Lit Brolly

Nōn omne quod nitet aurum est.

Aaron Tan
Lily Yang
Hengcheng Yu

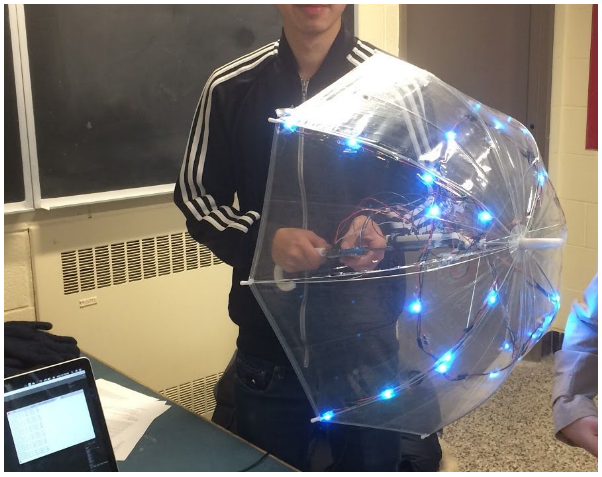
Group #25

Project Overview
The idea behind the project was to make a LED Blade Runner inspired umbrella—lining the spokes of a clear umbrella with LEDs and changing their colour and intensity based on the orientation and speed of the umbrella.

To implement this idea, we used the Omega2 to take an input from the accelerometer (through the SDA/SCL pins), mapping these to R, G, and B values from 0-255. We then sent them over to the Arduino over I2C, where the microcontroller sets colour and the intensity and colour of the Neopixel strips through its digital pinouts (6-10).

Originally we intended to communicate with the Neopixel LEDs through SPI; however, the very specific timings required by the Neopixels could not be controlled precisely enough by the Omega2. Instead, we used the I2C and the Arduino Adafruit Neopixel library, which were both better documented, to communicate with the LEDs. We also had stretch goals of allowing users to be able to switch between lighting profiles through a phone or web interface, but this was deemed unfeasible during the timeframe allotted.

Original Features	**Final Features**
Take input from the accelerometer (through either SPI or I2C)	Take input from the accelerometer through I2C
Take input from a gyroscope (through either SPI or I2C) (not included in our original project proposal but added shortly after submission)	Gyroscope was not used
Change intensity of LEDs (through SPI, backup plan: use Arduino Adafruit Neopixel library)	Control the brightness of the LEDs through the Arduino Adafruit Neopixel library
Change colour of LEDs (through SPI, backup plan: use Arduino Adafruit Neopixel library)	Control the colour of the LEDs through the Arduino Adafruit Neopixel library
Log accelerometer readings	Log accelerometer readings with timestamp, status upon writing I2C values to different registers, and histogram of past accelerometer readings to text files on the Omega
A strand of LEDs along each of the 8 spokes	A strand of LEDs along each of 4 spokes, plus a bonus circular pattern
Allow user to be able to switch lighting profiles through a phone or web interface	Was not implemented, but program can be controlled via SSH to test different colours

System Design

Hardware Components:

- Omega2
    - We used the Onion Omega2 that was provided for us.

- Arduino Dock R2
    - We used the Onion Arduino Dock for the Omega2. The dock has identical pin layouts to the Arduino and since they share the same microcontroller (the ATMega328P), using the dock with the Onion gives us access to Arduino hardware as a slave device to the Omega2.

- Neopixel LEDs
    - We used WS2812B APA102 individually addressable smart full-color LED pixel made by Kuman.
    - We soldered wires to connect them together into a daisy chain. One strip of LEDs contains 4 individual LEDs and are intended to line one spoke. We hot glued the back of each of the LEDs soldering pads to ensure that the wires would stay in place, then taped the strips to the spokes of the umbrella.

- Accelerometer
    - We used a ADXL345 accelerometer module made by OSEPP. It is a 3-axis MEMS accelerometer with 13-bit resolution measurement at up to +-16 g that can use both SPI and I2C.
    - We used I2C to interface with the accelerometer.

- 3.7V 2Ah Lithium-ion Battery and Xiaomi 20Ah Power Bank
    - We used a 2Ah battery to power the LEDs through a JST-PH connector connected to the breadboard.
    - We used a 20Ah portable power bank to provide power to the Omega through a Micro-USB B connector.

- Wires
    - Wires were used to make all the connections.

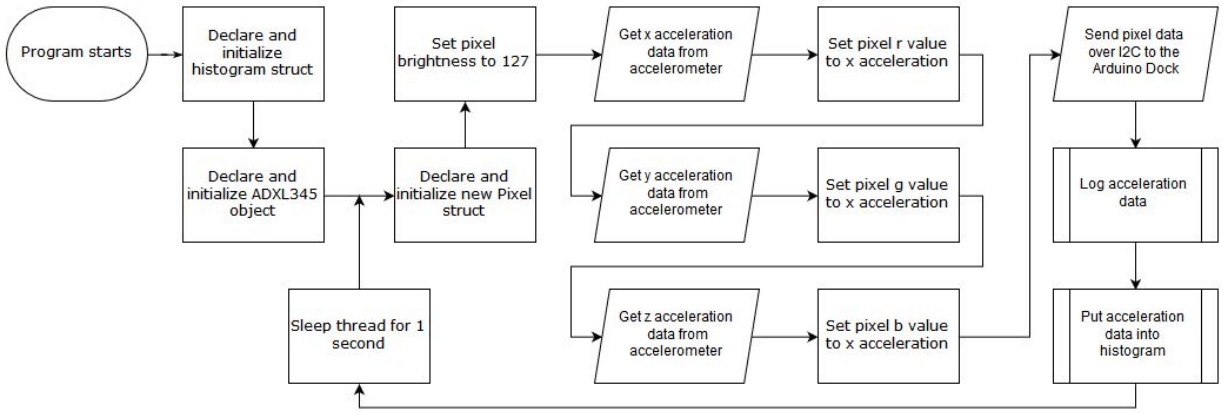
- Breadboard
    - A breadboard was used to house the accelerometer and connect the neopixels and accelerometer to the Arduino Dock.
    - The breadboard was strapped onto the handle of the umbrella using tape.

