Touch TrackIR

Matthew Kuczynski, Matthew Shen, and Darwin Torres

Department of Electrical and Computer Engineering,
Carnegie Mellon University

***Abstract*—Laptops equipped with touchscreens are starting to flood the market, however they often come at a steeper price tag. To provide the touchscreen experience to someone who lacks such a laptop, we are proposing the design of a frame equipped with LEDs and photodiodes that can be slipped onto the screen of a non touch-compatible laptop to make it touch-compatible. With the frame attached, users will be able to interact with applications by using touch controls, mimicking the experience of a touchscreen laptop.**

***Index Terms*—Arduino, Infrared, LED, Photodiode, Python, Touchscreen, Windows**

I. Introduction

In the modern world, touch screens have become an accessibility feature that consumers expect on nearly all of their new devices. However, laptops seem to be one set of devices where touch screen functionality is considered optional. Therefore, consumers are left with the need to choose between other general laptop specifications and this accessibility feature. We propose to solve these issues by designing a frame that attaches to the screen of a non-touch compatible laptop and makes it work as a touch screen. Our product is called Touch TrackIR.

The ideal consumer for this product is someone who wants to make their current laptop touch screen compatible, but does not want to buy a new device, either for cost or system specification reasons. Additionally, we would like Touch TrackIR to have accessibility features that do not currently exist on the market, which would increase the importance of the product.

One of the competing technologies that exists is normal touch screen laptops like the Microsoft Surface, however the most similar technology is called AirBar. Since the AirBar product is only $80 and is a simple bar that is placed across the bottom of the screen, it is relatively cheap and lightweight. However, our reach goal is to make our product have programmable functionality so the user can customize the product for their own accessibility needs. Furthermore, Touch TrackIR removes the need to buy a new laptop if a consumer wants touch screen functionality, and it creates a cheap accessory instead.

The goal of Touch TrackIR is to take non-touch compatible laptops of a specific size and make them touch screen devices. Ideally, the product will be able to work in both indoor and outdoor environments, and the frame will be secure enough to allow movement of the laptop. Like any touch screen device, we want Touch TrackIR to respond accurately and precisely to any finger touch, and it should respond in a timely manner. In addition to single finger tap responses, we want our product to respond to scroll functions that users are used to on their other devices. Similar to AirBar, we want our product to remain both cheap and lightweight, however as mentioned before, we ideally want Touch TrackIR to be able to have programmable functionality.

II. Use-Case Requirements

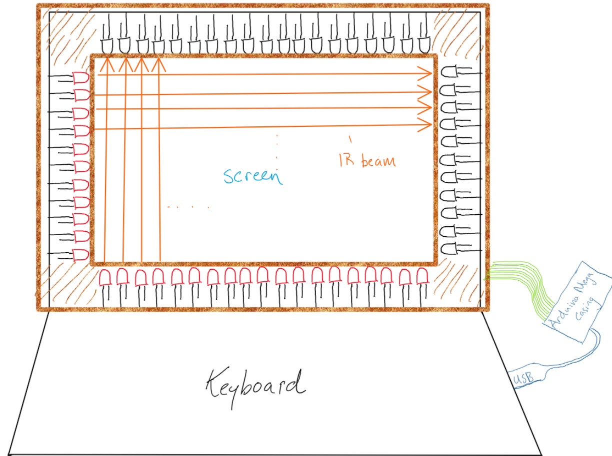
To ensure an enjoyable user experience comparable to that of a regular touchscreen laptop, we have decided on a set of quantitative requirements for the use-cases tackled by our product. For touch-precision, we want the distance between the physical point of contact with the screen (i.e. middle of finger) and the position registered by the OS to be within 0.2 inches of each other. This margin of error lies within the width of the average finger tip, and will guarantee that the registered point of contact lies directly underneath the user's finger.

For the false-positive rate, we want to ensure that at most 1 ghost touch is detected for every 5 minutes the user is idle, while for false-negative touches we are aiming for a maximum rate of 5% - that is for every 100 touches, at most 5 are not detected. Ideally, these rates would be 0, however in the early stages of our design, false-positives and false-negatives will be encountered frequently. The numbers we provide are low enough such that there will be minimal interference with the behavior that the user is expecting in response to their touches.

When it comes to the response time and refresh rate of our system, we want to make sure that our device is fast enough so that interactions do not feel "choppy" due to perceivable lag. A response time that is too low would result in large delays between a user's touch and an application updating in response to it. A refresh rate that is too low would make dragged touches less continuous, resulting in them mimicking a series of individual taps. In general, it would mean less taps can be detected per second, giving rise to a potentially higher number of false-negatives. To provide a good experience, we chose target requirements of 150 ms for response time, which is comparable to early tablets, and 15 Hz for refresh rate, which is about the bare minimum needed before continuous position updates, like when a finger is being dragged, become too slow for comfort.

Finally, our requirement for the final use-case, the weight of the frame, is a maximum of 0.5 pounds. This is the maximum amount of force the screen of our test laptop can withstand before it starts to rotate backward on its hinge. We want our frame to be lighter than half a pound so it does not tilt the screen when it is attached, especially when the user applies a force with their finger. Anything higher than that, and the hinge of the laptop experience high levels of torque that result in an unstable screen.

III. Architecture and/or Principle of Operation



Our system architecture consists of 3 main blocks:

- The Frame
- The Arduino Mega
- The Laptop

The Frame:

There are two main parts of the frame: an IR LED section and an IR Photodiode section.

For the IR LED half, the general principle of operation is that the Arduino will send select signals to decoders. These decoders will then output a high signal to some subset of MOSFETs. This signal will turn on the MOSFETs, thus allowing current to flow through a subset of IR LEDs, illuminating them.

For the IR Photodiode half, they utilize the same select signals from the Arduino as the LED array. These select signals control multiplexers which serve to read only from the Photodiodes that are positioned directly across from the illuminated LEDs.

