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

Virtual reality technology has lowering in cost and growing in popularity in recent years. This project uses a variant on virtual reality, augmented reality, to show the locations of tracked objects on an Android cellphone on top of images captured by the cellphone's camera.

The project allow the user to know whether a tracked object is to the left, right, or behind them, which would be useful in, for example, a military context for tracking the locations of fellow soldiers.

A proof of concept is presented which allows for accurate tracking in an in-door location. The system is composed of Decawave DW1000 ultra-wideband transceivers and Arduino Pro Mini microcontrollers forming a network of distance sensors, and Android cellphones running an augmented reality application created for the project.

Distances between objects are determined via time-of-flight measurements between a number of devices in a network. Trigonometry allows these distances to be turned into positions in 3D space. OpenGL is used to render these images on the cellphone screen in real time.

The system allows for distance determinations accurate to within 50 centimeters. Positions rendered on the augmented reality display, after calibration, can be similarly accurate.

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

Glossary

Anchor	A stationary device used as a part of a network of similar devices to triangulate a user's location
Billboarding	Technique used in 3D graphics to make an object face the camera
Graphical User Interface (GUI)	A visual interface which allows the user to interact with and know the state of a system
Heads Up Display (HUD)	A type of GUI which displays information to the user without the use of a dedicated screen
Homogenous Coordinates	An extension of coordinates which includes an extra component, w, which acts as a scaling factor on the other components of the coordinate
Model Matrix	A matrix used to transform the vertices of a 3D model from model space to world space. In equations, a model matrix usually uses the M symbol
Model Space	A coordinate system relative to the center of a 3D object
Nodes	Devices in a network, in this project either anchors or tags
Normalized Device Coordinates (NDC)	Coordinates transformed from world space to projection space and scaled such that all vertices of 3D objects visible to the camera will be contained within a cube from $(-1, -1, -1)$ to $(1, 1, 1)$
OpenGL	An API used to render 2D and 3D graphics

Perspective Projection Matrix	A type of projection matrix which mimics the human eye and will scale farther away objects so to they seem smaller
Printed Circuit Board (PCB)	A board with etched copper wires used in the construction of some electronics.
Projection Matrix	A matrix which can be used to transform coordinates in world space to projection space
Projection Space	A coordinate system relative to the center of the device's screen
Rotation Matrix	A transformation matrix which can rotate vectors a specified angle around any combination of axes, usually using the $\mathbf{R}$ symbol in equations
Tag	A mobile device, connected to the user's cellphone in this project, used as part of a network of similar devices to triangulate a user's location
Time of Flight (TOF)	A method used to calculate the distance between objects using the time it takes a signal to propagate between the objects
Transformation Matrices	Matrices that are used to transform vectors in 3D space, usually involving scaling, translation (movement in the space), and rotation
Translation Matrix	The transformation matrix which can be used to move a vector in space in a specified direction and distance
Ultra-wideband (UWB)	A type of radio technology useful in indoor ranging applications
View Matrix	A matrix used to transform coordinates in world space to view space
View Space	A coordinate system relative to the point at which the 3D scene is viewed from
World Space	A coordinate system relative to the 3D scene as a whole, similar to a coordinate system for the real world

Chapter 1

Introduction

1.1 Motivation

Valve recently released SteamVR which is to work with HTC Vive in order to promote virtual reality in their games. Videogames are the platform for interactive media. In the last few years, starting with Nintendo's Wii console, companies have started to heavily invest in developing ways for users to interact more intimately with their media, where they become "one with the game".

This movement started with the introduction of *Star Trek*'s holodeck and *X-Men*'s Danger Room, which use holographic images to create another reality within a room. This project is in some ways the opposite of that; the "game" is inserted into real life. The game is "augmented" into the current time and place, hence the term **augmented reality** (AR). Augmented reality is where portions of a graphical user interface from a program is seen in the eyes of a user right then and there through a **heads up display** (HUD). It is heavily inspired by many first person shooter videogames such as *Halo* and *Call of Duty*.

Augmented reality can be used in many applications, from a surgeon seeing detailed vital signs of a patient to a soldier keeping track of his teammates. This project looks at a practical application of tracking locations of objects and displaying them on a HUD.

1.2 Overview

This project takes the visual HUD very popular in video games such as *Planetside 2*, as seen in Figure 1.1, and reproduces it for use in the actual real world. While there are many applications and ideas for a heads up display, the project is focused on a proof of concept locating and tracking certain objects within the display. With further development and investment, the project could be 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 driver's destination on their windshield.

To locate a target, devices called tags are used to mark objects or locations of interest. Each tag is affixed to the target that is to be tracked. The tags determine

FIGURE 1.1: A screenshot of the videogame *Planetside 2*, showing its HUD. Locations of objectives and allies are indicated with triangles.

their distances to other tags, and these distances are triangulated and the positions of tracked objects determined. For 3D positioning, a minimum of four tags are necessary to precisely pinpoint an object's location.

There are three main parts to this project:

- Rangefinding (Chapters 2 and 3): rangefinding uses tags to determine the ranges (distances) between other tags.
- Position calculation (Chapter 4):pPosition calculation involves processing the range data from the tags to produce positions in 3D space for use within the augmented reality portion of the project.
- Augmented reality rendering (Chapter 5): the AR subsystem marks the locations of tracked targets on the screen.

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

Chapter 2

Determining Ranges

This chapter covers the rangefinding subsystem, the part of the project which determines the ranges between two sensors. The ranges are later fed into the position calculation subsystem, which determines the positions of the sensors in 3D space.

This chapter covers three main topics:

1. A description of what the rangefinding system does, and the devices composing it.
2. The math behind calculating the range between two sensors.
3. A description of the networking protocol developed for this project.

2.1 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 4.1.

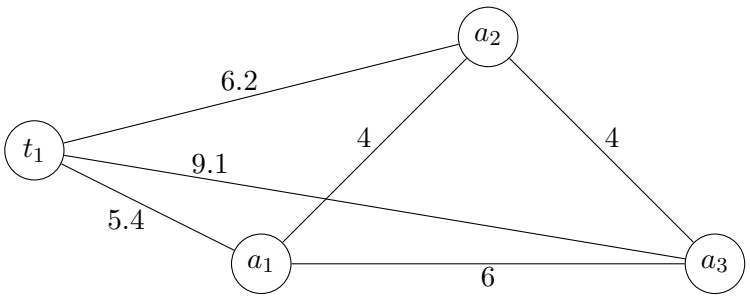

FIGURE 2.1: An example network, showing 1 tag t_1 and 3 anchors a_i and the reported distances between them. Note that each device calculates its own range to the other devices, which will be slightly different from the range calculated on the other devices. This is not shown on the diagram.

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.

2.2 Rangefinding

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

2.3 Requirements

The scope of the project is to handle precise rangefinding indoors in a room. The requirements for the system were:

- The system must be able to produce ranges accurate to within three meters or less that are not noisy (regular swings of $\pm$ one meter would be unacceptable). Otherwise, what was displayed in the augmented reality portion of the project would not be useful.
- The system must calculate ranges at frequencies greater than 3Hz. Otherwise, moving objects will have their positions displayed inaccurately (and the system would be misleading).
- The system must be able to rangefind within an area the size of a room (at least 5x5 meters).
- The system must be able to robustly handle nodes entering and leaving the network.

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

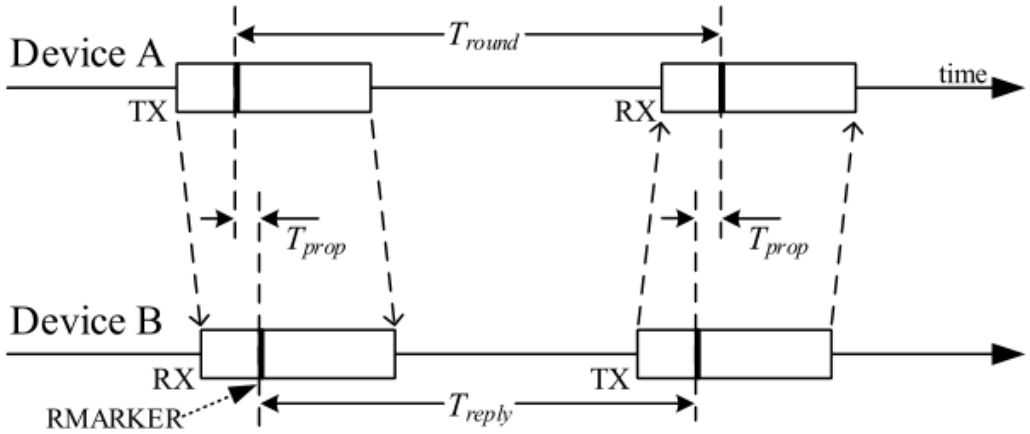

FIGURE 2.2: Single-sided two-way ranging [1].

