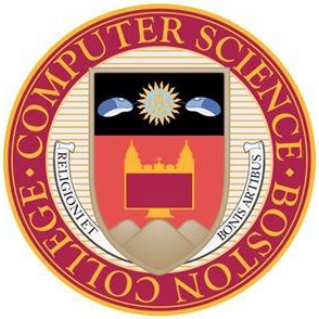


Boston College Computer Science

Local Sonic Positioning System

A Senior Honors Thesis
by
Steven Valeri

Advisor: Professor William Ames

May 5, 2014

Abstract

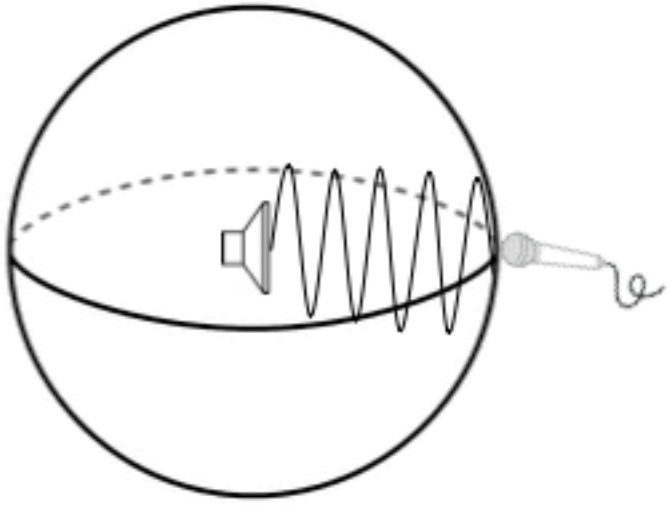
Based on recent innovations such as the Nintendo Wii and the XBox Kinect, gaming companies are looking to improve and drastically alter the way users interact with games. The Kinect uses cameras and image recognition algorithms to represent a user's body motions as the movement of virtual objects in 3D space. The Wii uses accelerometers and gyroscopes for the same purpose. This thesis project explores the possibility of using a process similar to GPS to locate the position of an object and move it around in a virtual 3D environment. A GPS receiver can calculate its position because it can determine how far away it is from multiple satellites. This thesis proposes that given three speakers and a microphone, the distance between one of the speakers and the microphone can be calculated from the time it takes sound to travel from the speaker to the microphone. The position of the microphone can then be calculated by finding the distance between the microphone and each of the three speakers. By locating the position of the microphone consistently, the changes in position can be used as input for a video game, resulting in the movement of an object in a virtual 3D space.

Intro

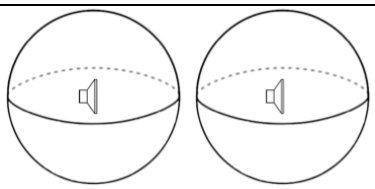
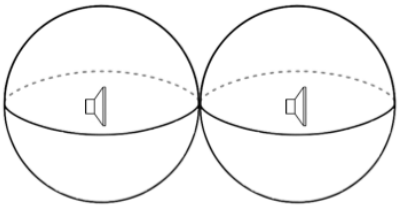
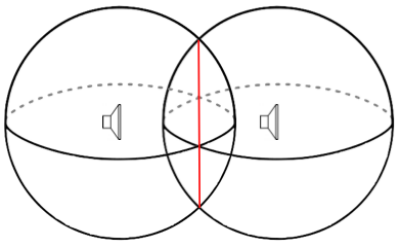
<u>The Basics:</u>

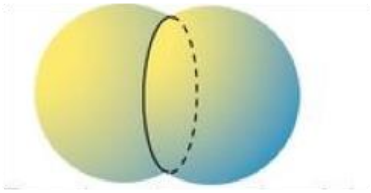
This location positioning system works relatively similar to a GPS, but instead of satellites and a GPS receiver, this system uses speakers and a microphone. This is possible because computers and microchips can function faster than the speed of sound. Imagine that you have a speaker and a microphone. The position of the speaker is known and is not moving and the position of the microphone is completely unknown. When the speaker beeps, there will be a certain time delay from when the beep is played on the speaker until the microphone hears the beep. This time delay is directly proportional to the distance the microphone is away from the speaker. The time delay can be multiplied by the speed of sound in order to find the distance in between the speaker and the microphone.

The distance between the microphone and the speaker can act as the radius of a sphere with the coordinates of the speaker acting as the center of the sphere. This means that in 3D space, the microphone could be located anywhere on the sphere's surface.

Sphere modeling possible coordinates of the microphone


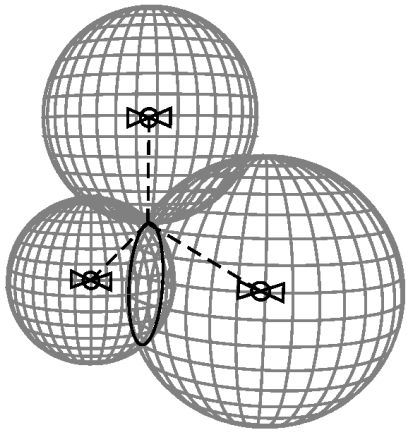
This clearly is not enough data to model the position of an object in 3D space. Imagine however that there is another speaker close enough for the microphone to process its sound. The position of this speaker is also known and not changing. Once the first speaker plays and the microphone picks up its beep, the second speaker beeps instantly and the microphone also picks up the second beep. Again, the time delay for each of these beeps to reach the microphone can be measured which allows for the distances between each speaker and the microphone to be calculated. Taken independently, this creates two spheres in 3D space where the microphone could be located. Taking the two spheres together, it is relatively apparent that the microphone can only be positioned at the intersection of these spheres. The intersection of these two spheres can exist at one point, exist along a circle, or not exist at all. In this implementation, a non-existent intersection results from a miscalculation in any of the time delays. Another possible result is that there is only one intersection point, but this only occurs if the sphere is equidistant from both speakers, which is rare. Most often the intersection of these spheres results in an equation of a circle. Below is a graphical portrayal of the different possible results of calculating the intersection of two spheres.

No Intersection	One Point	Circle Intersection
		

Two Sphere Intersection resulting in a Circle - 3D Model

http://www.punaridge.org/doc/factoids/gps/

Now imagine one more speaker is added to this set up and the distance from each of the speakers to the microphone is calculated. This will result in yet another sphere where the microphone could be located. The intersection of three spheres results in either two points or one point. Besides just taking this as a fact, visualize the circle found from intersecting the first two spheres. Any intersection point of all three spheres will need to lie on the circumference of this circle. Therefore, the intersection of this circle with the third sphere is the intersection of all three spheres. If the intersection circle is tangent to the sphere, there will only be one intersection point. Most likely, the intersection circle will not be tangent to the sphere and there will be two intersections points between the circle and the third sphere. The intersection circle could also be a "slice" of the third sphere, in which case every point of the circle would be contained by the third sphere. This possibility can be removed by making sure the speakers are not placed in a linear fashion. Below is an example of the intersection of three spheres where there is only one intersection point. Note that in this case the third sphere is tangent to the intersection circle of the other two spheres.

Intersection of Three Spheres at a Point



http://gnss.be/how_tutorial.php

If the microphone could be positioned anywhere in 3D space, a fourth speaker would be required to determine which of the two intersection points from the first three spheres is the actual location of the microphone; however, putting a few constraints on this system can eliminate the need for a fourth speaker. The first constraint is that all speakers must be placed in the same horizontal plane. This guarantees that if there are two intersection points, one will be above the speakers and one will be below the speakers. The second constraint is that the microphone must always be held at or above the plane of the speakers. Both of these constraints can simply be met by placing the speakers on the floor. This puts all of the speakers in the same *y* plane and forces the microphone to always be above the speakers. Finally, the speakers cannot be placed linearly with in the same *y* plane. Now rather than using a fourth speaker, the intersection coordinate of the three speakers will be the one with the *y* value greater than or equal to zero.

The Hardware

Speaker Setup:

A crucial aspect of the speaker and microphone system is that they are all wired together within the same system. This thesis project uses an Arduino Microcontroller for this purpose. Arduino is a popular open source electronics prototyping platform that allows users to develop electronic devices with programmable microcontrollers. The Arduino One, which is the specific microcontroller used in this project, functions at 16MHz and has fourteen digital pins as well as six analog pins. Within the Arduino code, each of these pins can be set as either an input pin or an output pin. The microcontroller also provides access to 5V, 3V, and ground pins.

Three digital ports (2,4, and 6) are set to output pins. Each of these pins is connected to a speaker. In the Arduino code, when the digital pin attached to one of the speakers is set to high, it results in a quick popping sound. It is the delay from when this sound is played on the speaker until the sound is received by the microphone that is used to calculate how far the microphone is from the speaker.

Each of the speakers used in this project initially came in a set of two with a 3.5mm audio jack as well as a two-prong power cord. The two-prong cord draws power from an external power outlet and does not get its power from the Arduino. The audio jack is stripped off, and within the audio wire, there are three separate wires. There is one wire per speaker and one wire for ground. Each of these wires is soldered onto a halved jumper wire which is plugged into the Arduino board. Speaker One is plugged into pin two, Speaker Two is plugged into pin four, and the ground wire is plugged into one of the ground ports on the Arduino.

The second set of speakers has the same setup except for the fact that only one of the two speakers is used. This speaker is Speaker Three, which is plugged into digital pin 6 on the Arduino board.

Microphone Setup:

The microphone used in this project is specifically built as an Arduino accessory. There are four pins on the microphone, one for power (5V), one for ground, one for the analog sound output, and one digital output which only outputs one if the sound input surpasses the microphone's capabilities of recording. The ground and power pins are both connected to the respective pins on the Arduino. The digital output is not used and the analog output goes through an amplification process and is sent to the Arduino.

The microphone includes an offset control, which adjusts the microphone's output when there is only silence. Initially, this output during silence was set to 0V. This resulted in the microphone only being able to record the top half of sound waves. Using an oscilloscope, the output during silence was adjusted to 2.5V, which is half of the 5V maximum. This allowed for the microphone to record both the top and bottom halves of sound waves.

The Arduino analog pins are capable of reading values from 0-1024. Based on the adjustment above, silence would theoretically be represented by a reading somewhere around 512 from the Arduino. Additionally, the Arduino analogs pins can be read at a speed of approximately .0001s, which allows for about 10,000 reads per second. Sound travels at a rate of about 1,116.4 feet/second. This rate can be inverted to find that it take sound about .0008957 seconds for sound to travel one foot.

