed enclosure with a window to see the STM32 OLED, a hole for the speaker, mounting areas for the four sensors, and their corresponding leds. The enclosure was modeled using autodesk fusion 360 and printed on a ender 3 3d printer. The board is held in with nylon nuts and bolts through its mounting holes and the speaker, cap touches, and leds are secured using hot glue. The case hides all circuits giving the game a more professional appearance.

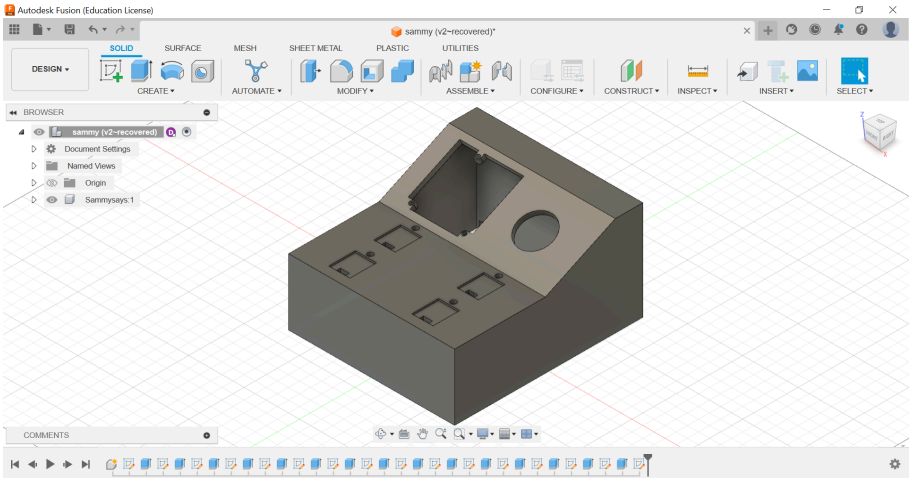

Fig. 11: Game enclosure.

Part 4: Evaluation

In order to quantitatively judge how well this project met the goals outlined in the proposal, this report will first touch on how well the frequencies of the square waves measured by the STM32 conformed to our expectations and the calculations in 3.2.2. Next, we also measured how many times the STM32 failed to register taps and registered taps when there were actually none.

4.1: Circuit Output

In order to determine the accuracy of our calculations with regard to the LM555 relaxation oscillator with capacitive touch sensor circuits, the following experiment was carried out. First, the entire game was assembled, with all the circuits inside the case. Next, the oscilloscope probes were connected to ground and the output of each LM555 circuit in turn, and used to measure the square wave and its frequency. The idea was to capture any changes in frequency induced by the case or the additional length in the wires. Next, the period of each square wave was captured using the STM32 interrupt setup and printed to the serial terminal, which should also allow us to observe any error introduced by the way the STM32 read the signal.

The results were as follows:

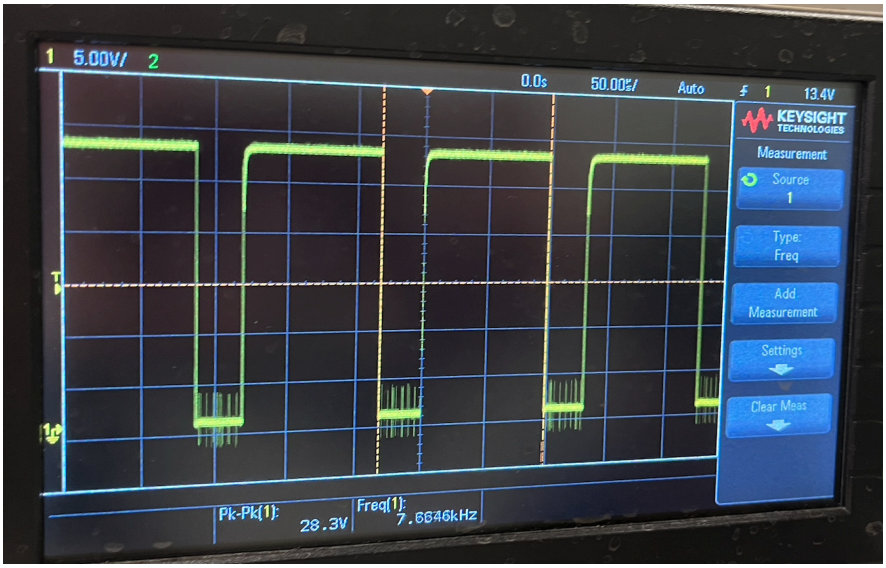

Fig. 12: Capacitive touch sensor #1, untouched square wave output.

