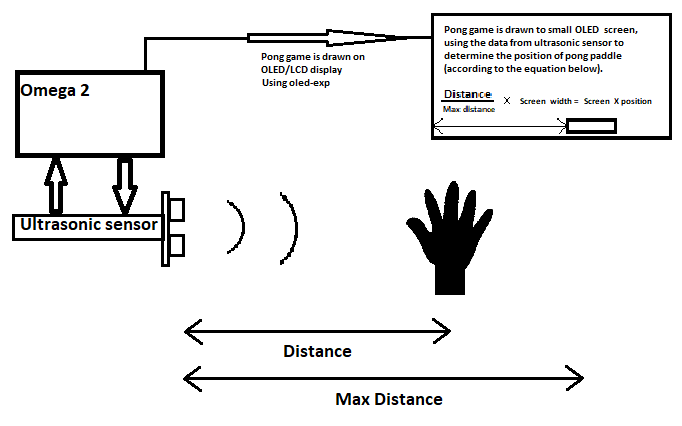
Project Report

### **By:** Patrick Hadlaw



# **Project Overview:**

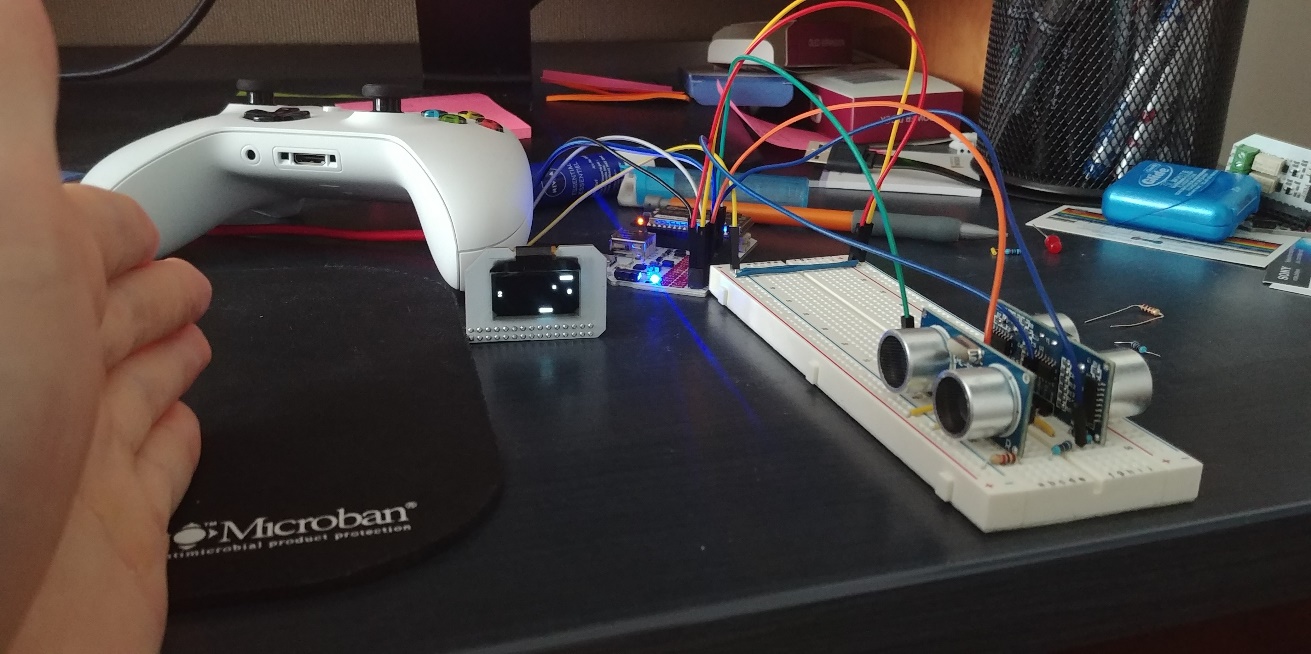
## Description:

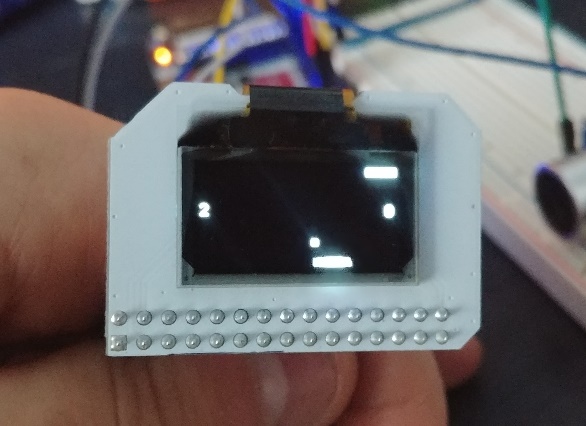
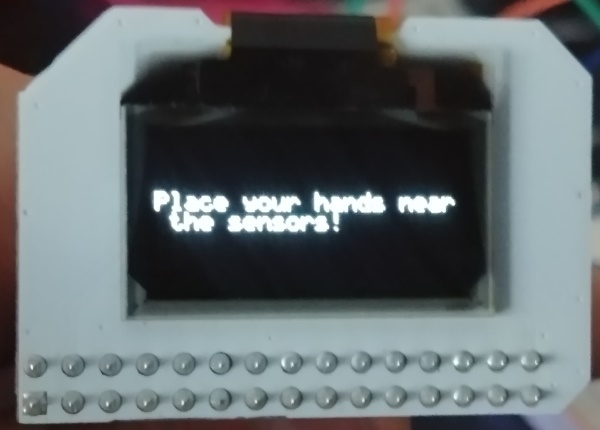
Our project is an interactive, motion controlled pong game using our Omega in conjunction with ultrasonic distance sensors and the Onion OLED Expansion. The sensors are used to detect the horizontal distance of the players hands from the sensor; this is used to approximate the position the pong paddles in the game relative to the width of the display.

The idea is to allow the players to move the pong paddles by simply moving their hands through the air. Ideally, this would create a very natural way to control an arcade game such as pong. The game features three game modes: player versus player, player versus computer and computer versus computer.

We managed to design and develop a system that works as described above. What we didn’t but would’ve liked to implement is a stronger approximation algorithm for the smoothing of ultrasonic sensor data, such as a smoothing Bayesian filtering technique. Our method uses multiple samples of sensor readings per each update of the game and averages this into a running average. We also reduce values that are greater than one standard deviation from the past five data points to avoid outliers. Each sensor keeps track of its own running average and the game then draws the paddles using this data to position each paddle. This method works well but suffers fits of noisy readings that cause the paddles to glitch around. In consumer electronics, more sophisticated smoothing algorithms such as Bayesian filtering are used to smooth noisy sensor data. Unfortunately, we did not have the time to research and implement such an algorithm.

Our project fully implements rendering moving geometric objects onto the OLED screen (it was not really designed to do this), reading and smoothing of ultrasonic data, and a variable delta time pong game calculating the velocity and collisions of the ball against the walls and paddles. To accomplish all of this with an OLED screen that only has one data transfer pin we had to do extensive optimizations including: exclusive pixel drawing/clearing using a double image (double buffer) technique, as well as concurrently running sensor reading functions while drawing the game using C++11 threads (all technical information about our software methods are in the software design section).



# **System Design:**

## Dependencies:

* Onion I2C libraries: oniondebug, onioni2c and onionoledexp
* GPIO library: ugpio
* C++11 standard: -std=c++11
  + For the use of C++11 multithreading