Using these facts, if a microphone is two feet away from the speaker, it should take about .0018 seconds for the sound wave to reach the microphone. This means that if a beep is played from a speaker, and at that same instant the Arduino starts reading from the microphone, the microphone will perform 17 analog reads and still receive a number around 512 indicating that there is still silence. It wouldn't be until the 18th reading that the microphone data would indicate there is sound. This delay of 18 reads takes .0018s which makes sense because theoretically, sound should take .0018s to travel two feet. If the microphone were 3 feet away, it should take the sound wave about .0027 seconds to reach the microphone and which would result in 27 analog reads before the microphone registered any sound.

This example proves that the number of analog reads before a sound is registered is proportional to the distance between the microphone and speaker. When the distance is increased by one half (2ft + 1ft = 3ft), the number of reads before a sound is registered is also increased by one half (18 reads + 9 reads = 27) which proves proportionality.

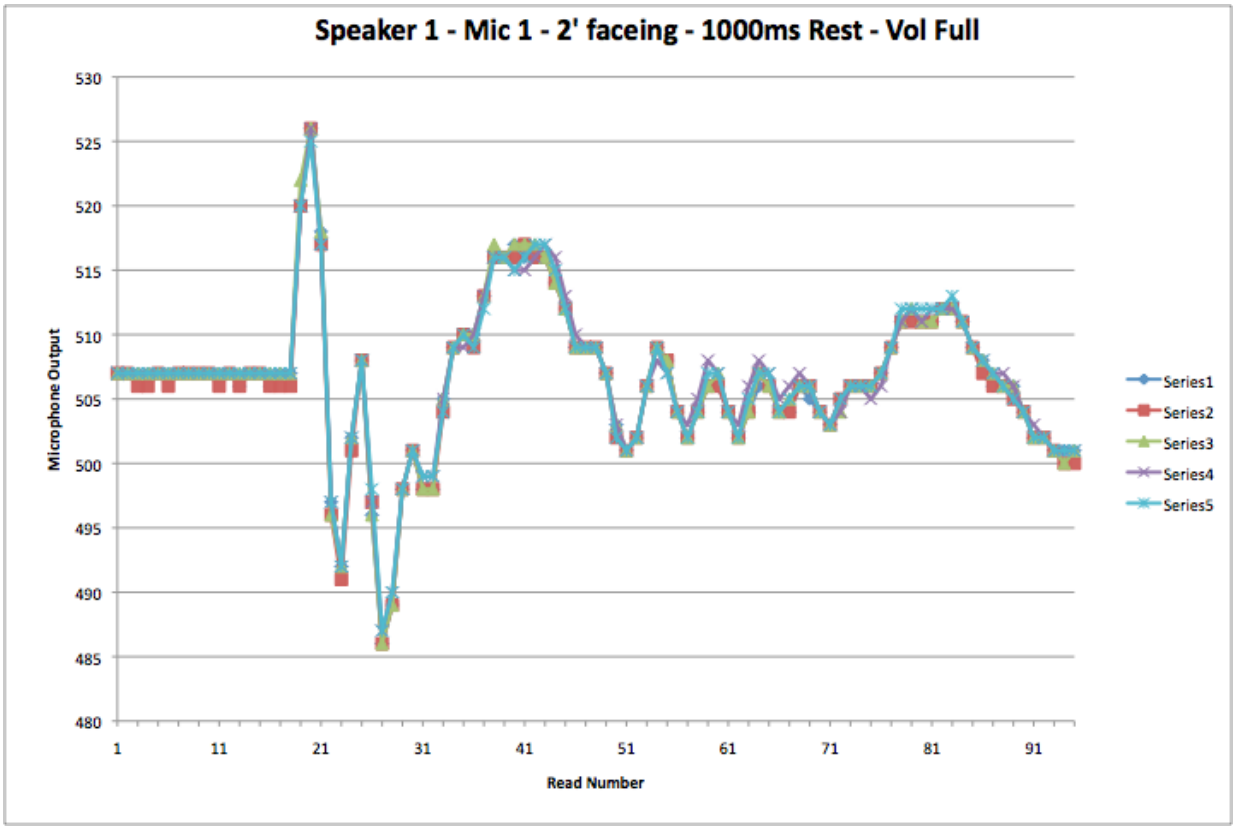
Analyzing the Sound

Initially, there was a question of whether the sound processing should be performed on the Arduino or on the computer. This issue was determined by the limited speed of the USB connection. In order for the processing to be performed on the computer, every single read from the Arduino (in this case, 100 integers per sound sample) needs to be written to the serial port from the Arduino and then read from the serial port by the code on the computer. This reading and writing process is so time costly, that it made sense to minimize the amount of data sent from the Arduino to the computer. In order to minimize

the data sent through the serial port, the analysis of the sound samples needs to be performed by the Arduino.

The crucial data point is the number of reads the microphone indicates it reads silence after a speaker has begun to play. This is always an integer and there are three speakers, therefore after each speaker beeps once, there are three integer data points written to the serial port. These are then read from the serial port and entered into sphere intersection equations on the computer.

The “Microphone” section above describes how the number of analog reads performed before a sound is recognized is proportional to the distance between the microphone and speaker. This presents the first serious question: How can the Arduino recognize the sound has changed and a “beep” or “pop” has occurred? This requires some testing and recording to find what the sound wave the microphone is recording looks like. When the microphone records 2 feet away from the speaker and starts recording exactly after the speaker “beeps”, the sound wave from the microphone output over five randomly selected trials can be seen in the following chart.



Based on this graph and the data it is derived from (Chart 1 in the Appendix), the spike in sound begins after read number 18. This is quite close to theoretically predicted value for the read number when the microphone is two feet from the speaker based on the speed of sound as discussed in the previous section. Thinking about this from an algorithmic standpoint and taking this graph into account, the easiest and most reliable way to find this spike in sound is to search for the greatest drop in amplitude because this is the most definable aspect of the sound wave. The drop after the sound spikes spans an average of 34 units, compared to the initial increase that only spans an average of 19 units. Also note that the drop after the spike occurs over three reads, which affects the code required to find this drop. The following is the pseudo-code used to find the read number at which this drop occurs:

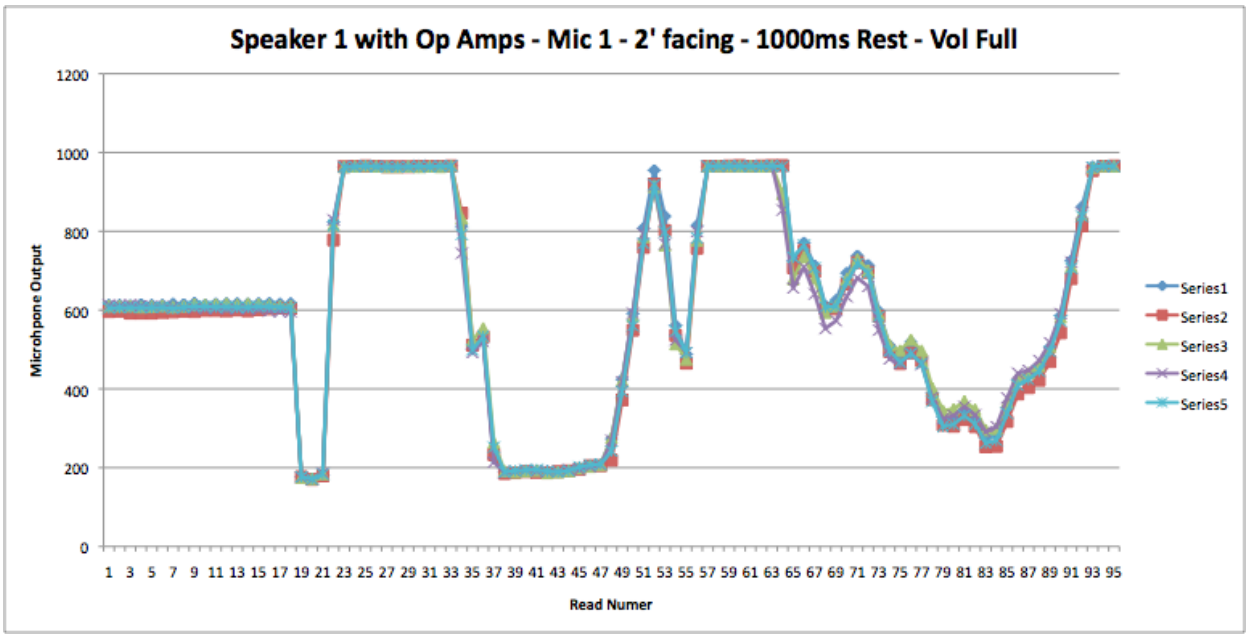
Find Max Drop Test 1 Arduino Pseudo-code

```
//initialize variables

speakerOne = digitalOutput(2);
microphoneInput = analogInput(0);
int buffer[100];
int ReadNumberMaxDrop = 0;
int maxJump = 0;
int delta = 0;

//set speaker high resulting in popping sound

digitalWrite(speakerOne, HIGH);

//read from the microphone 100 times into a buffer

for(int i = 0; i < 100; i++){
     buffer[i] = analogRead(microphoneInput);
}

//set the speaker back to low

digitalWrite(speakerOne, LOW);


//search through the buffer for the largest drop over three reads

for(int i = 0; i<95; i++){
   delta = buffer[i] - buffer[i+3];
   if(delta > maxJump){
       maxJump = delta;
       ReadNumberMaxDrop = i;
}
```

Unfortunately, using this in practice is ineffective because it is too inconsistent. The span of the drop is too small, which causes the wrong points to be recognized as the start of the greatest decrease. A thirty four unit span is just a small fraction of the 0-1024 units available. In order increase the span of this spike, the second version of the microphone circuit includes two operational amplifiers.

