Fruit Smash

CISC-340: Project #1 Report

TEAM 14
Tiffany Chan - 10181522 - tiffany.chan@queensu.ca
Garett MacGowan - 10197107 - 15gm1@queensu.ca
Michael Alarcon - 10172841 - 14ma61@queensu.ca
Quentin Petraroia - 10145835 - 13qp2@queensu.ca
Ryan Rossiter - 10177467 - 14rcr3@queensu.ca
Tyler Gawalewicz - 10135796 - 13tg19@queensu.ca

Table of Contents

Introduction

Fruit smash is a reaction based game, inspired by games such as Whack a Mole but with fruits instead. The project is comprised of a Raspberry Pi B+ and has three main components of a capacitive touch sensor, LCD display, and an ethernet based communication. The game will prompt the user one of the three fruits to touch through audio and the LCD display, within a given amount of time, and if the player is successful, the game will have an audio output of “Great job”. If the player fails to touch the correct fruit or runs out of time, the game will have an audio ouput of “You suck”, and the game is over. It will then follow with the points the player has gained, and will be saved accordingly to the HighScores list on the server. The game will then return to the beginning, and prompt a player to “touch to begin”. It is written in Python, and uses supportive software such as MPR121 Python Library, Python Package pyttsx, Python Package Flask, and Adafruit character LCD plate.

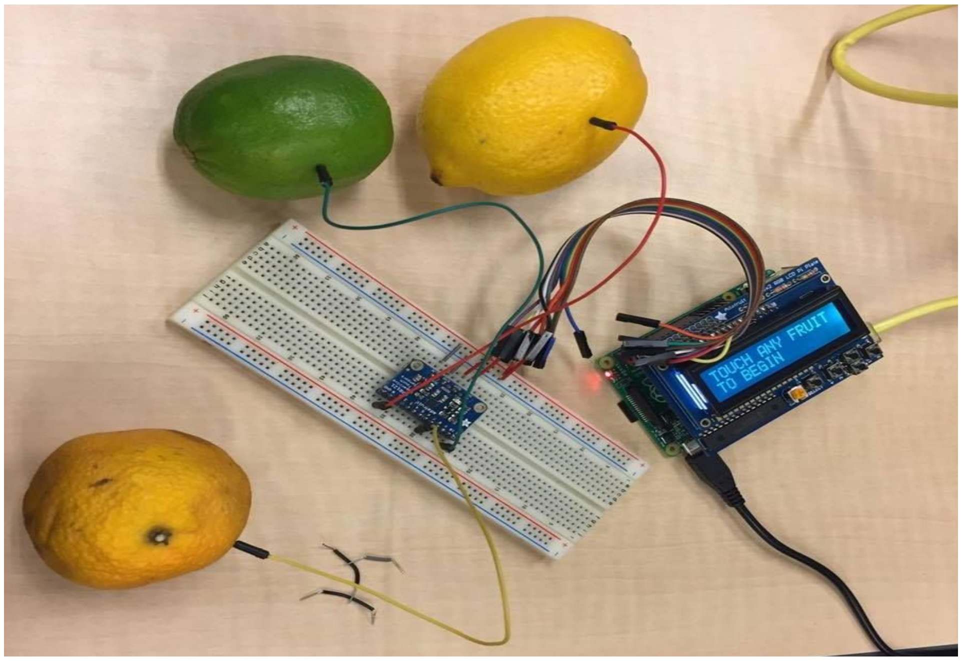

Figure 1. Fruit Smash Completed Project

Project Devices

Sensor: 12-Key Capacitive Touch Sensor

A capacitive touch sensor has many real-world applications in the devices people use almost everyday. It is a preferred option for many designs as it offers good stability, speed, resolution, and low power. It is also simple to integrate onto circuit boards, and has a wide range of applications. For example, touch-relative tablets, smartphones such as iphones, and control panels for appliances can have a capacitive touch sensor. With such a sensor in these devices, it can deliver a high performance but low-cost device to users. A capacitive touch sensor can also be applied in safety critical applications such as crash sensors in automobiles and thus, the capacitive sensor can be applied in a wide range of designs.

