Augmented Reality

Visual Orientation System (VOS)

Client: Abram Balloga

Team: Keith Risper

Wayne Landini

Start Date: May 14, 2012

End Date: June 22, 2012

Introduction

Our client is an independent inventor name Abram Balloga who came to us with an Augmented Reality project. Augmented reality is a view of the physical world enhanced by an added simulation. The specific name for his Augmented Reality project is called the Visual Orientation System or VOS for short. His vision for this project was to take a pair of rangefinder binoculars and add GPS capability to it as well as a visual display. There have been some products that have already implemented optics with visual displays but nothing quite like the product that our client had envisioned. His idea has been patented but not implemented in any way. We would attempt to accomplish this implementation by physically connecting a GPS chip to the circuit board of the binoculars and by using the FPGA chip on the board to handle all the calculations needed. The idea was to acquire a point of interest while looking through the binoculars and use the range finder to find the distance to that point of interest (POI). You would then calculate the GPS coordinate of that POI and save that point as a reference while moving around. The POI would be saved into the visual display of the binoculars and the user could then find the direction and distance of that POI at any position. All of these things would incorporate the use of inclinometer, rangefinder, GPS, and compass hardware. This final prototype that the client envisioned would benefit him by giving a rough prototype to demo to companies when explaining his VOS concept.