The function of operational amplifiers will be discussed further in the next section, but at a basic level, given two inputs, an operational amplifier takes the difference in voltage of the two inputs and scales the output based on the difference. Because the microphone is adjusted to have an output of approximately 2.5V during silence, a constant

2.5V can be used as one input for the operational amplifier and the output from the microphone can be used as the other. This means that when the microphone is silent, the difference between the two inputs will be minimal resulting in minimal amplification, but when the microphone receives sound, the two inputs will differ, resulting in significant amplification. This is the result of the same speaker and microphone playing and recording with the operational amplifiers in use:



Once again, the microphone reads 18 times before it receives any sound from the speaker (Chart B Appendix). There are three major differences between this new data and the data from the previous trial. The first is that this is an inverted version of the previous sound wave. Now rather than spiking first then decreasing, the wave decreases initially and then increases intensely. This means that rather than looking for the biggest decrease in the wave, the new goal is to find the biggest increase. The second difference is in the span of the greatest continuous change. In the first test, the biggest continuous change occurs

over thirty for units. With the operational amplifiers, the change occurs over more than 700 units, which makes the jump significantly easier to find. Finally, another change to note is that the largest change in the sound wave now only occurs over two reads. Searching for the max increase between reads is sufficient though because the first increase in the major spike is significantly larger than all other increases between two reads throughout the sound wave. The pseudo-code used to analyze the sound with these OpAmps is slightly different from the previous code in order to adjust for the changes described above. These changes are highlighted below.

Find Max Spike Test 2 Arduino Pseudo-code

```
//initialize variables

speakerOne = digitalOutput(2);
microphoneInput = analogInput(0);
int buffer[100];
int readNumberMaxSpike = 0;
int maxJump = 0;
int delta = 0;

//set speaker high resulting in popping sound

digitalWrite(speakerOne, HIGH);

//read from the microphone 100 times into a buffer

for(int i = 0; i < 100; i++){
     buffer[i] = analogRead(microphoneInput);
}

//set the speaker back to low

digitalWrite(speakerOne, LOW);


//search through the buffer for the largest drop over three reads

for(int i = 0; i<95; i++){
   delta = buffer[i] - buffer[i+1]; //looking for delta between each read
   if(delta < maxJump){             //find max increase, not decrease
       maxJump = delta;
       readNumberMaxSpike = i;
}
```

This is the basic algorithm implemented on the Arduino to find how many reads are required before the microphone registers the pop from one of the speakers. In order to perform this for all three speakers, the same code is used three times in a row with a few alterations. First of all, there are three different variables, “readNumberMaxSpike1”, “readNumberMaxSpike2”, and “readNumberMaxSpike3” in which the read number of the greatest spike is saved for each speaker so that data is not over written. Additionally, after analyzing each of the sound samples, the variable “maxJump” is reset to equal zero and the Arduino sleeps for 50ms. Resetting “maxJump” to zero allows the sound analysis to start over independently from the last run. After testing the Arduino using different rest periods, the 50ms pause resulted in the most accurate results, while still allowing the microphone position to be read often enough for it to be represented virtually. Once the sound is analyzed from each of the speakers, the values “readNumberMaxSpike1”, “readNumberMaxSpike2”, and “readNumberMaxSpike3” are sent over the USB connection to the serial port of the computer with a space in between each as a delimiter and a newline at the end of the third value. Below is psuedo-code outlining the code running on the Arduino. For a full version of the Arduino code refer to the Full Arduino Code in the Appendix.

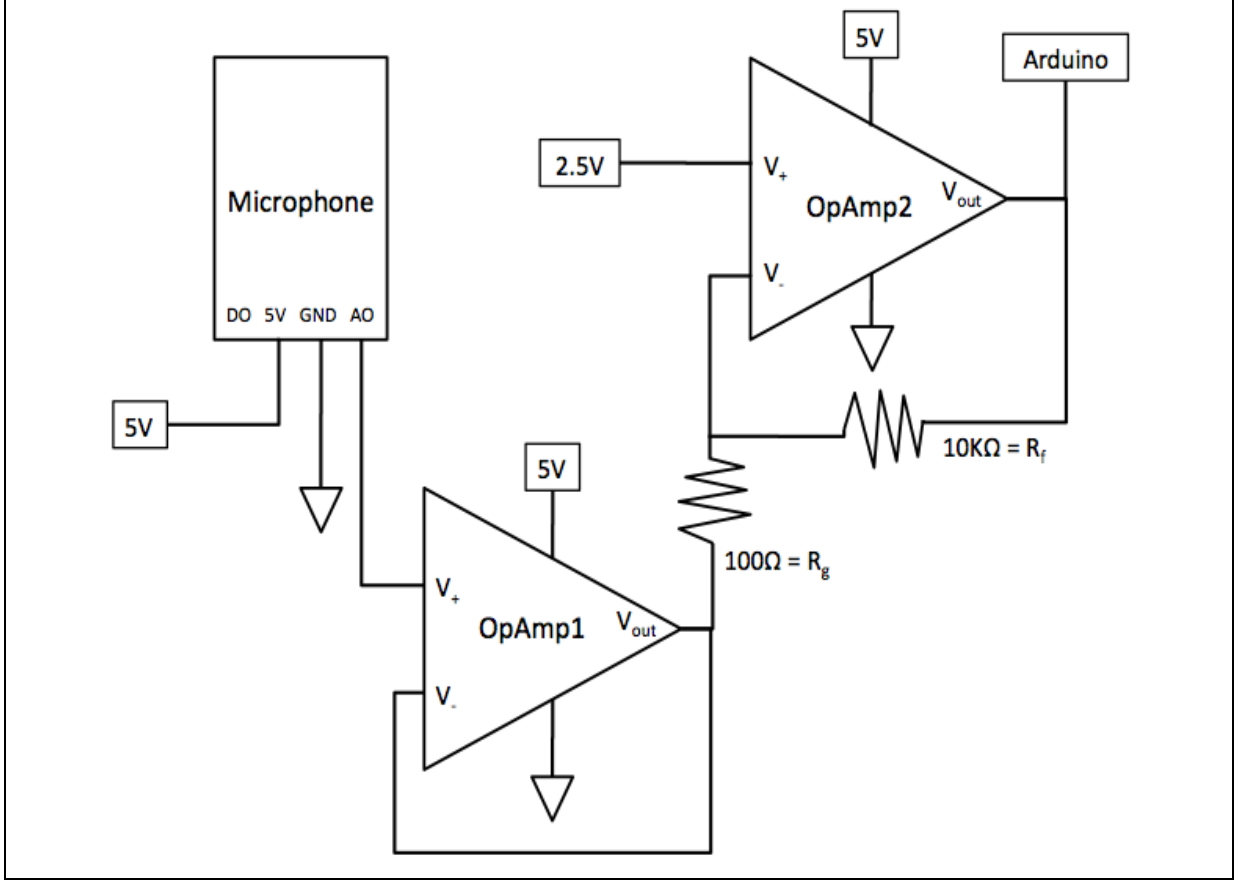
Full Arduino Pseudo-code

```
while true{
 //speaker 1

    set Speaker1 high
    read from microphone into buffer
    set Speaker1 low

    readNumberMaxSpike1 = buffer position at the start of the maxincrease

    maxJump = 0;
    sleep(50);

  //speaker 2

    set Speaker2 high
    read from microphone into buffer
    set Speaker2 low

    readNumberMaxSpike2 = buffer position at the start of the maxincrease

    maxJump = 0;
    sleep(50);

  //speaker3

    set Speaker3 high
    read from microphone into buffer
    set Speaker3 low

    readNumberMaxSpike3 = buffer position at the start of the maxincrease

    maxJump = 0;
    sleep(50);

 //send data

    send(readNumberMaxSpike1 + " " + readNumberMaxSpike2 +" "+
              readNumberMaxSpike3 + "\n")
}
```

Arduino Operational Amplifier Circuit

The microphone circuit requires two operational amplifiers. The first acts as a buffer between the microphone's output and amplification processes. The second operational amplifier multiplies any changes from silence. The following is a schematic of the microphone circuit:

Arduino Microphone Circuit



The inputs of the operational amplifiers are labeled V_+ and V_-. The V_- indicates the input that is inverted if the sound waves are different. The value of the output is equal to $(R_f/R_g)(V_+ - V_-)$. In the first case, there is no R_F or R_G and both of the inputs are the same. This means that the output remains the same as the input. The second operational amplifier has one input held at a constant 2.5V. The other input is the output from the microphone. The output of the microphone is connected to the inverted input, which is why the sound wave is inverted when it reaches the Arduino. Additionally, any time the sound wave is not equal to 2.5, the difference will be multiplied by R_f/R_g. This means that the formula for the output of the second operational amplifier is:

$$(10000\Omega/100\Omega)(2.5V - MicOutput) = V_{out} = 100(2.5V - MicOutput) =$$

This amplification process effectively multiplies any change in sound by 100 making the spike in the sound sample received by the microphone much easier to find in the analysis process. By increasing the jump in the sound sample, the microphone can pick up the speaker beeping from farther away which increases the range of the local positioning system. This amplification also decreases the number of misreads sent to the serial port which reduces the number of times an error results from intersecting the spheres.

The Math

Intersecting Spheres

In order to represent this local positioning system mathematically, think of how a sphere is represented three-dimensional space. Given a point *p* at (x_1, y_1, z_1), a sphere with radius *r* and center *p*, can be represented by the equation:

$$(x - x_1)^2 + (y - y_1)^2 + (z - z_1)^2 = r_1^2$$

This equation can be used to represent the possible coordinates of the microphone around each speaker. For each speaker, a value proportional to the distance between the microphone and the sphere is sent to the serial port. A conversion will be required to convert this number into an actual distance in order to be used as the radius in this equation, but this will be discussed later. For now assume that the radius is known in the appropriate units. There are three different speakers and located at three different constant positions: (x_1, y_1, z_1), (x_2, y_2, z_2), and (x_3, y_3, z_3) and each of the radii are acquired from the Arduino. The formulas representing the spheres where the microphone could be located for each speaker are:

$$(x - x_1)^2 + (y - y_1)^2 + (z - z_1)^2 = r_1^2$$

$$(x - x_2)^2 + (y - y_2)^2 + (z - z_2)^2 = r_2^2$$

$$(x - x_3)^2 + (y - y_3)^2 + (z - z_3)^2 = r_3^2$$

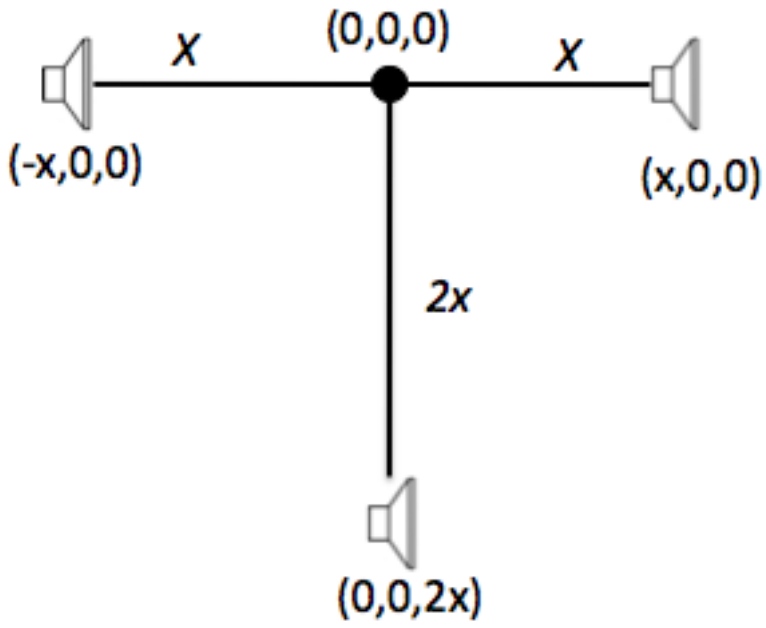
Because the speakers are not moving, x, y, and z are the only variables in each equation. Using Mathematica, these three equations can be set equal to each other and solved for the intersection of three spheres with undefined, but constant points as the positions of the speakers. Unfortunately, solving for intersection points using these equations results in the equations for each of the coordinates of the intersection points which are too complicated to be reasonably implemented in code.

This led to the realization that a custom coordinate system, proportional to real 3D space, could be used to represent the position of all of the speakers and simplify the three above equations. This means that each speaker's coordinates (x_1, y_1, z_1), (x_2, y_2, z_2), and (x_3, y_3, z_3) can all be defined as almost any number in the custom coordinate system, as long as there relative positions in the custom coordinate system are proportional to there actual positions.

As noted earlier, all of the speakers need to lie in the same horizontal plane to avoid using a fourth speaker. Here it makes the most sense to make the *y* coordinate of all speakers equal. This value could have been anything and worked mathematically, but in order to simplify the original three equations, it makes the most sense to use zero as the *y* value of each speaker.

There are two factors that affected the decision of what *x* and *z* coordinates should be used for each of the speakers. The first is that the numbers used for the position should simplify the intersection formulas for the spheres. The second is that the speakers need to be positioned in a manor that makes them effective for capturing the position of the microphone over as big a range a possible. Taking both of these factors into account, the following set up for the speakers was devised:

Birds Eye View of the Speaker Set Up



Above is an image representing a bird's eye view of the speakers (remember they all share a zero for the *y* coordinate and are therefore in the same horizontal plane). Speaker 1 and Speaker 2 are located an equal distance, *x*, away from the origin, on the x-axis. This makes Speaker 1's position (x, 0, 0) and Speaker 2's position (-x, 0, 0). Speaker 3 remains on the z axis, but is moved off of the origin by 2x, making Speaker 3's position (0, 0 , 2x).

At this point, any *x* value can be used, but *x* = 1 is the obvious choice in order to simplify all of the equations as much as possible. For the purpose of this thesis project, 1x in this custom coordinate system is equal to 20.5 inches. This is because 41 inches is as far as Speakers One and Speaker two can be spread apart because they are connected by a wire. Converting from this custom coordinate system into feet and inches is now possible, so if the read values from the Arduino can also be converted into feet and inches, these equations can be solved because all of the constants will be in the same units.

There are two possible ways to determine the distances between the speakers and the microphone based on the values sent from the Arduino. The first uses the speed of sound a sea level, which is approximately 1,125fps. A few conversion results in the following relationship between Arduino reads and distance

1,125 feet / second * 12inches / foot = 13,500 inches / second

13,500 inches / second * .0001 seconds / Arduino Read = 1.35 inches / Arduino Read

(1.35 inches/Arduino Read) * (1 custom units / 20.5 inches) =

.066 custom unit/Arduino Read

Another more realistic way to calculate the inches per Arduino Read is to use the built in Arduino Serial Monitor to continuously send back the number of delays it takes the microphone to read the beeps coming from one speaker. If the microphone is moved away from the speaker extremely slowly until the Serial Monitor shows that the delay number has increased, the position of the increase can be noted as the starting point of that Arduino read delay value. Now continue moving the microphone backwards until the delay value increases again. This is the ending position of this delay value. Subtracting the first

recorded value from the second will result in the distance sound travels during each Arduino Read. The result of these calculations came to an average of 1.60 inches/Arduino Read (see **Chart 3** in appendix). This allows for the following calculation:

*(1.60 inches/Arduino Read) * (1 custom units / 20.5 inches) =*

.078 custom/Arduino Read

This project uses the second method because of an initial delay that needs to be offset by a larger conversion factor. When the speaker initially beeps, even if the microphone is placed directly in front of the speaker, the Arduino returns a read delay of five even though the real read delay should be zero or one. For this reason, when the computer reads in the delay values from the Arduino, five is immediately subtracted from the delay value. These adjusted delay values are what are converted into custom units and used are radii for the spheres. For this reason, the .078 conversion factor returns more accurate results when the sphere equations are solved.

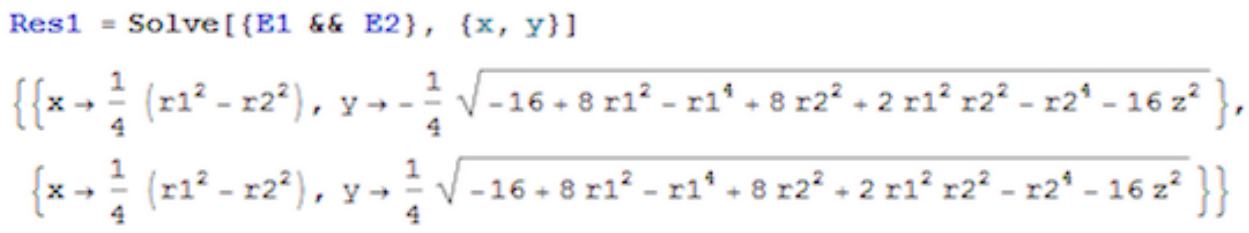
Now finally, this conversion factor can be used on each of the adjusted Arduino read delays in order to produce radii in custom units. Once the radii are in the same units as the positions of the speakers, the following equations can be solved for an intersection point:

$$(x - 1)^2 + y^2 + z^2 = r_1^2$$

$$(x + 1)^2 + y^2 + z^2 = r_2^2$$

$$x^2 + y^2 + (z - 2)^2 = r_3^2$$

At this point, E1 and E2 can be set equal to each other in order to solve for X as follows:



The result of solving these equations, as seen in the photo from Mathematica above, is the value of *x* for all intersection point of these two spheres. The *x* coordinate of the intersection point is equal to $.25(r_1^2 - r_2^2)$. Given that both r_1 and r_2 are constant, the *x* coordinate of the intersection point can be calculated. For now lets call the calculated *x* coordinate x_i. Exchanging the variable *x* for the value x_i in **E1** results in the following equation:

E1E2Circle $(x_i - 1)^2 + y^2 + z^2 = r_1^2$

Now there are only two squared variables which means this equation represents a circle. More specifically, this equation represents the circle that is the intersection of Sphere One and Sphere Two. The x_i coordinate shared by the first two spheres' intersection circle will also be shared by the third sphere's intersection point with both of these spheres. Therefore this x_i coordinate can replace *x* in **E3** as well, resulting in **E3** being rewritten as follows:

E3 $x_i^2 + y^2 + (z - 2)^2 = r_3^2$

Both **E1E2Circle** and **E3** are now equations of circles. They both contain two variables, *y* and *z,* so both *y* and *z* can be solved for. (note that x_i is denoted as XX in the following equations)

```
E1E2Circle = (XX + 1) ^ 2 + y ^ 2 + z ^ 2 = r1 ^ 2
```

$$(1 + XX)^2 + y^2 + z^2 = r1^2$$

```
E3 = XX ^ 2 + y ^ 2 + (z - 2) ^ 2 == r3 ^ 2
```

$$XX^2 + y^2 + (-2 + z)^2 = r3^2$$

```
Solve[{E1E2Circle, E3}, {y, z}]
```

$$\left\{\left\{y \to -\frac{1}{4}\sqrt{-25 + 10\,r1^2 - r1^4 + 6\,r3^2 + 2\,r1^2\,r3^2 - r3^4 - 20\,XX + 4\,r1^2\,XX - 4\,r3^2\,XX - 20\,XX^2}\,,\ z \to \frac{1}{4}\left(3 + r1^2 - r3^2 - 2\,XX\right)\right\},\ \left\{y \to \frac{1}{4}\sqrt{-25 + 10\,r1^2 - r1^4 + 6\,r3^2 + 2\,r1^2\,r3^2 - r3^4 - 20\,XX + 4\,r1^2\,XX - 4\,r3^2\,XX - 20\,XX^2}\,,\ z \to \frac{1}{4}\left(3 + r1^2 - r3^2 - 2\,XX\right)\right\}\right\}$$

```
Simplify[%]
```

$$\left\{\left\{y \to -\frac{1}{4}\sqrt{-r1^4 - r3^4 + r3^2\,(6 - 4\,XX) + 2\,r1^2\left(5 + r3^2 + 2\,XX\right) - 5\left(5 + 4\,XX + 4\,XX^2\right)}\,,\ z \to \frac{1}{4}\left(3 + r1^2 - r3^2 - 2\,XX\right)\right\},\ \left\{y \to \frac{1}{4}\sqrt{-r1^4 - r3^4 + r3^2\,(6 - 4\,XX) + 2\,r1^2\left(5 + r3^2 + 2\,XX\right) - 5\left(5 + 4\,XX + 4\,XX^2\right)}\,,\ z \to \frac{1}{4}\left(3 + r1^2 - r3^2 - 2\,XX\right)\right\}\right\}$$

Solving these two equations for *y* and *z* results in the two solutions for both *x* and *y* shown above. These two points are defined by the radii of the three spheres which are known, x_i, which is known and other constants, so *y* and *z* can be calculated. These coordinates, along with x_i form the two intersection points of the three spheres defined by

the original equations **E1**, **E2**, and **E3**. Because all of the speakers are in the same plane, one of these coordinates is below this plane and the other is above. In this model, it is assumed that the microphone will only be held above the speakers and therefore the intersection point with the positive *y* coordinate is the only one that is needed. The square root of a number is always positive, therefore the equation that multiplies the square root by positive ¼ in the picture above is the equation that represents the positive y coordinate.