With the 12-key capacitive touch sensor, it connects with the Raspberry Pi B+ through wires connected from the breadboard to the I2C bus, and the sensor itself sits on the breadboard and also has wires connecting to a conductive electrodes. It uses a mechanism called “mutual-sensing” to convert non-electrical info to electrical info. This is when sensing capacitance is formed between the sensor pad and the excitation pad. It measures the changes in space between two conductive objects, and how it responds to an electrical difference between them. Therefore, when an object such as a finger or hand comes closer, its’ capacitance decreases. Capacitance is an effective associate with sensors as it is more effortless compared to its’ resistive counterpart in which it recognizes the pressure of a finger to change the flow of electricity. However, a disadvantage in using capacitance is that it can only be used with conductors such a human finger or stylus.

The sensor device does not have a particular interface built onto the device, but instead uses a standard protocol of an I2C bus. This involves masters and slaves, in which the master is a device that drives the SCL clock line (Host computer) and slaves are a device that responds to the master (Capacitive Sensor). The master will then initiate a transfer over an I2C bus to the slave to perform a task. (Please refer to figure 2) Furthermore, the sensor requires the Python MPR121 library in order to use it. The user must install the software and all its’ dependencies through terminal for SSH, and have the Raspberry Pi connected to the internet either with a wired or wireless connection. Within the MPR121 library, there are many useful and available functions to be called. For example, the begin() function is used to initialize the capacitive touch sensor. The touched() function is used to determine if the conductive object is touched or not by returning a 12-bit value. Each bit represents one of the 12 inputs on the sensor where bit 0 represents input 0, bit 1 represents input 1, and so forth. If the bit is 0, the input is false, and therefore not being touched. If the bit is 1, input is true, and therefore it is being touched. There is also a is_touched() function, which is similar and simpler than touched() as it only checks if one input is being touched or not, and returns only true and false. In addition, there are two useful functions available for debugging, which are filtered_data() and baseline_data(). These functions enable the user to look up filtered data or baseline data register values for an input.

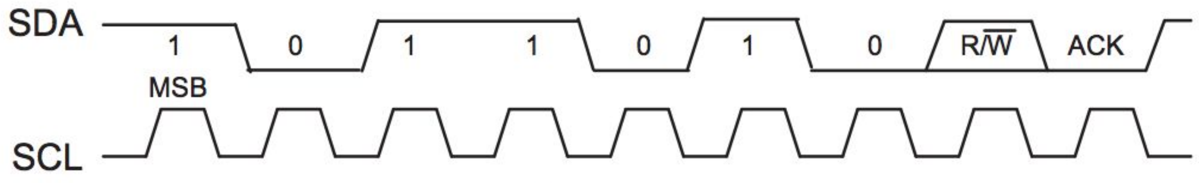

Figure 2. Waveform Diagram of I2C Protocol Data Transfer

The sensor device is powered by the Raspberry Pi B+'s 3.3 V to MPR121 VIN. This device did not have many special requirements. The only parts we needed were a breadboard and jumper wires. We placed the sensor on the board and connected the jumper wires to the sensor and the Raspberry Pi B+.

Team's code for the sensor is as follows:
This is file touch.py. It contains helper functions that we use in our actual game code.

```
import sys
import Adafruit_MPR121.MPR121 as MPR121

print('Initializing touch sensor...')

# Create MPR121 instance.
cap = MPR121.MPR121()

# Initialize communication with MPR121 using default I2C bus of
device, and
# default I2C address (0x5A).  On BeagleBone Black will default to
I2C bus 0.
if not cap.begin():
    print('Error initializing MPR121.  Check your wiring!')
    sys.exit(1)

def waitForTouch():
    touched = cap.touched()
    while touched == 0:
        touched = cap.touched()

    for i in range(12):
        pin_bit = 1 << i
        if touched & pin_bit:
            return touched

    return -1
```