2.4 Time-of-Flight

This section briefly covers the math behind time-of-flight range calculations. For more in-depth information, Decawave [1] has a comprehensive write-up of the different ways wireless ranging can be performed as well as an error analysis.

2.4.1 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:

$$d = cT_{prop}$$

2.4.2 Single-sided Two-Way Ranging

In the case where there are two nodes communicating with each other, [1] 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.

2.4.3 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. [1] presents, without proof, the following equation for more accurate rangefinding:

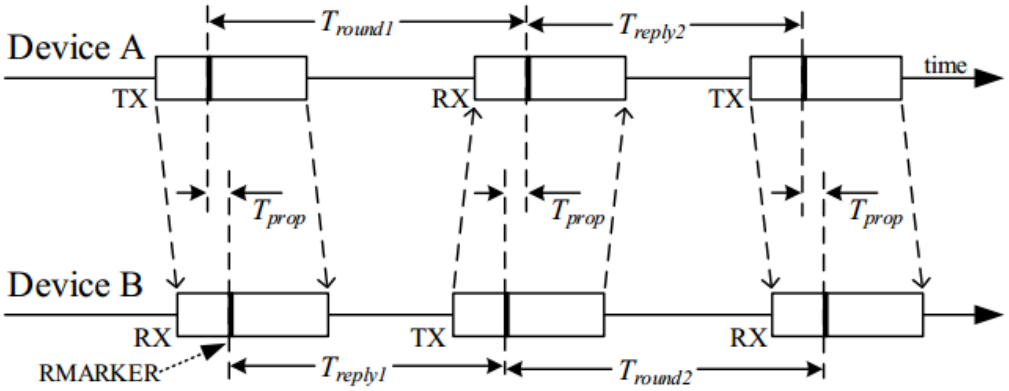

FIGURE 2.3: Double-sided two-way ranging with three messages [1].

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

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.

2.5 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. 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

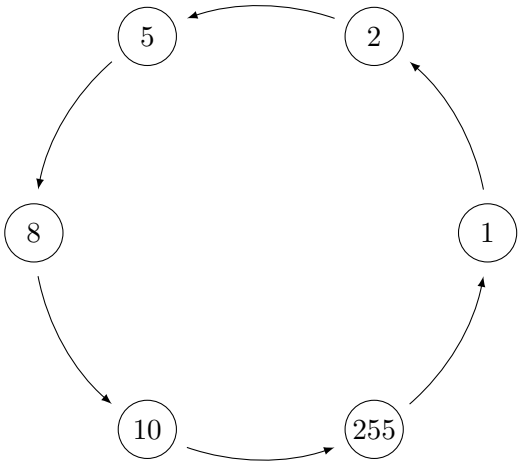

FIGURE 2.4: Transmission order of a network composed of five devices with IDs 1, 2, 5, 8, and 10. Note they transmit in order, and there is a dummy ID of 255 which serves as a marker for the end of the round.

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.

2.6 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.

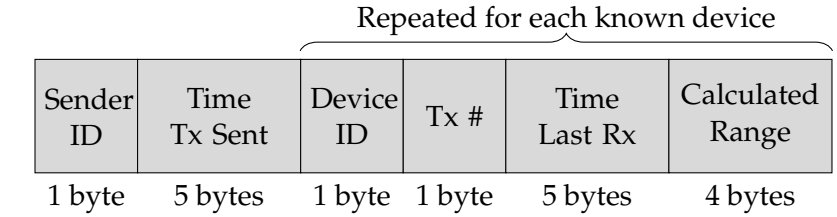

FIGURE 2.5: Data packet diagram for the rangefinding network.

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.

2.7 Range Information Protocol

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

- 1 byte for the ID of the transmitting node. The ID cannot not be 255.
- 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 a shared transmission counter. This counter is incremented by one whenever a message is received, and also incremented when one is sent. If a message is received or sent and the transmission counter on the device does not match the counter in the message, a transmission was lost. The message is thrown away and the transmission counter set to 0 to let the other device know there was an error on the next transmission.
- 5 bytes for timestamp of last received message from the device.
- 4 bytes for last calculated range to that device. Note that the two devices will each calculate slightly different ranges to each other.

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.

2.8 Summary

todo

Chapter 3

Rangefinding Hardware

3.1 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.

3.1.1 Bluetooth in Phones: A Failed Approach

(TODO REMOVE THIS OR APPENDIX IT WE SHOULD FOCUS ON OUR END PRODUCT AS PER AIDEN)

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 were able to use Bluetooth to form indoor positioning systems [2].

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.

3.1.2 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.

3.2 Design of the Hardware

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

3.2.1 The Microcontroller

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

- Others had used it with the DWM1000 and had good results [3].
- Group members had previous experience with programming Arduinos.
- It was inexpensive ($13 before tax).
- It worked off of 3.3V power, which was what the DWM1000 required. This obviated the need for voltage stepping.
- It was 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 had 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, though the USB cables required to connect the Arduinos ended up wiping out a lot of the cost savings.

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

3.2.2 Anchors

The anchors made for use with this project were made on a breadboard and based on the design by Thomas Trojer in his `arduino-dw1000` library [4]. A PCB design was included with this library which allowed a DWM1000 to be used with a breadboard. This PCB design was used in order to quickly prototype and determine whether the DWM1000 and Arduino would be feasible and meet the requirements for the project. Four anchors were constructed using these PCBs. An anchor can be seen in Figure 3.1.

As anchors are assumed to be stationary, and the project is intended to be used indoors, it was determined that the anchors could be powered via a standard electrical outlet. Designs using batteries were considered, but as the added flexibility in

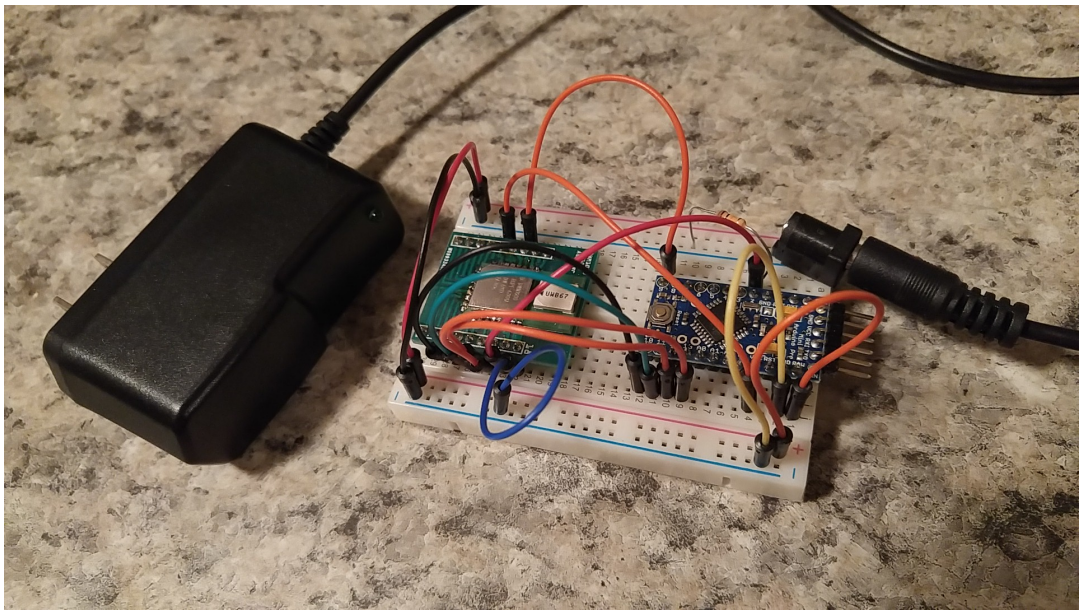

FIGURE 3.1: An anchor built on a breadboard using Thomas Trojer's PCB design [4]. The DWM1000 is on the left, and the Arduino is on the right.

placement of the anchors was of marginal benefit, and there was added complexity and maintenance required to deal with the batteries, the option was not pursued.

To save costs and time, the anchors' designed stayed constant throughout the project. There was only marginal benefit to designing a PCB, as their size was mostly irrelevant and their performance would not be improved.

3.2.3 Tags

The tags made for use with this project were required to be small enough to comfortably attach on a headset or cellphone. As a breadboard was too large, a custom PCB was designed.

Several designs were considered for the PCB connecting the Arduino and DWM1000. The most important factors were size and cost.