The Arduino Mega:

The Arduino is effectively the messenger of our architecture. It collects data from the frame and passes it to the screen control software running on the laptop.

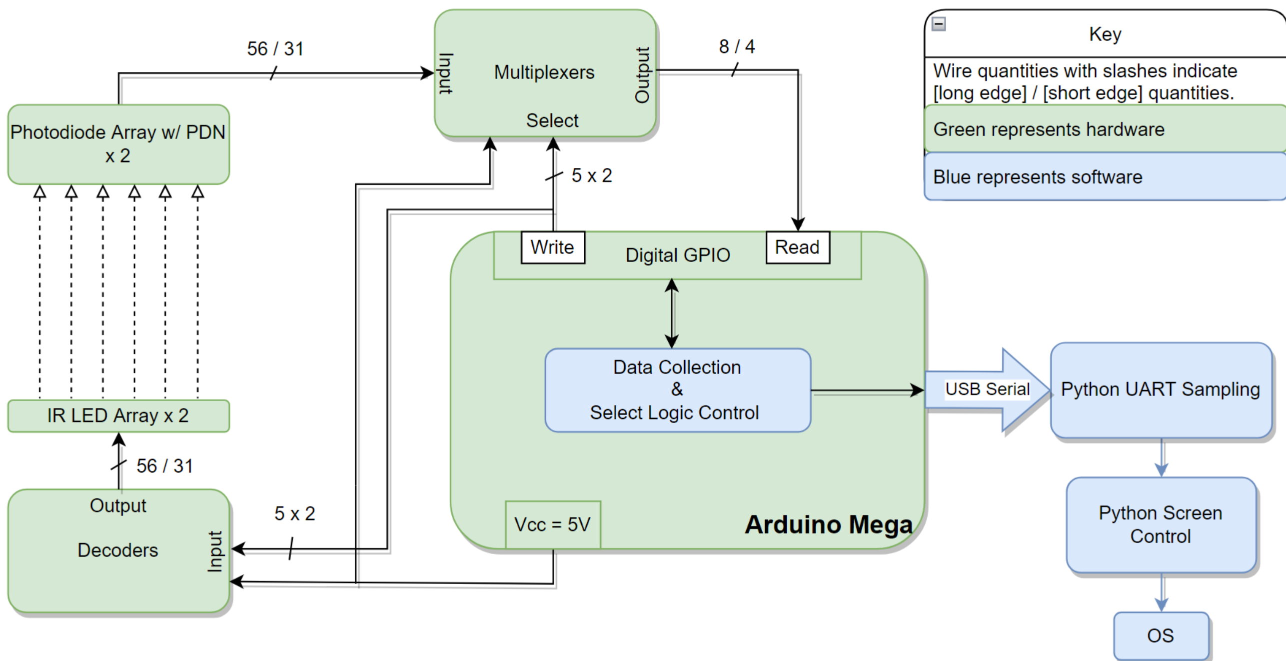
As mentioned above, the Arduino is responsible for sending digital control signals to the hardware in the frame. It is also responsible for reading the outputs of the multiplexers mentioned above. Additionally, the Arduino lends its 5V supply and ground to the frame.

On the other end of its operation, after a single sweep through the arrays of LEDs and photodiodes, the Arduino will report this data over a USB-serial interface for translation to screen-control.

The Laptop:

The laptop will be running Python to listen on a COM port for the serial data being sent by the Arduino. Using this collected data, Python will calculate the position of a finger and pass this data off to a screen-control library.

Since the design report, no changes were made to the architecture and/or principal of operation.



IV. Design Requirements

Our most important design decisions are split into two main categories - frame structure and software processing, with the exception of one design requirement related to power. Starting with those related to frame structure, touch-precision is dependent on how many LEDs we can pack on each axis - the more LEDs, the higher the resolution of our grid. To meet our

use-case requirement of a maximum error of 0.3 inches, we have set a design requirement of 30 LED/photodiode pairs on the vertical axis of the screen and 56 LED/photodiode pairs on the horizontal axis. This configuration results in 5.5 mm, or 0.22 inches, of spacing between the centers of each LED/photodiode. If the point of contact along an axis is calculated as the average position of activated photodiodes, we can then be precise up to 0.22/2 = 0.11 inches, as we can now register touches that occur halfway between LEDs. This is far less than our 0.3 inch goal. Moving on to false positive and false negative rates, these rely on the reliability and responsiveness of our sensors. To ensure we always get the most accurate readings, we want our frame to be strongly secured to minimize any variation that occurs in the alignment of LEDs and photodiodes due to the movement of the laptop. As a design requirement, as we handle the laptop, changing its position and orientation, we do not want deviations in values measured to exceed 0.1% error.

The remaining use-case requirements, response time and refresh rate, are mainly affected by the software processing design decisions. We found that our ability to meet these requirements depends on the speed of our code, so it is important that we choose a good language, use the right libraries, and maintain a high standard of quality when optimizing. Based on this, we have set the following design requirements: LED control and data collection will be written with Arduino code, while finger triangulation and screen control will be done in Python. To ensure the best performance, we will send touch control requests to the OS via python-compatible calls to the Win32 C++ API. Collectively, these all contribute to the response time and refresh rate requirements, so we can quantitatively measure their effects through our response time and refresh rate measurements.