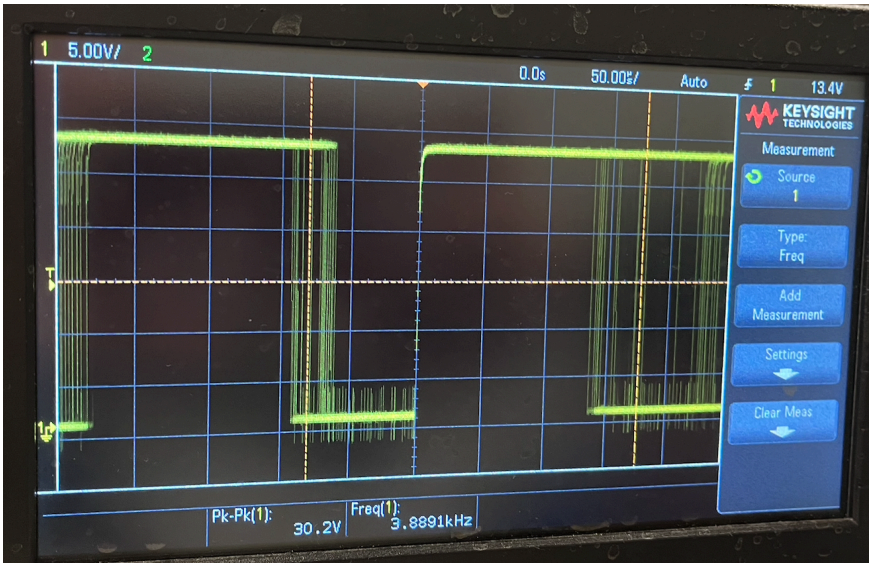

Fig. 13: Capacitive touch sensor #1, touched square wave output.

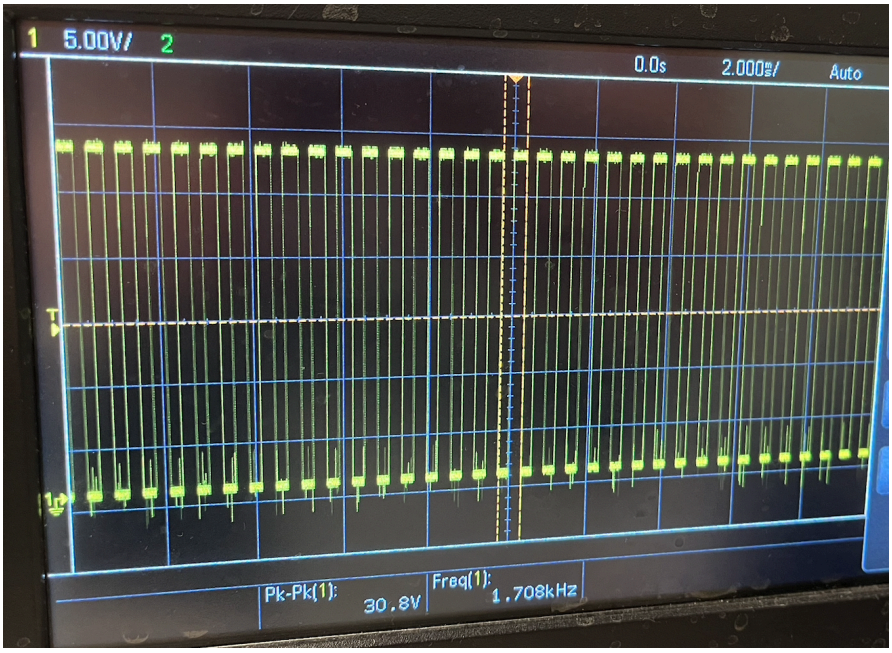

Fig. 14: Capacitive touch sensor #2, untouched square wave.