## Building:

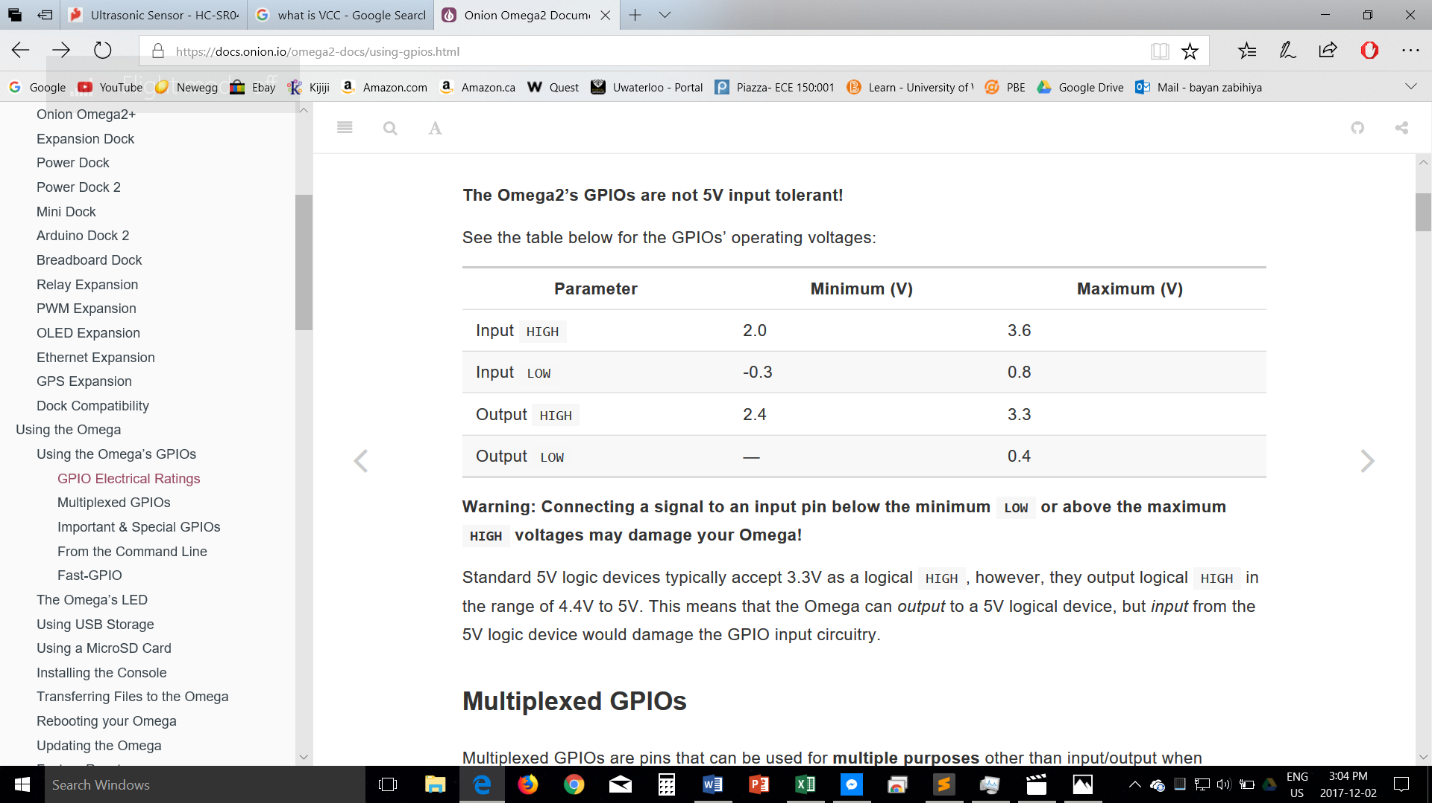
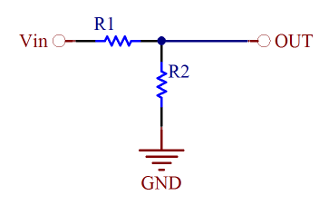
* Makefile using CXX required with compiler flag -std=c++11 at the end of recipe and -D P\_VS\_C for player versus computer, -D C\_VS\_C for computer versus computer and default is player versus player
* Ugpio, liboniondebug, libonioni2c and libonionoledexp shared objects must be within toolchain library directory or makefile library directory
* Build and upload to omega with the following commands in project directory:
  + sh xCompile.sh -buildroot ~/source/ -lib ugpio -lib oniondebug -lib onioni2c -lib onionoledexp
  + rsync -a motionPong [root@192.168.3.1:~/motionPong](mailto:root@192.168.3.1:~/motionPong)
    - This executable is for player versus player game-mode
  + rsync -a motionPongPVC [root@192.168.3.1:~/motionPongPVC](mailto:root@192.168.3.1:~/motionPongPVC)
    - This executable is for player versus computer game-mode
  + rsync -a motionPongCVC [root@192.168.3.1:~/motionPongCVC](mailto:root@192.168.3.1:~/motionPongCVC)
    - This executable is for computer versus computer game-mode