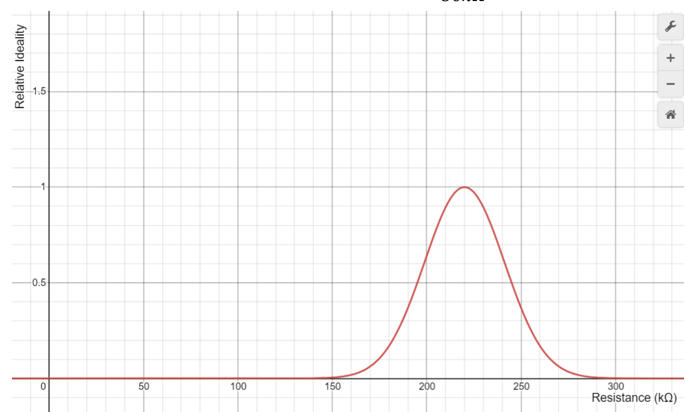
Our final design requirement focuses on the power consumption of our components. We want a power source capable of delivering 5 Volts to satisfy the needs of our hardware. Ideally, the Arduino's 5V pin should be able to get the job done.

V. Design Trade Studies

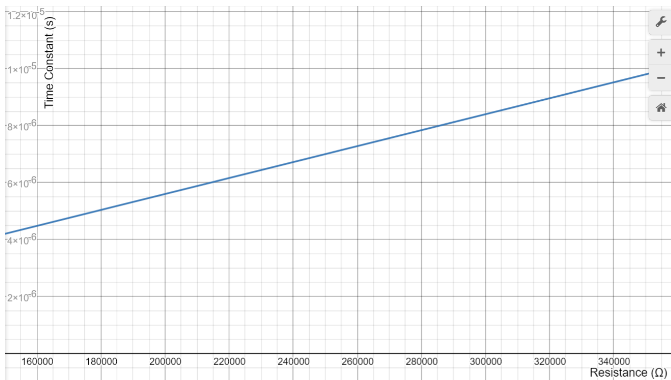
A. Number of LEDs Illuminated vs. Sensor Sensitivity

One of the major trade-offs in our design is determining a good balance between speed and sensitivity to light. The speed of the design is governed by a time constant τ=RC, where R is the resistance tied to the anode of a photodiode, and C is the sum of the input capacitance to the mux and the effective capacitance of the photodiode. As for sensitivity, there is a less well-defined governing equation, but two major aspects are ensuring the photodiodes are sensitive enough to detect light from a single LED operating at 100mA from 13 inches away (the approximate distance between the left and right edges). However, since we are turning multiple LEDs on at a time (separated by about 14 LED widths), we need to ensure that the circuit isn't too sensitive, since we only want the photodiodes to respond to the LED that is directly across from it (we will refer to an LED 14 widths down as a "neighboring LED"). Currently, we are using a 200kΩ pull-down resistor for each photodiode. In our tests, this resistor makes the photodiode sensitive enough such that when exposed to an LED 13 inches away, the worst-case voltage at the anode was measured to be 3.2V. This comfortably exceeds the value for VIH of the mux input (2.0V). However, this was a noticeable drop from a worst-case of 4.0V when using a 300kΩ resistor, but the 300kΩ resistor proved too sensitive when tested against a neighboring LED. Since the mux has a VIL of .7V, we want to ensure that neighboring LED exposure doesn't push the pull-down resistor's voltage above this. With the 300kΩ resistor, a single neighboring LED was enough to induce a voltage of .6V, which is dangerously close to VIL. Additionally, since the bottom frame runs 4 LEDs at a time, it is possible for multiple LEDs to further increase this voltage. So, we opeter for a smaller resistor. The 200kΩ experienced a mere .1V increase which we deemed safe. By our estimations, we estimate that the ideal resistance to balance this sensitivity trade-off follows a normal distribution. Given that the 200kΩ resistor overperforms regarding the neighboring LED test, we presume that something a little larger than 200kΩ is the ideal value. Additionally, given that the 300kΩ resistor nearly failed the neighboring LED test, we adjusted our standard deviation accordingly.

$$optimal\ resistance\ est. = \ exp(-\ (\frac{x-220k\Omega}{30k\Omega})^2)$$



It is also important to keep the time constant in mind, although it is likely that the serial communication will dominate the time per sweep. A graph of the time constant has been plotted below using a capacitance of 28pF (estimated from datasheets).



B. Layout Area vs. Hardware Complexity

Our design was massively limited by the narrow dimensions of the PCB. Especially on the LED arrays, which needed several thicker high-current traces, routing needed to be manually done, as the autorouter was never able to complete a job on its own. Thus, in order to fit all the necessary components onto the boards, considerations were taken into how we could reduce the footprint. For instance, in order to remove the need for 8 digital traces to be run back to the Arduino from the top edge and through the right edge, we decided to add another multiplexer to the top edge. For this, we would need 3 more control signals, but also 7 fewer output signals.

Another tradeoff involving area had to do with the number of LEDs illuminated in parallel. Since our decoders couldn't directly power the LEDs, we needed an additional stage of power NMOS transistors to act as switches. To avoid the need for 87 of these FETs, we decided that we should power several at once. This didn't come without its own set of drawbacks, however. Since our LEDs project IR at some nonzero angle off the ideal beam, we needed to ensure that a negligible amount of light from one LED would feed into the photodiodes across from any of the other illuminated LEDs.

To tune the sensitivity of our IR sensor arrays, we would have to modify the values of the pull-down resistors. This required a lot of trial-and-error before we finally arrived at a good balance of sensitivity from the intended LED, while rejecting light from other LEDs.

C. Programming Language vs. Speed

Another tradeoff that we had to consider was the programming language semantics and writability versus the speed of the language. A language like Python is known for being slower than lower-level languages, but it is much easier to write and has better APIs. In the end, we decided to use Python for the majority of our processing, and we would keep other languages in mind as part of a risk mitigation plan. As you can see in the testing results later, Python proved to be sufficiently fast for our use-case requirements.

D. Number of Serial Messages vs. Processing Speed

Another tradeoff that we chose to study was the number of serial messages we should send in each collection frame versus the combined Arduino and Python processing speed. Based on the processing that was done in the Arduino code, the data that needed to be sent over serial changed. Through initial testing, we noticed that serial communication would likely be a bottleneck in our processing speed, so we decided the minimize the total number of serial messages sent. In the end, we found that sending one value for protocol purposes followed by 3 encoded 32-bit integers that represent the 87 photodiode values would be best. This also allowed for calculations like the pixel location to be done in the Python code, which made our code much simpler overall.

