CAPSTONE PROJECT FINAL REPORT

Augmented Reality with Location Tracking

Authors:
Drew Barclay
Maricar Aliasut
Llandro Ojeda

Advisors:
Dr. Robert McLeod
Dr. Ekram Hossain

Final report submitted in partial satisfaction of the requirements for the degree of Bachelor of Science

in

Electrical & Computer Engineering

in the

Faculty of Engineering

of the

University of Manitoba

Spring 2017

Abstract

Drew Barclay
Maricar Aliasut
Llandro Ojeda

Augmented Reality with Location Tracking

- Statement of the problem
    - Procedures and methods used
    - Results and Conclusions

Acknowledgements

This project would not have been possible without a lot of people:

- Our advisors, Dr. Ekram Hossain and Dr. Bob McLeod for taking us on despite having a risky project. Bob McLeod's generous financial support was a great help as well.
- Zoran Trajkoski, for helping us learn to solder (and for doing a lot of the soldering, saving us hours of time).
- Derek Oliver for allowing us to form a group of three, rather than the minimum of four normally required.
- Aidan Topping for reading over the report and offering suggestions.
- Thomas “thotro” Trojer for his arduino-dw1000 project on Github, which allowed us to quickly determine whether we could do the project at all.
- Aleksandar “d3alek” Kodzhabashev and fractious for their work on the Texample2 3D text rendering library we used.

Thank you all.

Contributions

There were three main facets to the technical side of the project: rangefinding, position calculation, and augmented reality. On the management side of the project, there were the oral and written reports, and the project proposal.

Task	Drew	Maricar	Llandro
Research			
Bluetooth Rangefinding	•		
Google Cardboard		•	
DWM1000 and Arduino Code			•
Rangefinding			
PCB Design	•		
Soldering	•		•
Software	•		
Position Calculation			
Math	•		•
Coding	•		
Augmented Reality			
Camera Capture	•	•	•
3D Graphics	•		
Graphical Assets		•	
Administration			
Design Proposal (Oral)		•	
Design Proposal (Written)	•	•	•
Design Review 2			•
Design Review 3	•		
Design Review 4	•		
Final Report	•	•	•
Final Presentation	•	•	•

Contents

List of Figures

List of Tables

List of Abbreviations

PCB	Printed circuit board.
UWB	Ultra-wideband. A type of radio technology.

Physical Constants

Speed of Light $c_0 = 2.997\,924\,58 \times 10^{8}\,\mathrm{m\,s^{-1}}$ (exact)

List of Symbols

a	distance	m
P	power	$\mathrm{W}\ (\mathrm{J\,s^{-1}})$
ω	angular frequency	rad

Chapter 1

Introduction

Knowledge is power. The more you know, the more powerful you are. The constantly evolving situation requires that more information is being passed down even faster and even sooner. What better way of obtaining that information is to visually come up in front of you!

Augmented Reality, where the visual aspects from your graphical user interface (GUI) can now be seen in real time and in front of you is, is one of the many innovations coming up to solve this question. Imagine seeing the actual speedometer on your windshield rather than on on your dashboard. Or seeing where the person with the radio is when out in the field. This absolutely sounds like something that comes from a science fiction novel, but with how technology is quickly evolving nowadays, it seems closer to reality than later. This is already demonstrated in mobile games such as Pokemon GO! where the tiny virtual pocket monster is seen dancing about on the book sitting on your desk in front of you.

In order to apply a practical application to this, this project takes the visual heads up display very popular in video games such as Call of Duty, Halo and many others, and reproduces it for use in the actual real world. While there are many applications and ideas for a heads up display, this project is focused on locating and tracking certain points and other people within display. With further development and investment, we hope to find this project expanded and improved upon for many applications. For example, keeping track of a person's pulse while a surgeon is performing surgery or the marking of a very important target on a car's windshield.

[insert image of a HUD example]

To keep track of a target for the AR application to find, tags are used to mark points, both of which are arbitrary and needed. The more tags available to use as points, the more accurate a reading would be. The tags use triangulation in order to locate and measure distance from the user, and to the point. A minimum of four tags are necessary to determine from the user to his destination in a 3D plane.

There are three main parts to this project: range finding, position calculation and augmented reality rendering. Range finding requires the use of the tags mentioned above to get the data fed to the device. Position calculation involves data processing from the tags within the device for use within the phone. Rendering will then take the data processed and put it on the screen.

[Insert image of cellphone screen with arrows and tag locations]

Range finding is done wirelessly with tags receiving packets of information to and from devices. The time it takes to receive broadcasted signals determine distances between sensors. These tags uses Ultra Wideband (UWB) to communicate between each other and the device together.

[insert image of tags here]

Data processing is all done on our android devices, which can be read on a tag attached to an Arduino chip. From there, the android device can process the data which can be readied for rendering. From there, using our phones, the data points will be rendered in real time in conjunction with our cameras to point out the location of the tags or the other device also supporting these tags. (Might add: Using Google Cardboard, Google's own Virtual Reality SDK, we can emulate using it as our own HUD by mounting the device on a headset.)

[insert image of Google Cardboard if needed]

Chapter 2

Rangefinding

2.1 Overview

Rangefinding is the act of determining the distance between two things.

2.1.1 The System

The rangefinding subsystem is comprised of **nodes** in a network, each of which is capable of sending and receiving wireless signals.

Each node is either an **anchor** or a **tag**. Both tags and anchors use essentially the same hardware and code, but anchors are assumed to be stationary while tags are mobile. Stationary nodes are required so as to provide a consistent frame of reference for other nodes when calculating positions later on. More information on this can be found in Section 3.4.

The rangefinding subsystem's purpose is to determine the distances between every pair of nodes in the network. With this data, the position calculation subsystem can then determine the positions of every anchor and tag in 3D space. An example 2D rangefinding network is shown in Figure 2.1.