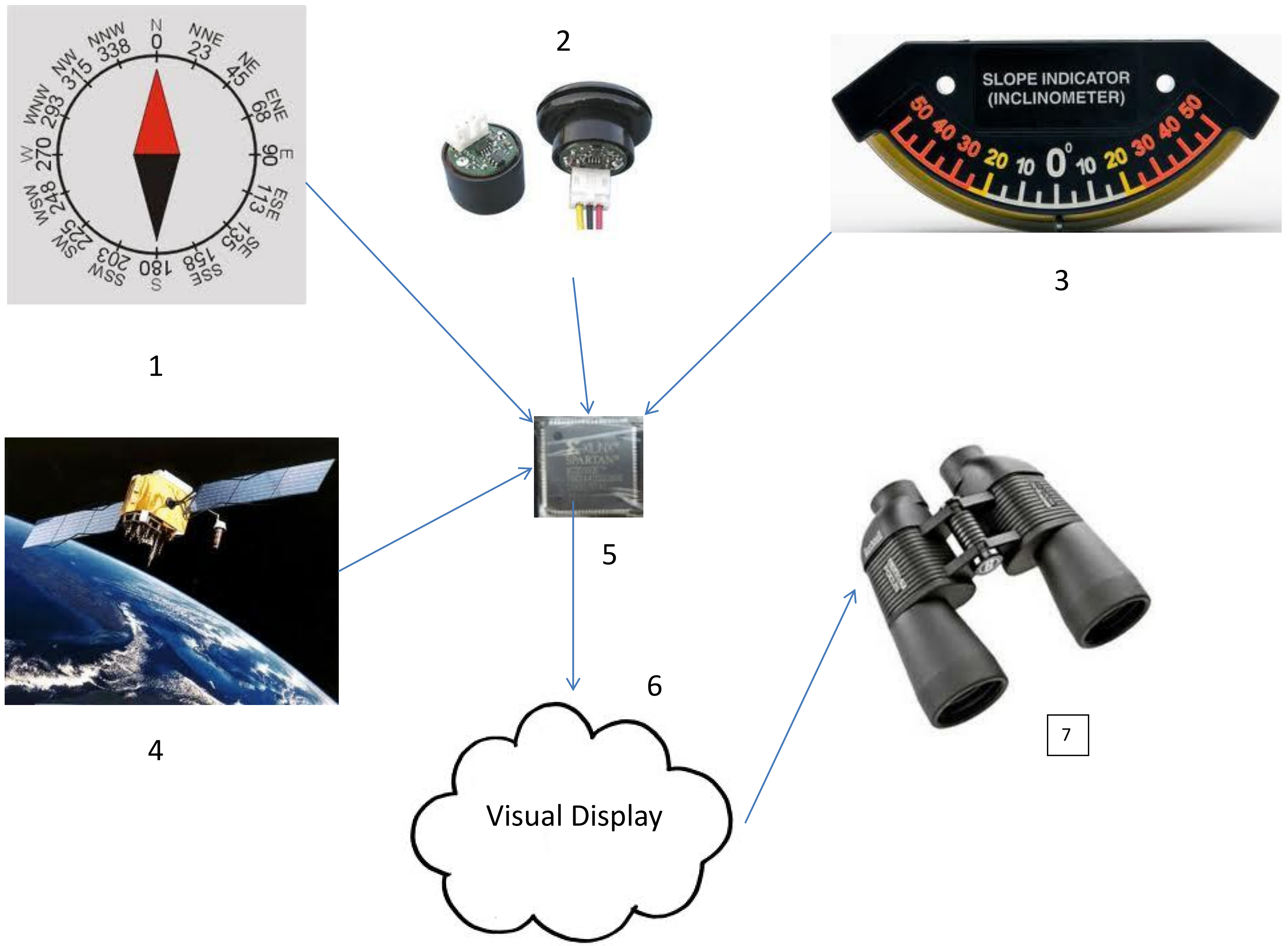
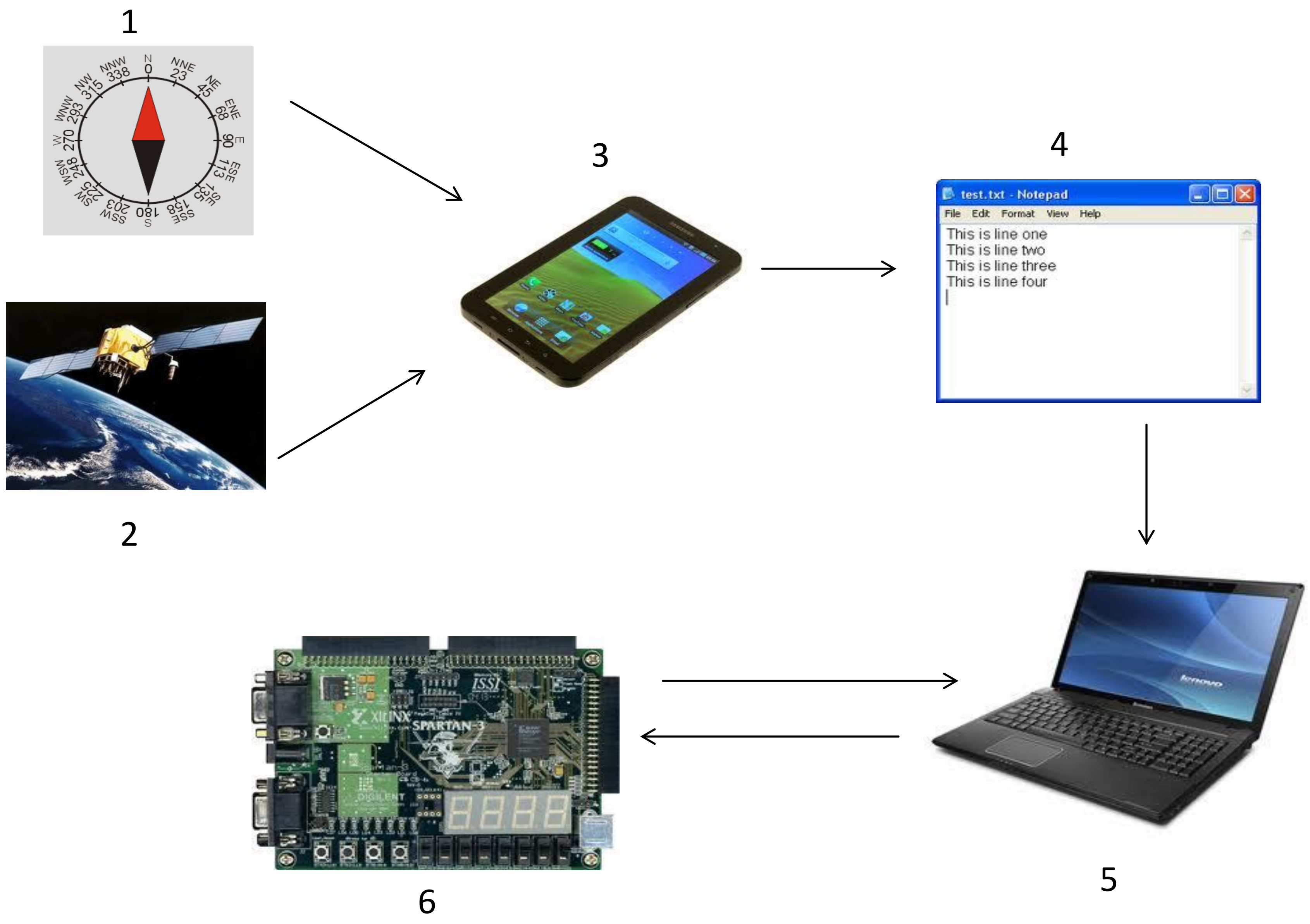


Figure 1.

System architecture of what the client had envisioned



This is line one
This is line two
This is line three
This is line four

Figure 2.