In practice, the most common source of error occurs at this point in the process. If the read values sent from the microphone do not result in three spheres with an intersection point on or above the y axis, the value under the square root of the y equation becomes negative which is not a real number. This normally only occurs while the microphone is in motion and there are controls in place to make sure this error does not crash the program. Most calculations accurately find the intersect point of the three spheres above the plane of the speakers as intended.

Unity

<u>The Game:</u>

The goal of this thesis is to represent the motion of the microphone from the local positioning systems in a virtual environment and use this as the input for a video game. "Save the Sheep" is the first game to utilize this local sonic positioning system as a controller. In this game, an evil alien species is attacking the user's farm, specifically targeting his sheep pen with missiles. Luckily the user has a magical sphere that can destroy the missiles and protect the sheep. The magical sphere is controlled by the microphone from the local positioning system and the goal is to have as many missiles as

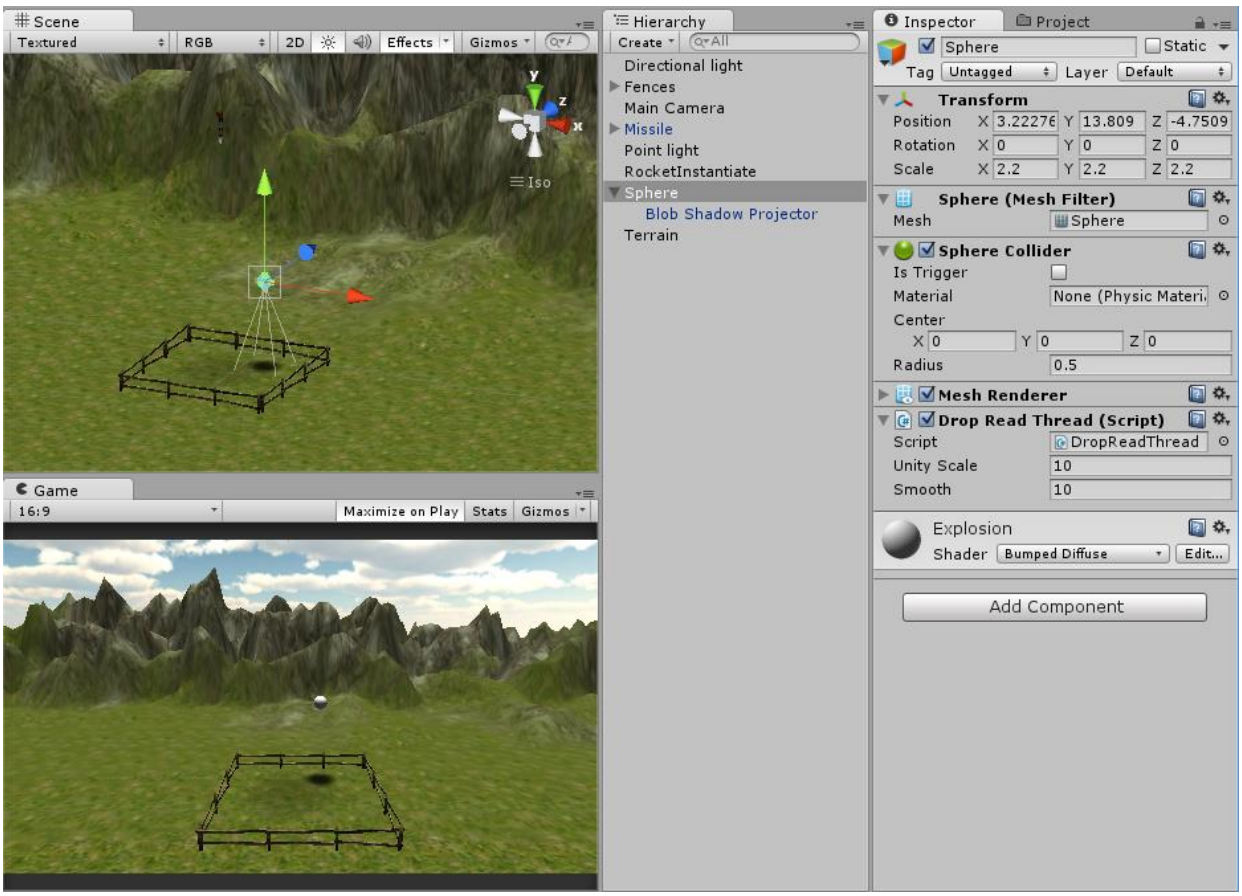
possible collide with the magical sphere before they reach the ground and injure any of the user's sheep.

Unity Essentials:

This project uses the program Unity to represent the microphone's movement in 3D space. Unity is a 3D game development tool used by many professional developers. Every "scene" within Unity is a virtual 3D space in which developers can insert a variety of "GameObjects." GameObjects can be a variety of things including 3D models, particle system, light objects, or many other types of objects. Unity can create its own basic 3D models, but custom 3D models can be imported from .fbx, .dae, .3DS, .dxf and .obj files. Particle systems are used to represent things like fires, explosions, or water in a water fountain. Each particle is a small image, which is assigned certain loosely defined behavior. Each particle in the particle system is created, goes through its behavior, and is then destroyed, but the particle system is continuously reproducing particles resulting in a non-static object. Light objects brighten the color of objects around them in the virtual 3D space. Another example of a GameObject is a Terrain, which are capable of being transformed into life like environments with mountains, trees, and grass. There are also GameObjects called Cameras which define what view the user will have while playing the game. This is brief description of a few core GameObjects within Unity, but this should be enough information to describe how "Save the Sheep" works.

Each GameObject can be given certain components. Once a GameObject is placed in the scene, it is automatically assigned a transform component. The transform component represents the object's position, rotation, and scale within the scene. Other examples of components that can be added to GameObjects are textures, colliders, Rigidbodies and

scripts. Textures are 2D images that can be applied to objects. Colliders detect if the GameObject collides with another colliders. Making a GameObject into a Rigidbody allows physics to be applied to the GameObject. This means that it will be affected by gravity, or if another Rigidbody collides into, the force will be transferred realistically from one object to the other. Scripts are custom behaviors that can be given to objects. These behaviors are defined by code which a developer can write in C#, JavaScript, or Boo. Below is a developer view of Unity which will hopefully make things more clear.



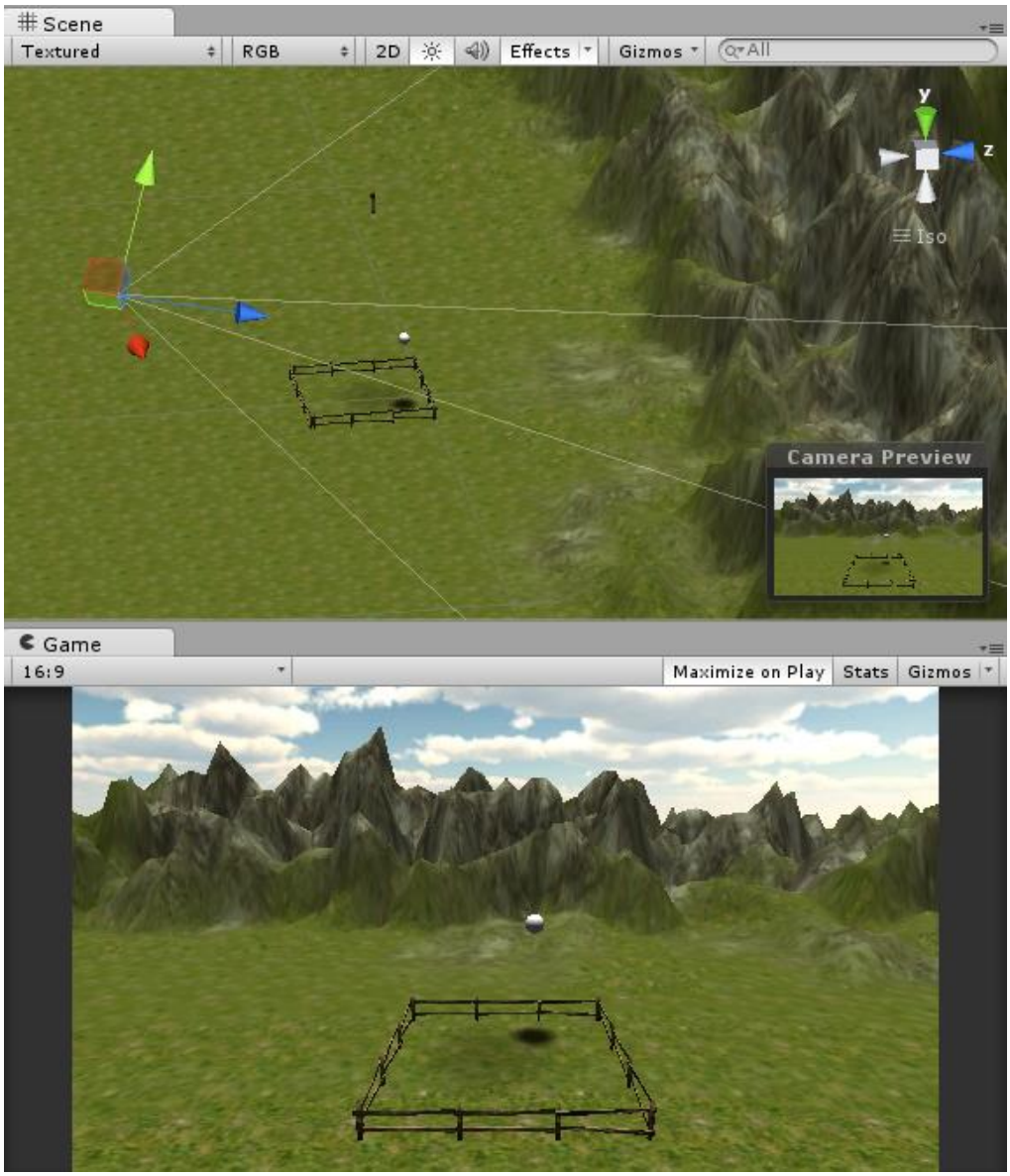
The first view is the scene view. The scene is the 3D virtual space that contains all of the GameObjects in the game. The scene view can be moved to show any part of the scene so the developer can ensure objects are in the right position and that everything is scaled properly.