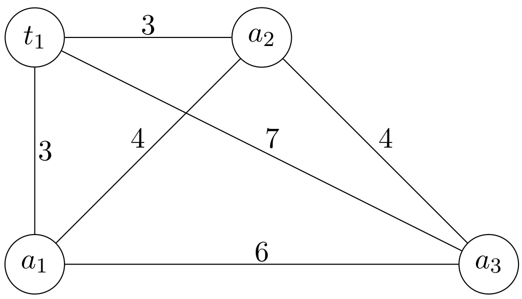

FIGURE 2.1: An example network, showing 1 tag t_1 and 3 anchors a_i and the reported distances between them. Note that each device calculates its own range to the other devices, which will be slightly different from the range calculated on the other devices. This is not shown on the diagram.

2.1.2 Rangefinding

Rangefinding is done wirelessly. The underlying concept is that if we precisely note the times at which we send and receive a signal, then – since light travels at a fixed speed – we can determine the distance the signal traveled, which is the distance between the nodes.

The basic algorithm for the network is:

1. Each node broadcasts a message to every other node, and every node responds.
2. The time it took for the message to travel from one node to another and then back (minus the time spent processing the received messages) is calculated.
3. With some simple math involving the speed of light the distance between the nodes is calculated.

This method of calculating range is known as **time-of-flight** (TOF).

Each node is comprised of a DWM1000, can send wireless signals, and an Arduino microcontroller.

2.1.3 Requirements

There are a number of useful characteristics a rangefinding system should have:

- Ranges should be accurate and not very noisy.
- Ranges should be calculated at a high frequency. If they are not, then we cannot calculate positions quickly and moving objects will have their positions displayed inaccurately.
- The system should be able to cover a large area.
- The system should be robust to nodes entering/leaving the network.

It will be demonstrated how we sought to satisfy these criteria.

(TODO TALK ABOUT WHAT IS COMING UP IN THE REST OF THE CHAPTER)

2.2 Time-of-Flight

This section briefly covers the math behind time-of-flight range calculations. For more in-depth information, (Decawave, 2016) has a comprehensive write-up of the different ways wireless ranging can be performed as well as an error analysis.

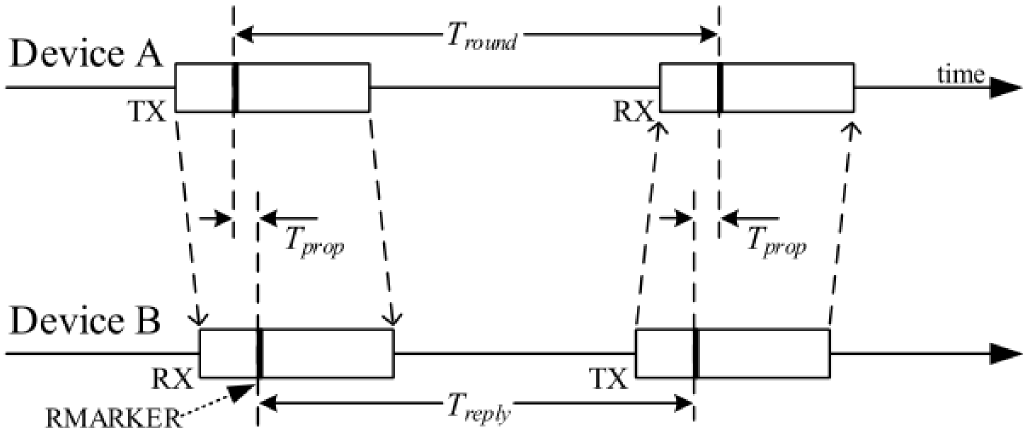

FIGURE 2.2: Single-sided two-way ranging. Figure from (Decawave, 2016).

2.2.1 Propagation Time

The goal behind time-of-flight is to measure the propagation time of a signal, T_{prop}. Once we obtain this, it is a simple measure to calculate the distance d between the two nodes using the speed of light, c, with the following formula:

$$d = cT_{prop}$$

2.2.2 Single-sided Two-Way Ranging

In the case where there are two nodes communicating with each other, (Decawave, 2016) states that one can calculate the time it takes a signal to propagate between them, T_{prop}, as:

$$T_{prop} = \frac{T_{round} - T_{reply}}{2}$$

where T_{round} and $T_{process}$ are the total durations between receiving and transmitting messages as can be seen in Figure 2.2.

2.2.3 Double-sided Two-way Ranging

Because the clocks of two nodes may not pass time at the same rate (clock skew), the above equation will suffer from significant error. This is because processing times far dwarf the time it takes a signal to propagate. (Decawave, 2016) presents, without proof, the following equation for more accurate rangefinding:

$$T_{prop} = \frac{T_{round1}T_{round2} - T_{reply1}T_{reply2}}{T_{round1} + T_{round2} + T_{reply1} + T_{reply2}}$$

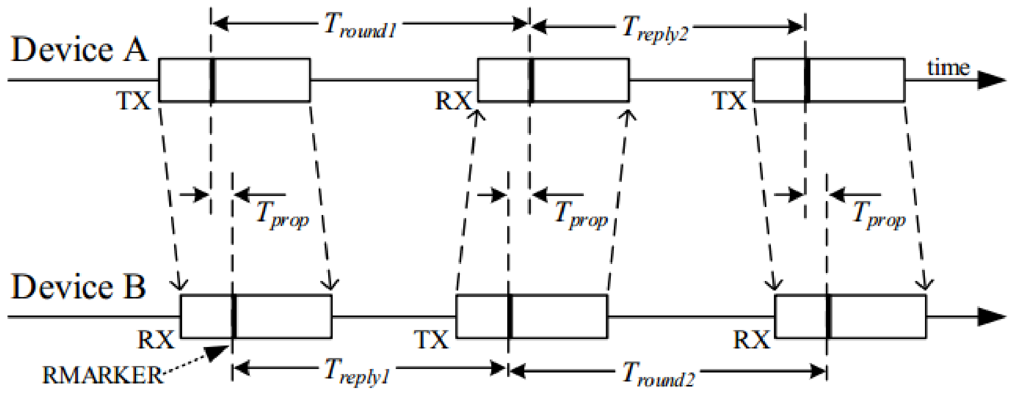

FIGURE 2.3: Double-sided two-way ranging with three messages. Figure from (Decawave, 2016).

where T_{round1}, T_{round2}, T_{reply1}, T_{reply2} are the durations between sending and receiving messages as seen in Figure 2.3. An independently derived proof of this equation can be found in Appendix B.