System Architecture of what we actually ended up doing

System Architecture

For figure 1, the FPGA chip (5) is handling all of the needed calculations for the VOS prototype. The inputs for the calculations are coming from the GPS chip (4), the compass hardware (1), the rangefinder (2), and the inclinometer (3). The results of these calculations are then fed to the visual display (6) which is then seen through the binoculars (7). For figure 2 the calculations are still handled by the FGPA but the FPGA chip is now on a FPGA starter kit (6). All of the hardware pieces of figure 1 are now being simulated with the hardware on the Android tablet (3). The Android has GPS hardware (2), compass hardware (1), but no rangefinder so the rangefinder had to be simulated by a human estimate of the distance. There had to be some way to transfer the GPS and compass data to the FPGA starter kit so that is where the text file comes in (4). This text file would be moved to a PC (5) and then into the FPGA code. While the Android does have the ability to produce inclinometer measurements, we did not try and simulate these values due to time constraints. Similarly, we also did not try and simulate a rangefinder using the camera on the Android because the calculations involved with doing that from a camera differ from the calculations one would use with lenses from a mono/binocular. It would have been outside the scope of this project.

State Diagram to Illustrate Calculations

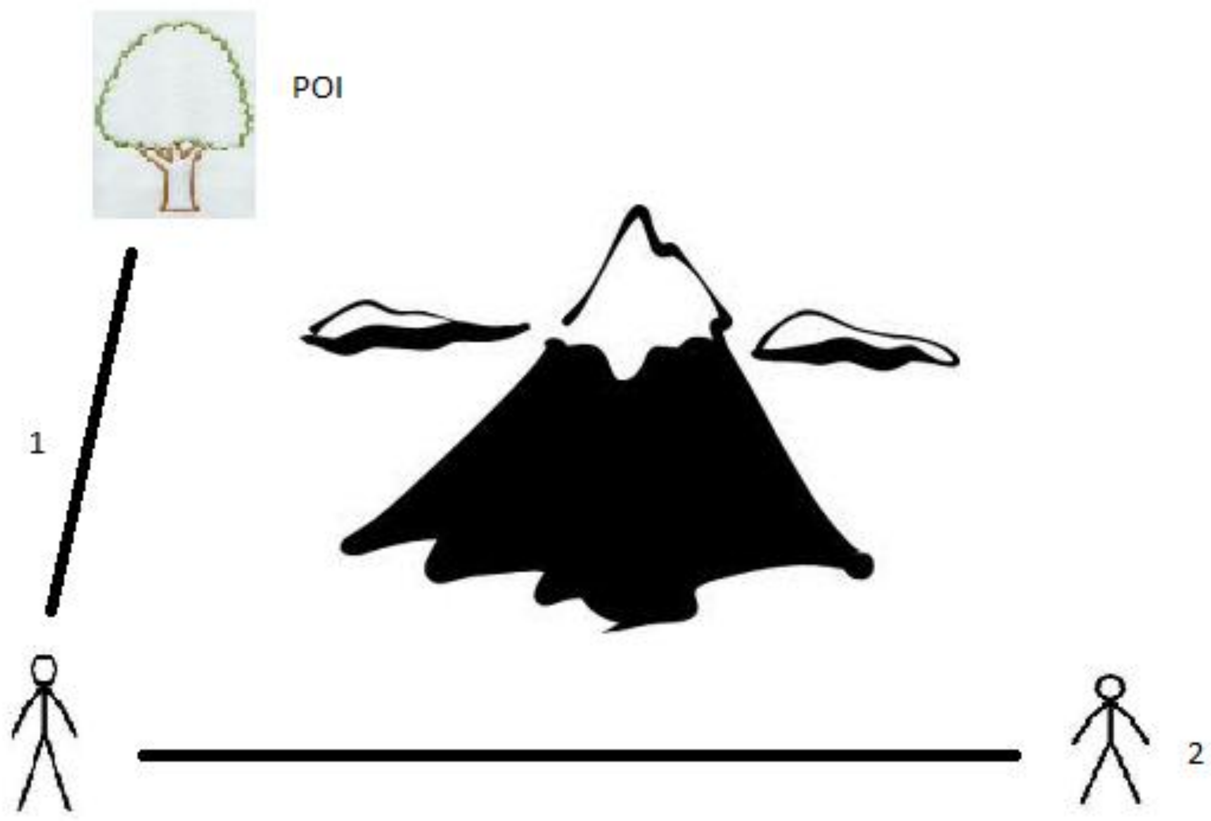


Figure 3.