Several designs were considered at first:

- The first idea considered was to place the Arduino and DWM on top of each other, resulting in the smallest possible size. However, the physical dimensions of the two components made this impossible.
- Another possibility was to place each component on opposite sides of the PCB, but the Arduino's design demanded breakout headers to solder it into the board, which meant the PCB had to have holes drilled, which again meant that the physical dimensions of the two components interfered.
- Yet another 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

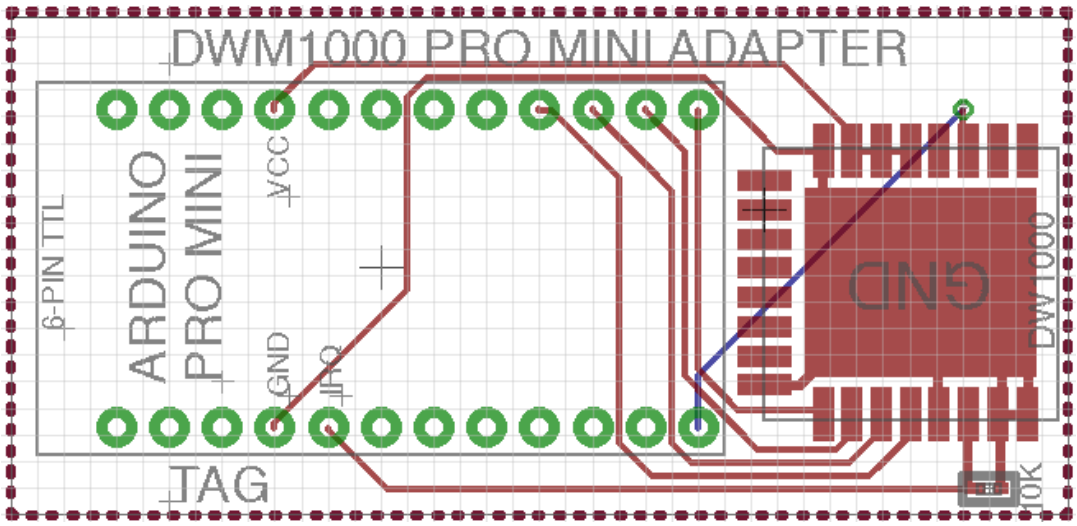

FIGURE 3.2: The design for the tag PCB in EAGLE.

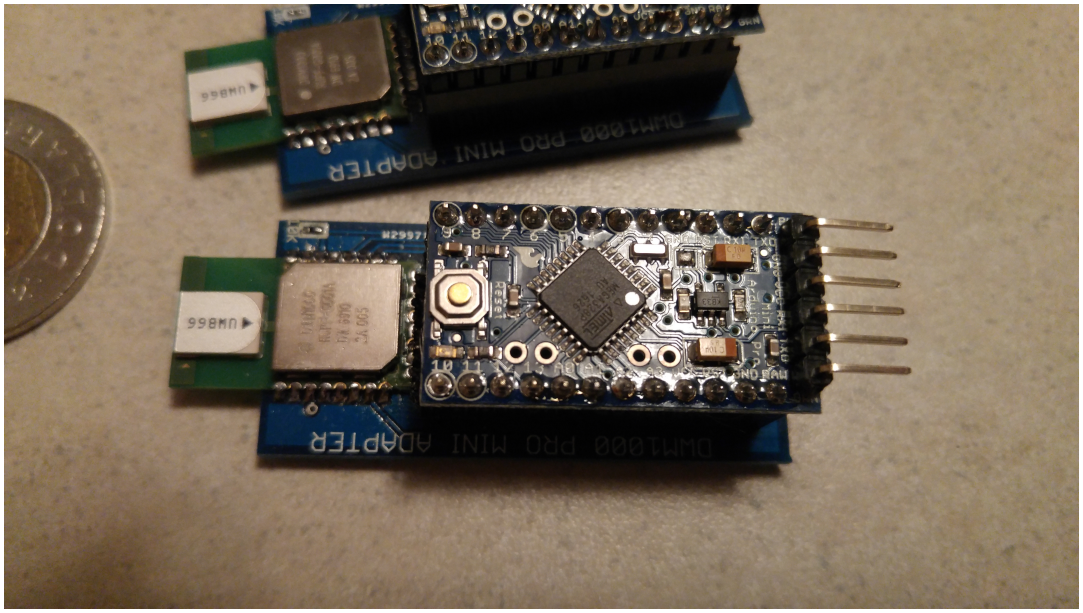

FIGURE 3.3: A soldered tag.

soldered to 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 the locations of their pins interfering with each other. This design was abandoned because the breakout pins would add a large amount of vertical length, it was twice as expensive, and because it was much more complex to design.

- The final idea considered was to have the Arduino and DWM1000 just be placed next to each other on the same side of a PCB. This design was physically realizable, the least expensive, and though it was not as compact as would be ideal, it was still relatively small.

The last idea was chosen due to the its cheap cost, simplicity of design, and satisfactory size.

As others who had used the DWM1000 recommended it [3], the PCB was designed so the antenna would not be on the PCB.

The PCB design was done in EAGLE and ordered from PCBWay. The PCB design can be seen in Figure 3.2 and a soldered tag can be seen in Figure 3.3.

3.3 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.

3.3.1 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.

3.4 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, the reader is advised to read the relevant sections in the DW1000 User Manual [1].

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 constan for the tags is roughly the same and was found to be 16470, and the antenna delay for the anchors is roughly the same and was found to be INSERT NUMBER HERE.

Changing the antenna delay in the main code required a slight tweak to the arduino-dw1000 library. A link to the code can be found in Appendix A.

3.5 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,

$$f_{op} = \frac{1}{T_{round}}$$

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:

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

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 3.3.1, 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,

[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.

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 3.6).

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

3.6 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 3.1 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 3.5. Predicted and actual values can be found in Table 3.2.

The max range between nodes was not determined due to space limitations but was at least 10m. The DWM1000 should get at least 60m if there are no obstructions **DWM1000UserManual**

TABLE 3.1: Accuracy of the rangefinding subsystem. Experiment performed with a tape measur and two tags lying on the ground. Antenna delay 16470.

Actual Distance (m)	**Mean Reported Distance (m)**	**Standard Deviation (m)**
0.5	0.62	0.045
1	1.08	0.025
2	2.24	0.03
3	3.21	0.022
4	4.22	0.104

TABLE 3.2: The frequencies of the rangefinding subsystem for various numbers of devices in the network. Experimented performed by placing an increasing number of devices in a network and calculating the duration between range reports from node 1 to node 2.

Number of Devices	**Round Time (µs)**	**Frequency (Hz)**
2	37000	27
3	58000	17.2
4	86000	11.6
5	93000	10.8
6	156000	6.4
7	170000	5.9

Chapter 4

Position Calculation

This chapter covers the position calculation subsystem, which takes the ranges from the rangefinding subsystem and calculates their positions. It also covers how the subsystem is integrated into the augmented reality subsystem. The position subsystem is not a separate piece of hardware. Rather, it is just a very complex mathematical function. An example of possible positions calculated from range data is shown in Figure 4.1.

The topics covered in this chapter are:

1. The infinite possible positions able to be calculated from a set of ranges.
2. A high-level example of calculating positions with basic trigonometry in 2D.
3. How this is extended to 3D.
4. The protocol used by the Arduinos to communicate with the cell phones of the users and transfer the ranging data.

4.1 Frame of Reference

One of the most important aspects to understanding the position of an object in space is knowing what it is *relative* to. If a point $p = (x, y, z) = (1, 2, 3)$, this is not useful unless it is known where, say, $(0, 0, 0)$ is and what directions the various axes point in.

The rangefinding subsystem determines the ranges between various devices. With these ranges, we can use trigonometry and determine positions in 3D space. However, it is impossible to place the positions in such a way that they correspond to the real world with only the range information. There are many different possible sets of positions which will result in the same rangefinding data. Figure 4.1 shows two possible sets of positions that will result in the rangefinding system reporting the same ranges.

In order to cut down on the infinite possible solutions to finding positions given only range data, this subsystem must arbitrarily create its own system of coordinates. The positions calculated by this subsystem must be further transformed so as to correspond to the real world. This transformation is not covered in this chapter, and instead is dealt with in Section 5.8.