Next we have touch_test.py. This is the test script provided by adafruit. We found it on adafruit's github.

```
# Copyright (c) 2014 Adafruit Industries
# Author: Tony DiCola
#
# Permission is hereby granted, free of charge, to any person import
sys
import time

import Adafruit_MPR121.MPR121 as MPR121
print('Adafruit MPR121 Capacitive Touch Sensor Test')

# Create MPR121 instance.
cap = MPR121.MPR121()

# Initialize communication with MPR121 using default I2C bus of
device, and
# default I2C address (0x5A).  On BeagleBone Black will default to
I2C bus 0.
if not cap.begin():
    print('Error initializing MPR121.  Check your wiring!')
    sys.exit(1)

# Alternatively, specify a custom I2C address such as 0x5B (ADDR tied
to 3.3V),
# 0x5C (ADDR tied to SDA), or 0x5D (ADDR tied to SCL).
#cap.begin(address=0x5B)

# Also you can specify an optional I2C bus with the bus keyword
parameter.
#cap.begin(busnum=1)
```

```
# Main loop to print a message every time a pin is touched.
print('Press Ctrl-C to quit.')
last_touched = cap.touched()
while True:
    current_touched = cap.touched()
    # Check each pin's last and current state to see if it was 
pressed or released.
    for i in range(12):
        # Each pin is represented by a bit in the touched value.  A 
value of 1
        # means the pin is being touched, and 0 means it is not being 
touched.
        pin_bit = 1 << i
        # First check if transitioned from not touched to touched.
        if current_touched & pin_bit and not last_touched & pin_bit:
            print('{0} touched!'.format(i))
        # Next check if transitioned from touched to not touched.
        if not current_touched & pin_bit and last_touched & pin_bit:
            print('{0} released!'.format(i))
    # Update last state and wait a short period before repeating.
    last_touched = current_touched
    time.sleep(0.1)

    # Alternatively, if you only care about checking one or a few 
pins you can
    # call the is_touched method with a pin number to directly check 
that pin.
    # This will be a little slower than the above code for checking a 
lot of pins.
    #if cap.is_touched(0):
    #    print('Pin 0 is being touched!')

    # If you're curious or want to see debug info for each pin, 
uncomment the
    # following lines:
    #print '\t\t\t\t\t\t\t\t\t\t\t\t\t 
0x{0:0X}'.format(cap.touched())
    #filtered = [cap.filtered_data(i) for i in range(12)]
    #print('Filt:', '\t'.join(map(str, filtered)))
    #base = [cap.baseline_data(i) for i in range(12)]
    #print('Base:', '\t'.join(map(str, base)))
```

```
def touchedFruit():
    last_touched = cap.touched()
    while True:
        current_touched = cap.touched()
        # Check each pin's last and current state to see if it was
pressed or released.
        for i in range(12):
            # Each pin is represented by a bit in the touched value.
A value of 1
            # means the pin is being touched, and 0 means it is not
being touched.
            pin_bit = 1 << i
            # First check if transitioned from not touched to
touched.
            if current_touched & pin_bit and not last_touched &
pin_bit:
                print('{0} touched!'.format(i))
                return '{0} touched!'.format(i)
```

Overall as a team, we did not have to many problems with the 12-key capacitive touch sensor. However, we did have to change some details regarding our project because of some miscommunication between the parts we were expecting and the parts we got. Originally we thought we were getting the Capacitive Touch HAT for Raspberry Pi, but instead we got MPR121 Capacitive Touch Sensor. When we realised we did not have the correct part we had to improvise quick. As a team we looked at the documentation for the MPR121 Capacitive Touch Sensor and realised we would need a breadboard and jumper cables. Once we had these we could set it up to work exactly like we had originally planned. Luckily one member in our group had a toolbox of extra parts so we didn't lose any time between figuring out we had the wrong part and deciding on what to do. It took our group a while to set up the touch sensor as not many of us had experience with a breadboard. However, once we had it set up, the programming of the sensors were easy. Overall, we were happy that we ended up getting a different part than what we originally wanted. It added an extra challenge and made us use our knowledge of what we learned in class and taught us that whenever we are stuck we could find a tutorial online.