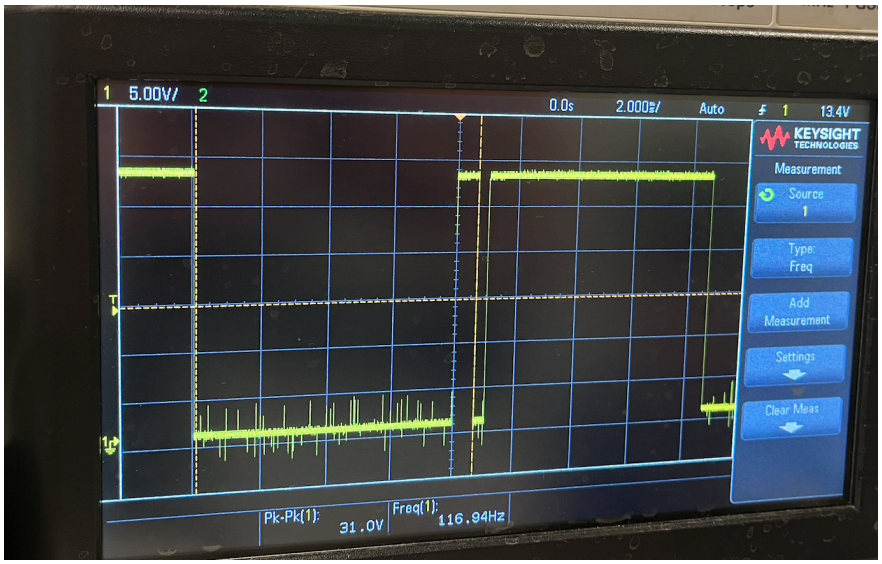

Fig. 15: Capacitive touch sensor #2, touched square wave output.

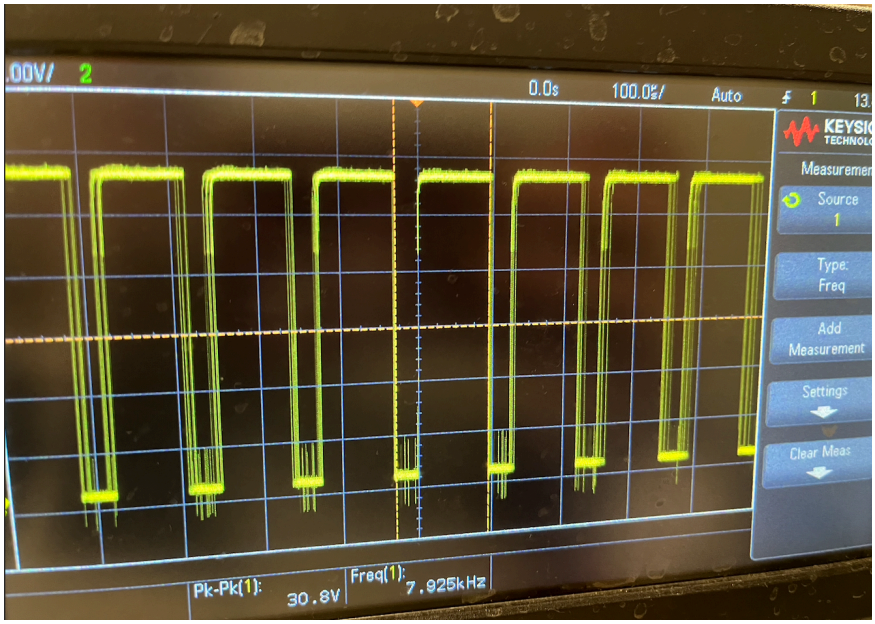

Fig. 16: Capacitive touch sensor #3, untouched square wave output.