The Hierarchy tab defines every GameObject that is used within your scene. When an object is selected in the Hierarchy tab, the object is selected and highlighted in the Scene View and all of the selected GameObject's components are seen in the Inspector Tab. In this example, the sphere is selected in the Hierarchy and is therefore highlighted in the Scene View. Also, looking at the Inspector, the sphere has a transform defining its position, rotation, and scale. The sphere also has a collider and the "Drop Read Thread" script attached to it which moves the sphere based on the movements of the microphone. The Hierarchy view also allows for what is called "parenting." In the sphere example, notice that the "Blob Shadow Projector" GameObject is listed below the sphere and is indented. This means that the sphere is the parent of the "Blob Shadow Projector" and the "Blob Shadow Projector" is the child. A child object's transform is defined relative to the transform of its parents. A child object also moves along with its parent object. In this case, the shadow will move with the sphere because it is its child even though there is no code specifically assigned to the shadow.

The Project tab displays all of the GameObjects and custom components available that can be included in a Scene or applied to a GameObject respectively. In order to add an object into the scene view, it can just be dragged from the Project tab into the Scene View. To apply a component from the Project tab to a GameObject in the Scene View, it just needs to be dragged from the Project tab to its listing in the Hierarchy.

Finally, the Game View shows what the end user will actually see. This is defined by a GameObject called a camera. This camera is placed in the scene by the developer with the desired position and rotation and is given a certain field of view, which it portrays to the end used. In this project, the camera remains in a constant position, but if this game were

modeled as a first person or third person game, the camera would move based on appropriately written scripts. Below is an image of the camera selected in the scene view. The field of view is represented by the white lines in the Scene View and the camera's view is seen in both the Camera Preview as well as the Game View.



The Scripting – Preparing the Environment

Scripts can be written in C#, JavaScript, or Boo, but for this project all of the code is written in C#. This is slightly limiting because in order to compile C# code on a Mac, the Mono compiler must be installed. Unfortunately even with the mono compiler, Macs are not capable of compiling C# code that uses serial ports, and for that reason this project currently only functions on Windows operating systems. The mono compiler can be expanded to include serial port accessibility, but that went beyond the scope of this project.

There is one more minor step that must be taken before a developer can access the serial ports through C# code in Unity, even when using a Windows machine. Initially, the default API compatibility level within Unity is a subset of .NET 2.0 which does not allow for the use of serial ports. In order to change this default setting, a developer can find the "API compatibility level" option under "Edit → Project Settings → Player → Other Settings." Once the user has navigated to this menu, the "API compatibility level" must be set to ".NET 2.0" rather than ".NET 2.0 Subset."

The Scripting – Actual Coding

All Unity friendly C# classes are derived from the MonoBehaviour class (in C#, a class being derived from another class is the same as a class extending another class in Java). Being derived from the MonoBehaviour class allows for a class to implement certain predefined methods such as "Start()", "Update()", and "OnApplicationQuit()." Start() is called once when the game is started. This is mostly used to instantiate variables, and in this project is where a thread is created and started. Update() is called every time the game updates its frame. This means the method is dependent on the frame rate of the game,

which at peak performance is generally around 60 frames/second. OnApplicationQuit() is called once when the game closes.

Moving the magical sphere requires three main classes. The first class is a MonoBehaviour class, responsible for updating the sphere's position in the scene view every time the OnUpdate() method is called. This MonoBehaviour class is called "DropReadThread.cs" and is a component of the sphere because it defines the sphere's motion. The DropReadThread class is also responsible for creating the other two major classes. In the Start() method, both of the other two major classes are instantiated. One class is a thread which reads the radii values from the Arduino and then performs the sphere intersection calculations. The other is a global data class which allows the thread and MonoBehaviour classes to synchronously get and set the microphone's coordinates.

DropReadThread

The main MonoBehaviour class, DropReadThread can be broken into three different sections, the Start(), Update(), and OnApplicationQuit() methods. The Start method is called when the user begins the game. In this method, the thread is instantiated and started and the global data class is also instantiated. In the Update method, the DropReadThread class reads the current coordinates of the microphone from the global data class, scales them appropriately into the virtual coordinates and then updates the position of the position of the sphere using the method Vector3.Lerp(). Vector3.Lerp() is a method which moves an object from one position to another smoothly and linearly over a defined period of time. This method takes three parameters: the starting position of a GameObject given as a vector with three variable (Vector3 class), the ending position of the GameObject, again

given as a vector with three variables, and the time over which the object should be moved from the starting position to the finishing position. In order to access the spheres position, because this code is attached to the sphere, all that is needed is the code "transform.position" which must be set equal to the result of Vector3.Lerp(startingPosition, endingPosition, .05). The ending position of the microphone is created from the position of the microphone, received from the global data class. The received position of the microphone is scaled from the custom coordinate system into values appropriate for Unity's coordinate system. In this project, the Unity virtual environment is 10x the scale of the custom coordinate system that was created. This will all be seen in the pseudo code below. Finally, the OnApplicationQuit() method only has one line of code which throws an interrupt exception in the thread it began in Start(), so that the thread knows the game is over and destroys itself.

DropReadThread Pseudo Code

```
GlobalData globalData
Thread readThread;
Float[] coords
Vector3 newPositon;

Start(){

       readThread = new Thread(ReadThread);
       readThread.Start();

       globalData = new GlobalData();

       coords = new Float[3];
}

OnUpdate(){

      coords = globalData.ReadCoords();

      newPosition = new Vector3(coords[0] *10, coords[1] *10, coords[2]*10);

      transform.position = Vector3.lerp(transform.position, newPosition, .05);
}

OnApplicationQuit(){
```

```
     readThread.Interrupt();
}

ReadThread(){
     //to be discussed
}
```

ReadThread

Before discussing the actual importance of the thread created in DropReadThread, there are two important points to make. The first is the relationship between Unity's MonoBehaviours and threads. MonoBehviours are not thread friendly. This means that none of the methods available to a class derived from a MonoBehaviour are available in a thread. For this reason, threads are most practically used in Unity for background calculations and not updating the game itself. This also results in a hindrance within the debugging process. In a MonoBehaviour, the method print(String string) is available to write debugging messages in the Console View within Unity. In a thread however, the Unity console is unavailable, making it impossible to print out debugging remarks through Unity. This project uses a workaround for this in which a log file is opened within the thread. The thread writes debugging remarks to the log file, which provides feedback for the developer instead of using the Console view. This debugging technique is no longer necessary, but proved extremely useful.

For native C# users, this next point may not be noteworthy, however developers who commonly use Java, may find this helpful. In Java, threads are created from classes that either extend the Thread class or implement the Runnable interface. In C#, there is a constructor for a thread which takes the name of a method within the current class as a

parameter. This method is what is started as a separate thread called when thread.Start() is executed.

In the Start() method of the DropReadThread, a thread, “readThread”, is instantiated using a thread constructor that is given the DropReadThread’s own method ReadThread() as a parameter. Upon calling readThread.Start(), the ReadThread() method from the DropReadThread class begins to run independently from the DropReadThread class itself.

This thread initially creates a serial port using the same serial port the Arduino writes to. The serial port is opened and then the thread enters an infinite loop that is only ended by a thread interrupt. Each time the thread enters the loop it initially waits on a read from the serial port. As described earlier, the Arduino sends three integers separated by spaces and ending with a newline to the serial port after every three beeps. These numbers represent the number of analog reads the Arduino performs after each speaker beeps before the microphone receives the sound. The readThread reads this line from the serial port and separates the string into substrings using the spaces as delimiter. Then each of these read values are parsed into integers, reduced by five and multiplied by the conversion number .078 in order to convert the read values into units of the custom coordinate system as described in the section “Intersecting Spheres.” Once the radii values are converted into the right units, the *x* coordinate of the microphone is calculated first, which is then used to calculate the *y* and *z* coordinates of the microphone using the equations derived in the “Intersecting Spheres” section. Once the coordinates of the microphone are calculated, they are sent to the global data class.

ReadThread Pseudo Code

```
ReadThread(){
     sp = new SerialPort();
     conversion = .078;

     while(true){ try {

           line = sp.readLine();
           radii = line.Split(“ ”);
           raduis1 = (radii[0]-5) * conversion;
           raduis2 = (radii[1]-5) * conversion;
           raduis3 = (radii[2]-5) * conversion;

           x = .25 * (radius1^2 - radius2^2);

           y = .25f * Sqrt( - radius1^4 - radius3^4
                      + (radius3^2 * (6 - 4 * x))
                      + (2 * radius1^2 * (5 + radius3^2 + 2x))
                      - 5 * (5 + 4 * x + 4 * x^2)
                          );

           z = = .25 * (3 + radius1^2 - radius3^2) - 2 * x);

           globalData.setCoords(x, y, z);

       }catch(InterruptException e){
              break;
       }
     }
}
```

GlobalData

The Global Data class is a basic class with two methods in order to get and set the coordinates of the microphone. The coordinates are set in the global data by the readThread and the coordinates are received from the global data class in the MonoBehaviour class. The most important part of the global data class is the lock that is placed around getting and setting the data. A lock is required so that the coordinates cannot be read in the middle of them being set, or the other way around. Because getting the data and setting the data are called from two different threads, if there were no locks, the MonoBehaviour could receive one coordinate from the GlobalData, then the

readThread could update the position, and the global data class would proceed to return the other two coordinates to the MonoBehaviour class from a different, more recent position. This would result in a completely in accurate position of the sphere. Using a lock forces the entire get or set method to finish before another GlobalData method can be executed.

The GlobalData class is also helpful because the position of the microphone can potentially be calculated multiple times before the frame refreshes once. If every position that is calculated were sent to the MonoBehaviour, the movements of the sphere would most likely be delayed as the sphere would have to move through each of past points each frame update and try to catch up to the current position. By having the readThread constantly update the position in the GlobalData class, the DropReadThread class can get the most recent position of the microphone and move the sphere there without laboring through all of the old positions. This could have been an issue without the lerp method, because the movements of the sphere may have appeared very jumpy, but the lerp transitions the sphere smoothly from one position to the next. This also could potentially be an issue if the frame rate was to slow. Say for instance, the microphone is moved in a zigzagging path from one position to another during the time span of one frame. This would result in the sphere just moving from the first point to the second, losing all of the zigzagging motion. This fortunately is not the case. Below is the pseudo code for the GlobalData class.