- Resistors
    - 5 1k ohm resistors were used for the neopixels (to protect the LEDs).
    - 1 10k ohm resistor was used as the pull up resistor to connect the accelerometer to the Arduino Dock (for I2C).
    - These resistors were placed on the breadboard.

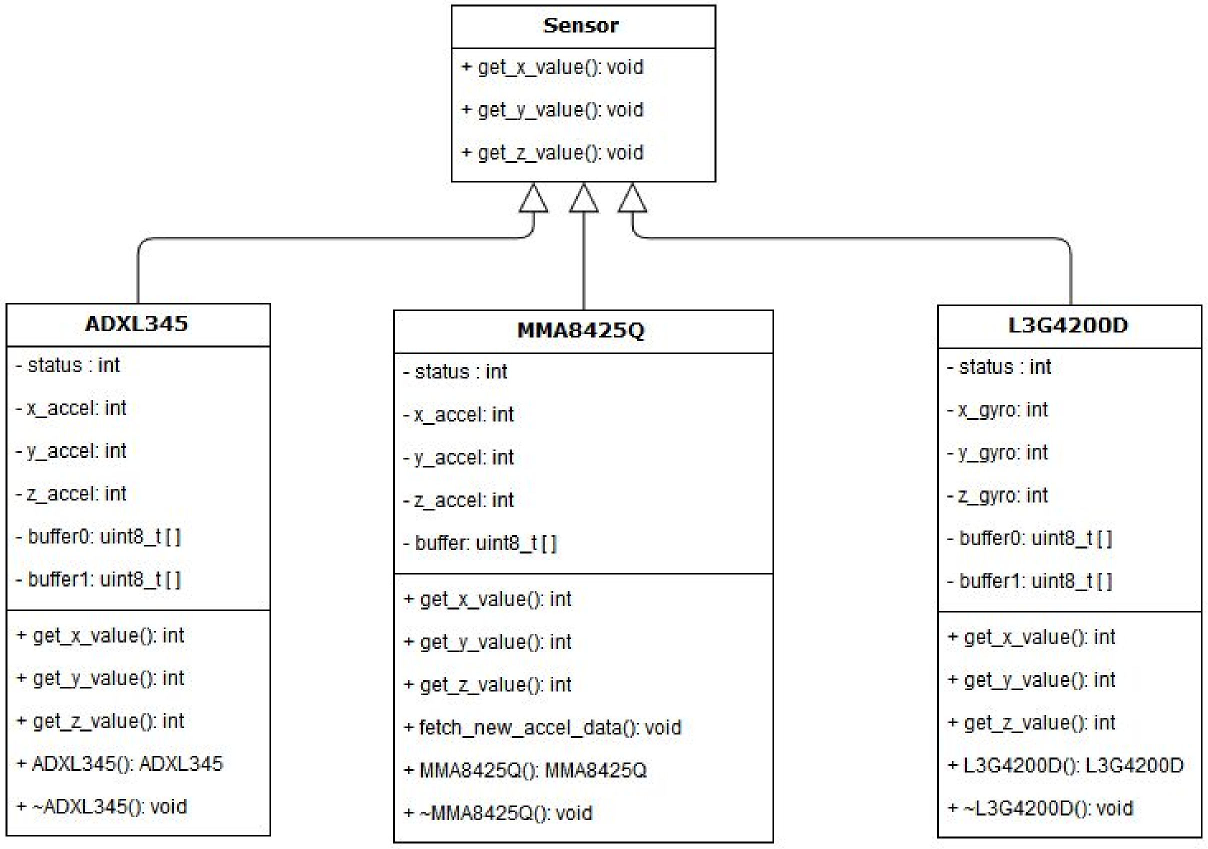
- Umbrella
    - 34 inch child sized clear umbrella.

Software Design

Flowchart for the main program


Program starts
Declare and initialize histogram struct
Declare and initialize ADXL345 object
Sleep thread for 1 second
Declare and initialize new Pixel struct
Set pixel brightness to 127
Get x acceleration data from accelerometer
Set pixel r value to x acceleration
Get y acceleration data from accelerometer
Set pixel g value to x acceleration
Get z acceleration data from accelerometer
Set pixel b value to x acceleration
Send pixel data over I2C to the Arduino Dock
Log acceleration data
Put acceleration data into histogram

UML class diagrams for Sensor abstract class and subclasses



Class explanations:

- Sensor is an abstract class that has only virtual member functions: `get_x_value()`, `get_y_value()`, and `get_z_value()`. These are meant to be overridden in any child classes, and generically serve to get the x, y, and z, values from any particular sensor.

- ADXL345 is a derived class from base class Sensor. Here, public member functions `get_x_value()`, `get_y_value()`, and `get_z_value()` have been overridden and return the x, y, and z accelerations respectively in the form of an int ranging from 0 to 255. We also define a constructor `ADXL345()` and destructor `~ADXL345()` in order to set up the accelerometer correctly (by writing to data registers to tell it to turn on, set sensitivity, and others). In terms of member variables, we have a private `status`, that changes depending on the success or failure of writing to I2C addresses, `x_accel`, `y_accel`, and `z_accel`, that contain the values of acceleration, and `buffer0` and `buffer1`, which contain the direct readings from the accelerometer (least significant bit and most significant bit respectively).