Display: Standard LCD 16x2

Another part being used is an RGB LCD Display which can display two lines of 16 characters. It connects to our Raspberry Pi B+ (which acts as our microcontroller) through the GPIO pins, using 8 pins in total: 5 to control the LCD and 3 for the RGB backlight. We use the LCD display to show information regarding the game, such as points and instructions on which fruit to click. The data being read by the LCD goes through the SDA and SCL pins that are a part of the 8 GPIO pins being used. The LCD itself is actually mounted onto a pi plate which included the 5 buttons as well as an input/output port expander. Had we built the plate from scratch, within the kit from Adafruit is the MCP23017 I2C 16 input/output port expander. Through this expander, we are able to connect more devices, such as the 12-key capacitive touch sensor.

Similarly to the 12-key capacitive touch sensor, an LCD display has a wide variety of uses. You can find LCDs in cars, television, monitors, cameras and pretty much any device that has a screen. LCD (Liquid Crystal Display) is one of the many different technologies when it comes to the screen. In cars, LCD displays often display your gas mileage, radio stations, time, etc. LCD's are used in various mobile devices, TVs and laptops to display wan output to the user. A display (not necessarily an LCD), has made the computer to human interactions a lot easier and efficient.

In our project, the LCD display is actually connected to a Raspberry Pi plate. The display uses liquid crystal modules and takes advantage of its light-modulating properties. While the liquid crystals don't emit light themselves, they are the medium that helps the light to bend or rotate within the display. These crystals are found between two transparent electrodes and two polarizing filters. When an electric current passes through them, it causes the crystal molecules to change and results in changing the angle in which the light passes through them. Due to this change in angle, the light hits the second polarizing filter at an angle rather than straight. This allows for some light to be seen, but can be fairly dark. To make up for this, a reflective mirror is set up in the back. Without this changing of angles, the light would pass through one polarizing filter, and then the next resulting in entirely blacked out screen. Each pixel has a certain amount of red, green or blue sub-pixels. The pixel colour shown will depend on how much voltage is being passed through each pixel. Although there is no special current required for the device to work, it does require proper voltage handling when emitting the different coloured light. The amount of volts applied to each pixel

In order for the LCD to run on the raspberry pi, we first need to set up the raspberry pi for the LCD. We found these library when we looked up the part on Adafruit and opened up its datasheet. In the datasheet, Adafruit provided a github repository link where we found the library. The libraries didn't isn't a part of the code, but it was necessary for us to use the LCD. While connected to the pi (usually by SSH), we put the following lines into the terminal to set up the raspberry pi for the LCD:

1. sudo apt-get install python-smbus
2. sudo apt-get install i2c-tools

For us to actually use the LCD and have it display what we wanted, we used a special Python library called Adafruit CharLCD. This library allowed us to use built in functions to communicate with our LCD display. First, we needed to create a lcd object, which we called by initializing it to LCD.Adafruit_CharLCDPlate(). We only used two functions to operate the LCD: message(text) and clear(). The message(text) function takes in a string value which will be displayed on the screen. The clear() function simply erases whatever is currently on the screen leaving the screen empty. This made it really easy for us to use the LCD as they were straightforward and only two to be used.

Aside from this special library, our code is simply made in Python, which runs a loop while the game is still active. Inside the loop, a random number between 0 and 2 will be generated. Depending on what number was generated, or if the game was over, a function would be called to display the message. The function we called *displayText()* to communicate the text we wanted to be displayed to our LCD, where it contained functions from the special library . We quickly determined that characters would get cut off if the word length exceeded the LCD display's screen size. We then decided to incorporate new lines (\n) in our text to utilize the display's capacity to output multiple lines. An example of this was *displayText("BEGIN\CONFIGURATION")*. We then used the function *displayConfigFruit()* which would display the name of the configured fruit that was put into the function. An example of this was *displayCongfigFruit(f)* where f was the current fruit the user had to touch. These functions made it extremely efficient and easy for our group to communicate with the LCD display.