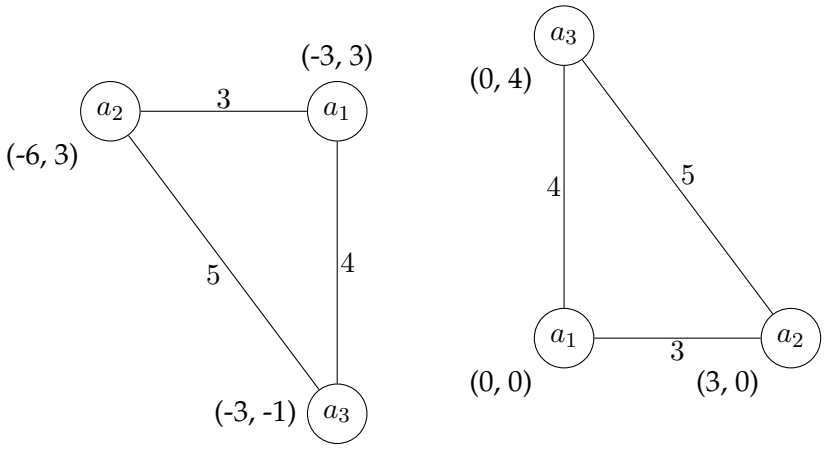

FIGURE 4.1: Two example networks with different positions reporting the same ranges.

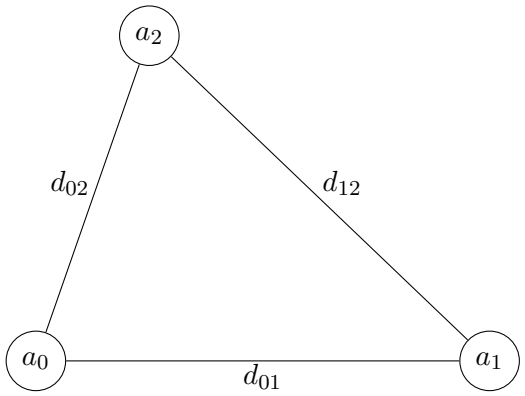

FIGURE 4.2: A network with 3 anchors a_i and the reported distances between them.

4.2 High-Level Example in 2D

As 3D position calculation is much more complex, position calculation is done in 2D will be explained first. We start with the distances between three anchors, a_0, a_1 and a_2. The distance between anchor i and anchor j is d_{ij}.

In order to assign positions in 2D space to them that keep them the same ranges from each other, we start by setting a_0 arbitrarily to be at the origin. Next, we arbitrarily declare a_1 to be on the x-axis, which puts it at the position $(0, d_{01})$. With these two positions, we can calculate the angle Θ between the x-axis and the vector from a_0 to a_2 with the cosine law:

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

With Θ, we can calculate x and y coordinates of a_2:

$$x = cos(\Theta)d_{12}$$

$$y = sin(\Theta)d_{12}$$

4.3 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.

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

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

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

$$z_{temp} = 0$$

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.

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

$$x_3 = x_{temp}$$

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

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

4.4 Integrating Rangefinding and Position Calculation

The position calculation code runs on the cellphone. In order to calculate positions, ranges must be transferred to the cellphone from the Arduino Pro Mini.

The Arduino's processor can communicate serially via UART TTL. Most cellphones do not have the ability to communicate serially, but Future Technology Devices International (FTDI) manufactures a USB cable that interfaces between the serial output of the Arduino and the micro USB port of a cellphone. With this cable connecting an Arduino and a cellphone, the cellphone can provide power to the Arduino and send and receive data to it.

The project uses the Android `D2xx` library, provided by FTDI, to communicate with the Arduino while running an Android app. It allows the easy sending and receiving of bytes.

4.5 Protocol

There are a number of design considerations when dealing with the communications between the cellphone and the Arduino:

1. The stream of data may begin in the middle of a message, as the buffer storing the serial data sent from the Arduino has only a limited capacity. If a cellphone were connected to an anchor that had already been running for some time, it is quite likely that the buffer would have overflowed. So, the protocol must include a way to determine the start of a message.

2. The protocol should be human readable for easy debugging.

With this in mind, a simple protocol was developed. Lines of text, delineated by newline characters, are sent. If the Arduino is reporting the range between two devices, it sends a line with the format:

```
!range <from ID> <to ID> <range in meters>
```

If the Arduino is informing the cellphone of its id, it sends a line with the format:

```
!id <tag's ID>
```

If the cellphone reads a line which does not start with '!', the line is assumed to either have started in the middle of a ranging information line, or else debug data has been sent. Either way, the line is discarded and the system waits to read a new line.

An example of the output from the Arduino is:

```
DW1000 initialized ...
Committed configuration ...
New device found.  ID: 1
Transmission received from tag 1 with transmission count 1
!range 1 2 0.00
Transmission received from tag 1 with transmission count 3
!range 2 1 3.26
!range 1 2 91659.28
Transmission received from tag 1 with transmission count 5
!range 2 1 3.48
!range 1 2 3.38
Transmission received from tag 1 with transmission count 7
!range 2 1 3.37
!range 1 2 3.42
```

Note the briefly extremely large range while the timestamps settle.

Parsing is done on the cellphone with the `java.util.Scanner` class. The parsing code can be found in (INSERT THIS LATER).

4.6 Summary

This chapter covered the issues with the infinite possible positions that can be calculated from a set of ranges between nodes, the math behind calculating positions in 3D given ranges, and how the cellphones and Arduinos are interfaced.

Chapter 5

Augmented Reality

This chapter covers the AR portion of the project. The AR subsystem takes the positions from the position calculation subsystem and renders them on the screen of a cellphone using OpenGL. If they cannot be directly rendered, an arrow is drawn on the edge of the screen to show which way the object is from the user. An example of this can be found in Figure **??** (DO THIS FIGURE).

The code for the AR subsystem can be found at (PROVIDE LINK HERE).

This chapter covers the following topics:

1. A brief overview of the math used later in this section, including homogenous coordinates and transformation matrices.
2. A description of OpenGL and the basics of how it is used.
3. How objects and text can accurately be drawn on the screen overlaid on a camera image.
4. How the coordinate system of the position calculation system can be transformed into real world coordinates through the cell phone's sensors.
5. Examples of the accuracy of the subsystem.

5.1 3D Math Overview

This section covers the basics of using matrices to render 3D scenes. A full treatment of the subject is beyond the scope of this report, though other resources on the subject exist (PUT CITATION HERE). The reader is assumed to be familiar with some linear algebra, including the multiplication of matrices.

5.1.1 Transformation Matrices

Transformation matrices are matrices which can describe transformations in space such as translation, rotation, and scaling. They are regularly used in computer graphics.

Figure 5.1 shows some of the transformations that can be applied to vertices with matrices.

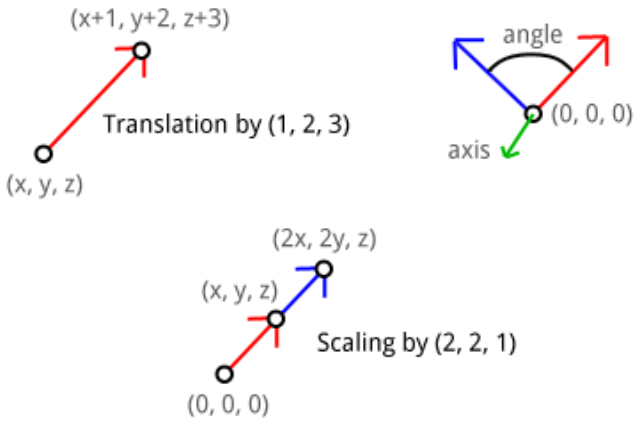

FIGURE 5.1: Translation, rotation, and scaling of vectors are all possible with matrices [5].

Transform matrices can be composed with multiplication. For example, one could describe the translation of a point with a translation matrix, then multiply the matrix by a rotation matrix to create a new matrix which, when multiplied by a vector, cause that vector to be rotated and then translated.

Order matters when composing transformation matrices. The matrix $\mathbf{TR}$ is not the same as $\mathbf{RT}$.

Normally, the transformation of a vector v to a new vector v' with a transformation matrix $\mathbf{M}$ is written as:

$$v' = \mathbf{M}v$$

Or, with two transformation matrices $\mathbf{M_1}$ and $\mathbf{M_2}$ composed:

$$v' = \mathbf{M_2}\mathbf{M_1}v$$

When written like this, the transformation matrix $\mathbf{M_1}$ can be viewed as applying to the vector first, and then the new transformed will be further transformed by $\mathbf{M_2}$.

In this report, points and vectors will be used interchangeably. When a matrix is said to multiply a point $p = (x, y, z)$, it should be considered as multiplying a vector $v = \begin{bmatrix} x & y & z \end{bmatrix}^T$.

The definitions of matrices which scale, translate, and rotate are mostly beyond the scope of this report. OpenGL provides utility functions to create them, so the exact definitions are unnecessary.

5.1.2 Homogenous Coordinates

For a given 3×3 matrix M and an arbitrary point in 3D space $p = (x, y, z)$, there is no $\mathbf{M}$ such that will multiplying it by any p will cause p to be translated by a specified number of units. There is, however, a way to do it with a homogenous point and a 4x4 matrix. This is one of the reasons homogenous coordinates are used in OpenGL.