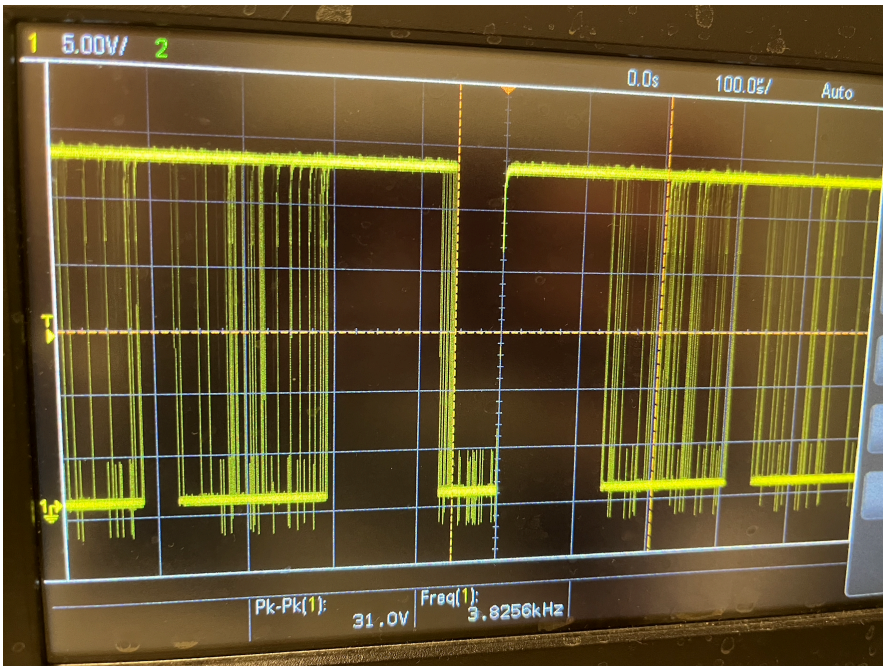

Fig. 17: Capacitive touch sensor #3, touched square wave output.

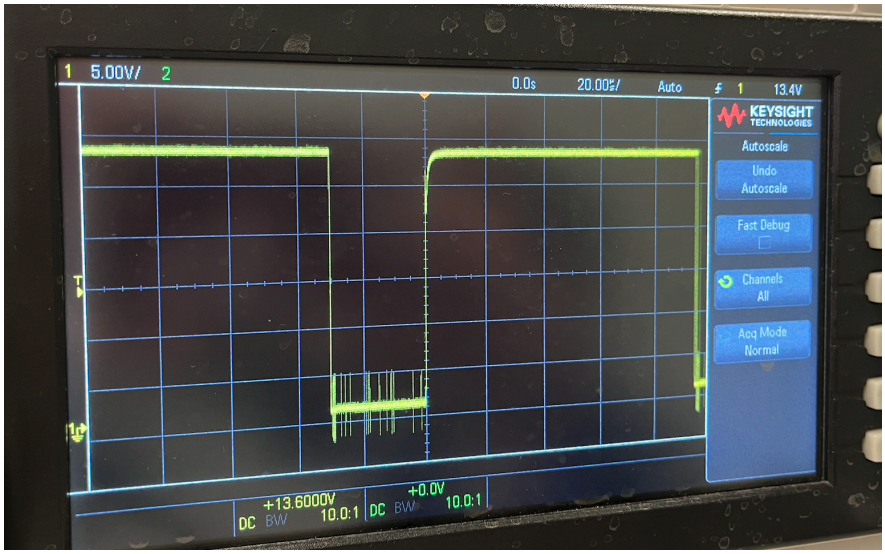

Fig. 18: Capacitive touch sensor #4, untouched square wave output.