Although a minor problem, which was solved quite quickly, we did have some confusion of how to actually operate the LCD. We had it plugged in and we assumed we would see it turn on right away. We even turned the contrast up and down to see if the LCD was actually on. Seeing as it was not on, we decided to check Adafruit for the part and search if there was something we needed to download for it to work. Lo and behold, we did need to install a couple of things, as mentioned earlier, for the LCD to work on the Raspberry Pi. Aside from setting up the device for use, there were no glaring issues itself with the product.

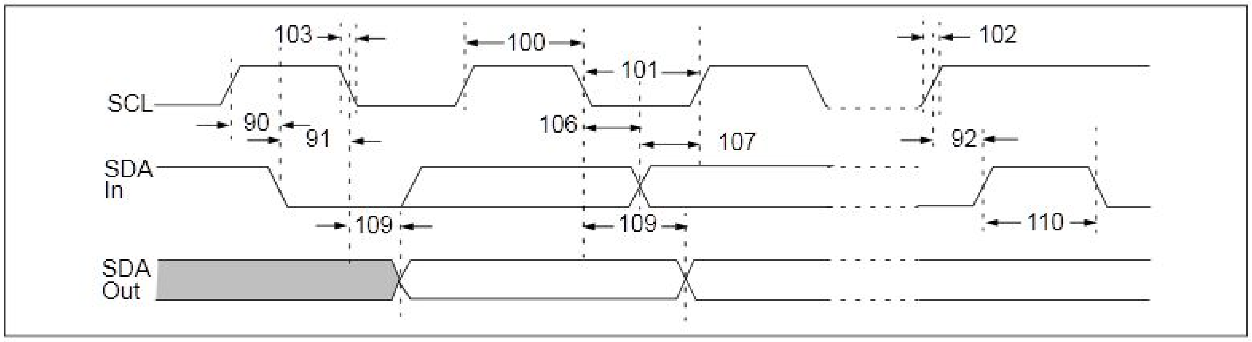

Figure 3. I2C bus data timing diagram for the MCP23017 I/O port expander.

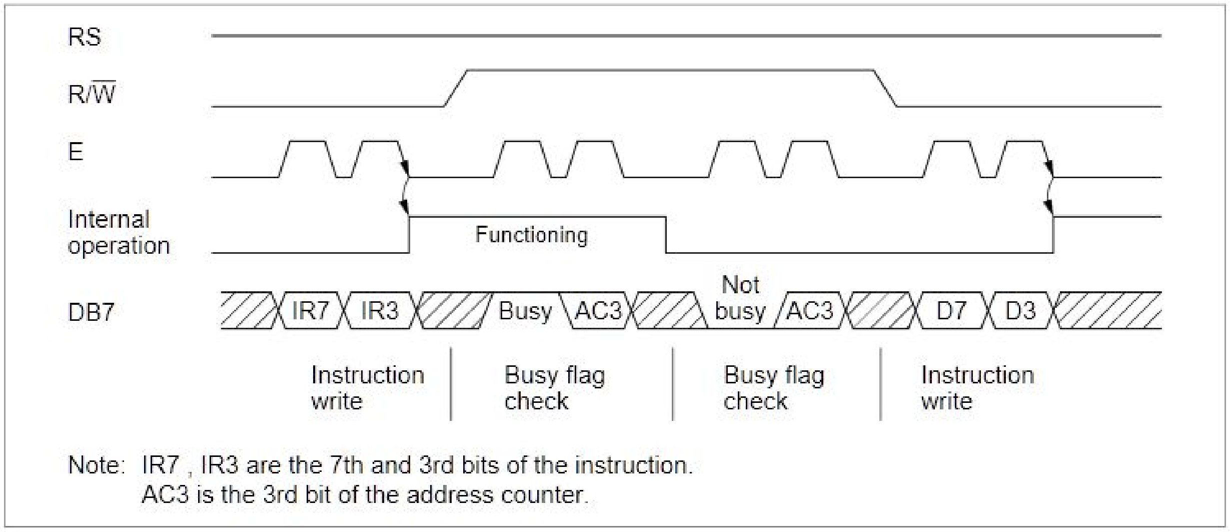

Figure 4. A 4-Bit data transfer timing diagram of the LCD.