Homogenous coordinates are coordinates in space which contain an extra element w which acts as a scaling factor on the three elements. A 3D homogenous coordinate p_h could be written as:

$$p_h = (x_h, y_h, z_h, w)$$

This relates to a regular point in 3D space:

$$(x, y, z) = (\frac{x_h}{w}, \frac{y_h}{w}, \frac{z_h}{w})$$

For example, the homogenous point $(1, 1, 1, 1)$ maps to the regular 3D point $(1, 1, 1)$. The homogenous point $(2, 2, 2, 2)$ does as well.

A translation matrix T that moves a point p by (T_x, T_y, T_z) units, is:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To prove this, we'll multiply out a homogenous point $p_h = (x_h, y_h, z_h, w)$ by $\mathbf{T}$ and show that, when converted to a regular 3D point, it equals $(\frac{x}{w} + T_x, \frac{y}{w} + T_y, \frac{z}{w} + T_z)$.

$$p_h\prime = \mathbf{T} p_h$$

$$p_h\prime = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_h \\ y_h \\ z_h \\ w \end{bmatrix}$$

$$p_h\prime = \begin{bmatrix} x_h + wT_x \\ y_h + wT_y \\ z_h + wT_z \\ w \end{bmatrix}$$

Now, if we convert $p_h\prime$ to a regular 3D point p, we get:

$$p = (\frac{x_h}{w} + \frac{wT_x}{w}, \frac{y_h}{w} + \frac{wT_y}{w}, \frac{z_h}{w} + \frac{wT_z}{w})$$
$$p = (x/w + T_x, y/w + T_y, z/w + T_z)$$

which is the translation of p by (T_x, T_y, T_z) units.

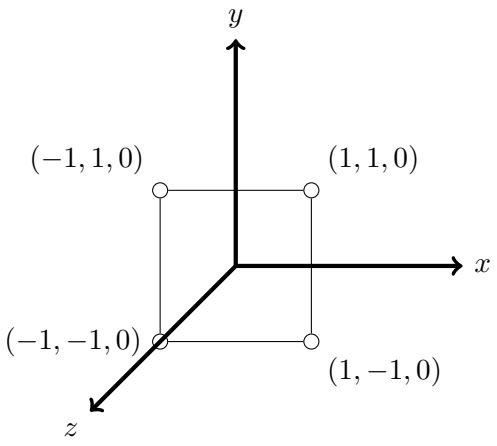

FIGURE 5.2: An example 3D model of a square in model coordinates.

5.2 OpenGL

OpenGL is an API used to render 2D and 3D graphics. For the project, OpenGL was used to render position markers in 3D space overtop what the cell phone's camera sees.

OpenGL works by taking in collections of points making up a 3D model of an object and then applying various matrix transforms to the points, translating and rotating them. Afterwards, it projects them onto a 2D plane and renders them. A full treatment of how OpenGL works is beyond the scope of this report, but (INSERT CITATION HERE FOR OPENGL BOOK).

The three main matrices that OpenGL uses to render a 3D scene are called the **projection matrix**, **view matrix**, and **model matrix**. This project uses several novel techniques which use and manipulate the matrices that OpenGL uses to render a 3D scene. This report will only cover them briefly, though other resources explore them more fully [6].

5.3 The Model Matrix

The model matrix exists to take **model space**, in which the coordinates of a 3D model are relative to the center of the model, and transform it into **world space**, in which coordinates are relative to the center of the "world". Most of the code in this project considers the locations of objects in world space. For example, the position subsystem might say that a tag is at $(5, 5, 1)$. This coordinate is considered to be in world space, though it should be noted that this is not the same as *real* world coordinates - a further transform is required to match up the positions given by the position calculation subsystem with the real world as shown by a camera.

Figure 5.2 shows an example of a square in model space. The square's 3D model has its vertices placed at $(-1, -1, 0)$, $(1, -1, 0)$, $(1, 1, 0)$, and $(-1, 1, 0)$. An example of creating a 3D scene with two squares at $(5, 0, 0)$ and $(-5, 0, 0)$ in world coordinates

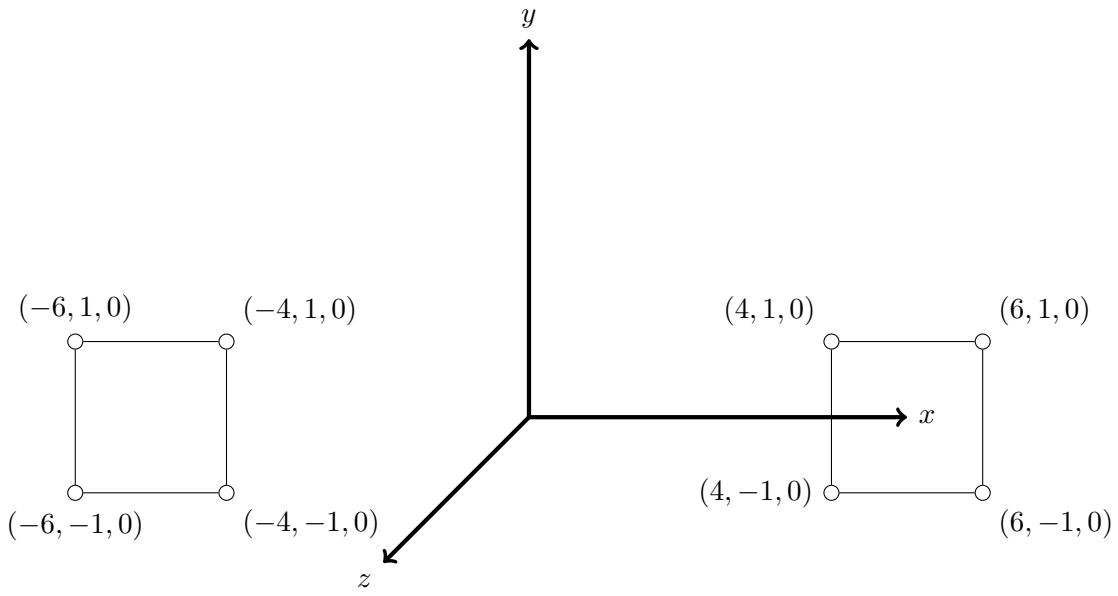

FIGURE 5.3: A 3D scene with two squares, the centers of which are placed at $(5, 0, 0)$ and $(-5, 0, 0)$ in world coordinates by transforming the vertices of Figure 5.2 with two model matrices.

can be seen in Figure 5.3. A model matrix is created for each of the squares. The first model matrix encodes a translation $(-5, 0, 0)$ units and the second model matrix encodes a translation of $(5, 0, 0)$ units. To get the vertices of the first square in world coordinates, the first model matrix is applied to the square's vertices. The same is done for the second square and model matrix.

That is, for the model matrix $\mathbf{M}$ and each vertex v of the square, the vertex v is transformed into world coordinates vertex v' by the formula:

$$v' = \mathbf{M}v$$

5.4 The View Matrix

In a 3D scene, a position is declared from which the scene is viewed. The view matrix exists to transform world coordinates to **view space**, in which coordinates are relative to the position from which the scene is viewed. It as if an imaginary camera has been placed in the scene. Figure 5.4 shows an example of this.

The view matrix is typically just a simple translation of all the vertices (if a camera is placed at $(5, 5, 5)$, then all the vertices will be translated by $(-5, -5, -5)$), and then a rotation based in which direction the camera is looking at.

For the rest of this project, the imaginary camera is assumed to be located at (0, 0, 0), making the view matrix a simple rotation matrix. The positions of markers and objects are calculated relative to the position of the tag connected to the cellphone.

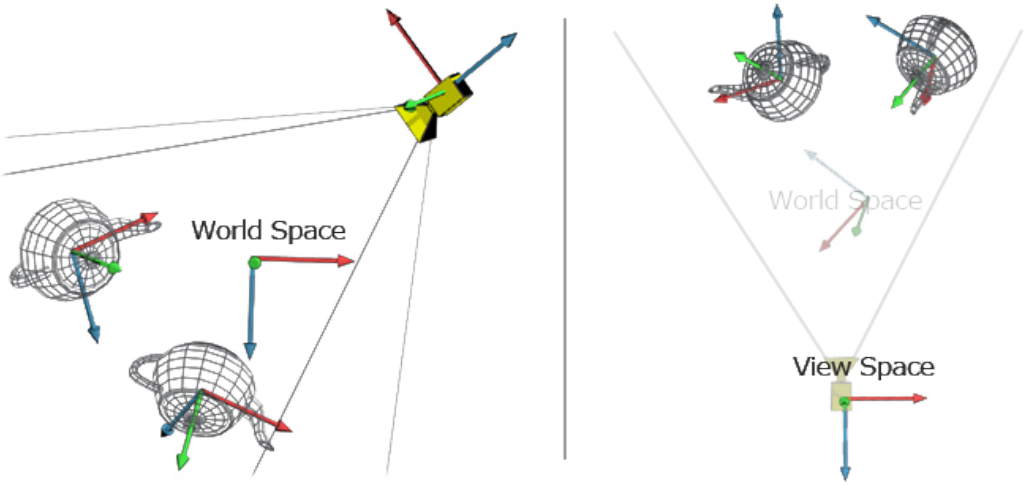

FIGURE 5.4: A figure showing an object in world coordinates and an object in camera coordinates [6].