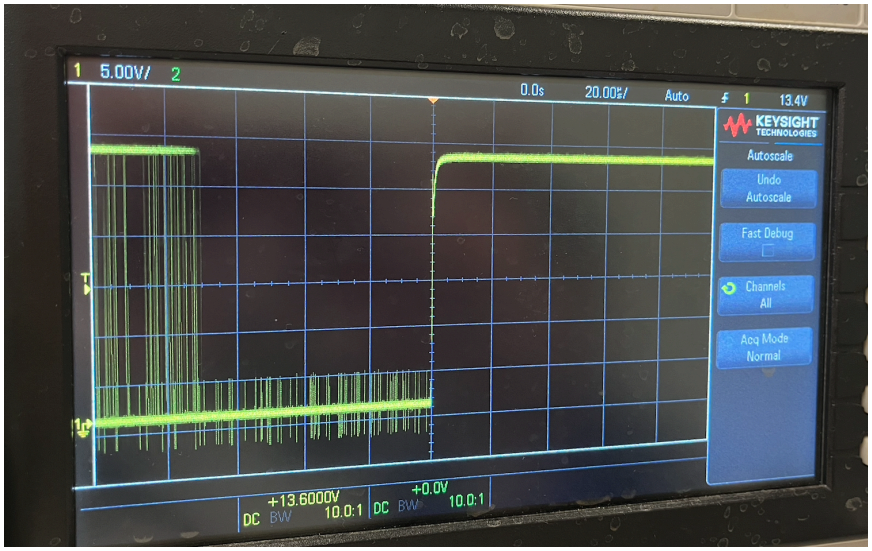

Fig. 19: Capacitive touch sensor #4, touched square wave output.

	Untouched frequency (kHz)	Touched frequency (kHz)	Estimated untouched frequency (kHz)	%error
C1	7.665	3.881	9.119	15.9447
C2	1.708	0.1169	8.325	79.4835
C3	7.925	3.825	9.119	13.0935
C4	7.9	3.5	9.119	13.3677

Table 1: Touched and untouched frequencies of each capacitive touch sensing circuit.

For all these circuits, we can see that the frequency is lower than expected, suggesting additional unmodeled capacitance. This is particularly noticeable for circuit #2, the one with the

extra 1MΩ resistor. It is very possible that we underestimated the capacitance of the touch sensors, especially since we did not measure their capacitances individually. An assumption was made that they all had roughly the same capacitance. However, since we calibrated the sensors twice, we also know that putting them in the case raised the length of the period somewhat, also lowering the frequency.

Here are the final values for the ranges of the period, after applying a moving average filter.

	Untouched period (us)	Touched period (us)	Cutoff (us)
C1	135	200	140
C2	574	1000	580
C3	129	230	130
C4	125	230	130

Table 2: Period as read by the STM32 microcontroller and chosen cutoff values, in microseconds.

These are all much as we expected from the oscilloscope readings. The microcontroller reads the frequency of each fairly accurately. Although the output as seen on the oscilloscope is very noisy, a 300-value moving average filter applied to each sensor successfully eliminated most of the noise, at the cost of being more likely to miss very short touches.

4.2: Game Touch Accuracy

In order to determine how well these chosen cutoff values actually worked from the perspective of someone playing the game, we assessed the number of missed touches (times when the player touched one of the sensors, but the game didn't register a touch) and false positives (times when the sensors were not touched, but the game registered a touch anyway). The procedure for this assessment was simply to play the game and record 100 attempted touches. The ground truth was measured by writing down a touch whenever the player felt his finger touch one of the sensors.

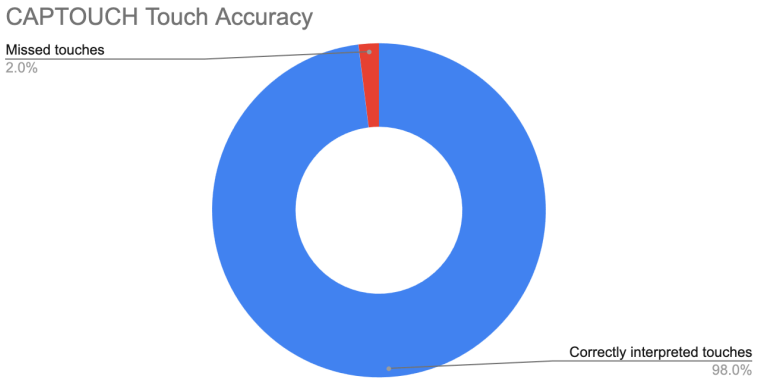

Fig. 20: Visualization of the number of missed touches to correct touches.