Demo Code:

```
import time
import Adafruit_CharLCD as LCD
# Initialize the LCD using the pins
lcd = LCD.Adafruit_CharLCDPlate()

#displays the points when game is over
def displayScore(points):
      lcd.clear()
      lcd.message('POINTS\n')
      lcd.message(str(points))

#displays the fruit to be touched
def displayText(text):
```

```
          lcd.clear()
          lcd.message(text)

    #displays game over when player takes too long to touch fruit
    def displayTimesUp():
          lcd.clear()
          lcd.message('GAME OVER')

    #displays the game configuration
    def displayConfigFruit(fruit):
          lcd.clear()
          lcd.message('TOUCH FRUIT:\n')
          lcd.message(fruit)
```

Communication: Peer-to-Peer Ethernet

The most simplistic use case of Ethernet communication requires “Ethernet Cards” or “Network Interface Cards”, most of which are integrated into the PCB’s of the devices which use the standard (this is also the case in our implementation). Additionally, a Cat5e or Cat6 cable is required to connect two Ethernet cards together. Cat5e and Cat6 refer to the most commonly used twisted pair cables for Ethernet communication. The difference between Cat5e and Cat6 is most apparent in two areas, bandwidth and reduced crosstalk. Crosstalk is a word used to describe the presence of electromagnetic interference between two intertwined, but separate cables. The Cat6 cable has more plastic shielding between its twisted wires, which in-turn reduces the electromagnetic interference between its wires. If there is high electromagnetic interference between wires, packet loss will be more likely to occur from one network card to another. Cat6 has a performance standard of 250MHz with 10000Mbps maximum theoretical throughput, whereas Cat5e has a performance standard of 100MHz with 1000Mbps maximum theoretical throughput.

At Ethernet’s lowest level, packets are sent to and from network cards over a Cat5/6 type cable. Packets consist of binary data which can be subdivided into three main components, a packet header, a payload, and a checksum. The packet header defines the source MAC address, the destination MAC address, and an EtherType which defines the protocol being used for the packet. The next component, the payload, primarily contains the actual data being sent by the source address. Finally, the checksum is a sequence of information that can be validated when the packet is received to detect any packet corruption that may have occurred during the transfer process.

Internet Protocol or IP defines the formatting, sending, and receiving of packets. IPv6 is the most recent protocol, it contains security revisions and increased addressing capacity. IPv4 uses 32-bit addressing whereas IPv6 uses 128-bit addressing. This means that IPv4 has an addressing capacity of 2^32 whereas IPv6 has an addressing capacity of 2^128. IPv6 resolves concerns about IPv4 running out of address space as the internet grows, because IPv6's address space is magnitudes larger.

To keep our implementation simple, we decided to use Flask, a Python micro web framework which handles protocol to allow for quick development, while also also keeping unnecessary features at a minimum, making it optimal for rapid development.

Our Flask API consists of three endpoints:

1. **/tts/**
   A text-to-speech endpoint that will use the pyttsx Python library to convert the contents of the request to speech and playing it using the host computer's speakers.
2. **/score/**
   An endpoint to record the score of a game in the leaderboard table.
3. **/leaderboard/**
   An endpoint that will return an HTML page containing the current leaderboard.