E. Refresh Rate vs. Sensitivity

One design tradeoff that we are unaware of until we start testing was the sensitivity of finger movement versus the refresh rate. We found that our system had trouble distinguishing many simple taps, and it was instead interpreting them as small drags. This hurt the overall performance of our system because it made it quite difficult to perform a simple task like clicking on a new tab in a browser. We found that this could be fixed by creating a drag threshold distance that the finger needed to move in order for the touch subsystem to update the finger position. Additionally, we discovered that averaging multiple frames together and sending fewer updates helped this issue as well. As you can see in the system implementation, we made two settings for this that could be controlled at the command line so that the user could have the device fit their use-case as perfectly as possible.

VI. System Implementation

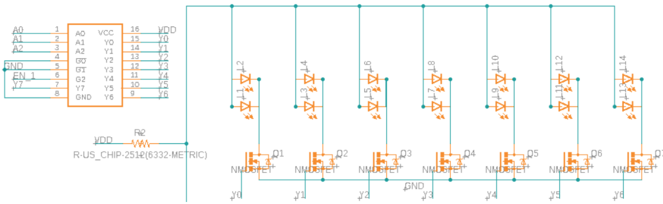
A. Subsystem A – Hardware

For the hardware behind our project, there is a frame consisting of a PCB and either wooden or plastic casing. We also use the Arduino Mega to act as the brain for the frame.

The frame consists of a PCB on each edge of the screen. The aim is to connect the four PCBs by soldering them together with header pins at the corners of the frame. The wires from the Arduino will be fed into the bottom-right of the frame where they will attach to header pins on the PCB. Signals that the left-edge PCB needs are routed through the bottom-edge PCB. Similarly, signals that the top-edge PCB needs are routed through the right edge PCB. Thus, the header pins that are used to fasten the edges together also serve a dual purpose of electrically connecting everything in the frame. We also intend to use the header pins to attach the casing.

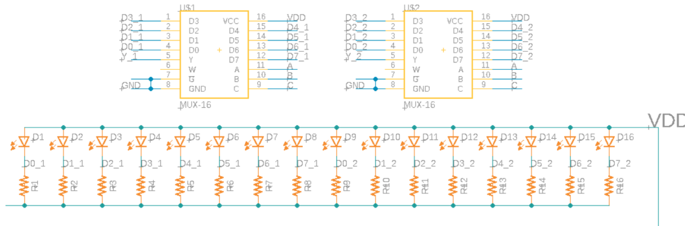
Regarding the actual functionality of the frame, there are two main components as mentioned earlier in Section III. One component consists of two LED arrays, one along the left-edge of the screen and the other along the bottom-edge. The other is

the pair of photodiode arrays, positioned along the top and right edges of the frame. Only two edges will be "active" at a given time: either the top and bottom, or the left and right. The operation of each pair is for the most part the same, with some minor differences. The left and right edges have arrays of 31 LEDs and photodiodes respectively, while the top and bottom edges have arrays of 56 of each. Below is a small subset of the left-side LED array.



The box on the left is a decoder. It receives control signals from the Arduino and feeds a high signal to one of the MOSFETs shown. In this case, a pair of LEDs is illuminated (the bottom LED array illuminates 4 consecutively). Also note the current-limiting resistor shared by the anodes of the LEDs. The main reason for having LEDs share a MOSFET is to reduce the number of components on-board.

Also see below a subset of the photodiode array:



Firstly, it is important to note that the diodes above should have their terminals reversed. The image above is from Fusion 360 PCB software, and we will just have to ensure that we solder the LEDs in the proper direction when the time comes. Each node atop a pull-down resistor is fed into a mux input, and the Arduino is responsible for instructing the mux which node to read. For the top edge, we have an additional mux whose inputs are outputs from 8 different muxes, which effectively creates a 64:1 mux, plenty for our array of 56. The reason for adding this mux was to reduce the number of wire traces. Originally, 8 wires needed to be sent back to the Arduino to be read. With the added mux, 3 more control signals are required but with only 1 wire to be read, thus reducing the number of wires by 4.

For the last piece of hardware, our Arduino Mega simply uses a USB cable to connect to the computer. As a result, it is effectively the computer that is responsible for powering the frame.

B. Subsystem B – Python-Arduino Software Interface

Subsystem B, which is the Python-Arduino interface, connects the information recorded in subsystem A to the screen control logic in subsystem C. This subsystem covers several important components that will be explained in this section, including the select logic control, data collection, serial communication to Python, and finger detection algorithms.

Since only a small subset of the LEDs and photodiodes will be in use at any given time, the select logic control is used to determine which LEDs are turned on and which photodiodes are being read from. This logic is written in the Arduino language and is uploaded directly on the Arduino board, which we believe is the quickest way for this to be performed. The basic idea is that we will loop over all of the LEDs and photodiodes by setting the Arduino pins that control the multiplexers and decoders. There are 13 select logic pins in total, with there being four enable pins (one for each of the two decoders on the left and bottom edges), three select pins shared by the decoders and multiplexers on the top and bottom edges, three select pins shared by the decoders and multiplexers on the left and right edges, and an additional three select pins going to the additional multiplexer on the top that is used to clean up the output logic. These pins are all set to OUTPUT mode in the Arduino code, and they are looped over to ensure that all LEDs are turned on in a cycle. Then, for all LEDs that are turned on at once, the code takes turns reading from the opposing photodiodes.