Calculation Illustration

Calculations Explanation

This section explains which calculations are used for each situation you would encounter when developing the VOS prototype. State 1 in the diagram is when the user sees a point of interest that they want to save. There is one calculation with 2 parts for this situation. It is the following:

$$lat_2 = asin(sin(lat_1)*cos(d/R) + cos(lat_1)*sin(d/R)*cos(\theta))$$

$$lon_2 = lon_1 + atan2(sin(\theta)*sin(d/R)*cos(lat_1), cos(d/R) - sin(lat_1)*sin(lat_2))$$

Lat1 and lon1 are the GPS coordinates of the user and these would be acquired by the GPS chip that would eventually be attached to the binoculars. d is the straight line distance to the POI from the user and would be acquired by the range finder. R is the radius of the earth and can be in any units but the units must be the same as the distance (d). Theta(θ) is the bearing to the POI and this would be acquired using the compass hardware implemented in the binoculars. Note that this would be in degrees clockwise from north (0-360). Also note that all arguments to the trig functions must be converted to radians. Once you have all these things acquired from their respective hardware components you can input them into the equation which would be executed by the FPGA. The results (lat2, lon2) would be the GPS coordinate of your desired POI and would have to be saved for further reference.

The next situation is when the user moves out of the line of sight of the POI but must find it again. This is situation number 2 in the diagram. Notice the mountain blocking the line of sight to the original POI. You would need 2 different equations for this situation, one to find the distance to your saved POI (the haversine equation) and one to find your bearing to the POI. To find the distance to your POI you would use this equation:

$$a = sin^2(\Delta lat/2) + cos(lat_1) * cos(lat_2) * sin^2(\Delta long/2)$$

$$c = 2 * atan2(\sqrt{a}, \sqrt{(1-a)})$$

$$d = R * c$$

Δlat and Δlong are the difference in latitude and longitude between the user's current point and the saved POI. Lat1 and lon1 refer to the user's GPS coordinate and lat2 and lon2 refer to the GPS coordinate of the saved POI. The GPS coordinate of the POI was calculated and saved with the first equation above. With all these inputs you can then find the distance to your POI despite not being in line of sight. The next equation that you would need in order to find what bearing you should be at to face the POI is as follows:

$$\Theta = \text{atan2}(\sin(\Delta\text{long}) * \cos(\text{lat2}) , \cos(\text{lat1}) * \sin(\text{lat2}) - \sin(\text{lat1}) * \cos(\text{lat2}) * \cos(\Delta\text{long})$$

All of the variables refer to the same objects that they do in the haversine equation above. The result of this equation is the bearing you should be at to be facing your POI. With all of the results from these equations you should be able to implement all of the visual display that the client had envisioned. The system would have to constantly be updating the user's GPS coordinates (from the GPS chip) and compass bearing (from the compass hardware) in order for the graphical display to work in real time. This also means that the FPGA would have to constantly be executing the second and third equation with the updated user coordinates. Also in order for the arrow of the graphical display to point in the right direction, you would have to find the user's current bearing and subtract it from the required bearing and use the result as a reference as to where the arrow should be drawn but this an algorithm that depends on how the visual display is implemented.