Flask leaderboard endpoint demo code:

```
app = Flask(__name__)

@app.route('/leaderboard/', methods=['GET'])
def getLeaderboard():
    return """
        ... HTML ...
    """
```

On the Raspberry Pi, we used the requests Python library to make HTTP POST requests to the Flask server. This is the code used to test sending tts/HTTP requests, 10.42.0.1 being the IP of the Flask server:

```
import requests
r = requests.post("http://10.42.0.1:5000/tts/", data={'text': 'woooo'})
r = requests.post("http://10.42.0.1:5000/tts/", data={'text': 'this is a test'})
r = requests.post("http://10.42.0.1:5000/tts/", data={'text': 'multiple test phrases'})
r = requests.post("http://10.42.0.1:5000/tts/", data={'text': 'ORANGE APPLE PEAR BANANA'})
```

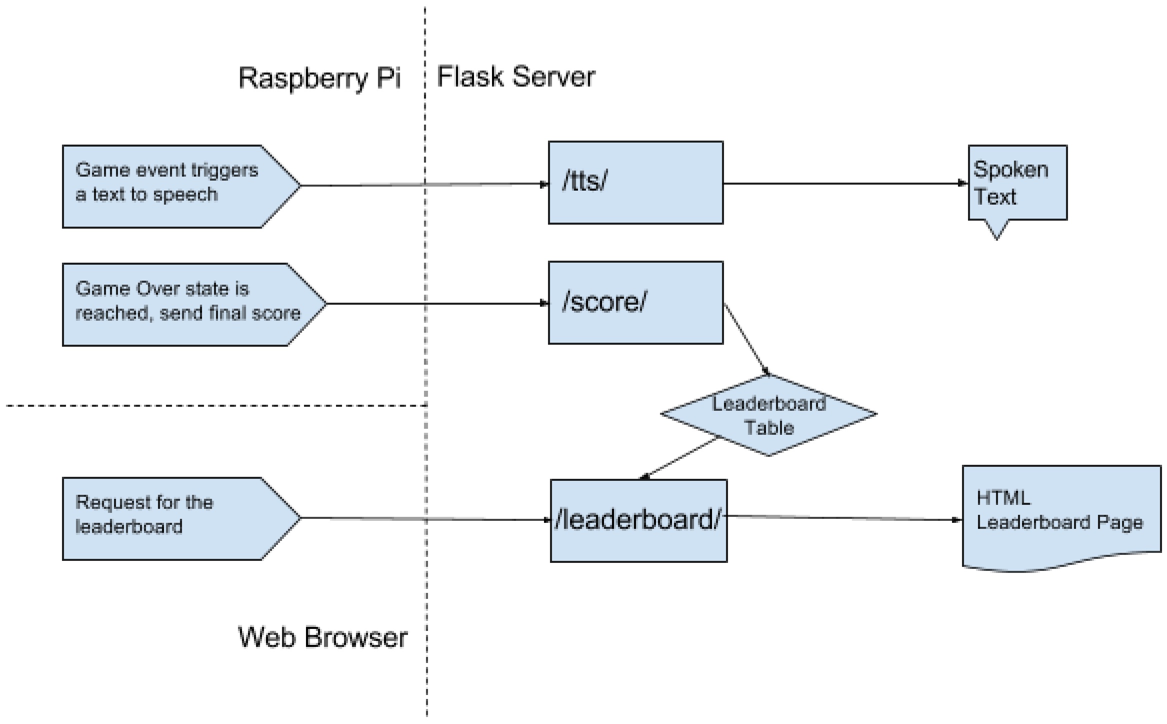

Figure 4. Control flow between Raspberry Pi and Flask Server

In our original project proposal, we were planning to use a Bluetooth USB adapter on the Raspberry Pi to connect to a Bluetooth speaker for text-to-speech. When we tested our Bluetooth adapter it didn’t work, so instead we implemented the text-to-speech as a Flask server over Ethernet. By overcoming the difficulty in this way, it provided us the opportunity to easily implement a leaderboard system.