- MMA8425Q is another derived class from base class Sensor. Here, public member functions `get_x_value()`, `get_y_value()`, and `get_z_value()` have also been overridden and return the x, y, and z accelerations respectively in the form of an int ranging from 0 to 255. We define `get_new_accel_data()` to update the buffer in which acceleration data occurs; here, all the data is contained in one register as opposed to the many in ADXL345 and L3G4200D. We also define a constructor `MMA8425Q()` and destructor `~MMA8425Q()` in order to set up the accelerometer correctly (by writing to data registers to tell it to turn on, set sensitivity, and others). In terms of member variables, we have a private `status`, that changes depending on the success or failure of writing to I2C addresses, `x_accel`, `y_accel`, and `z_accel`, that contain the values of acceleration, and `buffer`, which contains 7 bits describing the x, y, and z accelerations detected by the sensor.

- Finally, we have L3G4200D as another derived class from base class Sensor. Here, public member functions `get_x_value()`, `get_y_value()`, and `get_z_value()` have also been overridden and return the x, y, and z tilts respectively in the form of an int ranging from 0 to 255. Unlike the other two subclasses, the L3G4200D returns tilts rather than acceleration from the same function name; a great example of runtime polymorphism in C++! We also define a constructor `L3G4200D()` and destructor `~L3G4200D()` in order to set up the gyroscope correctly (by writing to data registers to tell it to turn on, activate three axes, and more). In terms of member variables, we have a private `status`, that changes depending on the success or failure of writing to I2C addresses, `x_gyro`, `y_gyro`, and `z_gyro`, that contain the values of tilt in each axis, and and `buffer0` and `buffer1`, which contain the direct readings from the gyroscope (least significant bit and most significant bit respectively).

Struct explanations:

```
typedef struct Pixel {
      int brightness;
      int r;
      int g;
      int b;
} Pixel;
```

The Pixel struct is a structured data type that groups together information about a pixel: how bright it should be, as well as its red, green, and blue intensities.

```
 typedef struct Histogram {
       int accel[5];
       int histx[5];
       int histy[5];
       int histz[5];
 } Histogram;
```

The Histogram struct is a structured data type that declares integer arrays of 5 buckets that acceleration data can be grouped into.

System dependent/independent components:
When writing software for this project, because our project was very close to the metal and interfaced very directly with the hardware that we used, there were many system dependent components:

- Each accelerometer/gyroscope class is system dependent because the slave addresses are unique for each piece of hardware (e.g. 0x1C, 0x53, and 0x68 for ADXL345, MMA8425Q, and L3G4200D respectively). In addition, each sensor has different data registers to read/write from in order to extract useful data.
- The Sketch running on the Arduino dock can be used by any Arduino, but it must be an Arduino-compatible board (such as the Arduino Dock we have!). Therefore, it is system dependent.
- The code to control the Neopixels uses a library specifically written to control these types of LEDs, thus that software component is also system dependent.

Nonetheless, our project also had several system independent components in software:

- The logging infrastructure was completely software independent and could be tested by feeding dummy values into the function. It does not depend on a specific piece of hardware to work (theoretically, all it needs is three values that change over time); thus, it is system independent.
- The abstract class Sensor is system independent, as all of its functions are purely virtual. Because of that, it cannot be instantiated and all its member functions must be overridden in any subclasses. Thus, it could be used for any Sensor that returns x, y, and z values and is system independent.

Logging infrastructure:
During every loop of the main function, logging is performed, with timestamps.

- The status of the I2C write action is logged (where 0 is success and !0 is failure) in a log.txt file.
- The accelerometer values are logged in the log.txt file.
- Acceleration data is added to a histogram and then written every ten seconds to a histogram.txt file. Data manipulation is performed on the acceleration x, y, and z-values to calculate a total acceleration; this is also logged in histogram.txt.
- These files can be viewed from the command line with `cat log.txt` or `cat histogram.txt`

Source code files:

- Synergy.cpp
  - This is the main program, with all the class definitions and logic.
- Makefile
  - This is the file that tells the compiler how to compile the program
- xCompile.sh
  - This is the shell script that we use to compile the program.
- Ardu.ino

    - This is the sketch that runs on the Arduino Dock to receive commands over the SDA/SCL pins from the Omega2.

Testing

Testing all the components took the most time during this project. We tested our components first using Arduino and then with the Omega2, because using an Onion was very time-consuming and it would be hard to tell whether a problem was hardware or software related. The pins on the Arduino Dock and the Arduino are also identical which makes the connections very similar for both the boards when we would eventually transfer the hardware over.

First, we used an Arduino and breadboard to test each individual LED. Once we verified that each LED was functional we soldered them together into a strand and tested the strand. This was necessary since if one LED in the strand was not functioning then the strand would not light up.

We also used the Arduino to test to accelerometer (both of them) and the gyroscope. The gyroscope was originally working but then after and soldering accident, it ceased to function. This was evident when we tested it with the Arduino code. The accelerometer never gave out an output when we tried to test it leading us to purchase another accelerometer. When we verified that the new accelerometer was in working condition (using the Arduino), we tested it with the Omega2.

We wrote a small command line program to output accelerometer readings every second to the command line in order to test the new accelerometer. We tilted, left it in place, and played around with it to make sure that our readings were consistent with the motion that the accelerometer was experiencing.

After we wired up the Neopixels and the accelerometer, we mashed the code together and tested the lights to make sure that they were lighting up in ways that we expected. When large amounts of movement happened, lights glowed in beautiful, strange ways; when left alone, it usually reverted to a bright white colour. These results were consistent with what we expected to see.

Limitations

Our project has some limitations as an umbrella, as the components of our project affect some primary functions of the umbrella. The wiring made it very difficult to close and open the umbrella, due to stretching and the breadboard being glued to the center handle. The Omega2, Arduino Dock and battery were very bulky and reduced the portability of the umbrella.