GlobalData pseudo code

```
public GlobalData(){
     lockThis = new System.Object();
}

public void SetRead(float x, float y, float z){
       lock(lockThis){
           position[0] = x;
           position[1] = y;
           position[2] = z;
       }
}

public float[] GetReadValue(){
      float[] positionCopy;
     lock(lockThis){
                 positionCopy = position;
      }
           return positionCopy;
}
```

These three classes work together effectively to represent the movement of the microphone as a sphere in 3D space.

RocketInstantiation

GameObjects can be also added to the scene programmatically during game play rather than by the developer before hand. This is done using the Instantiate(Object, Position, Rotation) method. This method creates a given object at a specific position and rotation. In this project, there is an empty GameObject called a spawn point positioned above the sheep pen out of view of the main camera. The script is "RocketInstantiation.cs" and is attached to this missile spawn point. When the game begins, a rocket is instantiated with the same *y* coordinate as the spawn point, but randomly positioned above the sheep pen every four seconds. Additionally, because the missiles have the RidgidBody component,

gravity is applied to them causing them to fall with the acceleration of gravity as soon as they are instantiated.

Rocket Instantiation Code

```
prefab = missileObject;

while (true) {
      xRand =  Random.Range(-5.0f, 5.0f);
      zRand = Random.Range(-5.0f, 5.0f);
      yield return new WaitForSeconds(4.0f);
      Instantiate(prefab,
                              new Vector3(transform.position.x+xRand,
                                          transform.position.y,
                                          transform.position.z+zRand),
                        prefab.transform.rotation);
}
```

DestroyRocket

Unity can automatically detect when collisions occur between two GameObjects using components called colliders. When these collisions occur, the OnCollisionEnter(Collision collision) method of any scripts attached to either of the colliding objects is called. These colliders are what allow this game to determine when the missiles collide with the sphere or the terrain. The OnCollisionEnter method comes with a Collision parameter which allows the developer to identify what type of GameObject the collision was with as well as where the collision occurred. In this case, the OnCollisionEnter method is in a MonoBehaviour attached to each missile. Every time there is a collision with a missile, the behavior checks to see if the collision is with the sphere. If the collision is with the sphere, a firework particle system is instantiated at the point of collision, a sheep GameObject is randomly instantiated within the pen, and the missile is destroyed. If the collision is with the terrain, an explosion particle system is created, one of the sheep is

blown away and the missile is still destroyed. This collision code is where all of the GUI scores would be updated.

Destroy Rocket Pseudocode

```
OnCollisionEnter(Collision collision){
      if(collision.gameObject.tag == "Player"){

            Instantiate(fireworks,  collision.position, noRotation);

            xRand =  Random.Range(-5.0f, 5.0f);
            zRand = Random.Range(-5.0f, 5.0f);

            sheep[i++] = (Transform) Instantiate(sheepPrefab,
                                    new Vector3(transform.position.x+xRand,
                                                0,
                                                transform.position.z+zRand),
                                    sheepPrefab.transform.rotation);

            }
      else{
            Instantiate(explosion,
                        new Vector3(transform.position.x,
                                    transform.position.y-4,
                                    transform.position.z),
                        noRotation);

            Destroy(gameObject);    //destroys the missile
      }
}
```

Conclusion

The results of this thesis show that a local sonic positioning system can effectively locate the position of a microphone in 3D space using three speakers in known locations. Additionally, the movements of the microphone can be used as input for a video game. The equipment used for this project is less than ideal. If possible, the project would benefit from a microcontroller that could perform analog reads from a microphone at a quicker pace than 1 read per .0001 seconds. This would mean sound would travel less per each analog read which in turn would increase the accuracy of the positioning system. Additionally, if

the speakers were capable of playing a sound more recognizable than the pop of a digital increase from 0 to 1 the sound would most likely be much easier to recognize. This would result in less erroneous radii values sent from the Arduino in the sound analysis portion. This project proves that even with somewhat crude supplies, finding the position of the microphone is possible, which indicates that with optimized hardware, this local sonic position could be a legitimate solution for finding the position of objects in 3D space.

As currently implemented, there are some limitations to this system. The cacophony of the current system is quite impractical but there are a few solutions that would still allow this system to be used without all of the mind-numbing beeping. One possible solution to make this system more practical is to use ultrasonic speakers and an ultrasonic microphone rather than the ones currently in place. Another possibility is to have the speakers play a song of some type that has definable aspects. The microphone would recognize these specific aspects, which would still allow the microcontroller to calculate how many Arduino reads are required from when the speaker plays that portion of the song until the microphone receives it.

Although this project proves that locating the position of a microphone using three speakers with known locations is feasible, it does suffer from some boundaries. The first is that the microphone must remain above the plane of all of the speakers. This limitation could be removed by adding an additional speaker into the project however trying to find an intersection point between four spheres makes the calculations more difficult and greatly increases the chance of an error somewhere in the calculations. Another limitation is that this system requires almost complete silence from the user. For some applications of this positioning system, this might not be an issue, but this causes multiple issues for video

game users. The first is that they cannot talk with their friends while playing. The second is that the game cannot have any volume.

Finally, the greatest limitation of this system is brought on by microphone reads while the microphone is in motion. While the microphone is moving, the Arduino rarely sends a set of three legitimate reads that result in an intersection point between the three spheres. One major reason for this is that all three of the reads need to occur while the microphone is in the same position. If one read occurs, then the microphone is moved for the second two, the intersection point that is calculated is either going to be inaccurate or is not going to exist altogether.

With some limitations, this local sonic positioning system can effectively find the position of a microphone in 3D space. Furthermore, the movements of the microphone can be represented in a virtual environment in order to act as the controller for a video game. With more research, experimentation, and some customized hardware, this local positioning system could be developed into a legitimate solution for tracking the position of objects in 3D space.

Appendix

Chart A – First sound wave test

Read#	Speaker 1 - Mic 1 - 2' faceing - 1000ms Rest - Vol Full				
1	507	507	507	507	507
2	507	507	507	507	507
3	507	506	507	507	507
4	507	506	507	507	507
5	507	507	507	507	507
6	507	506	507	507	507
7	507	507	507	507	507
8	507	507	507	507	507
9	507	507	507	507	507
10	507	507	507	507	507
11	507	506	507	507	507
12	507	507	507	507	507
13	506	506	507	507	507
14	507	507	507	507	507
15	507	507	507	507	507
16	506	506	507	507	507
17	507	506	507	507	507
18	507	506	507	507	507
19	520	520	522	520	520
20	526	526	526	526	525
21	518	517	518	517	517
22	497	496	496	497	497
23	492	491	492	492	492
24	502	501	502	502	502
25	508	508	508	508	508
26	496	497	496	497	498
27	486	486	486	487	487
28	489	489	489	490	490
29	498	498	498	498	498
30	501	501	501	501	501
31	498	498	498	499	499
32	498	498	498	499	499
33	505	504	505	505	504
34	509	509	509	509	509
35	510	510	510	509	510
36	510	509	510	510	509
37	513	513	513	513	512
38	516	516	517	516	516
39	516	516	516	516	516
40	517	516	517	515	515
41	516	517	517	515	516
42	516	516	517	516	517
43	516	516	516	517	517
44	514	514	514	516	515

45	512	512	512	513	512
46	509	509	509	510	509
47	509	509	509	509	509
48	509	509	509	509	509
49	507	507	507	507	507
50	503	502	503	503	502
51	501	501	501	501	501
52	502	502	502	502	502
53	506	506	506	506	506
54	509	509	509	508	509
55	508	508	508	507	507
56	504	504	504	504	504
57	502	502	502	503	502
58	504	504	504	505	504
59	506	506	506	508	507
60	506	506	507	507	507
61	504	504	504	504	504
62	502	502	502	503	502
63	504	504	504	506	505
64	506	507	507	508	507
65	506	506	506	507	507
66	504	504	504	505	504
67	504	504	505	506	505
68	506	506	506	507	506
69	505	506	506	506	506
70	504	504	504	504	504
71	503	503	503	503	503
72	505	505	504	504	505
73	506	506	506	506	506
74	506	506	506	506	506
75	506	506	506	505	506
76	507	507	507	506	507
77	509	509	509	509	509
78	511	511	511	511	512
79	511	511	512	512	512
80	511	511	511	511	512
81	511	511	511	512	512
82	512	512	512	512	512
83	512	512	512	512	513
84	511	511	511	511	511
85	509	509	509	509	509
86	508	507	508	508	508
87	506	506	507	507	507
88	506	506	506	507	506
89	506	505	506	506	505
90	504	504	504	504	504
91	502	502	502	503	502
92	502	502	502	502	502
93	501	501	501	501	501

94	501	500	500	501	501
95	501	500	501	501	501

Chart B - Second sound wave test – Microphone with OpAmps

Read#	Speaker 1 - Mic 1 - 2' faceing - 10ms Rest - Vol Full				
1	575	568	557	560	555
2	574	574	559	561	559
3	575	570	559	559	563
4	575	570	558	555	565
5	578	565	556	552	564
6	575	561	557	547	566
7	579	562	557	549	569
8	576	559	559	548	569
9	575	555	560	545	567
10	575	559	561	550	565
11	573	555	561	554	565
12	572	555	559	552	564
13	566	554	559	556	563
14	563	553	558	557	562
15	549	536	546	542	550
16	170	171	169	169	168
17	169	169	167	169	170
18	185	186	184	186	189
19	963	963	964	964	963
20	964	964	965	964	964
21	965	965	966	966	966
22	965	966	966	966	966
23	964	964	965	964	964
24	962	963	963	963	963
25	963	963	963	962	964
26	964	965	965	965	964
27	965	965	965	964	964
28	965	964	964	965	964
29	964	965	966	966	965
30	966	966	966	966	966
31	678	675	682	683	667
32	584	574	574	578	566
33	542	523	516	530	519
34	196	197	194	197	198
35	185	186	182	186	184
36	189	186	185	187	186
37	186	184	182	183	184
38	186	188	186	188	184
39	190	191	189	188	188
40	189	190	189	188	187
41	189	190	189	191	189
42	198	197	197	197	196
43	202	200	203	204	200