Because the ranging has two rounds, after the initial calculation of range we can calculate a new range value for every single following transmission by re-using the last timestamps received for the beginning of the next round.

2.3 Wireless Communications

At the start of the project, it was determined that a technology would need to be chosen to handle wireless communications for ranging purposes. A number of options were considered. The ideal technology would:

- Be inexpensive.
- Have good range for in-door use.
- Allow for extremely precise measurements of time. Due to the speed of light, a nanosecond of error in timing calculations would lead to approximately 30cm of error in the calculated distance.
- Be small. As tags are attached to cellphones, they must be small.

2.3.1 Bluetooth in Phones: A Failed Approach

Originally, the goal was to use Bluetooth for ranging. The reason for this was that Android cellphones, which usually have Bluetooth transceivers, could be used for tags and anchors. This would save a large amount of time, as cellphones include batteries, are easy to program, and almost everyone has one (which would make letting people use our project as easy as downloading an app). We would just need to put a few phones running our app around a room, and we'd get ranging. Others, such as (Bekkelien, 2012), were able to use Bluetooth to form indoor positioning systems.

Unfortunately, it became clear early on that Bluetooth – specifically, Bluetooth used on Android cellphones – was not suitable.

There were two ways to use Bluetooth for ranging, and we checked the viability of both:

- RSSI (received signal strength indicator), which is essentially a measure of how strong a received signal is. Because signal power drops off with the square of distance, RSSI can be used to determine distance from a cellphone. Indeed, there this is a cellphone app to do just that on the Google Play Store. Measurements showed that this was method had very low range, was very noisy (power levels varied wildly), and had a large latency between measurements. As well, RSSI values are not standard among cellphones, requiring many calibrations for each model of phone used. Due to these factors, it was determined that using RSSI for distance measurements was not well suited for the fine-grained location tracking this project sought.
- Time-of-flight measurements. Experiments showed that the time it took to send a Bluetooth message itself through the Android OS suffered massive variance of milliseconds (ADD IN APPENDIX CITATION WITH OUR CODE?), which would lead to 300 km of error in calculated distances! Android does not have any guarantees on timing, and does not allow low-level programming access to its internals. This method was proven unworkable.

With all the avenues to use Bluetooth exhausted, the idea of using phones and their Bluetooth transceivers was rejected.

2.3.2 Ultra-wideband and the DWM1000

After doing some research, we discovered the Decawave DWM1000 ultra-wideband transceiver. This chip is advertised as specifically being suited for ranging applications. It uses ultra-wideband technology rather than Bluetooth, Wi-Fi, or similar technologies.

Ultra-wideband, in contrast to Wi-Fi and other radio technologies, occupies a large bandwidth and transmits information via high-bandwidth pulses. Ultra-wideband is suited to tracking applications due to its resistance to multipath propagation, a phenomenon where signals reflect off of surfaces and thus reach the antennae via multiple paths (causing interference). An in-depth look at ultra-wideband is beyond the scope of this report, but interested readers might look more at (INCLUDE SOURCES HERE!!!).

The DWM1000 is advertised as:

- Allowing one to locate objects with up to 10cm accuracy.

- Having a range of up to 290m.
- Having a data rate of up to 6.8Mb/s.
- Having a small physical size.

As well, there was an already an open source library written to use it with an Arduino, which would allow us to quickly prototype with the chip and ensure it would fit for our application.

Because these qualities satisfied our requirements, the DWM1000 was chosen for the foundational technology of the ranging part of the project.

2.4 Design of the Hardware

The hardware design for tags is an Arduino Pro Mini 3.3V connected to a DWM1000 over a PCB.

2.4.1 The Microcontroller

To interface with the DWM1000, a microcontroller was needed. The Arduino Pro Mini 3.3V was chosen because:

- Group members had previous experience with programming Arduinos.
- It was inexpensive.
- It worked off of 3.3V power, which was what the DWM1000 required. This obviated the need for voltage stepping.
- It is capable of floating point math, which is useful for asynchronous two-way ranging. As well, barely any processing power or RAM was perceived to be needed (this was not quite true). Each microcontroller only needed to hold a small number of timestamps, so the small amount of memory and slow processor was not important.
- It has a small physical size. As tags are attached to cellphones, they must be small.
- Batteries would not be needed to power tags, since power could be delivered via USB from the cellphone. This further simplified the design and kept costs low.

The only real downside of the Pro Mini was that it required a lot of soldering.

2.4.2 The PCB

Several designs were considered for the PCB connecting the Arduino and DWM1000. An important factor was size. A PCB was constructed only for the tags, and anchors - which did not need to be reduced in size - were left in breadboard form in order to save costs and time. Making alterations to the PCB design to allow for a barrel jack power adapter (the only distinguishing feature between tags and anchors) would, however, have been extremely simple.

The first design idea we tried was to place the Arduino and DWM on top of each other, resulting in the smallest possible size. See (INSERT FIGURE HERE). However, the physical dimensions of the two components made this impossible. The other possibility was to place each component on opposite sides of the PCB (INSERT FIGURE HERE), but the Arduino's design demanded breakout headers to solder it into the board, which meant the PCB had to have holes drilled. This essentially ran into the same problem as with the original.

The second idea was to make two PCBs. The Arduino would connect to a PCB above it, and that PCB would connect to a board above it with the DWM1000 it. Because it was layered, the pins connecting the layers could be arranged such that the Arduino and DWM1000 were essentially above each other (but without interfering with their pins, as before). See (INSERT FIGURE). In the end, this design was abandoned because the breakout pins would add to much vertical length, it was expensive, and because it was much more complex to design.

As such, the end design required the Arduino and DWM1000 to be spread out over the board, resulting in the PCB being quite long and narrow. The PCB design was done in Eagle and ordered from PCBWay. The result can be seen in (INSERT FIGURE).

2.5 Arduino Software

The software to control the DWM1000 was written in C++ in Arduino IDE. The basic code to control the DWM1000 (handling memory address constants, communication with it via SPI, and a few high-level functions like send/receive) was freely available in Thomas "thotro" Trojer's `arduino-dw1000` library. The library served as the foundation of our code to network the devices.