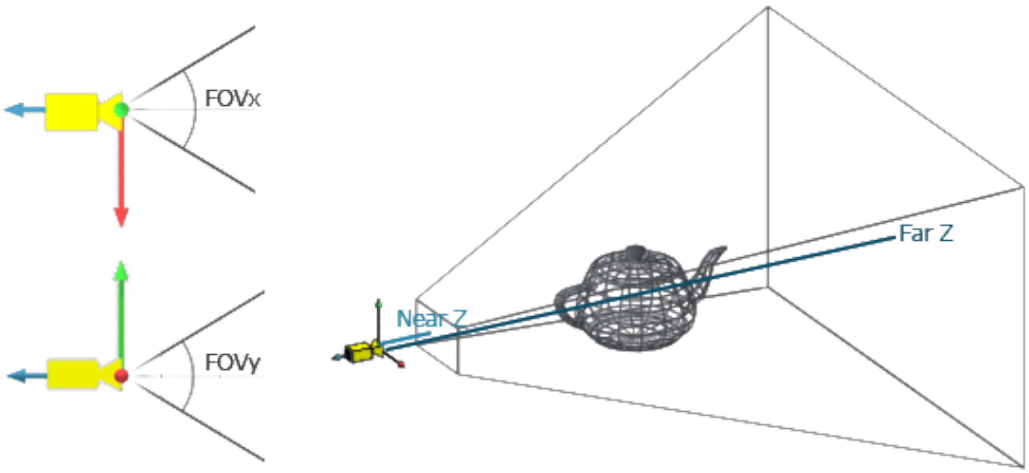

FIGURE 5.5: The transformation from view space to projection space [6].

5.5 The Projection Matrix

In one of the last steps in rendering a 3D scene in OpenGL, the scene must be projected onto a plane so it can be displayed on a monitor. The projection matrix is involved in this process. View space is transformed into **projection space**, in which coordinates are transformed into **normalized device coordinates** (NDCs). Vertices which fall within the field of view will be contained within a cube with corners $(-1, -1, -1)$ and $(1, 1, 1)$ and are drawn. Vertices outside this box are culled.

There are different types of projection matrices. For the purposes of this projection, only the **perspective projection matrix** is considered. This projection takes into account that objects which are further away from the camera should appear smaller. Figure 5.5 and Figure 5.6 show the transform from view space to projection space via a perspective projection matrix.

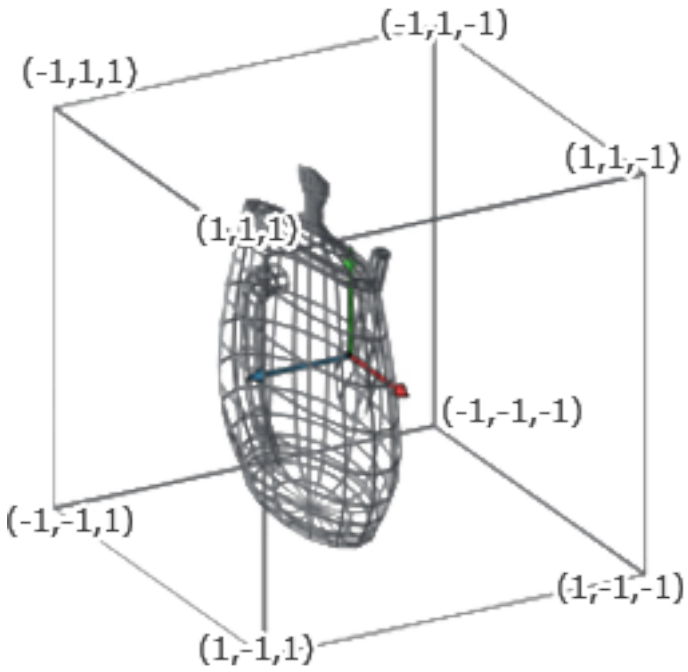

FIGURE 5.6: A teapot in projection space [6].

To actually render the image in 2D, the NDCs are converted to the coordinates of pixels as determined by the size of the surface being rendered to. The z coordinates handle depth, which allows OpenGL to determine whether a shape is on top of another during the rendering process.

5.6 Camera Perspective and OpenGL Perspective

In order to accurately display the locations of the devices, it is important that the 3D scene rendered by OpenGL match up with the real world as shown by the cellphone's camera. Figure 5.8 shows how a camera can only see see a limited part of the world, as dictated by its field of view.

As previously shown in Figure **??**, the projection matrix has a field of view which can be changed. The field of view for the OpenGL perspective matrix needs to be set on a per-cellphone basis to match that of the cellphone's camera for the project to display locations accurately.

(TODO FINISH THIS UP LATER WITH MORE DETAIL IF TIME PERMITS.)

5.7 Cellphone Rotation

In order to display the positions of markers on the screen correctly, the direction the camera is facing must be known. The cellphone can determine this by requesting the phone's rotation from Android's Sensor API. Android does not specify *how* the

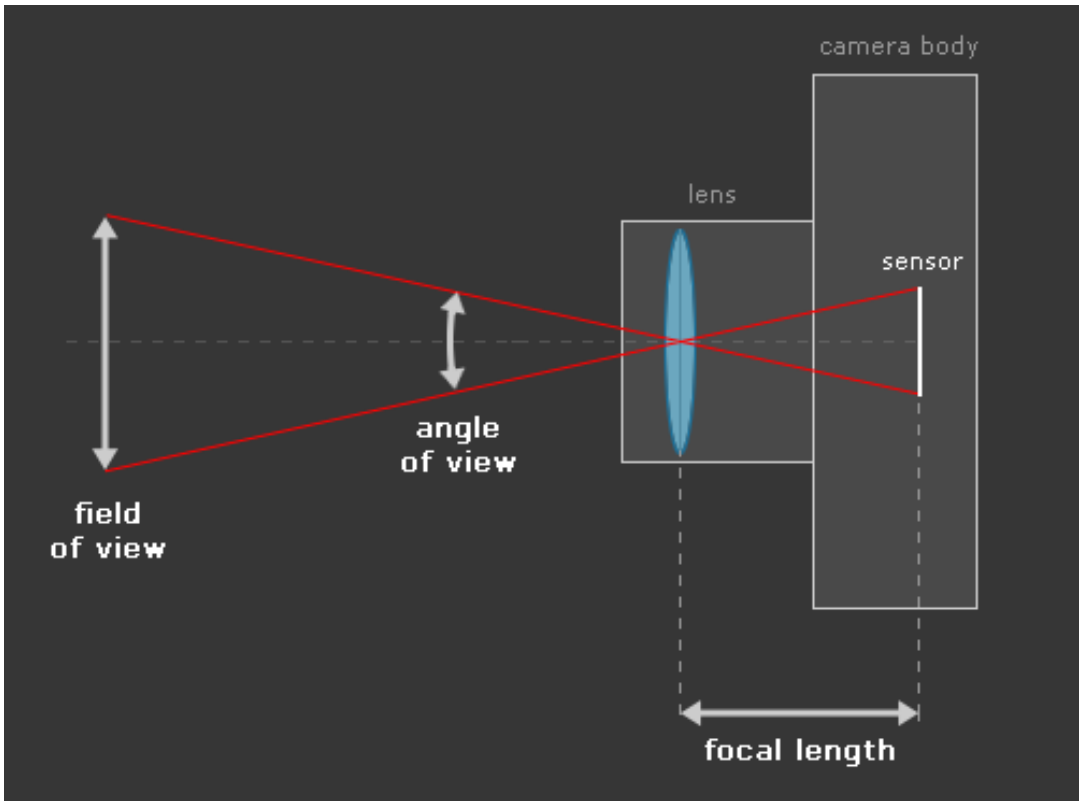

FIGURE 5.7: A diagram depicting the field of view of a camera [7].