Technical Design

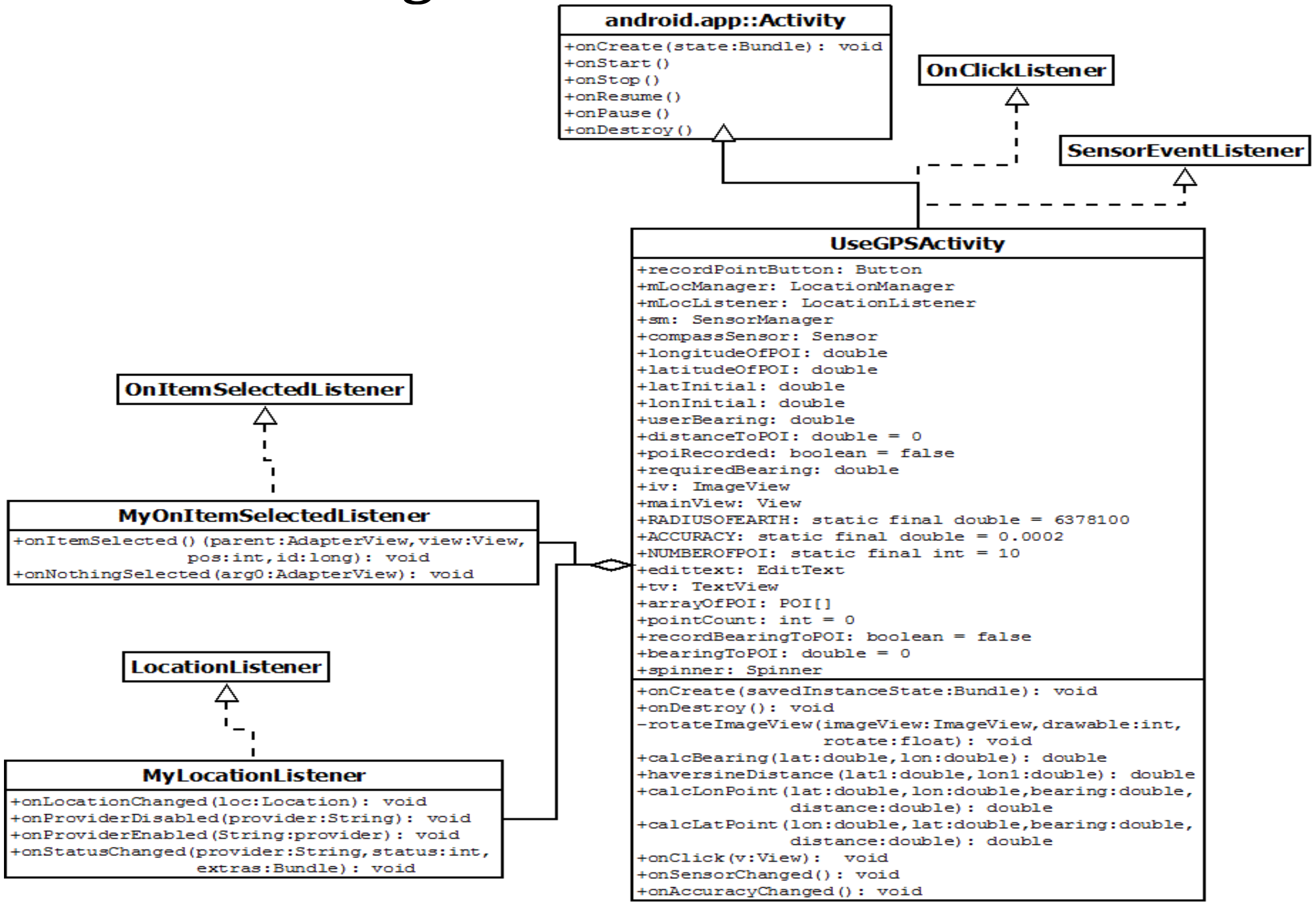
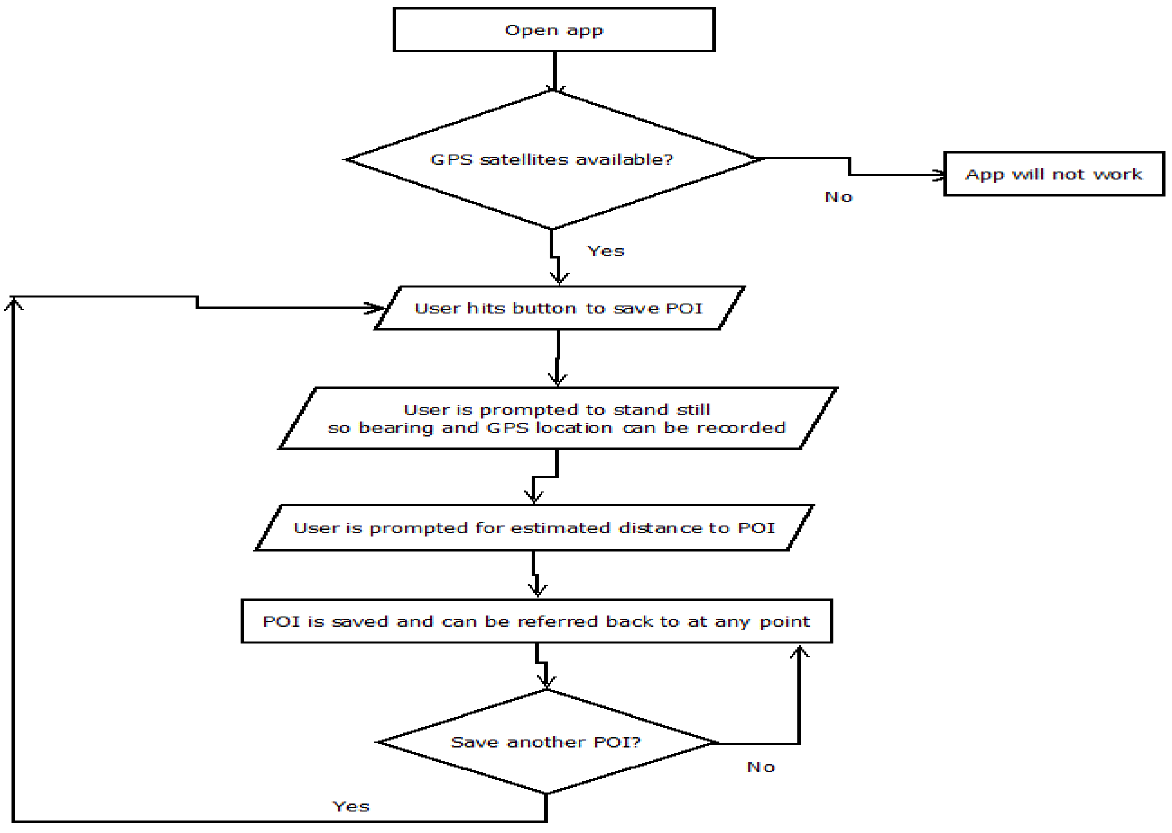