In addition, we initially believed that we could run our NeoPixel LEDs directly off the Omega2 with C or C++ code. However, after testing extensively for over 30 hours we could not get the NeoPixel libraries to work with the Omega2. This testing included using the provided libonionneopixel, the neopixel_Cwrapper, looking up the source code for the implementation of the functions defined in these header files, and trying some code ourselves. In the end, we decided to use the Omega2 to communicate with Arduino sketches to control the NeoPixels.

Lessons Learned
We learnt many lessons throughout this project.

First, hardware is very fragile. At first we bought both an accelerometer and a gyroscope but we weren't able to use either of these original components in our project. While soldering the header pins onto the gyroscope, an accident rendered the gyroscope unuseable. No matter how many times we tested the accelerometer, we could not get an output, leading us to conclude either we broke it somehow accidentally or it was dead on arrival. We ended up buying another accelerometer which we used for the project.

If we were to start the project again, we would buy a different accelerometer and gyroscope. The original ones we bought were hard to work with and might have been defective upon arrival. We also would probably have started with I2C instead of SPI to test the accelerometer and the gyroscope, because I2C lead to better results with the second accelerometer.

Secondly, we learned that soldering can be a difficult beast to tame. During the project, soldering took up a significant portion of the time, and required full concentration. To improve on our process, we would also have used a better soldering iron (such as one with variable temperature and a new tip) to avoid accidents and make the process faster and smoother. We would have also soldered in an environment with better ventilation to avoid long term health damage.

Lastly, we would have also done more research on the Neopixels and its compatibility with the Omega2. We spent a lot of time trying to control the Neopixels directly (without Arduino code) from the Omega2, a task we now think is borderline impossible. If we started over today, we would skip directly to controlling the Neopixels through the Arduino Dock instead of trying to work them directly through the Power Dock.

Overall, this project was a lot of fun and gave us a pretty cool introduction to embedded computing!

Appendix

Peer Contribution:

- Aaron Tan:
    - Software: Classes, I2C, main program logic, style editing
    - Hardware: Wire stripping and hardware QA with Omega2 and Arduino Dock
    - Miscellaneous: Set up the virtual machine on three different laptops from source
- Lily Yang:
    - Software: Logging infrastructure
    - Hardware: Setting up the circuit on breadboard with accelerometer and complete strands, soldering
    - Miscellaneous: Bought most of the materials, making multiple hour-long trips to distant lands
- Hengcheng Yu:
    - Software: Arduino sketch, debugging, code review
    - Hardware: Lots of soldering, setting up the strands
    - Miscellaneous: Wired up umbrella and attached bread board to handle, supplied bubble tea

We also give a special thanks to Austin Jiang for driving us to places so we could buy emergency accelerometers!

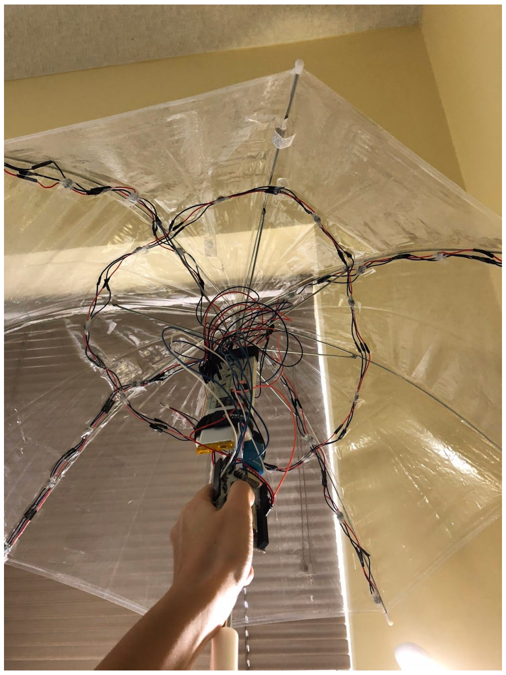


Figure 1: The finished umbrella.

Figure 2: The accelerometer that was used in this project(ADXL345).

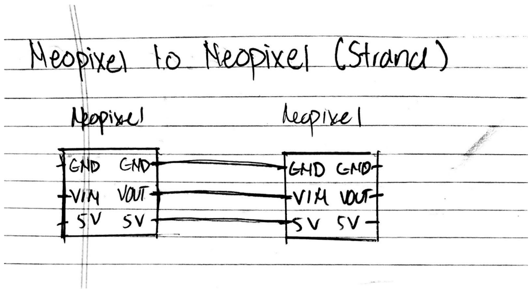


Figure 3: Neopixel to Neopixel connection.

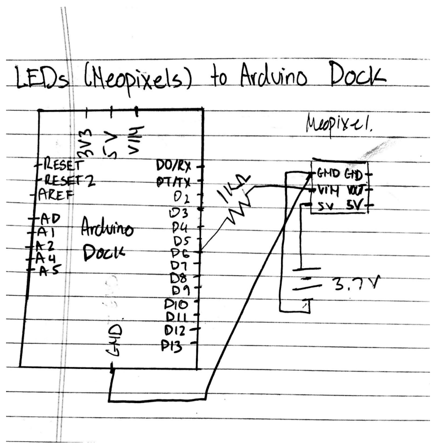


Figure 4: Neopixel to Onion (using Arduino Dock) connection.

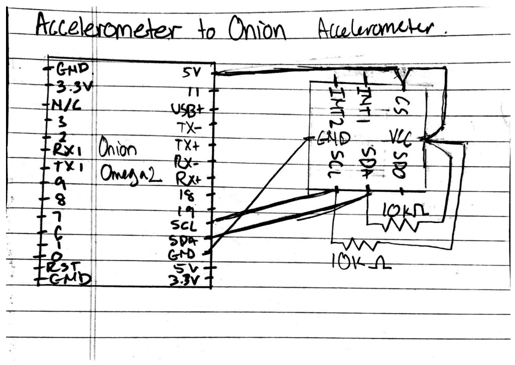


Figure 5: Accelerometer to Onion (using Arduino Dock) connection.