# **Hardware Design**

Of course, the Onion Omega 2 served as the primary microcontroller, memory, and storage in our project while the power dock was the IO that interacted with all other components. The interesting thing is that the Power Dock is entirely optional as documentation for all pins on omega exists. However, the Power Dock made physical assembly much simpler. The game uses 2 ultrasonic sensors (datasheet below)

<http://www.electroschematics.com/wp-content/uploads/2013/07/HCSR04-datasheet-version-1.pdf>

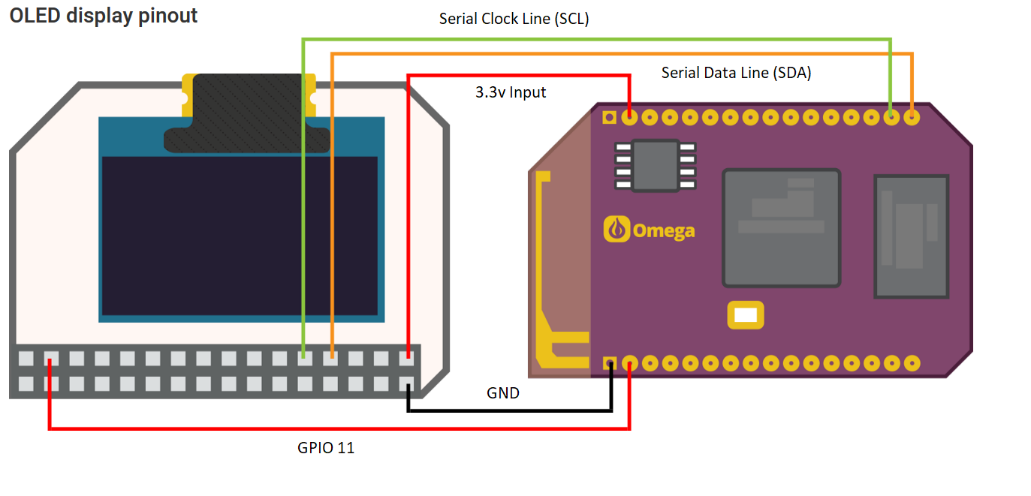
to track players hand location to update pong paddles positions and the Onion OLED Expansion to display game assets.

This is all “fun and dandy” except that the Omega’s GPIO’s doesn’t take 5V, the GPIO’s instead use 3.3V. This is a problem since most simple electronic components not exclusive to onion run off 5V since that is the standard for Arduino which is the standard in this market. For this reason, 3.3v sensors were both expensive and rare and we had to settle for the readily available 5v sensor HC-SR04, lucky for us the omega does offer 5v VCC to power it, but the GPIO’s have a max rating for 3.6v. 

<https://docs.onion.io/omega2-docs/using-gpios.html>

<https://www.daycounter.com/Calculators/Voltage-Divider-Calculator.phtml>

To get around this problem we needed to reduce the 5v output from the Echo pin on the sensor down to 3.3v for the GPIO. This is where we learned about voltage dividers in which an input voltage is “partially” shorted with GND to decrease it. The formula is Vout = Vin x R2 / (R1 + R2) since we had access to 1k ohm resistors. We plugged in 1000 for R1 then solved for R2 which we found was 1941ohms, very close to 2k ohms so we just used two 1k ohm resistors instead. This is shown in the diagrams above.

Even the Onion OLED expansion display had a few issues, most notably the single pin for sending data to the display. I2C communicates with 2 pins, one pin for transferring data (SDA), and one pin for when data is being transferred (SCL). 2 pins are for power and the last pin sends some data back to omega regarding state used by the oled-exp library.

<https://community.onion.io/topic/260/pinouts-for-omega-docks-and-expansions/2>

The underlying issue was revealed when trying to use the oledClear function to clear before redrawing the new position of paddles and ball. What oledClear does is write a zero byte to all 8 rows of 128 columns on the screen (screen is divided into rows, for use with text). That is 1024 bytes of data sent just to clear the screen. Trying to do this after every single draw command with only one pin causes a massive bottle neck when trying to do this at our desired refresh rate. Effects were input lag, screen tearing, display flickering and noting displaying on the OLED, which is plain awful. We solved this issue by implementing our own handler for clearing and drawing to the screen, this is described in detail in the software section. This fix basically solved all the issues by reducing the amount of data being sent via the single I2C data transfer pin.