2.5.1 Networking Basics

Each node in the network broadcasts in a round robin fashion, with a small break after each node has transmitted. As part of the transmitted message, the node transmits the timestamp of when it is sending the message, a list of the timestamps when it last received a communication from every other node in the network, and a list of the last computed ranges to the other nodes. Every other node in the network will receive this information and use it to compute a new range to the node in question.

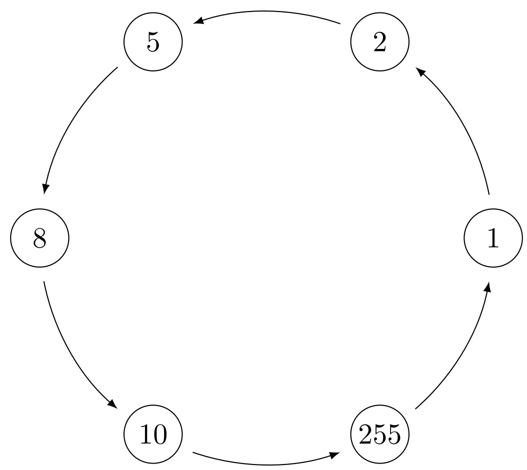

FIGURE 2.4: Transmission order of a network composed of five devices with IDs 1, 2, 5, 8, and 10. Note they transmit in order, and there is a dummy ID of 255 which serves as a marker for the end of the round.

Thus, every node in the network receives complete range information for the whole network.

The DWM uses 40 bits (5 bytes) for its timestamps. The Arduino internally represents these as 64-bit integers (the last byte being meaninless), but transmits them as 40 bit (5 byte) numbers.

Every node in the network has a pre-determined ID, which is set when it the code is programmed onto the device. A single byte is used for this ID to save as much space as possible, though the code could easily be modified to support longer IDs if the devices were to be mass produced. Alternately, something like DHCP could be implemented. As there are only 7 devices currently operational, our project only uses IDs 1 through 7, though the IDs do need not be consecutive. ID 255 is a special dummy value used in the code and should not be selected for use with an actual device.

Communications between the Arduino and the DWM1000 are handled through SPI. When the DWM1000 receives a message, it triggers an interrupt on the Arduino, which can then obtain the received data through SPI.

2.5.2 Communications Timing

In order to maximize the operating frequency of the system and provide as smooth a visual experience as possible on the phone, we want to minimize the amount of time a node is not transmitting. In order to do this, it was decided that nodes should transmit in order of their ID in a round robin fashion. A round of transmissions are performed, each node transmitting once, and then it repeats. When one node receives a transmission, it checks to see if it is its turn and then transmits as soon as it can.

This round robin ordering is done by creating an ordered array of IDs in the network. When a transmission is received, the network increments the index of the next expected device to transmit by one. A device can tell whether it should transmit by checking to see if its ID is equal to the ID of the next expected node to transmit. If so, it transmits.

The downside to this approach is that if any transmission were to be lost, for example by interference or electrical noise, the network would grind to a halt as it waits for a message that is never going to come. A solution to this is to include a timer that tracks a window in which a message should be received. If it takes too long to receive a message, the device was assumed to have to failed to transmit for some reason and the next device will take their turn and transmit.

To help the network be more robust, for example if one node has a clock that runs faster than the others (making it think a transmission is late when it is not), the network assumes whatever device last transmitted was right in doing so, and sets the index of the next node to transmit in the ordered array as being one higher than it.

This approach also raises the question of how new nodes will join the network if a node is constantly transmitting. To solve this, a small delay is added at the end of each round. When a device wishes to enter the network, it waits for the end of a round, and then does a transmission in this added space. This transmission lets all devices know it is part of the network, and they will add it to their ordered lists of IDs appropriately.

In the code, this delay at the end of the round is implemented by adding a dummy node with the ID of 255 (the highest ID possible with one byte) to the network. The nodes will wait for a transmission from it at the end of each round, but it will never come, at which point the round starts over with the lowest ID transmitting. An example ordering can be found in Figure 2.4.

An unsolved issue here is what happens when the transmission to join the network is sent and it is not received. This will cause the joining device to have a wrong order, and it will transmit at the same time as another device every round. Since the chances of this happening are quite low and could be solved by resetting the joining device, this is not dealt with. A possible solution to deal with this would be having the receiving device check the responses of all the other nodes to make sure they all received its transmission, and, if they have not, then to retransmit in the space at the end of the round until they do.

Code for this logic can be found in the `loop` function of Appendix A.

2.5.3 Range Information Protocol

The protocol for a transmission from a node is as follows:

- 1 byte for the ID of the transmitting node.
- 5 bytes for the timestamp of the sending of the message.

- For each device the transmitting node has knowledge of:
  - 1 byte for the ID of the device
  - 1 byte for the shared counter (used to detect lost transmissions)
  - 5 bytes for timestamp of last received message from the device
  - 4 bytes for last calculated range

Parsing a received message and updating range values from the parsed data is fairly straightforward. The code in question can be found in the `parseReceived` function in Appendix A.

2.5.4 Calculating a Delay

As part of the time of flight calculation, a timestamp is needed for when the message was received and for when the reply was sent. The DWM1000 does not offer a way to automatically set the time upon transmission, but does offer the ability to set a time when a message will be transmitted in the future. This timestamp can then be embedded in the message itself.

Ideally, this delay is short. However, if the delay is too short, the Arduino will not be able to transmit the data the DWM1000 should send before the timestamp is passed. This causes a silent error, and the DWM will not transmit anything. As well, the delay specified is not the delay at which the DWM1000 will begin transmitting, it is the delay that the DWM1000 will begin transmitting the data itself. There is a "premable" sent before any transmission to allow the other devices in the network time to wake up and begin sniffing the air and know a message is coming. This time to send the preamble lasts quite a long time (approximately 1 microsecond per symbol in the problem (ADD CITATION HERE), which adds up to almost 2 ms). This was the most difficult to solve bug that was encountered in the design of the system.