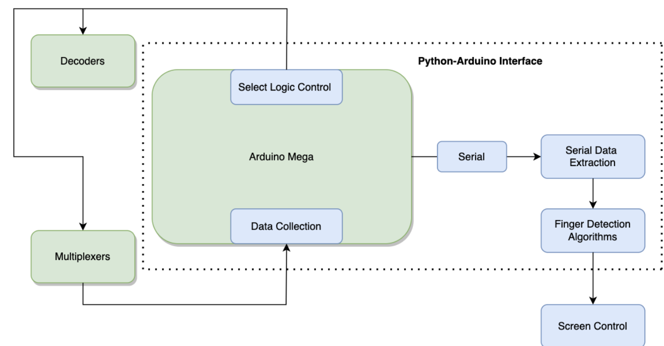


Fig. 1. Block diagram of Python-Arduino interface.

The data collection section of this subsystem requires 5 pins to be read, which correspond to the photodiode values. The right edge has four pins to be read, while the top only has one due to the extra multiplexer. Unlike for the select logic, these pins are set to INPUT mode. The photodiodes for which these pins correspond to change with the select logic, and all of the photodiodes are looped over. To our surprise, the data could be read accurately without any delays in the Arduino code.

While the data collection aspect is relatively simple, it ties in closely to the serial communication component of the subsystem. We have determined that the fastest way to communicate the data from the Arduino to a Python script is by using serial prints on the Arduino and the Python serial library. As mentioned in the design trade studies, we send four integer values over serial for each collection frame. The first is the value -1, which is used for protocol purposes and is expected by the Python code at the beginning of a data sequence for each collection frame. The second value is half of the 56 photodiode values on the top array encoded into the lower 28 bits of a 32-bit integer, and the third value the other

half encoded the same way. The fourth integer is the 31 photodiode values from the right array encoded into the lower 31 bits. Since the most significant bit is always 0 for these data-encoded integers, -1 works as a good value for the first value in the protocol. In the end, we were able to achieve a maximum baud rate of 115200.

Next, the Python code receives the integers and converts them into bit arrays. First, it checks that the protocol mentioned before has been satisfied by looking for a -1 followed by 3 positive integers. Next, it checks if the data is valid since there is occasionally information lost over serial communication in general. The data is then converted into bit arrays, and then it checks whether there is valid touch, which occurs when there are photodiodes covered on both dimensions.

Calculating the pixel location of the finger is done by taking the average position of the LEDs covered. Using the screen size in pixels and number of LEDs, we can determine which pixel locations correspond to which LEDs. Keep in mind that this device only works for single finger touch commands, which is a limitation of the system.

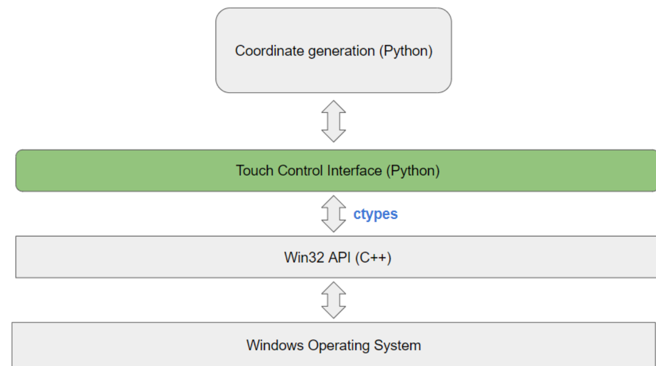
When the finger first touches the screen, a down command, which is mentioned in the next subsystem, is called. Subsequent touches following a down command will trigger an update command to be called. Another point that was mentioned in the design trade studies is that there is a drag threshold that must be met in order for an update command to be called. Therefore, the code checks the finger position against the position from the initial down command to decide whether or not to send an update. This drag threshold value can be set at the command line.

Another value that can be set at the command line is the number of frames to average for smoothing. By averaging more frames, we send update commands less frequently, which makes actions like drawing much smoother. To do this, we simply average the pixel location over the specified number of frames before sending any command.

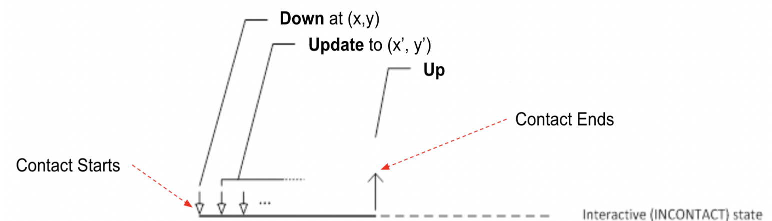
One component that failed in this subsystem was the detection of multiple fingers. This is because it was very difficult to pair covered pixels on the horizontal dimension with those on the vertical dimension. For example, if two fingers were down, we found two clusters of covered LEDs on the horizontal dimension and two on the vertical dimension, but we did not know which two clusters belonged together, and therefore we could not determine the positions of the two fingers.

C. Subsystem C – Screen Control

Subsystem C is the software interface that is responsible for using the coordinates detected by the hardware to tell the Operating System where to emulate touch inputs. In order to perform this task, we must communicate with the Win32 API provided by Microsoft for Windows application developers. This API provides a set of functions for injecting touch inputs through software, however our code is written in Python, while the library's functions are written in C++. Luckily, Python has a foreign function library called ctypes that allows Python code to make calls to the C++ functions in the Win32 .dll libraries.



Microsoft's Touch Injection API supports 2 states, the hover state and the interactive state. In the hover state, a touch input is in close range with no physical contact being made, and applications cannot be interacted with. This is mainly meant for Windows-specific electronic styluses, which do not apply to our use case. Thus, we focus only on the interactive state, in which physical contact is being made and applications can be interacted with. With our touch control interface, we hide the existence of the hover state, allowing us to simplify the commands we send to the OS. It should also be noted that the Win32 API also allows for a contact area to be specified. This means instead of contact being made at an exact point in space, it is spread across a group of pixels.