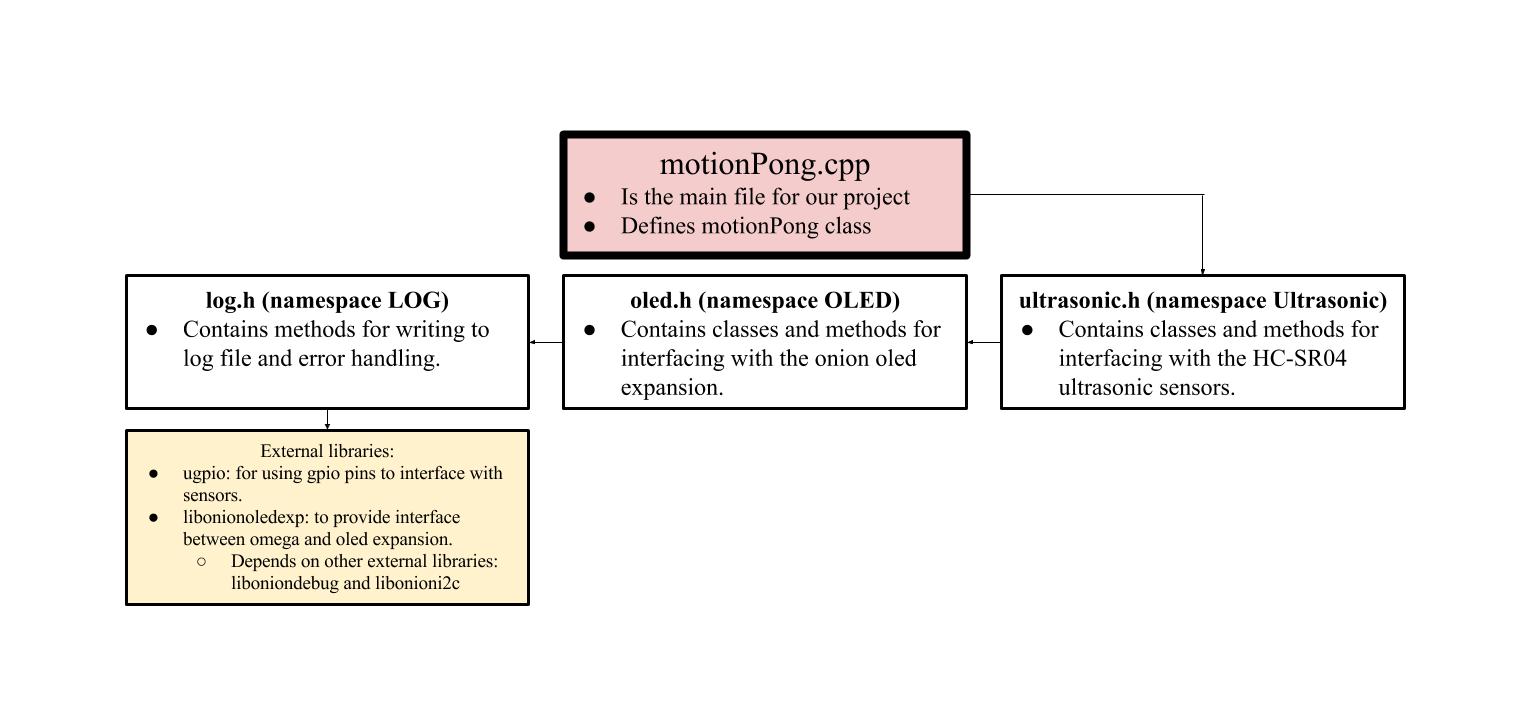
# **Software Design: 4 Components**

## Class Diagram:

## System Dependant and Independent Components:

Everything within motionPong.cpp, log.h and Stats namespace in ultrasonic.h are system independent while everything in oled.h and in the Ultrasonic namespace in ultrasonic.h are system dependant.