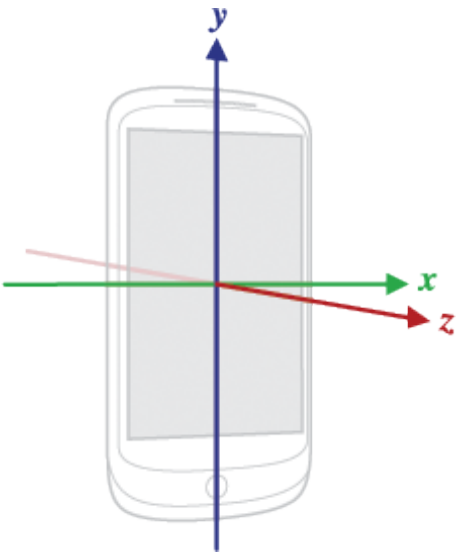

FIGURE 5.8: The coordinate system used by the Android Sensor API [8] and by AR subsystem, assuming the phone's top points towards geomagnetic North and the screen points toward the sky. For the purposes of rotation, X points approximately East, Y to geomagnetic North, and Z towards the sky [9].

rotation of the phone is determined, but in practice uses a subset of the magnetometer, accelerometer, and gyroscope (if the phone contains these sensors; Android does not force phone manufacturers to include any of these sensors).

For example, if the phone has a magnetometer and an accelerometer, by determining the direction of magnetic north and the direction of the force of the gravity on the phone, the direction of the phone's rotation can be determined [9].

The cellphone has its own coordinate system which must be taken into account. To deal with the coordinate system differences, the Android Sensor API includes a method to remap the coordinate system of the rotation it reports. The coordinate system used by the project for its real world coordinates was chosen to match the coordinate system of the Android Sensor API, which can be seen in Figure **??**.

The rotation value returned by the Android OS was found to be quite accurate in tests so long as the device was not in motion. If the device was in freefall or moving, it became more difficult for the Android OS to determine the direction of gravity, which in turn made the calculated rotation values inaccurate.

The Android Sensor API can directly return a rotation matrix. However, the way it is stored in memory is not quite the same as how OpenGL stores matrices, so the matrix must be transposed before being used in OpenGL.

The rotation matrix returned is used in the project to create the view matrix by taking a "lookAt" vector equal to $(0, 0, -1)$ (which represents that the camera, when at 'rest' on a table looks down the z axis) and "up" vector equal to $(0, 1, 0)$ (arbitrarily stating that the 'up' direction of the cellphone screen points north) and rotating each by the rotation matrix returned by the phone. The new vectors produced are then used in the OpenGL utility function `setLookAtM` which creates a view matrix based on an up vector and the point at which the camera should be looking at.

The rotation matrix could have been used directly as the view matrix without going through these steps. However, for later processing it is necessary to have the lookAt and up vectors rotated.

Thus, if we ignore the transformation required to correct for the arbitrary coordinate system of the position calculation subsystem, the view matrix, $\mathbf{V}$, every frame is to set it equal to the rotation matrix the Android OS last returned, $\mathbf{R_{cur}}$:

$$\mathbf{V} = \mathbf{R_{cur}}$$

5.8 Calibration

As noted in Section 4.1, the positions calculated by the position calculation subsystem have an arbitrary coordinate system where the XY plane is formed by the positions of the three anchors with the lowest ID and the Z axis is determined by the fourth anchor. In order to display positions obtained from this subsystem accurately on top of the images captured by the cellphone's camera, there is a necessary

calibration step on system startup to create a transformation matrix which can convert coordinates into the cell phone's coordinate system. The user determines the transformation needed to convert to the coordinate system used by the cellphone by rotating and positioning the phone until it displays the position of a device correctly, at which point the system knows the transformation needed.

The steps a user follows to calibrate the system are:

1. The user taps the phone's screen to enter calibration mode.
2. The system creates a new view matrix every frame which will cause the screen to display the position of an arbitrary device in the network in the middle of the screen.
3. The user then points the phone directly at the tag being displayed and rotates the phone until the position of a second tag matches its location on the screen.
4. The user then taps the screen to end calibration mode. The system stores the view matrix at this time, $\mathbf{V}_{end}$, as well as the raw rotation matrix at the time, $\mathbf{R}_{end}$.
5. If the locations of other tags are incorrect due the z-axis being flipped (the z axis of the position calculation subsystem flips based on the elevation of the anchor with the highest ID), the user taps the phone's screen twice to correct it. The system stores this in a boolean and, when calculating the positions of other devices relative to the user's location, knows whether to flip the z component.

(TODO INSERT PICTURE HERE)

Each frame, to correctly position the objects, the calibration matrix should be changed only by the rotation of the device from its rotation at the end of the calibration. Before, each frame the view matrix was set equal to the current raw rotation matrix, $\mathbf{R}_{cur}$. Now, the view matrix must multiplied by a matrix to correct for the coordinate transform. Intuitively, this calibration matrix should reverse the effects of the rotation at the end of the calibration mode and multiply by the view matrix at the time. A rotation can be reversed by inverting the matrix. Thus, the new calibrated view matrix used every frame, $\mathbf{V}_{cal}$, is:

$$\mathbf{V}_{cal} = \mathbf{V}_{end}\mathbf{R}_{end}^{-1}\mathbf{R}_{cur}$$

5.9 Billboarding

So far, this chapter has discussed how a collection of vertices can be transformed, and more specifically how the positions of other devices can be correctly placed. This would be sufficient if the system had to render a cube at the position of the object in question. Aesthetically, rendering a cube would leave a lot to be desired. This section covers a method to draw a 2D image at a 3D location.

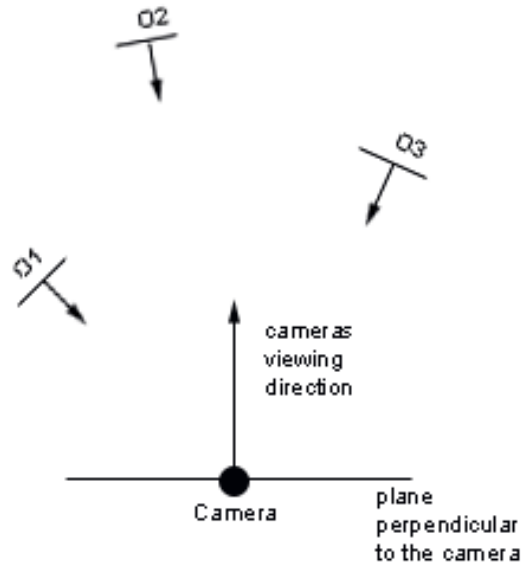

FIGURE 5.9: Billboarding used to make objects O1, O2, and O3 face the camera [10].

Billboarding is a technique to make an object face the screen no matter where it is situated. It is used, for example, in some 3D games to render trees as 2D images to lower the number of vertices that need to be processed in a scene. Figure 5.9 shows the end result of using billboarding for 2D objects.

There are many ways to do billboarding, depending on whether you want to only have the object rotate on a specific axis [10]. For this project, it was determined that the best way to represent an object's position on the screen was to have a rotating circular 2D image as a marker. The exact graphic used was chosen for aesthetic reasons.

To have this 2D image always face the screen, it must be able to rotate on all three axes. The technique to accomplish this rotation is to multiply the model matrix before it is translated by the inverse of the portion of the view matrix corresponding to rotation. That is, given a model matrix $\mathbf{M}$, making it rotate to always face the camera is accomplished by creating a new billboarding model matrix $\mathbf{M}_b$:

$$\mathbf{M}_b = \mathbf{M}\mathbf{V}^{-1}$$

Intuitively, when a model's vertices are multiplied by the view matrix, they are rotated about depending on the angle the camera is facing. If we reverse this rotation effect but not the translation part, the model will always face the camera.

As the camera is always situated at $(0, 0, 0)$ in this project, multiplying the model matrix of the textured square by the inverted view matrix will produce a billboarding effect. If we first rotate the square about the Z-axis (assuming the square's vertices are on the XY plane), it will produce the rotating effect on the final image.

5.10 Marking the Positions of Objects Off-Screen

One of the advantages of AR is the ability to put more information on the screen than can be seen with the human eye. As part of the goal of the project to increase the user's awareness of the locations of objects, the system informs the user of where an object is relative to them when the object is not within the viewing area of the camera.

First, the system needs to determine whether or not the objection is within the viewing area. This is done by multiplying the object's position in space relative to the camera by the view and projection matrices. That is, given a position p, we find the NDCs of p, p_{NDC}, by:

$$p_{NDC} = \mathbf{PV}p$$

If the point is outside the bounding box from $(-1, -1, -1)$ to $(1, 1, 1)$, then the point is known to not be visible on the screen. Or, since p_{NDC} is in homogenous coordinates, we can check whether any of the components are greater than or less than w.