The three main commands supported by our interface are **touchDown, touchUpdate,** and **touchUp**. **touchDown** tells the OS that contact has been made at a specified pair of coordinates, and generates a new touch instance. A touch instance describes a continuous contact event, and lives as long as contact is being made with the screen. **touchUp** terminates a specified touch instance, signifying to the OS that contact is no longer being made. **touchUpdate** moves the point of contact of a specified touch instance to a new set of coordinates.

The interface's main job is to relay information about the location of the point of contact and how it moves across the screen. It is up to each Windows application to determine what happens in response to each sequence of touch events. In a web browser, a drag across the screen may correspond to scrolling through a webpage, while in an application like Microsoft Paint, it may correspond to drawing a line on a canvas. Examples of Windows gestures that many applications take advantage of are taps, holds, and drags. Taps are generally

recognized when touch contact starts and then immediately ends without a change in coordinates. To emulate a tap with our interface, a **touchDown** can be sent followed by a **touchUp** within a short time frame. Given our high refresh rate, we may send multiple **touchUpdates** in between due to the detection of continued contact across multiple frames, but all that matters to the application is that the finger stays on the same coordinates for a very short time. To emulate a hold, a **touchDown** is sent followed by a series of **touchUpdates** at the same coordinates for a prolonged period of time. A drag is similar to a hold, except that subsequent **touchUpdates** result in a change of coordinates. With the support of these principal gestures, a user can navigate through most touch-compatible applications with ease.

VII. Test, Verification and Validation

We have outlined a specific test for each of the use-case and design requirements that were mentioned in the earlier sections. Almost all of the tests are quantitative, with a couple being qualitative. Tests A-F are for the use-case requirements, while tests G, H, and I are for the design requirements.

A. Results for Touch Precision

To test the touch precision, we measured the distance between the center of the finger and the position of the click. One method for doing this initially was to create really small boxes in Microsoft Excel, touch them, and see which boxes were actually clicked on. Our goal was to have a touch precision that was less than 0.3 inches, and we ended up achieving a touch precision around 0.13 inches. This makes perfect sense because the distance between two LEDs is ¼ of an inch, so ⅛ of an inch would be the position found if two neighboring LEDs were averaged together. Therefore, this test passed.

B. Results for False Positive Rate

Our false positive rate was tested by leaving the laptop unattended for 1 hour with the script running and checking how many touches were detected. Our false positive rate goal was an average of 1 false positive per 5 minutes, or less than 12 total false positives for the hour. When running the test, we received 0 false positives, which means we had a false positive rate of 0%. This likely happened because the environment of the system did not change at all for that hour, however we also have never noticed any false positives when testing. In the end, we consider this a successful test.

C. Results for False Negative Rate

For the false negative rate, we touched the screen 100 times and counted the number of touches that were not recognized by the system. To provide diversity to the touches, we touched all different areas of the screen. We found that all 100 touches were detected by the system, which gave us a false negative rate of 0%. Since our original goal for the false negative rate was 5%, this test passed.

D. Results for Response Time

In order to test the response rate, we took a slow motion video of a finger tapping an online stopwatch, and we looked at the difference in time between when the finger touched the screen to when the stopwatch stopped. Our goal for this test was 150 ms based on the performance of other touch screen devices. Repeating this test a couple of times, we found that the mean response time was around 120 ms, with very little variation. Since our system performed better than our goal, we consider this test to be passed successfully.

E. Results for Refresh Rate

To test the refresh rate, we removed all smoothing algorithms, and we calculated the time that it took to run a collection frame using software. Our goal for this test was 15 Hz, and we were able to measure a refresh rate of 66 Hz for our system. As mentioned previously, the system is not very user friendly when all smoothing algorithms are disabled, so the performance is better with a lower refresh rate in practice. However, since our system was able to pass our goal, the test was successful.

F. Results for Frame Weight

In order to measure the weight of our frame, we simply placed the frame on a scale. We set our goal for this test at ½ lb since we determined that this was the weight that we could place at the end of the laptop screen and have the screen remain open. In practice, the weight is distributed across the screen, so we could have had a heavier frame. However, we measured a weight of 0.4 lb, so we met our goal.

G. Results for Stationary Frame

To test for a stationary frame, we decided to shake the laptop with the frame on for 15 seconds and recalculate the false positive and false negative rates. Our intuition here was that the light may not be properly detected on all of the photodiodes if any of the components move while the frame is shaking. We performed the false positive and false negative rate tests again after shaking and saw no change in error. We set a goal of <0.1% error change on both tests, so the test passed.

H. Results for Power Source

The power source test was run by simply using a multimeter to check the voltage that came out of the Arduino. We measured a voltage of 4.73V, despite 5V being expected.. Since the system still worked properly, we found the results of this test to be insignificant.

I. Results for Software Processing

For our software processing to be sufficient, we wanted to ensure that our combination of languages, libraries, and algorithms were efficient enough to have our response time and refresh rate tests passed. Since both tests passed, we consider our software processing to have met its goal.

VIII. Project Management

A. Schedule

For the most part, our schedule didn't change. There were some tasks that took longer than planned and some that took shorter than expected, but the end result was as expected. For instance, the hardware debugging took a few days less than expected, but some of our software algorithms took more time to iron out than expected.

Some of our less important tasks were dropped to focus on improving the actual functionality of our product, such as making the hardware look nicer.