## File diagram:



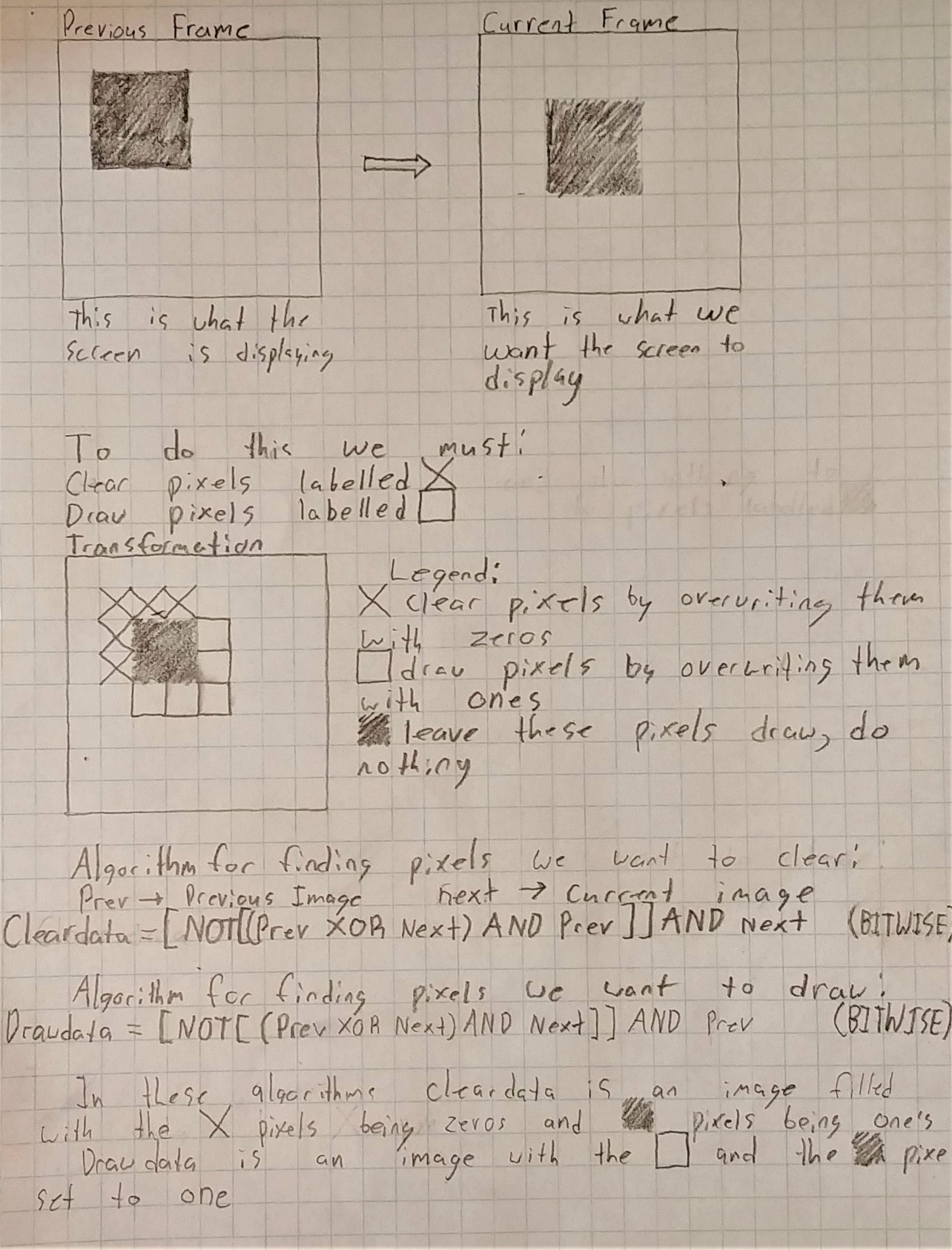
## OLED Expansion: implemented in oled.h

We very quickly learned that using the OLED expansion for drawing moving game geometry would not be trivial. This is because the OLED expansion uses I2C communication to update screen data over only one data transfer pin. Also, we discovered that the OLED clear command takes upwards of up to half a second. Not only is it not possible to update a game with a 2Hz framerate but the clear command takes so long that you can’t see anything show up on the screen.