android.app::Activity
+onCreate(state:Bundle): void
+onStart()
+onStop()
+onResume()
+onPause()
+onDestroy()
OnClickListener
SensorEventListener
UseGPSActivity
+recordPointButton: Button
+mLocManager: LocationManager
+mLocListener: LocationListener
+sm: SensorManager
+compassSensor: Sensor
+longitudeOfPOI: double
+latitudeOfPOI: double
+latInitial: double
+lonInitial: double
+userBearing: double
+distanceToPOI: double = 0
+poiRecorded: boolean = false
+requiredBearing: double
+iv: ImageView
+mainView: View
+RADIUSOFEARTH: static final double = 6378100
+ACCURACY: static final double = 0.0002
+NUMBEROFPOI: static final int = 10
+edittext: EditText
+tv: TextView
+arrayOfPOI: POI[]
+pointCount: int = 0
+recordBearingToPOI: boolean = false
+bearingToPOI: double = 0
+spinner: Spinner
+onCreate(savedInstanceState:Bundle): void
+onDestroy(): void
-rotateImageView(imageView:ImageView,drawable:int, rotate:float): void
+calcBearing(lat:double,lon:double): double
+haversineDistance(lat1:double,lon1:double): double
+calcLonPoint(lat:double,lon:double,bearing:double, distance:double): double
+calcLatPoint(lon:double,lat:double,bearing:double, distance:double): double
+onClick(v:View): void
+onSensorChanged(): void
+onAccuracyChanged(): void
OnItemSelectedListener
MyOnItemSelectedListener
+onItemSelected()(parent:AdapterView,view:View, pos:int,id:long): void
+onNothingSelected(arg0:AdapterView): void
LocationListener
MyLocationListener
+onLocationChanged(loc:Location): void
+onProviderDisabled(provider:String): void
+onProviderEnabled(String:provider): void
+onStatusChanged(provider:String,status:int, extras:Bundle): void

Figure 4.

UML Diagram for VOS Simulator App


Open app
GPS satellites available?
No
App will not work
Yes
User hits button to save POI
User is prompted to stand still so bearing and GPS location can be recorded
User is prompted for estimated distance to POI
POI is saved and can be referred back to at any point
Save another POI?
No
Yes

Note: Every time the user changes position which is almost always, the app displays a toast(text display) that shows the GPS coordinates of the currently selected POI, the current GPS coordinates of the user, the distance to the current POI, and the GPS coordinates of each saved POI.

Note: User can change the current POI to any previously saved POI via a drop down box.

Figure 5.

Flow Chart for VOS Simulator

Design and Implementation Decisions

We decided it would be feasible for us to model the VOS so we decided to continue working on the Visual Orientation System project. When making the decision to continue with the project, instead of pursuing a new one, we had to choose how we were going to model all the various parts of the VOS prototype. To model the FPGA chip located on the board of the rangefinder binoculars we decided to obtain a FPGA starter kit that had a similar chip. This was chosen because we could prove that the FPGA in the binoculars could do all the calculations needed if there was a way to interface with it. The starter kit provided an easy way to interface with the FPGA. At first glance we assumed all we would need for calculations was GPS coordinates so we had to find a way to get GPS coordinates to the FPGA for calculations. We decided to use either a smart phone or tablet for this since they have GPS hardware built in. We had access to either an IPhone or an Android Tablet. We chose the tablet because Android uses Java to code with and iOS uses Objective C and we were much more familiar with Java. We would use the Android to acquire several GPS points and write them to a file. This file would then be sent to a PC and the FPGA would use these points to do calculations with.