Source Code

This is our main code that runs the game. It is called game.py.

```
import random
from fs_api import say, postHighscore
#from touch_test.py import touchedFruit
from touch import waitForTouch
import signal
from lcd import displayScore, displayText, displayTimesUp,
displayConfigFruit

import time
import thread
import threading


def waitForFruitTouch(timeout):
    timer = threading.Timer(timeout, thread.interrupt_main)
    pin = None
    try:
        timer.start()
        pin = waitForTouch()
    except KeyboardInterrupt:
        displayTimesUp()
        say("TIME'S UP")
    timer.cancel()

    if pin in pins:
        return pins[pin]
    return None

fruits = [
    "lime",
    "lemon",
    "orange"
]
pins = {}

def getNextFruit():
    return fruits[random.randint(0, len(fruits) - 1)]

# Tells user to touch fruits in order
def configureFruits():
    displayText("BEGIN\nCONFIGURATION")
    say("BEGIN CONFIGURATION")
    time.sleep(1) # sleep for 2 seconds

    for f in fruits:
        pin = None
        while pin is None or pin in pins:
            displayConfigFruit(f)
            say("Touch " + f)
            pin = waitForTouch()

            if pin in pins:
                displayText("Fruit already\nconfigured")
                say("Fruit already configured")

        pins[pin] = f


#game structure
def main():
    pointsToWin = 10
    mainLoop = True
    configureFruits()
    time.sleep(0.5)

    while mainLoop:
        points = 0
        gameOn = True
        timeToHit = 2.4

        displayText("TOUCH ANY FRUIT\nTO BEGIN")
        say("TOUCH ANY FRUIT TO BEGIN")
        waitForTouch()
        time.sleep(0.5)

        while (gameOn):
            #if points % 5 == 0 and 5 < points < 50:
            #    timeToHit -= 0.3

            currentFruit = getNextFruit()
            displayText("TOUCH FRUIT:\n" + currentFruit)
            say(currentFruit)

            #somehow link touchedFruit() function to the interrupting
function.

            fruitTouched = waitForFruitTouch(timeToHit)

            if fruitTouched == currentFruit:
                # did it
                say("GREAT JOB")
                points += 1

                #if points >= pointsToWin:
                #    say("CONGRATULATIONS YOU HAVE TOUCHED %d FRUITS"
% pointsToWin)
                #    say("YOU ARE A WINNER")
                #    gameOn = False
            else:
                say("YOU SUCK")
                gameOn = False

            time.sleep(0.2)
            timeToHit = 2 * ((10 - min(points / 2, 10)) / 10.0) + 0.1

        postHighscore(points) # post to highscore server

        say("GAME OVER")
        say("YOU SCORED %d POINTS" % points)
        displayScore(points)
        time.sleep(3)

main()
```

Conclusion

Overall, the project of designing and developing Fruit Smash allowed Team 14 to gain hands on experience with hardware, and experience collaborating with a team on a project that reflects on how real industries do today. Fruit Smash is comprised of three of the four required devices of sensors, display, and communication through the capacitive touch, LCD, and peer-to-peer ethernet. We are very pleased with the final product, as it is an interactive application, where players can actually have an enriched gaming experience. In addition, we were able to take an everyday object such as fruits, and transform it into a concept that goes beyond the purpose of it. Team 14 is feeling positive and accomplished, and we look forward to the next project to see what else our team can create and develop.

References

1. https://learn.adafruit.com/adafruit-mpr121-12-key-capacitive-touch-sensor-breakout-tutorial
2. https://learn.adafruit.com/character-lcds
3. http://www.robotshop.com/media/files/pdf/mrp121-datasheet-dfr0129.pdf
4. https://cdn-shop.adafruit.com/datasheets/HD44780.pdf
5. https://learn.adafruit.com/adafruit-16x2-character-lcd-plus-keypad-for-raspberry-pi/usage
6. http://www.zseries.in/electronics%20lab/displays/lcd/#.WevDf4hryUk
7. https://en.wikipedia.org/wiki/Liquid-crystal_display
8. https://cdn-shop.adafruit.com/datasheets/mcp23017.pdf
9. https://www.adafruit.com/product/732
10. https://en.wikipedia.org/wiki/Category_5_cable
11. https://en.wikipedia.org/wiki/Category_6_cable
12. https://customcable.ca/cat5-vs-cat6/
13. http://www.informit.com/articles/article.aspx?p=131034&seqNum=5
14. https://en.wikipedia.org/wiki/Ethernet_frame
15. https://en.wikipedia.org/wiki/Internet_Protocol
16. https://en.wikipedia.org/wiki/Network_packet