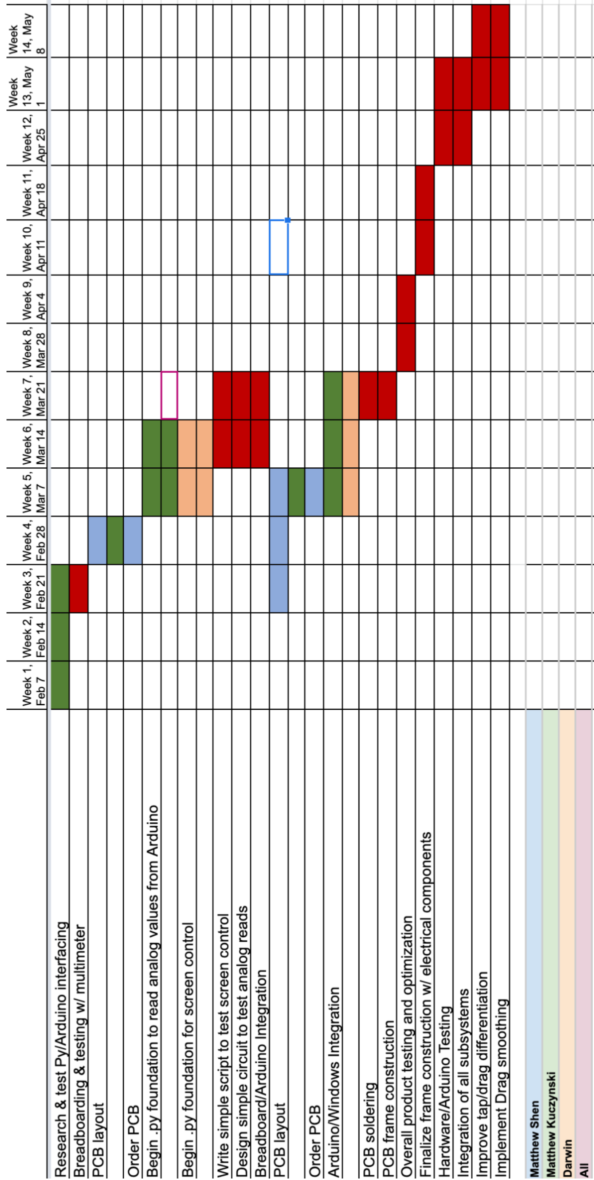


Fig. 2. Gantt chart for all weeks

B. Team Member Responsibilities

Each team member has a different subsection of the block diagram for which they are primarily responsible, however all of the components work together, so there is significant overlap. Matthew S.'s main focus is the design of the hardware side of the project, which involves determining which components to use, how they will be connected, and also performing the majority of the PCB design. Matthew K. is primarily responsible for the software interface with the Arduino, which involves writing the Arduino code for performing data collection, controlling the select logic, and sending data over the serial interface to Python. This also involves using the data sent to the Python sampling code to determine the finger position(s) and quantity. Darwin is primarily responsible for the Python screen control backend, which involves interpreting the finger position as a command that is sent to the OS, as well as allowing for programmable multi-finger functionalities.

The secondary responsibilities for all of the team members are the same, and they include frame design and construction, soldering, and integration of the components that were worked on individually by each team member. More specific information about the secondary responsibilities can be found in the Gantt chart.

Since the design review, the responsibilities of the team members have not changed

C. Bill of Materials and Budget

See page 9 for Bill of Materials.

D. Risk Management

For each of our tests, we developed a risk mitigation plan in case the assessment fails. If touch precision was not high enough, we would have tried reading multiple photodiodes for each of the LEDs to extract more information about the finger location. For the false positive test, we planned to use software to filter out the touches that do not span across multiple photodiodes since we know that a normal finger will cover multiple. If our false negative rate was too high, response time was too long, or refresh rate was too low, we would improve the software quality by switching from Python to C++. If the stationary frame test failed, then we would add a mechanism to tighten the frame to the screen. Since we are dependent on the Arduino and a 5V power source based on our design, our only possible risk mitigation plan for the power source test failing was to have a backup Arduino Mega. As mentioned above, issues with the software processing tests could have been fixed by moving from Python to a faster language.

As we worked on our project, we discovered that many of the risks we faced had to do with our PCB potentially not working. We also did not have too many of the hardware components that needed to be soldered, so we needed the ones that we did have to work properly. In the end, we were fortunate enough to have everything work, but that did not come without encountering any problems along the way.

When we soldered the components to one of the boards, we realized too late that there was a misprint in the board. Therefore, we had to develop a mitigation plan on the spot, which was to reconnect the board properly using wires. Another option would have been to desolder the components, however that would have been extremely time consuming

given the time it took the solder the components onto the bad board in the first place. We were fortunate enough to not have problems large enough to make the PCB completely unusable, but we certainly had to come up with risk mitigation plans on the spot.

Once everything was integrated, we were able to successfully pass all of our tests, so we did not need to implement any of the risk mitigation plans mentioned previously. In an absolute worst case scenario where the frame did not work, we had mentioned potentially creating the necessary circuit with a breadboard and having a touch pad on the side of the laptop. Similarly, if the frame did not fit onto the screen, we would have kept the frame to the side of the laptop and used that as a touch pad.

IX. Ethical Issues

Touchscreens on laptops can provide a more user-friendly experience to certain individuals who have impaired motor control and struggle to use a mouse to interact with the computer. For example, individuals suffering from Parkinson's Disease often prefer touch screens as computer mice can be very sensitive to fine movements. In this case, such users are extremely vulnerable to even a minor hardware/software failure as malfunctioning touch controls might limit their ability to use the laptop. In these situations, it is important that the quality of the product meets the needs of such individuals, otherwise it will severely limit their ability to communicate with others given that they have a very limited set of options due to their condition. Quality assurance is an important step in the development and manufacturing process, and it is crucial that no shortcuts are taken just for the sake of convenience, otherwise it can negatively affect the end user in ways that do not seem obvious at first.