In the code, the delay before we can transmit is the sum of the:

- Number of symbols in the preamble $\times$ 1µs (128 is the value used here, though the DWM offers a choice of preamble length)
- Time required to calculate and send the timestamp using SPI, about 1000µs (empirically determined)
- Bytes of data to transmit $\times$ 4.5µs, or 85 $\times$ number of devices in the network besides this one (empirically determined)

Adding in a fudge factor of about 200µs, the delay we use in code for 6 devices is roughly 1800 µs.

It is important to minimize this delay so to increase the maximum frequency the system can update ranges at. The tradeoff of having a short preamble versus a long one is that a long preamble means there is less chance of a transmission being missed, at the cost of using more power and taking longer to send. The value of 128 was chosen as it was the smallest possible. Tests indicated that transmissions were not being lost enough to matter with such a short preamble.

2.6 Calibrating

DWM1000s need to be calibrated to give correct distances due to the differences in hardware. There are a number of factors affecting the DWM1000, such as temperature. Some of these factors are controlled for in software in the arduino-dw1000 library. For a detailed overview of the possible errors and how to correct for them, consult the INCLUDE SOURCE HERE IT WAS A PDF ON THE SITE I READ.

The primary factor which could not be controlled for by Decawave is the antenna delay. The capacitance of the hardware the DWM1000 is hooked up to can cause nanosecond-level delays in transmission (in experiments, it was found that this could cause up to several meters of error incorrectly configured). This is a constant, and is determined empirically. The antenna delay for the tags is the same and was found to be INSERT NUMBER HERE THAT WE FOUND, and the antenna delay for the anchors is roughly the same and was found to be INSERT NUMBER HERE.

These constants required a slight tweak to the DWM1000 library. INSERT LINK TO SOURCE CODE OF OUR CHANGED VERSION HERE.

2.7 Operating Frequency Analysis

In order to estimate the operating frequency of the rangefinding subsystem – that is, how many times we can get the range from all devices to a single device per second[1] – we see that the frequency will be the inverse of the time taken to complete one round (the system performs one range update per device per round), assuming no lost transmissions. That is,

$$f_{op} = 1/T_{round}$$

The time it takes a round is equal to the time it takes the devices to each parse a received message and transmit a message. Determining the time it takes to transmit a message is a little tricky to calculate due to the requirement to include the delay before the message is sent. A rough estimate of the time it takes to do a round of transmissions is:

[1] It should be noted that this definition of operating frequency is somewhat misleading and calculates a minimum operating frequency. Because each pair of devices compute their range twice, once on each device at different points in the round, the true operating frequency of the system will be higher on average. The analysis here does not take this into account.

In the best case scenario, this will almost double the system frequency. In the worst case scenario, where two devices are next to each other in the transmission order in a large network, the system operating frequency will barely increase. The figures given for operating frequencies will thus give a range from the minimum by the definition here to twice it.

$$T_{round} = N(T_{device} + T_{prop}) + T_{end}$$

where N is the number of nodes in the network (the 4 anchors and 3 tags that were made make 7), T_{device} is the time it takes a device to fully receive and transmit, T_{prop} is the time it takes the message propogate (this will not be constant, but it is so small as to be ignorable in this analysis), and T_{end} is the duration of the pause at the end of the round.

We can estimate T_{device} as

$$T_{device} \approx T_{rx} + T_{tx} + T_{txDelay}$$

where T_{rx} is the time it takes to parse a received message (7-8 ms empirically), T_{tx} is the time it takes to create the packet to send to the DWM1000 (1-2 ms empirically), $T_{txDelay}$ is the delay before the ranging packet can be sent (calculated in Section 2.5.4, about 1800 µs).

Putting it all together, the equation estimating the operating frequency of the system is:

$$T_{round} \approx N(T_{rx} + T_{tx} + T_{txDelay}) + T_{end}$$

T_{end} is implemented in the code as a dummy device which the network will never receive a message for, allowing it to expire. Perhaps unintuitively, it is not actually equal to the time it takes a normal device to be rejected. This is because, when the round ends, it will not have transmitted anything, meaning that the time it takes to parse a received message will be 0 from it and the next round will start as soon as the first device in the transmission order array can transmit. T_{end}, then, is about equal to $T_{tx} + T_{txDelay}$.

Plugging the above numbers into the equation, we find we should expect that a network composed of 2 devices can operate at a frequency of about 40Hz and a network of 7 devices can operate at about 13Hz. These estimates are quite close to the empirical measurements made (see Section 2.8).

As can be seen from the equations, the system is primarily bottlenecked by the speed at which messages can be processed and transmitted. These numbers are well above what the minimum requirements for the system, but they are below an ideal 60Hz (at which point the updates would be so smooth as to seem mostly continuous to the human eye and there would be marginal benefit to improving the frequency). Originally, it was thought that the Arduino's speed would not matter, but this turned out to not be the case. Future work on this project would benefit from finding a faster microcontroller, or perhaps offloading all processing to the cell phone each tag is hooked up to.

An optimization that was considered and rejected due to time constraints was offloading only the time-of-flight calculation to the cell phones, instead transmitting timestamps instead of calculated ranges in the ranging packets. The Arduino Pro Mini was found to take roughly 1ms to calculate each range. Implementing this

TABLE 2.1: The effects of treatments X and Y on the four groups studied.

Groups	**Treatment X**	**Treatment Y**
1	0.2	0.8
2	0.17	0.7
3	0.24	0.75
4	0.68	0.3

TABLE 2.2: The effects of treatments X and Y on the four groups studied.

Groups	**Treatment X**	**Treatment Y**
1	0.2	0.8
2	0.17	0.7
3	0.24	0.75
4	0.68	0.3

would increase the operating frequency by about 1Hz - not enough to justify any time spent on it.

2.8 Results

Results showed that the DWM1000 was close to as accurate as Decawave claimed (within 10cm). Of note, sometimes the reported ranges would glitch and be off by a meter or more. Table 2.1 shows the accuracy of the system. Though the accuracy of the rangefinding system was not a direct requirement of the project as a whole, the accuracy of the rangefinding system translates into accuracy of the position calculating system and thus the accuracy of the markers on the AR display.

The operating frequency was close to the values predicted by the theory in Section 2.7. Predicted and actual values can be found in Table 2.2.