In code:

```
//Start by determining whether or not this is on the screen
float[] p = {x, y, z, 1}; //w=1 by default
float[] pNDC = new float[4];
 Matrix.multiplyMV(pNDC, 0, vpMatrix, 0, p, 0);

boolean onScreen = true;
for (int i = 0; i < 3; i++) {
        //if v outside the bounds -w <= x_i <= w
        if (pNDC[i] <= -pNDC[3] || pNDC[i] >= pNDC[3]) {
                onScreen = false; //is clipped
        }
}

//onScreen is now false if the point is off the screen
```

The system displays an arrow on the edge of the screen pointing towards where the object is. To determine the angle and the part of the screen where the arrow should be drawn, we continue to use p_NDC.

The direction of the arrow can be determine by the vector from the center of the screen, $(0, 0, 0)$ in NDCs, to p_{NDC}. The z component is ignored, and we only look at the xy components of the point. Then, the edge of the screen where the arrow should go is given in NDCs as where that vector intersects the bounding box from $(-1, -1)$ to $(1, 1)$. This intersection is calculated by dividing p_{NDC} by the larger of its

components, so that the magnitude of the larger component is 1 and the magnitude of the smaller component is some number smaller than 1. The new point is p_{edge}.

The point is transformed back into world coordinates by multiplying the inverse of the view-projection matrix by it. Then, the square textured with an arrow image on it is rotated an amount determined by where the point is on the screen so it will point in the same direction as the vector from the NDCs $(0, 0, 0)$ to p_{NDC}.

In code:

```
float divisor = Math.max(Math.abs(x), Math.abs(y));
//homogenous component set to 1, z value determines how large the arrow is lat
float[] edgeVector = {x/divisor, y/divisor, 0.6f, 1};

//Rotate based on vector in NDC, then rotate on inverted view matrix so it fac
float[] rotateToFaceEdgeMatrix = new float[16];
Matrix.setIdentityM(rotateToFaceEdgeMatrix, 0);
float angle = (float)Math.atan2(edgeVector[1], edgeVector[0]);
Matrix.rotateM(rotateToFaceEdgeMatrix, 0, angle*180f/(float)Math.PI, 0, 0, 1f)
//correct so the right edge is on the right border
Matrix.translateM(rotateToFaceEdgeMatrix, 0, -0.5f, 0, 0);

//Face camera...
float[] rotationMatrix = new float[16];
Matrix.multiplyMM(rotationMatrix, 0, invertedViewMatrix, 0, rotateToFaceEdgeMa

//Convert from NDC to world coordinates via inverted VP matrix, translate matr
float[] worldCoords = new float[4];
Matrix.multiplyMV(worldCoords, 0, invertedVPMatrix, 0, edgeVector, 0);
Math3D.fixHomogenous(worldCoords); //w may be non-1

float[] translationMatrix = new float[16];
Matrix.setIdentityM(translationMatrix, 0);
Matrix.translateM(translationMatrix, 0, worldCoords[0], worldCoords[1], worldC

float[] modelMatrix = new float[16];
Matrix.multiplyMM(modelMatrix, 0, translationMatrix, 0, rotationMatrix, 0);

return modelMatrix;
```

An example of the rendered off-screen arrow can be seen in (TODO).

5.11 Results

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

5.12 Conclusion

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

Chapter 6

Budget

The budget for the project can be found in Table 6.1.

As a whole, the cost of the project exceeded the proposed budget of $285. This was due to the fact that the actual cost of the DWM1000 chips were much more than expected including shipping and taxes. Also, the PCBs were made in China, which drove shipping costs up. The creation of more tags was necessary to get more data points available for the devices to pick up. It was a necessary expenditure to expand the network beyond the original four anchor tags to bring to a total of seven. The budget only consisted of $226.24 from the ECE department, the wondrous donation from Dr. McLeod for all out of pocket expenses was greatly appreciated.

During the compilation of all expenditures for the project, $1.08 was not accounted for. It must have been wrong (yet very minor) calculation, or a missing cent or two in taxes. The documentation of the expenditures is something to improve on for further progress, but nothing of major importance was missing.

What was interesting to note that the Arduino chips costs $12.76 each only for the first order. The second order of the chips that price increased to $26.56 per chip, but that was because additional materials for soldering the chips were included in that cost as well.

TABLE 6.1: A table listing the project expenses.

Item	**Quantity**	**Price per unit ($)**	**Total Price of Item ($)**
Pro Mini Arduino Microcontroller 3.3V$_D$	4	12.76	51.04
Wall Adapter Power Supply 5 V DC 2 A$_D$	4	7.63	30.52
2.1mm Barrel Jack Adapter - Breadboard Compatible$_D$	4	1.22	4.88
400 Tie Point Interlocking Solderless Breadboard$_D$	4	4.76	19.04
FTDI USB-to-TTL (Serial) Cable 3.3V$_D$	2	23.01	46.02
Break Away Headers - Straight$_D$	4	1.92	7.68
66 x 22 Gauge Assorted Jumper Wires$_D$	1	5.07	5.07
40 pin Breaker Header - Right Angle$_D$	1	2.50	2.50
USB-A to Micro USB Adapters$_M$	4	6.99	27.96
DWM1000 chips (1st Order - Anchors)$_D$	4	37.96	151.83
DWM1000 chips (2nd order - Tags)$_D$	3	37.09	111.27
10K SMD Resistors$_D$	25	0.0116	0.29
Google Cardboard Headset$_M$	1	26.72	26.72
PCB (1st Order)$_D$	1	48.04	48.04
Arduino Chips (Tags and soldering parts)$_D$	3	26.56	79.68
PCBs (2nd Order)$_D$	2	20.00	40.00
		Duty Taxes	23.84
		Shipping	33.00
		Taxes (Total)	27.54
		Grand Total	736.92

Note: D, L and M specify which group member bought which item: D for Drew, L for Llandro, M for Maricar.

Chapter 7

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

As Decawave did not provide a proof for their double-sided two-way ranging formula, others have independently done so [11].

Clock drift is an issue where a clock does not run at the same rate as the reference clock, the clock gradually 'drifts' apart and desynchronizes from the other clock. The main objective here is to get rid of the clock drift difference between the nodes which is a major source of error for us. 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:

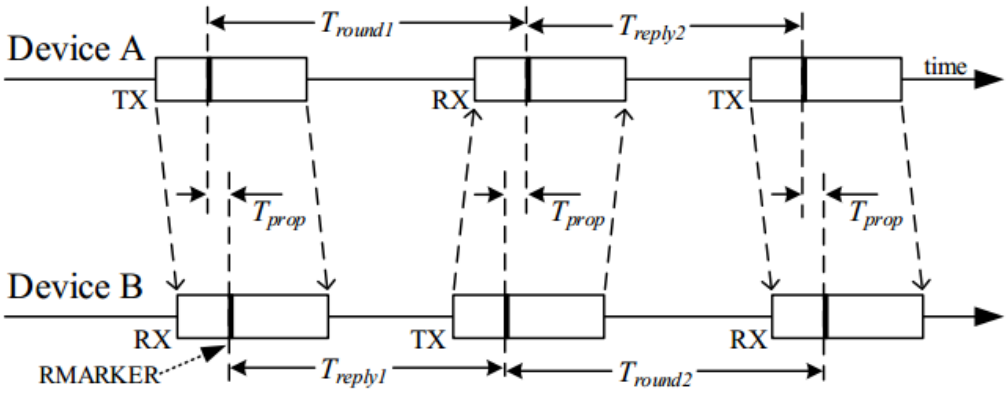

FIGURE B.1: Double-sided Two-way ranging with three messages[1].

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

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

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

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

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

[1] Decawave, "DW1000 User Manual", version 2.10, no., 2016. [Online]. Available: http://www.decawave.com/support.

[2] A. Bekkelien, "Bluetooth indoor positioning", no., Mar. 2012. [Online]. Available: https://pdfs.semanticscholar.org/1919/037541b4a84d0b5a6dd4d425c0891282ae8f.pdf.

[3] G. Nordahl. (2016). LPS Mini, [Online]. Available: https://hackaday.io/project/7183/logs.

[4] T. Trojer. (2016). arduino-dw1000, [Online]. Available: https://github.com/thotro/arduino-dw1000.

[5] ().

[6] C. Labs. (2017). Coding Lab: World, View, and Projection Transform Matrices'.

[7] M. Pot. (2017). Understanding Your Camera: Focal Length, Field of View and Angle of View Defined.

[8] Google. (2017). Sensors Overview.

[9] ——, (2017). Motion Sensors.

[10] A. R. Fernandes. (2017). Billboarding Tutorial.

[11] arnaud. (2016). DWM1000 DS-TWR method calculate distance method?, [Online]. Available: https://forum.bitcraze.io/viewtopic.php?t=1944#p9959.