Figure 6: The umbrella in the dark.

Source code:

Synergy.cpp:

```
#include <onion-i2c.h>
#include <iostream>
#include <fstream>
#include <ctime>
#include <cmath>

#define NEOPIXEL_I2C_DEVICE_NUM 0
#define DEV_ADDRESS 0x08

// Abstract Sensor class
class Sensor {
public:
	// Pure virtual functions to get x, y, and z accelerations
	virtual int get_x_value() = 0;
	virtual int get_y_value() = 0;
	virtual int get_z_value() = 0;
};

/****************************** ADXL345
******************************/
/* The ADXL345 is a small ultralow power 3-axis accelerometer.
 * There are many others like it, but this one is mine.
 */

// Derived class ADXL345 is a subclass of the Sensor
class ADXL345 : public Sensor {
public:
	// Concrete implementation of virtual functions
	int get_x_value();
	int get_y_value();
	int get_z_value();

	// Constructor and destructors
	ADXL345();
	~ADXL345();
private:
	int status, x_accel, y_accel, z_accel;
	uint8_t *buffer0, *buffer1; // buffer0 contains least significant
bit, buffer1 contains most significant bit
```

```
};

// Setup the ADXL345
ADXL345::ADXL345() {
	// Initialize some variables
	status = 0;
	buffer0 = new uint8_t[2];
	buffer1 = new uint8_t[2];

	// i2c_write(devNum, devAddr, addr, val)
	// The ADXL345 has an address of 0x53 (83); writing 0x0A (10) to
0x2C (44, bandwidth rate register) sets the output data rate to 100Hz.
	status = i2c_write(0, 0x53, 0x2C, 0x0A);

	// Writing 0x08 (08) to 0x2D (45, power control register)
disables auto-sleep.
	status = i2c_write(0, 0x53, 0x2D, 0x08);

	// Writing 0x08 (08) to 0x31 (49) sets the sensitivity to +/-2g,
with self-test disabled and 4-wire interface.
	status = i2c_write(0, 0x53, 0x31, 0x08);

	std::cout << "Status after setup is " << status << std::endl;
}

// Clean up after ourselves!
ADXL345::~ADXL345() {
	delete[] buffer0;
	delete[] buffer1;
}

// Get x-values
int ADXL345::get_x_value() {
	status = i2c_read(0, 0x53, 0x32, buffer0, 1); // i2c_read(devNum,
devAddr, addr, buffer, numbytes)
	status = i2c_read(0, 0x53, 0x33, buffer1, 1); // 0x32 to 0x37
contain acceleration data

	// Now that we have the data, convert it to 10-bits with some
magic numbers
	int x_accel = (buffer1[0] & 0x03 * 256) + buffer0[0];
	if (x_accel > 511) {
	x_accel -= 1024;
```

```
    }

    return x_accel;
}

// Get y-values
int ADXL345::get_y_value() {
    status = i2c_read(0, 0x53, 0x34, buffer0, 1);
    status = i2c_read(0, 0x53, 0x35, buffer1, 1);

    int y_accel = (buffer1[0] & 0x03 * 256) + buffer0[0];
    if (y_accel > 511) {
    y_accel -= 1024;
    }

    return y_accel;
}

// Get z-values
int ADXL345::get_z_value() {
    status = i2c_read(0, 0x53, 0x36, buffer0, 1);
    status = i2c_read(0, 0x53, 0x37, buffer1, 1);

    int z_accel = (buffer1[0] & 0x03 * 256) + buffer0[0];
    if (z_accel > 511) {
    z_accel -= 1024;
    }

    return z_accel;
}

/***************************** MMA8452Q
*****************************/
/* The MMA8425Q is another 3-axis accelerometer.
 * Ours broke. Also, SPI doesn't seem to work on this thing. Very sad.
:(
 */


// Derived class MMA8425Q is a subclass of the Sensor
class MMA8452Q : public Sensor {
public:
    // Concrete implementation of virtual functions
	int get_x_value();
	int get_y_value();
	int get_z_value();

	// Put accelerometer data into the buffer
	void fetch_new_accel_data();

	// Constructor and destructors
	MMA8452Q();
	~MMA8452Q();
private:
	int status, x_accel, y_accel, z_accel;
	uint8_t *buffer; // buffer contains all accelerometer data
};

// You get an MMA8452Q! You get an MMA8452Q! Everybody gets an
MMA8452Q!
MMA8452Q::MMA8452Q() {
	// Initialize some variables
	status = 0;
	buffer = new uint8_t[7];

	// i2c_write(devNum, devAddr, addr, val, numBytes)
	// The MMA8452Q has an address of 0x1C (28); writing 0x01 to 0x2A
activates the accelerometer.
	status = i2c_write(0, 0x1C, 0x2A, 0x01);

	// Writing 0x00 to 0x0E sets the sensitivity to +/-2g. This could
be changed to 01 or 10 for 4 and 8 respectively.
	status = i2c_write(0, 0x1C, 0x0E, 0x00);

	std::cout << "Status after setup is " << status << std::endl;
}

// Clean up, clean up, everybody clean up
MMA8452Q::~MMA8452Q() {
	delete[] buffer;
}

// Read accelerometer values over I2C and put them into the buffer
void MMA8452Q::fetch_new_accel_data() {
	status = i2c_read(0, 0x1C, 0x00, buffer, 7); // i2c_read(devNum,
devAddr, addr, buffer, numbytes)
```

```
}

// Get x-values
int MMA8452Q::get_x_value() {
     // Now that we have the data, convert it (see datasheet AN4076
for data manipulation details)
     int x_accel = (buffer[1] * 256 + buffer[2]) / 16;
     if (x_accel > 2047) {
     x_accel -= 4096;
     }

     return x_accel;
}

// Get y-values
int MMA8452Q::get_y_value() {
     int y_accel = (buffer[3] * 256 + buffer[4]) / 16;
     if (y_accel > 2047) {
     y_accel -= 4096;
     }

     return y_accel;
}

// Get z-values
int MMA8452Q::get_z_value() {
     int z_accel = (buffer[5] * 256 + buffer[6]) / 16;
     if (z_accel > 2047) {
     z_accel -= 4096;
     }

     return z_accel;
}

/****************************** L3G4200D
******************************/
/* The L3G4200D is a 3-axis gyroscope / angular rate sensor.
 * Ours got into a sad soldering accident and perished bravely on the
front lines.
 */

// Derived class L3G4200D is a subclass of the Sensor
class L3G4200D : public Sensor {
```

```
public:
      // Concrete implementation of virtual functions
      int get_x_value();
      int get_y_value();
      int get_z_value();

      // Constructor and destructors
      L3G4200D();
      ~L3G4200D();
private:
      int status, x_accel, y_accel, z_accel;
      uint8_t *buffer0, *buffer1; // buffer0 contains LSB, buffer1
contains MSB
};

// Set up the gyroscope
L3G4200D::L3G4200D() {
      // Initialize all the variables
      status = 0;
      buffer0 = new uint8_t[2];
      buffer1 = new uint8_t[2];

      // i2c_write(devNum, devAddr, addr, val, numBytes)
      // The L3G4200D has an address of 0x68 (104); writing 0x0F (15)
to 0x20 (32) activates X, Y, and Z axis.
      status = i2c_write(0, 0x68, 0x20, 0x0F);

      // Writing 0x30 (48) to 0x23 (35) sets mode to continuous update,
data lsbat lower address, fsr 2000dps, self-test enabled, and 4-wire
interface
      status = i2c_write(0, 0x168, 0x23, 0x30);
}

// Clean up after ourselves
L3G4200D::~L3G4200D() {
      delete[] buffer0;
      delete[] buffer1;
}

// 0x28 (40) and 0x29 (41) contain all the x-axis data from the gyro,
LSB first.
int L3G4200D::get_x_value() {
      status = i2c_read(0, 0x68, 0x28, buffer0, 2); // i2c_read(devNum,
devAddr, addr, buffer, numbytes)
      status = i2c_read(0, 0x68, 0x29, buffer1, 2);

      // Now that we have the data, convert it
      int x_gyro = (buffer1[0] * 256 + buffer0[0]);
      if (x_gyro > 32767) {
      x_gyro -= 65536;
      }

      return x_gyro;
}

// Now do for y-values from 0x2A and 0x2B
int L3G4200D::get_y_value() {
      status = i2c_read(0, 0x68, 0x2A, buffer0, 2);
      status = i2c_read(0, 0x68, 0x2B, buffer1, 2);

      int y_gyro = (buffer0[2] * 256 + buffer1[0]);
      if (y_gyro > 32767) {
      y_gyro -= 65536;
      }

      return y_gyro;
}

// Now do for z-values from 0x2C and 0x2D
int L3G4200D::get_z_value() {
      status = i2c_read(0, 0x68, 0x2C, buffer0, 2);
      status = i2c_read(0, 0x68, 0x2D, buffer1, 2);

      int z_gyro = (buffer0[1] * 256 + buffer1[0]);
      if (z_gyro > 32767) {
      z_gyro -= 65536;
      }

      return z_gyro;
}


/****************************** Pixel *****************************/
/*  Struct declaration to hold data about how to light the Neopixels
      Typedef for convenience when initializing */
```

```
typedef struct Pixel {
     int brightness;
     int r;
     int g;
     int b;
} Pixel;

/****************************** Logging
******************************/
// We are the lumberjacks

// Histogram container struct
typedef struct Histogram {
     int accel[5];
     int histx[5];
     int histy[5];
     int histz[5];
} Histogram;

// Function to log data in a file
void file_log(Pixel pixel, int status) {
     std::ofstream file; // Make the text file to log everything in
     file.open("log.txt"); // Open the file

     if (file.is_open()) {
     std::time_t time = std::time(NULL); // Current time and date
using ctime

     char* dt = std::ctime(&time); // Convert to string form

     // Put it in the text file
     if (pixel.r > 255 || pixel.g > 255 || pixel.b > 255 || pixel.r <
0 || pixel.g < 0 || pixel.b < 0) {
          file << dt << '\t' << "[ERROR]: The accelerometer values are
incorrect" << std::endl;
     } else {
          file << dt << '\t' << "[DEBUG]: Accelerometer values are
acceptable" << std::endl;

     }

     if (status != 0) {
            file << dt << '\t' << "[ERROR]: Something went wrong while
sending data over I2C" << std::endl;
      } else {
            file << dt << '\t' << "[DEBUG]: No errors while sending data
over I2C" << std::endl;
      }

      file << dt << '\t' << "[INFO]: Intensity of the pixels: " <<
pixel.brightness << std::endl;
      file << dt << '\t' << "[INFO]: Acceleration in the X-Axis: " <<
pixel.r << std::endl;
      file << dt << '\t' << "[INFO]: Acceleration in the Y-Axis: " <<
pixel.g << std::endl;
      file << dt << '\t' << "[INFO]: Acceleration in the Z-Axis: " <<
pixel.b << std::endl;
      }

      file.close();
}

// Function to initialize histogram
void initialize_histogram(Histogram &histogram) {
      for(int x = 0; x < 5; x++) { // Initialize values to 0
      histogram.accel[x] = 0;
      histogram.histx[x] = 0;
      histogram.histy[x] = 0;
      histogram.histz[x] = 0;
      }
}

// Function to put values into their buckets
void input_histogram(Histogram &histogram, Pixel pixel) {
      double accel = sqrt(pow(pixel.r , 2) + pow(pixel.b , 2) +
pow(pixel.g , 2)); // Calculate total acceleration

      if (accel >= 0 && accel < 50) { // Drop some accelerations into
those bins
      histogram.accel[0]++;
      } else if (accel >= 50 && accel < 100) {
      histogram. accel[1]++;
      } else if (accel >= 100 && accel < 150) {
      histogram.accel[2]++;
      } else if (accel >= 150 && accel < 200) {
```

```
      histogram.accel[3]++;
      } else if (accel <= 200) {
      histogram.accel[4]++;
      }

      if (pixel.r >= 0 && pixel.r < 50) { // Drop some Reds into those
bins
      histogram.histx[0]++;
      } else if (pixel.r >= 50 && pixel.r < 100) {
      histogram.histx[1]++;
      } else if (pixel.r >= 100 && pixel.r < 150) {
      histogram.histx[2]++;
      } else if (pixel.r >= 150 && pixel.r < 200) {
      histogram.histx[3]++;
      } else if (pixel.r >= 200 && pixel.r <= 255) {
      histogram.histx[4]++;
      }

      if (pixel.b >= 0 && pixel.b < 50) { // Drop some Blues into those