For the FPGA part we chose to go with a Basys2 FPGA starter board from Digilent Inc. It was very affordable and most likely had more features than we needed. It gave us a functional way to interface with a FPGA chip identical to what we found on the binoculars and had a VGA port for possibly modeling a visual display. When interfacing with the FPGA, we had to choose a hardware description language (HDL) to program the calculations onto the chip. The FPGA maker, Xilinx, provides the programmer with an IDE, called Xilinx ISE, for programming their new FPGA. The two main languages supported within the IDE are Verilog and VHDL. There is some support for using other languages though. We chose to go with Verilog because it is supposed to be the simpler of the two popular languages to learn and syntactically it is "C-like." We also had to choose what number type we would be using when programming with the FPGA. We initially went with double precision floating type numbers but later on in the project we realized this wouldn't work because of the difficulty to do this with the FPGA hardware. Slowly it became clearer that we were not going to be able to carry on with our original plan of writing GPS points to a file, sending that file to a PC, and then having the FPGA do calculations on those points. This led us to our next decision. We decided to forget about trying to write the points to a file. We decided to do the entire modeling of the VOS system with the Android and to do some calculations with some arbitrary points with the FPGA in order to prove that it was capable of doing these calculations. Even this task became very difficult with the FPGA so our last and final decision was to illustrate an example of how you would have to go about doing calculations on an FPGA. This would involve look up tables for those types of functions.

Requirements

Initial requirements

- Physically attach GPS chip to circuit board of rangefinder binoculars
- Incorporate GPS coordinates into calculations that are done by the FPGA chip already attached to the rangefinder binocular circuit board
- Create a visual display that is seen while looking through the binoculars

Requirements Version 2

We discovered that it was not possible to attach the GPS chip to the board that the binoculars used because the board was not made for expansion. We also discovered that we could not actually reprogram the FPGA on the binoculars because for the same reason. New requirements:

- Obtain a FPGA starter kit to model the FPGA chip in the binoculars.
    - Get familiar with HDLs (Verilog and VHDL) (non-functional requirement)
    - Be able to interface with the FPGA starter kit (non-functional requirement)
- Find a way to obtain GPS coordinates and be able to send them to the FPGA. (This would be an Android tablet.)
- Create an Android application that acquired the GPS coordinates at several points and write these points to a file. The overall purpose of the app was to simulate as much hardware as possible.
- Transfer the file over to a PC.
- Use the PC to interface the file with the FPGA starter kit and write code to get the FPGA on the starter kit to do the calculations that the FPGA in the binoculars would have done.

- Write the results from the FPGA calculations back onto a file.
- Send this file to a PC and use Java to model the visual display

Requirements Version 3

File I/O is another process that is not trivial when programming an FPGA. We ran into difficulties when trying to read data from a file into the FPGA. New requirements:

- Expand the app into modeling the visual display.
- Continue to attempt to do the calculations via the FPGA hardware but with a better direction of how to do it.

Requirements Version 4

Floating point operations were feasible but could not be used because the HDL support library that we used would not work with on the FPGA, only in simulations on the computer. New requirements:

- Scrap trying to do floating point calculations with the FPGA and focus on the method (look up tables) of how you would do fixed point calculations.
- Extend the Android app into completely modeling the entire VOS prototype.
  - Visual display that points the user back to the saved POI.
  - Display the distance of the current POI from the user.
  - Be able to save several POIs. (non- functional requirement)

FPGA Results

We encountered a number of issues related to programming the FPGA. All the calculations that we needed to use involved GPS coordinates and trigonometric functions. GPS coordinates have latitude and longitude values formatted in decimal degrees (eg. 112.8654121°). HDLs have a data type called "real" that handles values of this sort. When using the real data type though, your HDL code is not synthesizable and unable to be run on the FPGA. For HDL code to be synthesizable there must be a hardware construct able to handle the data type. While reals are a data type, there is no way to represent a decimal number in hardware. It initially made just reading GPS coordinates and file I/O a real challenge. One way around this is to use floating or fixed point number formats. Based on past experiences we chose to go with a floating point representation because we weren't sure of what kind of precision and accuracy the customer wanted or needed. Although there is HDL support (much like a library in software languages) for floating point, the library could be used but was not synthesizable.

This left only the possibility of fixed point numbers being used to represent the latitude and longitude. A fixed point number has an integer part and a fractional part and is represented in the format of Q**integer**.**fraction**. An example of a fixed point number could use the format Q2.2. So the number 3.25 would be represented in binary as 1101. This number format would be precise up to two places after the radix (decimal) and accurate up to 2^{-2} = 0.25 / 2. Addition/Subtraction works the same as integer addition/subtraction with 1101(3.25 decimal) - 0010(0.5 decimal) = 1011(2.75 decimal). Multiplication/Division would work in a similar manner. 1101(3.25 decimal) * 0010(0.5 decimal) = 00011010(1.625 decimal). It is of particular interest to note that multiplication returns a fixed point number in the format of double the size of the original Q[integer+integer].[fraction+fraction]. In order to get the same size floating point number back, the programmer would have to shift the bits left by the original number of fixed point bits and drop the rest of the bits. So your actual value after multiplication would be 0110(1.5 decimal).