Chapter 3

Position Calculation

3.1 Overview

Our location tracking system consists of two different anchors, anchors and tags. The anchors which we can think of as static anchors doing the postion tracking, and the tags which are our dynamic anchors that we need to track. The overall goal of this subsystem is to calculate the position of each anchor given only the distances from one another. The anchors use ultrawideband technolgy to communicate with each other and send the position data to our android application using FTDI.

3.2 High-Level Example in 2D

To start the position calculation we take three different anchors for example a_0, a_1 and a_2. We set an arbitrary point as the origin a_0, then we proceed to make a horizontal line from a_0 to another anchor say a_1, since we have all the distances between the anchors we can now use basic trigonometry to calculate the position of our last anchor a_2. An Example calculation is as follows:

distance from anchor x to anchor y: d_{xy}, we use cosine law to determine the angle

$$\Theta = \cos^{-1}\left(\frac{d_{01}^2 + d_{02}^2 - d_{12}^2}{2 * d_{01} * d_{02}}\right)$$

when we have the angle we can now calculate the x and y coordinates:

$$x_2 = cos(\Theta) * d_{12}$$

$$y_2 = sin(\Theta) * d_{12}$$

3.3 Extending This to 3D

Extending the subsystem to three dimensions does not really change the calculation all that much except that we now need four anchors to get a reliable position in three

dimensions. We have our four anchors: a_0, a_1 , a_2 and a_3, we arbitrarily set one as the origin a_0(0, 0, 0) and we now make a horizontal line to another anchor say a_1 and since we know the distance between these two anchors we can get the position to be a_1(0, d_{01}, 0), where d_{01} is the distance between a_0 and a_1. We can now form a triangle with the third anchor and use cosine law to determine the position and we arbitrarily set the z value to 0.
We now have the coordinates for the third anchor, but we do not know the correct orientation in the z axis, it could be either positive or negative. This is where the fourth anchor comes in, we use the anchor to create a temporary point using the same method we use above.

$$\Theta = \cos^{-1}\Big(\frac{d_{01}^2 + d_{03}^2 - d_{13}^2}{2 * d_{01} * d_{03}}\Big)$$

$$x_{temp} = cos(\Theta) * d_{13}$$

$$y_{temp} = sin(\Theta) * d_{13}$$

$$z_{temp} = 0$$

We will rotate around the x-axis by keeping the x value constant and taking the y distance and z distance to be a circle of radius of y_{temp}.We calculate the distance between the x value of the temporary point and the x value of anchor 2, we denote this value Δx.

$$\Delta x = x_{temp} - x_2$$

Now that we have Δx we can now calculate the position of our fourth anchor using cosine law.

$$\Theta = \cos^{-1}\Big(\frac{\Delta x^2 - d_{23}^2 + y_2^2 + y_{temp}^2}{2 * y_2 * y_{temp}}\Big)$$

$$x_3 = x_{temp}$$

$$y_3 = cos(\Theta) * y_{temp}$$

$$z_3 = sin(\Theta) * y_{temp}$$

3.4 Frame of Reference

The positions calculated from ranges do not correspond to the real world! We must use the cellphones' accelerometer (for gravity) and magnetic sensor (for compass direction/inclination) to map these positions to reality.

Go over how this is done.

3.5 Getting the Ranges

Talk about FTDI and our protocol to communicate from Arduino to Android. Go over how they are parsed (put in appendix of parsing code)

3.6 Conclusion

This subsystem calculates positions; we figured out how ranges are obtained, how they are calculated. Go over this in more detail.

Chapter 4

Augmented Reality

4.1 Overview

Talk about the goal of this section: to take positions from the position calculation subsystem, then render them on the screen to show where things are (even behind walls, etc.)

Make sure to note that we'll have youtube videos up and give links.

Don't forget an appendix with major code.

4.2 Tech Used

Go over OpenGL, the 3D text rendering library we're using, Android's rotation calculations.

4.3 3D Math Overview

Touch on matrices, projection/view/model matrices. Talk about how we use OpenGL's implementation to place things in 3D space.

4.4 Camera Field of View and OpenGL Field of View

Talk about how we make positions accurate here. Go over how we just overlay the camera and OpenGL surface on each other. Discuss Android camera limitations/FPS results.

4.5 Cellphone Rotation

Discuss in detail more about how Android implements getting the rotation, the limits of it when moving + in strong magnetic fields.

4.6 Billboard Effect

Talk about how doing the inverted view matrix multiplication causes things to always face the screen. Give pictures! Motivation here is to make 2D images that we can place in 3D space and have them face the screen.

4.7 HUD

Go over how the HUD is made. (Still need to finish that too so we can get pictures.)

4.8 Calibration

Talk about how we can take the cellphone's rotation matrix and calibrate the positions we calculate from anchors/tags.

4.9 Results

Show off how accurate we are. Have Youtube videos displaying such.

4.10 Conclusion

Conclusion of this section: we can render positions on the screen and have 3D math to do so etc.

Chapter 5

Conclusion

Lastly, the our project shows that with much more refinement, a practical application of augmented reality is closer to reality than previously envisioned. This is because it is operating on a device that can be easily obtained by the average consumer. While devices such as the HTC Vive and Occulus Rift demonstrate the power of virtual reality, and the potential to be augmented in real life, it is currently out of the practical price range of many consumers. Most of the time, a user will also require a powerful personal computer to run the programs, which can range at least one thousand dollars. Google Cardboard shows that with an Android device, that this is subjectively cheaper, where the tags and chips must be obtained on top of the Android device. There are three main topics that were covered in the report: augmented reality rendering, position calculation and range finding.

Most of the data is processed on our Android devices, read from tags attached to an Arduino chip. Using 3D space equations to translate the data to a 2D plane, allows data processed from the chips. The phones then render the graphical overlay in real time with a live video feed to show the direction of other devices or tags in the network. (to edit later if we put in images with the complete HUD: The report shows an demonstration of what the screen can be.) There will be a live demonstration in the final presentation.

Locating devices and tags are done with the tags to send directly to the Android device. It is done UWB to communicate between each part of the network. A minimum of four tags are necessary to determine a user's distance and direction to his destination in a 3D plane. The more tags available, the more accurate the reading will be. This is essentially the bread and butter of the project; the device on which this can be processed and rendered on does not matter. Without the tags, there would be no data to be drawn about a user's location and the destination.