bins
      histogram.histy[0]++;
      } else if (pixel.b >= 50 && pixel.b < 100) {
      histogram.histy[1]++;
      } else if (pixel.b >= 100 && pixel.b < 150) {
      histogram.histy[2]++;
      } else if (pixel.b >= 150 && pixel.b < 200) {
      histogram.histy[3]++;
      } else if (pixel.b >= 200 && pixel.b <= 255) {
      histogram.histy[4]++;
      }

      if (pixel.g >= 0 && pixel.g < 50) { // Drop some Greens into
those bins
      histogram.histz[0]++;
      } else if (pixel.g >= 50 && pixel.g < 100) {
      histogram.histz[1]++;
      } else if (pixel.g >= 100 && pixel.g < 150) {
      histogram.histz[2]++;
      } else if (pixel.g >= 150 && pixel.g < 200) {
      histogram.histz[3]++;
      } else if (pixel.g >= 200 && pixel.g <= 255) {
      histogram.histz[4]++;
      }
```

```
}

// Function to write histogram to a file
void write_histogram(Histogram histogram) {
      std::ofstream file;
      file.open("histogram.txt");

      if(file.is_open()){
      std::time_t time = std::time(NULL); // Current time and date
using ctime
      char* dt = std::ctime(&time); // Convert to string form
      file << "Generated " << dt << std::endl;

      file << "Value" << '\t' << "X-acc: " << '\t' << "Y-acc" << '\t'
<< "Z-acc" << '\t' << "Total Acceleration" << std::endl;
      file << "0-49" << '\t' << histogram.histx[0] << '\t' <<
histogram.histy[0] << '\t' << histogram.histz[0] << '\t' <<
histogram.accel[0] << std::endl;
      file << "50-99" << '\t' << histogram.histx[1] << '\t' <<
histogram.histy[1] << '\t' << histogram.histz[1] << '\t' <<
histogram.accel[1] << std::endl;
      file << "100-149" << '\t' << histogram.histx[2] << '\t' <<
histogram.histy[2] << '\t' << histogram.histz[2] << '\t' <<
histogram.accel[2] << std::endl;
      file << "150-199" << '\t' << histogram.histx[3] << '\t' <<
histogram.histy[3] << '\t' << histogram.histz[3] << '\t' <<
histogram.accel[3] << std::endl;
      file << "200-255" << '\t' << histogram.histx[4] << '\t' <<
histogram.histy[4] << '\t' << histogram.histz[4] << '\t' <<
histogram.accel[4] << std::endl;
      }

      file.close();
}

/****************************** Main Logic
******************************/
int main(const int argc, const char* const argv[]) {

      Histogram histogram; // Set up the histogram
      initialize_histogram(histogram);

      int status = 0; // Keeps track of whether things worked
```

```
	ADXL345 accelerometer; // Instantiate ADXL345 object (and set up
the accelerometer)

	// Our MMA8425Q broke so we are no longer using this code
	// MMA8452Q accelerometer; // Instantiate MMA8425Q object (and
set up the accelerometer)

	// Our L3G4200D got into an accident involving solder so we are
no longer using this code
	// L3G4200D gyro; // Instantiate L3G4200D object (and set up the
gyroscrope)

	sleep(1); // Sleep for a second to make sure that changes have
propagated

	// Infinite loop to continuously grab data from the accelerometer
and then push the changes to the LEDs
	for (int i = 0; i > -1; i += 0) {
	std::cout << "Looping..." << std::endl;

	// Initialize a pixel with default brightness values and RGB
values based on the X, Y, and Z readings from accelerometer
	Pixel pixel;
	// pixel.brightness = gyro.get_x_value; // The gyro can no longer
be used.
	pixel.brightness = 127;
	pixel.r = accelerometer.get_x_value();
	pixel.g = accelerometer.get_y_value();
	pixel.b = accelerometer.get_z_value();

	// FOR TESTING PURPOSES DO NOT UNCOMMENT UNLESS YOU KNOW WHAT
YOU'RE DOING
	// i2c_writeBytes(NEOPIXEL_I2C_DEVICE_NUM, DEV_ADDRESS,
pixel.brightness, 0x01, 0);
	// i2c_writeBytes(NEOPIXEL_I2C_DEVICE_NUM, DEV_ADDRESS, 255,
0x01, 0);
	// i2c_writeBytes(NEOPIXEL_I2C_DEVICE_NUM, DEV_ADDRESS, 0, 0x01,
0);
	// i2c_writeBytes(NEOPIXEL_I2C_DEVICE_NUM, DEV_ADDRESS, 0, 0x01,
0);
```

```
      /*  Use I2C to communicate with the Arduino dock: change values
with #defines at top of file
            int i2c_writeBytes (int devNum, int devAddr, int addr, int
val, int numBytes)
            devNum is always 0 for Arduino dock
            devAddr for the ATMega is 0x08 by default unless you change
it on the Omega itself
            addr to write to is the value that we want to transmit to
the microcontroller, to be read with Wire library
            val to be written is a dummy value because we specify that
we write 0 bytes of data */
      status = i2c_writeBytes(NEOPIXEL_I2C_DEVICE_NUM, DEV_ADDRESS,
pixel.brightness, 0x01, 0);
      status = i2c_writeBytes(NEOPIXEL_I2C_DEVICE_NUM, DEV_ADDRESS,
pixel.r, 0x01, 0);
      status = i2c_writeBytes(NEOPIXEL_I2C_DEVICE_NUM, DEV_ADDRESS,
pixel.g, 0x01, 0);
      status = i2c_writeBytes(NEOPIXEL_I2C_DEVICE_NUM, DEV_ADDRESS,
pixel.b, 0x01, 0);

      // Print out the data for verification in terminal
      std::cout << "Acceleration in the X-Axis: " << pixel.r <<
std::endl;
      std::cout << "Acceleration in the Y-Axis: " << pixel.g <<
std::endl;
      std::cout << "Acceleration in the Z-Axis: " << pixel.b <<
std::endl;

      // Log ALL THE THINGS
      file_log(pixel, status);
      input_histogram(histogram, pixel); // Put new values into
histogram
      if (!(i % 10)) { // Runs only once every ten seconds to minimize
load on Omega
            write_histogram(histogram);
      }

      // Sleep for a second to make it look nicer
      sleep(1);
      }

      return 0;
}
```