Another issue we ran into was that we thought there would be HDL support for trigonometric functions. We were unable to find a synthesizable library that supported trigonometric functions. We found that in order to properly model these functions, a programmer would have to code a lookup tables for these functions to be read into ROM during compilation and then be referenced from within their program. Choosing the format for the lookup table would come down to how efficient, precise and accurate ones calculations needed to be. This would need to be considered for all fixed point arithmetic, trigonometric functions and square root approximations that the programmer decided to use along with approximations for square root. Once the programmer figured out what type of accuracies are ok within the application, then it would be just a matter of generating a useful table. An example of this would be using a sin lookup table. You could generate a table value for each degree in a circle (0-360) value but depending on what hardware and how precise values need to be, this could get costly. Instead you could try and optimize it a little by only calculating 0-90 degrees since sin is a periodic function and adjust signs where needed for anything above 90. The same rules would apply for a cosine lookup table. If you give it a closer look though, room could be saved in ROM if you just do an additional fixed point arithmetic step using the sin lookup table and use the formula cos(x) = 1 - sin(x). This is the kind of details a programmer would have to work out before doing any hardware code.

VOS Simulator (Android App) Results

The final product is an app that has an arrow dynamically pointing in the direction of any one of 10 POI's that the user has saved. The background color changes from white to red when the user is in a certain vicinity of the current POI. This vicinity is set at about a 20 m radius. There are some features and bugs, that given more time, we would have liked to work out. These are the following:

<u>Minor bugs that we would have fixed given more time</u>

- If the GPS location has not been set by the Android hardware then the app will crash. (If you are somewhere where you can't get a GPS signal or the signal hasn't been acquired yet the app will crash.)
- There is a toast (text display) that displays the user's lat/lon, the current POI lat/lon, the numerical bearing the user should be at, and the lat/lon of all the saved POI's. This numerical display is very slow to update. The arrow direction updates just fine but the numbers in the display do not. Would like to try and make this update faster but not sure if it is even possible.
- App only runs when tablet is in the landscape view orientation. App hasn't been tested on smart phone so not sure how it would react on a smart phone.

Android GPS hardware

The GPS and compass hardware that is found on most Android devices is not ideal for this type of project. Tablets and smart phones are not meant to do this kind of thing with high accuracy and precision. Therefore the manufacturers use cheaper hardware. When continuing on this project, this is something that you would want to consider. In our experience working with the app, we would personally suggest getting a GPS chip that is as accurate and precise as possible but still within the price range of the client. Although this all depends on how accurate the client wants his product to be.

Appendix A – Digilent Inc Basys2 board

Product Information and Pricing

http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,790&Prod=BASYS2

Software

- Adept 2.0
- Xilinx ISE design suite

Appendix B – Xilinx XC3S100E Specs

Data Sheet

http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf

Appendix C – VOS Simulator(Android App)

Product Installation Instructions

- Get a tablet or smart phone running the Android operating system.
- Connect the Android device to a PC with USB connectivity.
- Make sure USB debugging is enabled on the Android device.
- Install the Eclipse IDE, the Android SDK, and all of the Eclipse plug-ins needed to develop Android apps in Eclipse.

- Import the provided Android project into the Eclipse environment.
- Set up the Eclipse device manager to work with your specific Android device.
- Run the project.
- Choose your device when prompted.
- The app should automatically install on your Android device.

Development Environment Description

- App developed in the Eclipse Indigo IDE using the Android SDK.
- App tested on a Samsung Galaxy Tablet (Model: GT-P7510)

Calculation Details

- Obtained equations from http://www.movable-type.co.uk/scripts/latlong.html
- Used double precision floating point variables in order to get most accurate results possible.
- Used equations from the website under the headings labeled **Bearing** and **Destination point given distance and bearing from start point**

Modeling Technique

- Used a dynamically turning arrow to model the visual display originally wanted for the binocular rangefinder.
- Modeled the GPS chip originally given to us by taking advantage of the Android GPS hardware capabilities.
- Modeled the compass hardware of the binoculars by taking advantage of the Android compass hardware capabilities.
- Modeled the range finder capabilities of the binoculars by having the user of the app estimate the distance.