To solve the limitations of the OLED expansion we ditched the clear command in our update loop and instead used a technique which uses to images or buffers storing the pixel data of the current frame and the pixel data of the previous frame. This method works by overwriting pixel data on the OLED expansion that needs to be altered while not doing anything to the other pixels on the screen. To do this we compare the previous and current frame to find pixels that are exclusive to the previous frame, we know these pixels are the only ones that need clearing, so we overwrite the pixel data of these pixels to zero or off. We do the same for pixels that are exclusive to the current frame, for which we overwrite the pixel data of these pixels to one or on.

This method was implemented in the class DrawContext. In this class we also added methods which would fill rectangles into the current frame. This allows us to draw rectangles on the screen on a per frame basis, as well as update the rectangles positions smoothly (at a high refresh rate with no flickering or tearing).

**The following diagram shows in detail how this is accomplished:**



## Ultrasonic Sensors: implemented in ultrasonic.h

We used ultrasonic transducing distance sensor as the motion control for our game. We started by programming the reading of the sensor. To do this, we send a ten-millisecond pulse through the GPIO hooked up to the trigger pins of sensors. We then wait for a response in the echo pin at which we time the output pulse of the echo pin, this is the time between emitting ultrasonic wave and receiving this wave. To get the distance detected from the sensor we multiply the time in seconds by the speed of sound then divide by two to account for the fact that the wave travels back and forth.

Once we got the sensor reading working we realised that the sensor data was very noisy and that sometimes the sensor completely misses the echo. To account for this our sensor methods take multiple samples of sensor readings then take the median of those readings. We chose the median of the dataset because we needed to avoid invalid values which were either far too low or far too high.

The sensor readings were still very noisy making the pong paddles moves very unpredictably. To solve this problem, we implemented smoothing using a moving average. In this moving average we would track the past five sensor values, calculate the sample standard deviation of these values and reduce any value being averaged that is higher than the deviation. We did this because these values are most likely outliers. The previous methods smoothed our sensor data and the pong paddles started to behave as intended.

To optimise the sensor code, we return the max distance for any reading that takes more than it takes sound to travel that distance. We realised that the OLED and ultrasonic methods within our program structure were both slow because they waited on data transfer between our components, and that by multithreading the ultrasonic and OLED update methods we could save time to be able to refresh the OLED at a higher speed. To accomplish this, we launched threads running the sensor reading methods promising future return values which would be retrieved after OLED methods are called. This was done using C++11’s std::async and std::future.

## Logging: implemented in log.h

Our logging methods write to: runtime.log. We have four functions which are described in the class diagram above. Our implementation simply adds a line to the beginning of the log file with a prepended timestamp that has the following format: (hour:minute:seconds). The log file can write messages, warnings, errors or just print a line of text with the following formats.

(hour:minute:second)[LOG][MESSAGE]: <message>

(hour:minute:second)[LOG][WARNING]: <warning>

(hour:minute:second)[LOG][ERROR]: <error>

Logged errors are usually fatal and will cause the program to abort, so we also got our OLED screen to write any error on screen if it can.

## Pong Game: implemented in motionPong.cpp

Our pong game is defined within the MotionPong class which contains the highest-level interface for our project. The MotionPong class defines two PongPaddle members which contain all state relating to the pong paddles. It also contains a OLED::DrawContext member to handle the game drawing. The init method initialises sensors, OLED and all initial state required to begin game simulation. The update method updates ball position and velocity vectors through a variable timestep calculation. We also increase speed by twenty percent every time the ball collides with paddle and calculate speed of paddle which we add to horizontal velocity of ball every collision with paddle. The draw method writes all data to draw context and draws to screen. The reset method waits for players to put hands close to sensor before starting the next round. After one player reaches 4 points the winner is announced on OLED and the game closes.

## Testing

Our project required a lot of testing, researching and debugging to get working. As a result our final result works almost as intended.

## Limitations:

Due to limited time we were not able to improve our smoothing algorithm and thus the pong paddles can sometimes not be as responsive as we would like. Additionally, the OLED has a lower refresh rate than we would like but this is due to the limitations of the hardware.

## Lessons Learned:

* Hardware doesn’t work straight out the box, it requires a lot of fiddling and researching to get small hardware working as intended. Especially if it wasn’t designed for the task at hand such as the OLED expansion which was only designed to display static data.
* Sensors produce very noisy readings and that the software does a lot of work to approximate the ideal value for the sensor