Ardu.ino:

```
#include <Adafruit_NeoPixel.h>
#include <Wire.h>
#ifdef __AVR__
  #include <avr/power.h>
#endif

// Data pins for the eight Neopixel strands
#define PIN_NEO_0 6
#define PIN_NEO_1 7
#define PIN_NEO_2 8
#define PIN_NEO_3 9
#define PIN_NEO_4 10

#define NUMBER_OF_STRANDS 5

// Struct declaration, where member variables change how a pixel is
lit
typedef struct Pixel {
  int r;
  int g;
  int b;
  int brightness;
} Pixel;

// Initialize an array of strands with each strand's data pin set
Adafruit_NeoPixel strip[NUMBER_OF_STRANDS] = {
  Adafruit_NeoPixel(60, PIN_NEO_0, NEO_GRB + NEO_KHZ800),
  Adafruit_NeoPixel(60, PIN_NEO_1, NEO_GRB + NEO_KHZ800),
  Adafruit_NeoPixel(60, PIN_NEO_2, NEO_GRB + NEO_KHZ800),
  Adafruit_NeoPixel(60, PIN_NEO_3, NEO_GRB + NEO_KHZ800),
  Adafruit_NeoPixel(60, PIN_NEO_4, NEO_GRB + NEO_KHZ800),
};

int transmission_number = 0;
Pixel pixel, previous_pixel;

void setup() {
  pinMode(LED_BUILTIN, OUTPUT); // Turn off that blue LED on the
Arduino dock (because it is ugly)
  digitalWrite(LED_BUILTIN, LOW);
  Wire.begin(0x08); // Listen for input on 0x08, the device address of
the Arduino dock
  Wire.onReceive(i2c_on_receive_handler);

  // Loop through all the strands to set them up
  for (int i = 0; i < NUMBER_OF_STRANDS; i++) {
     strip[i].begin(); // Prepares data pin for Neopixel output
     strip[i].show(); // Turns off all Neopixels at the beginning
  }
}

void loop() {
  // Nothing to see here friendos! :)
}

// Fires when data is passed from the Omega to the Arduino dock
void i2c_on_receive_handler(int bytes_num) {

  /* Depending on which transmission over I2C this is, we set
different member variables of the Pixel
   * First, brightness, then R, G, B, and finally, we show the pixels.
   */

  switch(transmission_number) {
     case 0:
     pixel.brightness = Wire.read();
     transmission_number++;
     break;
     case 1:
     pixel.r = Wire.read();
     transmission_number++;
     break;
     case 2:
     pixel.g = Wire.read();
     transmission_number++;
     break;
     case 3:
     pixel.b = Wire.read();
     for (int i = 0; i < NUMBER_OF_STRANDS; i++) { // Loop through
every strand and every pixel to set colour
     for (uint16_t j = 0; j < strip[i].numPixels(); j++) {
          strip[i].setPixelColor(j, pixel.r, pixel.g, pixel.b);
```

```
           strip[i].show();
      }
      }
      transmission_number = 0;
      break;
  }
}
```

Makefile (slightly modified from sample in class):

```
# main compiler
#CC := g++

TARGET1 := Synergy

all: $(TARGET1)

$(TARGET1):
    @echo "Compiling C++ program"
    $(CXX) $(CFLAGS) $(TARGET1).cpp -o $(TARGET1) $(LDFLAGS) $(LIB)

clean:
    @rm -rf $(TARGET1) $(TARGET2)
```

Bibliography

Burgess, P. (2017, October 13). Adafruit Neopixel Uberguide. Retrieved December 1, 2017, from https://cdn-learn.adafruit.com/downloads/pdf/adafruit-neopixel-uberguide.pdf

Onion Omega2 Documentation. (2017, November 13). Retrieved December 04, 2017, from https://docs.onion.io/omega2-docs/