44	206	206	207	207	205
45	208	206	205	207	208
46	312	269	305	276	273
47	549	518	551	538	522
48	776	754	792	762	759
49	901	883	917	873	889
50	858	838	860	814	843
51	772	744	770	732	751
52	754	723	744	710	732
53	748	724	739	702	732
54	702	680	685	646	690
55	628	608	618	580	615
56	677	665	688	635	674
57	837	837	866	802	860
58	960	964	965	935	965
59	952	965	966	907	965
60	853	875	910	822	886
61	816	845	866	805	851
62	902	941	955	904	945
63	964	966	966	964	966
64	912	966	967	907	966
65	732	809	822	726	771
66	602	640	636	603	626
67	650	679	702	671	694
68	745	761	782	757	775
69	774	779	785	764	781
70	675	667	672	659	672
71	634	612	628	627	632
72	643	610	631	630	638
73	613	575	582	585	591
74	459	414	402	404	414
75	307	253	252	256	270
76	239	204	205	207	213
77	224	205	203	206	204
78	208	202	201	204	202
79	203	200	198	200	200
80	203	201	200	203	202
81	207	204	201	204	204
82	216	209	203	206	209
83	280	213	218	214	276
84	338	289	300	285	343
85	423	378	398	379	433
86	537	498	520	487	542
87	621	591	607	574	625
88	690	668	676	642	699
89	784	758	784	739	801
90	961	950	965	946	965
91	965	965	964	964	964
92	965	964	966	965	965

93	965	965	965	965	965
94	965	965	966	965	966
95	966	965	965	964	965

Chart C

ReadNumber	Distance of Read
14	1.6875
15	1.5
16	1.625

Full Arduino Code

```
int speakerOne = 2;
int speakerTwo = 4;
int speakerThree = 6;
int buffer[100];
int counter = 0;
int maxJump = 0;
int delta = 0;
int ar = 0;
int rest = 50;

int a = 0;
int b = 0;


void setup(){
    Serial.begin(9600);
    pinMode(speakerOne, OUTPUT);
    pinMode(speakerTwo, OUTPUT);
    pinMode(speakerThree, OUTPUT);

    for(int i = 0; i<100; i++){
       buffer[i] = i;
    }

    digitalWrite(speakerOne, LOW);
    digitalWrite(speakerTwo, LOW);
    digitalWrite(speakerThree, LOW);

    pinMode(ar, INPUT);
}

void loop(){

  //speaker @ 2
    digitalWrite(speakerOne, HIGH);

    for(int i = 0; i < 100; i++){
        buffer[i] = analogRead(ar);    //microphone input in 0
    }

    digitalWrite(speakerOne, LOW);

    for(int i = 0; i<95; i++){
      delta = buffer[i] - buffer[i+1];
      if(delta < maxJump){
```

```
          maxJump = delta;
          counter = i;
      }
    }

    a=counter;
    delay(rest);
    counter = 0;
    maxJump = 0;

  //speaker @ 4
    digitalWrite(speakerTwo, HIGH);

    for(int i = 0; i < 100; i++){
        buffer[i] = analogRead(ar);    //microphone input in 0
    }

   digitalWrite(speakerTwo, LOW);

   for(int i = 0; i<95; i++){
      delta = buffer[i] - buffer[i+1];
      if(delta < maxJump){
          maxJump = delta;
          counter = i;
      }
    }

   b=counter;
   delay(rest);
   counter = 0;
   maxJump = 0;

  //speaker @ 6
    digitalWrite(speakerThree, HIGH);

    for(int i = 0; i < 100; i++){
        buffer[i] = analogRead(ar);    //microphone input in 0
    }

    digitalWrite(speakerThree, LOW);
    for(int i = 0; i<95; i++){
      delta = buffer[i] - buffer[i+1];
      if(delta < maxJump){
          maxJump = delta;
          counter = i;
      }
    }

    delay(rest);
    Serial.print(a);
    Serial.print(" ");
    Serial.print(b);
    Serial.print(" ");
    Serial.println(counter);
    Counter = 0;
    maxJump = 0;

}
```

DropReadThread.cs

```
using UnityEngine;
using System.Collections;
using System.Threading;
using System.IO.Ports;
using System.Linq;
using System.Text;
using System.IO;
using System;


public class DropReadThread : MonoBehaviour {

	// Use this for initialization
	private GlobalData gd;
	private float currentRead;
	//private float delta;
	//private float lastRead;
	private Thread readThread;
	private float[] coords;
	public float unityScale = 10;
	private int i;
	private Vector3 newPosition;
	public float smooth = 10;

	void Start () {
		gd = new GlobalData();
		//create and start thread
		/*ReadThread rt = new ReadThread(gd);
		Thread readThread = new Thread(rt.Read);
		readThread.Start();
		*/
		readThread = new Thread(ReadThread);
		readThread.Start();
		print("in start thread, created. ");
		coords = new float[3];
		newPosition = transform.position;
		i = 0;
		//coords[0] = 0;
		//coords[1] = 0;
		//coords[2] = 0;

	}

	void Update(){
		//print("in Update");
		coords = gd.GetReadValue();


		//coords = gd.GetReadValue();
		transform.position = Vector3.Lerp(transform.position,
                                          new Vector3(
                                               coords[0]*10,
                                               coords[1]*10,
                                               coords[2]*10),
                                          .05f
                                          );

	}

	void OnApplicationQuit () {
		readThread.Interrupt();
```

```
	}


	private void ReadThread()
	{
		using(StreamWriter writer = new StreamWriter(@"C:\Users\Steven
					Valeri\Desktop\ThreadLog.txt", true))
		{
			writer.WriteLine("ReadThread Started");

			gd.SetRead(5f, 5f, 5f);
			SerialPort sp = new SerialPort("COM3",9600);
			sp.Open();
			gd.SetRead(10f, 10f, 10f);
			float r1 = 0;
			float r2 = 0;
			float r3 = 0;
			string line = "";
			string[] radii;
			char[] rams = {' '};
			float conversion = .08f;
			float x = 0; float y = 0; float z = 0;
			int i = 0;

			while(true){
				try{
					line = sp.ReadLine();
					writer.WriteLine("Line read: " + line);
					radii = line.Split(rams);

					r1 = (Single.Parse(radii[0])-5) * conversion;
					r2 = (Single.Parse(radii[1])-5) * conversion;
					r3 = (Single.Parse(radii[2])-5) * conversion;

					x = .25f * (float)(Math.Pow(r1, 2)-
							Math.Pow(r2, 2));

					y = .25f *(float)(Math.Sqrt(
							-Math.Pow(r1, 4)
							- Math.Pow(r3, 4)
							+ (Math.Pow(r3, 2) * (6 - 4 * x))
							+ (2 * Math.Pow(r1, 2) * (5 +
								Math.Pow(r3, 2) + 2 * x))
							-  5 * (5 + 4 * x + 4 *
									- Math.Pow(x, 2))
							)
					);

					z = .25f* (float)(3+Math.Pow(r1, 2)-
							Math.Pow(r3, 2) - 2*x);

					writer.WriteLine(x + " " + y + " " + z);

					gd.SetRead(x, y, -z);
					writer.WriteLine("counter: " + i++);
				}
				catch(ThreadInterruptedException e){
					break;
				}
			}
		}
	}
}
```

GlobalData.cs

```
using UnityEngine;
using System.Collections;
using System.IO;

public class GlobalData{

	public int readValue;
	private System.Object lockThis;
	private float[] position;

	public GlobalData(){
		position = new float[3];
		lockThis = new System.Object();
		position[0] = 25f;
		position[1] = 25f;
		position[2] = 25f;
	}

	public void SetRead(float x, float y, float z){
		using(StreamWriter writer = new StreamWriter(@"C:\Users\Steven
					Valeri\Desktop\GlobalDataLog.txt", true))
		{
			writer.WriteLine("xyz: " + x + " " + y + " " + z);
			lock(lockThis)
				{
					position[0] = x;
					position[1] = y;
					position[2] = z;
				}
			writer.WriteLine("positions: " + position[0] + " " +
						position[1] + " " + position[2]);
		}
	}

	public float[] GetReadValue(){
		float[] positionCopy;
		lock(lockThis)
		{
			positionCopy = position;
		}
		return positionCopy;
	}
}
```

RocketInstantiation.cs

```
using UnityEngine;
using System.Collections;

public class RocketInstantiation : MonoBehaviour {

	public Transform prefab;
	void Start() {
		StartCoroutine("MakeBomb");
	}
	// Update is called once per frame
	void Update () {

	}

	IEnumerator MakeBomb() // Not void
	{
		float timeDelay = 2;
		float xRand = 0;
		float zRand = 0;
		while (true) {
			xRand =  Random.Range(-5.0f, 5.0f);
			zRand = Random.Range(-5.0f, 5.0f);
			yield return new WaitForSeconds(4.0f);
			Instantiate(prefab,
					new Vector3(transform.position.x+xRand,
							transform.position.y,
							transform.position.z+zRand),
					prefab.transform.rotation);

		}
	}
}
```

DestroyRocket.cs

```
using System.Collections;

public class DestroyRocket : MonoBehaviour {


	public Transform explosion;
	public Transform fireworks;
	public Transform sheepPrefab;
	private float xRand;
	private float zRand;
	private int i;
	private Transform[] sheep;
	// Use this for initialization
	void Start () {
		i=0;
		sheep = new Transform[10];
		xRand =  Random.Range(-5.0f, 5.0f);
		zRand = Random.Range(-5.0f, 5.0f);
	}

	void OnCollisionEnter(Collision collision){
		if(collision.gameObject.tag == "Player"){

			Instantiate(fireworks,  collision.transform.position,
					Quaternion.identity);

			xRand =  Random.Range(-5.0f, 5.0f);
			zRand = Random.Range(-5.0f, 5.0f);

			sheep[i++] = (Transform) Instantiate(sheepPrefab,
								new Vector3(
							transform.position.x+xRand,
							0,
							transform.position.z+zRand),
						sheepPrefab.transform.rotation);

		}
		else{
			Instantiate(explosion,
					new Vector3(transform.position.x,
					transform.position.y-4,
					transform.position.z),
					Quaternion.identity);
		}

		Destroy(gameObject);    //destroys the missile
	}
}
```