The realization of a HUD becoming a real and practical application to everyday lives can be a fascinating endeavor. It cannot be limited to science fantasy and video games no longer. The applications of this project are astounding and very broad. It can range from surgeons knowing the status of their patient, location tracking, entertainment. The amount is limitless!

Pokemon GO! was the start of many augmented reality applications available on Google Play. The hope is that other developers realize the potential of Augmented

Reality and not only make it accessible to normal, everyday citizens but to also improve it to such a point where it will be applied as part of a more accessible everyday accessory such as watches and glasses.

To have all the information ready at your fingertips, or shall we say, in front of your face is a blessing to behold. It allows for the quicker passing of information quickly and received in such a fast and direct manner. If one obtains and processes of the situation at hand, he shall receive the edge. To him, he shall be the victor.

Appendix A

Arduino Code

```
//Created by Drew Barclay
//Code uses thotro's dwm1000-arduino project
//Protocol: one byte for ID, then five bytes for the timestamp of the sending
//For each device, append one byte for the ID of the device, one byte for the

#include <SPI.h>
#include <DW1000.h>

class Device {
public:
  byte id;
  byte transmissionCount;
  DW1000Time timeDevicePrevSent;
  DW1000Time timePrevReceived;
  DW1000Time timeSent;
  DW1000Time timeDeviceReceived;
  DW1000Time timeDeviceSent;
  DW1000Time timeReceived;
  float lastComputedRange;
  bool hasReplied;

  Device() : lastComputedRange(0.0f), id(0), hasReplied(false) {}

  void computeRange();

  float getLastComputedRange() {
    return this->lastComputedRange;
  }
};

// CONSTANTS AND DATA START
//number of devices that can form a network at once
```

```
#define NUM_DEVICES 6
Device devices[NUM_DEVICES];
int curNumDevices;

// connection pins
const uint8_t PIN_RST = 9; // reset pin
const uint8_t PIN_IRQ = 2; // irq pin
const uint8_t PIN_SS = SS; // spi select pin

// data buffer
#define LEN_DATA 256
byte data[LEN_DATA];

//id for this device
const byte OUR_ID = 5;

long lastTransmission; //from millis()

// delay time before sending a message, should be at least 3ms (3000us)
const unsigned int DELAY_TIME_US = 2048 + 1000 + NUM_DEVICES*83 + 200; //shoul

volatile bool received; //Set when we are interrupted because we have received
// CONSTANTS AND DATA END

void Device::computeRange() {
  // only call this when timestamps are correct, otherwise strangeness may res
  // asymmetric two-way ranging (more computationly intense, less error prone)
  DW1000Time round1 = (timeDeviceReceived - timeDevicePrevSent).wrap();
  DW1000Time reply1 = (timeSent - timePrevReceived).wrap();
  DW1000Time round2 = (timeReceived - timeSent).wrap();
  DW1000Time reply2 = (timeDeviceSent - timeDeviceReceived).wrap();

  DW1000Time tof = (round1 * round2 - reply1 * reply2) / (round1 + round2 + re
  this->lastComputedRange = tof.getAsMeters();
}

void setup() {
    received = false;
    curNumDevices = 0;
    lastTransmission = millis();

    Serial.begin(115200);
```

```
    delay(1000);
    // initialize the driver
    DW1000.begin(PIN_IRQ, PIN_RST);
    DW1000.select(PIN_SS);
    Serial.println(F("DW1000 initialized ..."));
    // general configuration
    DW1000.newConfiguration();
    DW1000.setDefaults();
    DW1000.setDeviceAddress(OUR_ID);
    DW1000.setNetworkId(10);
    DW1000.enableMode(DW1000.MODE_LONGDATA_RANGE_ACCURACY);
    DW1000.commitConfiguration();
    Serial.println(F("Committed configuration ..."));

    // attach callback for (successfully) sent and received messages
    DW1000.attachSentHandler(handleSent);
    DW1000.attachReceivedHandler(handleReceived);
    DW1000.attachErrorHandler(handleError);
    DW1000.attachReceiveFailedHandler(handleReceiveFailed);

    receiver(); //start receiving
}

void handleError() {
  Serial.println("Error!");
}

void handleReceiveFailed() {
  Serial.println("Receive failed!");
}

void handleSent() {

}

void handleReceived() {
  received = true;
}

void receiver() {
    DW1000.newReceive();
    DW1000.setDefaults();
```

```
    // so we don't need to restart the receiver manually
    DW1000.receivePermanently(true);
    DW1000.startReceive();
}

void parseReceived() {
  unsigned int len = DW1000.getDataLength();
  DW1000Time timeReceived;
  DW1000.getData(data, len);
  DW1000.getReceiveTimestamp(timeReceived);

  if (len < 6) {
     Serial.println("Received␣message␣with␣length␣<6,␣error.");
     return;
  }

  //Parse data
  //First byte, ID
  byte fromID = data[0];
  //Second byte, 5 byte timestamp when the transmission was sent (their clock)
  DW1000Time timeDeviceSent(data + 1);

  //If this device is not in our list, add it now.
  int idx = -1;
  for (int i = 0; i < curNumDevices; i++) {
   if (devices[i].id == fromID) {
     idx = i;
   }
  }
  if (idx == -1) { //If we haven't seen this device before...
    Serial.print("New␣device␣found.␣ID:␣"); Serial.println(fromID);
    if (curNumDevices == NUM_DEVICES) {
      Serial.println("Max␣#␣of␣devices␣exceeded.␣Returning␣early␣from␣receive.
      return;
    }
    devices[curNumDevices].id = fromID;
    devices[curNumDevices].timeDeviceSent = timeDeviceSent;
    devices[curNumDevices].transmissionCount = 1;
    idx = curNumDevices;
    curNumDevices++;
  }
```

```
devices[idx].hasReplied = true;

//Now, a list of device-specific stuff
for (int i = 6; i < len;) {
  //First byte, device ID
  byte deviceID = data[i];
  i++;

  //Second byte, transmission counter
  byte transmissionCount = data[i];
  i++;

  //Next five bytes are the timestamp of when the device received our last t
  DW1000Time timeDeviceReceived(data + i);
  i += 5;

  //Next four bytes are a float representing the last calculated range
  float range;
  memcpy(&range, data + i, 4);
  i+= 4;

  //Is this our device? If so, we have an update to do and a range to report
  if (deviceID == OUR_ID) {
    //Mark down the two timestamps it included, as well as the time we recei
    devices[idx].timeDeviceReceived = timeDeviceReceived;
    devices[idx].timeDeviceSent = timeDeviceSent;
    devices[idx].timeReceived = timeReceived;

    Serial.print("Transmission␣received␣from␣tag␣"); Serial.print(devices[id

    //If everything looks good, we can compute the range!
    if (transmissionCount == 0) {
      //Error sending, reset everything.
      devices[idx].transmissionCount = 1;
    } else if (devices[idx].transmissionCount == transmissionCount) {
      if (devices[idx].transmissionCount > 1) {
        devices[idx].computeRange();
        Serial.print("!range␣"); Serial.print(OUR_ID); Serial.print("␣"); Se
      }
      devices[idx].transmissionCount++;
    } else {
      //Error in transmission!
```

```
        devices[idx].transmissionCount = 0;
        Serial.println("Transmission count does not match.");
      }

      devices[idx].timeDevicePrevSent = timeDeviceSent;
      devices[idx].timePrevReceived = timeReceived;
    }

    Serial.print("!range "); Serial.print(fromID); Serial.print(" "); Serial.p
  }
  //TODO if our device was not in the list and we think it should have been, r
}

void doTransmit() {
  data[0] = OUR_ID;

  //Normally we would set the timestamp for when we send here (starting at the

  int curByte = 6;
  for (int i = 0; i < curNumDevices; i++) {
    data[curByte] = devices[i].id;
    curByte++;
    data[curByte] = devices[i].transmissionCount;
    curByte++;
    devices[i].timeReceived.getTimestamp(data + curByte); //last timestamp wil
    curByte += 5;
    float range = devices[i].getLastComputedRange();
    memcpy(data + curByte, &range, 4); //floats are 4 bytes
    curByte += 4;

    if (devices[i].hasReplied) {
      devices[i].transmissionCount++; //Increment for every transmission, the
      devices[i].hasReplied = false; //Set to not for this round
    }
  }

  //Do the actual transmission
  DW1000.newTransmit();
  DW1000.setDefaults();

  //Now we figure out the time to send this message!
  DW1000Time deltaTime = DW1000Time(DELAY_TIME_US, DW1000Time::MICROSECONDS);
```

```
  DW1000Time timeSent = DW1000.setDelay(deltaTime);
  timeSent.getTimestamp(data + 1); //set second byte (5 bytes will be written)

  DW1000.setData(data, curByte);
  DW1000.startTransmit();

  for (int i = 0; i < NUM_DEVICES; i++) {
    devices[i].timeSent = timeSent;
  }
}

void loop() {
  unsigned long curMillis = millis();

  if (received) {
    received = false;
    parseReceived();
  }

  if (curMillis - lastTransmission > 300) {
    doTransmit();
    lastTransmission = curMillis;
  }
}
```