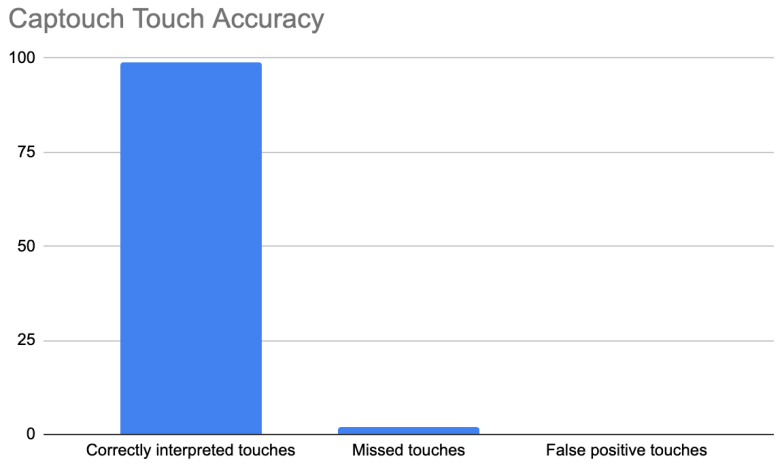

Fig. 21: Bar plot of correct touches, missed touches, and false positive touches.

Out of 101 total touches to the capacitive touch sensor, over the span of fourteen levels, 98 were interpreted correctly and 2 were missed. We observed zero false positive touches to the capacitive touch sensor during the duration of this experiment.

This indicates that our filtering and chosen cutoffs work very well. Choosing the cutoffs is a choice between risking more missed touches or risking more false positive touches; the fact that we have erred on the side of more missed touches may be seen as a good thing, because false positive touches are more likely to erroneously make the player lose. Our choice of sample size for the moving average filter may also have contributed to the presence of missed touches.

Part 5: Discussion and Conclusion

5.1: Previous versions

This project was largely unchanged from its first conception, as outlined in the project proposal. The greatest change was that we abandoned the idea of multiplexing the inputs to the STM32. This presented some challenges, as the method for setting up an ISR was difficult to find in the STM's documentation.

At some point, we also conceived of a version where the time for the player to answer got shorter and shorter as the games progressed, but this was abandoned in the state machine testing stage. Rather than adding enjoyment to the game, it only added confusion, and trying to find a way to communicate how long the player had left to answer made the UI needlessly complicated.

5.2: Shortcomings

As seen above, the game does miss touches very occasionally. The best way to fix this and to further improve the accuracy of the sensors would probably be to adjust the size of the moving average filter. Alternatively, we could lower the cutoffs, although this would increase the possibility of false positive touches. A balance between these two methods and significant time spent experimenting would work best.

At the beginning, there was also a plan to expand the game using more capacitive touch sensors. This would not be overly difficult, particularly if five additional sensors were added. The best STM32 pins for these would be PC12, PC5, PC4, PD2, and PB14, due to their input capabilities. The interrupt handling, sensing circuits, and calibration would all be handled as in the earlier sections of this report. Adding additional sensors would be more difficult, as this would begin to overwrite pins used by other STM libraries, but certainly not impossible. However, at some point, the question becomes not whether additional sensors can be added, but whether or not they contribute any enjoyment to the game.

Finally, since all of the circuits in this project were implemented on breadboards, the durability could be greatly improved by soldering them properly and creating a PCB.

5.3: Conclusion

Overall, we successfully created a game with four capacitive touch sensors, as outlined in the project proposal. It responds to touch with 98% accuracy, switches states as intended, and is contained by a 3D printed case. Although there are still some shortcomings, and certainly things which might improve this project, it can be considered a success.