A more general situation that can affect anyone who uses the product is being able to obtain the ability to track or control a user's inputs. As designed, Touch TrackIR does not collect and store data about the user, only using any data received about finger positioning and movement to instantly construct commands to send to the OS. If someone is able to successfully modify the software or hardware, they could, in theory, intercept the touch commands sent by Touch TrackIR, and possibly even send commands of their own. This would not only be a major invasion of privacy as it can expose information about the user to the attacker, but it would also allow the attacker to take full control of a user's laptop to perform malicious actions. An example of a common attack that can be deployed through such means is ransomware, in which the files on the victim's machine are encrypted and the only means of getting them back is to pay a ransom to the attacker, and even then there is no guarantee that they will comply. To prevent such an incident from happening, it is imperative that shipped software is equipped with anti-adversarial measures that prevent the exposure of sensitive information. Many programming languages today are designed with security in mind, but it is also the programmer's responsibility to be knowledgeable about what is offered by the development tools they use in terms of security.

X. Related Work

A similar project that provided inspiration during the early stages of the project was an approach to providing touch-screen compatibility using light-triangulation. This project also used an array of LEDs, but instead utilized a CIS scanner from a printer instead of a self-built array of photodiodes. While his idea uses far fewer LEDs than our design, it is compensated by the insanely high resolution and accuracy of a CIS scanner. This high resolution and accuracy enables him to triangulate light from the LEDs to calculate the position of a finger. However, the low resolution of LEDs meant that touches made close to the LEDs experienced drop-offs in accuracy. For our approach, we wanted equal effectiveness across the board, so we opted for a dense array of diodes. Additionally, we decided to design our own sensor array since a CIS sensor would be far to bulky to put on a laptop frame.

Another similar product is Airbar, which somehow uses a single bar along the base of the screen. Initial research for our product aimed at a similar approach. Unfortunately, we couldn't find distance sensors that were small enough to achieve this. After hours of research, the only option that remained would be to buy an Airbar to reverse engineer. We decided this would not be an effective use of our budget.

XI. Summary

Touch TrackIR is an attachable frame intended to transform the screens of non-touch-compatible laptops into touchscreens. This is done via an array of Infrared LEDs and photodiodes that detect the presence of a finger on the screen through sweeps of LED/photodiode pair activations. To make sure the user has the best experience, we aim to have precise contact detection, with errors less than 0.3 inches, low response times of under 150ms, and a refresh rate of at least 15Hz, providing smooth screen updates and an imperceivable delay. In addition, we want to ensure that every touch counts, with responsive sensors that provide near-zero false-positive and false-negative rates, removing any hiccups that may occur due to unexpected behavior. By providing a sturdy and light frame, we will also make the system as non-invasive as possible, allowing for more freedom in handling and transporting the device without having to worry about the hinges being under too much stress or changes in sensor accuracy due to movement.

Overall, we feel that the project was a success because all of the design specifications were met, and the project works quite well. The main limiting factor for our system is the touch precision since this depends on the spacing between the LEDs, which was minimized. Additionally, if we were given more time, we would try reading from multiple photodiodes per LED in order to get a better estimate of the finger location. The only thing that we could not achieve was multi-finger functionality, however this would have been very difficult to

complete successfully given our system implementation.

A. Lessons Learned

Overall, this project taught us a lot about the integration of separate subsystems. Although we were not able to fully test our project until the final week, we were confident that it would work because we had already tested individual and neighboring subsystems. As mentioned before, we would definitely try reading from multiple photodiodes per LED if we were to try a project using this technology again. While we were initially not sure how well the system would respond to different environments and user testing, we were pleased to discover that our LED-photodiode array approach worked extremely well. If another group were to try a project using this technology, we would definitely encourage it, and we would suggest that they perform additional software processing.

Glossary of Acronyms

API - Application Programming Interface
CIS - Contact Image Sensor
GPIO - General Purpose Input/Output
IR - Infrared
LED - Light Emitting Diode
OS - Operating System
PCB - Printed Circuit Board
PDN - Pull-Down Network
USB - Universal Serial Bus

References

[1] "AirBar." *Neonode Inc.*, https://neonode.com/airbar
[2] Perardel, Jean. "Magic Frame : Turn Everything into a Touch Area." *Hackaday.io*, 5 Sept. 2017, https://hackaday.io/project/27155-magic-frame-turn-everything-into-a-touch-area.
[3] "Photodiode Basics." *Wavelength Electronics*, Wavelength Electronics, 11 Feb. 2020, https://www.teamwavelength.com/photodiode-basics/.
[4] Nasir, Syed. "Introduction to LM317." *The Engineering Projects*, 2 July 2020, https://www.theengineeringprojects.com/2017/06/introduction-to-lm317.html.
[5] Shawn. "Types of Distance Sensors and How to Select One?" *Latest Open Tech From Seeed*, 29 June 2021, https://www.seeedstudio.com/blog/2019/12/23/distance-sensors-types-and-selection-guide/.

Item	Quantitiy	Price Per ($)	Total Price ($)	# used in design	Part #	
Arduino Mega	2	18	36	1	varies	
IR LED	100	0.3272	32.72	87	SFH 4555	
IR Photodiodes	100	0.1729	17.29	87	INL-5ANPD80	
210k Resistors	100	0.0184	1.84	87	ERJ-3EKF2103V	
Muxes	18	1.051	18.918	13	CD74HCT151M	
Decoders	10	0.41	4.1	4	74HC138D	
PCB	5	14	70	1	none	
PCB Stencil	1	60	60	1	none	
MOSFETs	50	0.298	14.9	30	PJA3430_R1_00001	
Low Ohmic Resistors	10	0.8	8	3	varies	
Solder Paste	1	10	10	yes	none	
Header Pins (pack)	1	5	5	yes	none	
Not Ultimately Used:			0			
TH Power FETs	24	0.71	17.04	0		
TH Muxes	20	0.74	14.8	0		
TH Decoders	20	0.68	13.6	0		
Jumper wires (pack)	1	7.12	7.12	0		
Various TH logic ICs	20	0.75	15			
			0			
			0			
			0			
		Total:	346.328			