Appendix B

Asynchronous Two-Way Ranging Proof

We begin by assuming that the clock of one node is off by alpha (α) and the other by beta (β), the key here was to find the time of flight in a virtual clock that would be the mean value of alpha and beta.

$$\alpha a = 2t + \beta c \tag{1}$$

$$\beta d = 2t + \alpha b \tag{2}$$

$$1 = \frac{\alpha + \beta}{2}$$

We simplify and solve for β:

$$\beta = 2 - \alpha \tag{3}$$

We then subsititute equation (3) into equation (1) and equation (2).

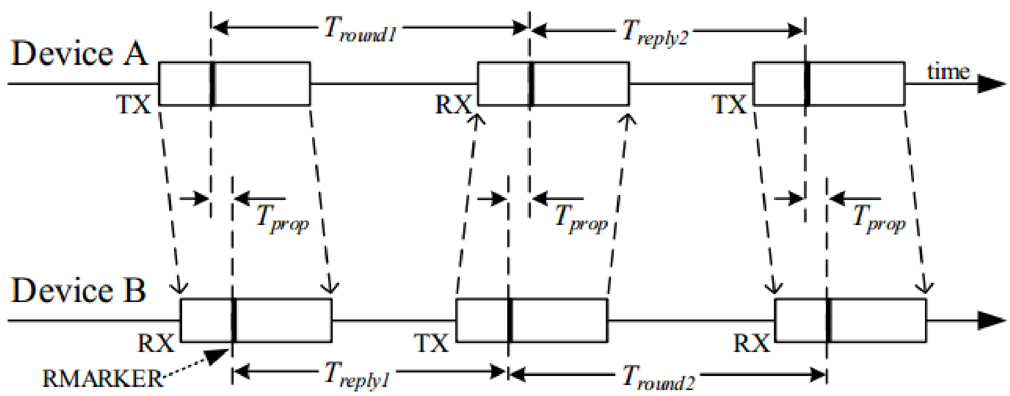

FIGURE B.1: Double-sided Two-way ranging with three messages. Figure from Decawave (2016).

$$\alpha a = 2t + 2c - \alpha c \tag{4}$$

$$2d - \alpha d = 2t + \alpha b \tag{5}$$

Isolate α for both equations (4) and (5) we get the following:

$$\alpha = \frac{2(t+c)}{a+c} \tag{4.1}$$

$$\alpha = \frac{2(d-t)}{b+d} \tag{5.1}$$

Setting (4.1) and (5.1) equal to each other we can solve for propagation time, t.

$$2(t+c)(b+d) = 2(d-t)(a+c)$$

$$tb + td + bc + dc = da + dc - ta - tc$$

$$t(b+d+a+c) = da - bc$$

$$t = \frac{da - bc}{a+b+c+d} \tag{6}$$

Bibliography

Bekkelien, Anja (2012). "Bluetooth Indoor Positioning". In: URL: https://pdfs.semanticscholar.org/1919/037541b4a84d0b5a6dd4d425c0891282ae8f.pdf.

Decawave (2016). "DW1000 User Manual". Version 2.10. In: URL: http://www.decawave.com/support.