# Newmont 4- TabletMapper

Justin Pinkul, Michael Coughlin
June 23, 2011

# Abstract

Newmont Mining employs many different scientists and engineers in their mining operations, responsible for gathering data to be used in company operations. This data is often a simple set of polygons representing geologic areas, that is stored in a database and later analyzed using an existing system. This information is currently gathered using pen-and-paper techniques, but can be extended to a mobile application based off the Android platform.

This project aimed to create an Android application that measures and uploads this data into a database in a format that is compatible with the current system in use by Newmont. The primary goals of this project were to 1) create an easy to use interface contained in an Android application that allows the mapping of geological segments using GPS, and 2) sync the collected data with an external database. To accomplish this, we used several tools, including the Android SDK and Eclipse IDE in order to develop the application as well as SQL and SQLite databases to store the data. We primarily programmed this project using Java, C++ and XML.

During this semester, we developed an Android application that takes advantage of the GPS module to gather geologic data. We also developed an external Java application that interfaces with a SQL database that is used to store the data in a form that is compatible with the current system in use by Newmont Mining. Several integration tests of this entire system were run successfully, and the project meets all of the requirements that were initially set forth.

# Table of Contents

## Contents

# Introduction

The Newmont 4 project is intended to solve Newmont Mining's issues of recording data in the field. Currently, Newmont must collect their data by hand and then enter it into a single computer upon completion. This project will develop an application for the Android platform that will measure data in the form of polygons that represent boundaries of waste or mineral deposits, which will be measured by geologists in the field. Once all the data is collected, the application will then synchronize the data with an external database in use by Newmont.

# Requirements

Our client requested that we create an Android application that is designed for use on a tablet. This program should be able to track the users location and allow the user to record that location, as well as allow user to add lines and polygons between recorded locations.

Our client defined these requirements for the project:

**Functional Requirements**

1. Record user location as they walk around a geological site.
2. Allow the user to create lines and polygons using their location as vertices.
3. Save the location and polygon data onto the tablet device.
4. Sync the data from the device with the existing system.
5. Read in a topographical background map from a binary DIG file.
6. Output the data into Newmont's currently existing system.

**Non-Functional Requirements**

1. Using Android and the Android Software development kit (SDK).
2. Programming using Java and Extensible Markup Language (XML), which are the languages that Android uses.
3. Using a SQLite database to store the data.
4. Using Newmont's dynamically linked libraries (DLLs) to read DIG files.
5. Utilize OpenGL ES for rendering.

**Optional Requirements**

1. Provide the option of mapping continuously while walking.
2. Store attributes related to the polygons in the form of text.
3. Create and store video or pictures related to the polygons using an on-board camera.

# High Level Design

Currently Newmont uses mapping software that requires powerful machines, which is a problem because it requires full sized computers in the field. Their existing software is capable of many advanced manipulations of geological data, however this is not necessary in the field where the data is being collected, and our Android application aimed to fill that gap.

Our Android application acquires location data from the on-board Global Positioning System (GPS) module in the Android device, and stores data in a local Structured Query Language Light (SQLite) database. We also wrote a Java application that is intended to be installed on a computer that the Android device will be connected to over the Internet or a local network. Additionally, prior to being used in the field, the application will need to be loaded with the topological map for the area, as well as any existing polygons that have already been collected. This data will be loaded onto the application using the external Java application before the device leaves a Newmont office.

When connected to a computer over a network, an installed Java application will establish a connection to the Android device and retrieve any data that is stored in the database. This data is then processed and stored in one of Newmont's databases in a format compatible with the existing system.

## System Architecture

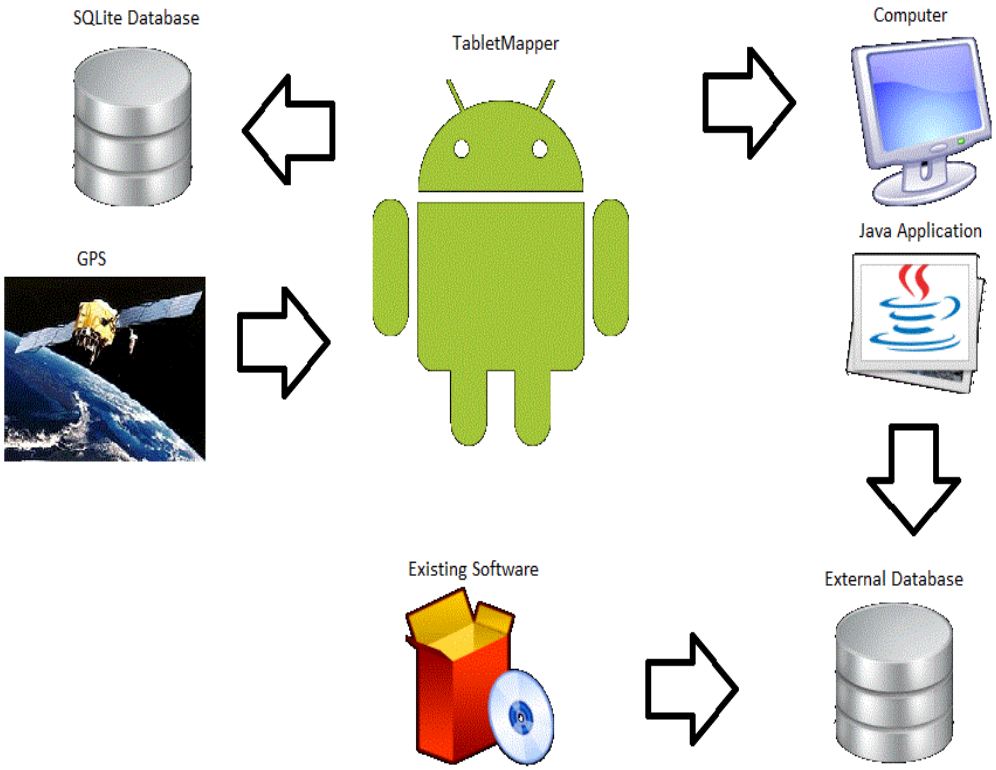


*Figure 1: System Architecture Diagram*

In order for the java application to read in the topographical map, from a DIG file (a custom Newmont binary file), it needed to be able to use Newmont's existing code, provided in the form of several DLLs. Theses DLLs are written in C/C++ and cannot be directly used by a Java application, therefore there must also be a Java Native Interface (JNI) bridge between the

application and Newmont's DLLs. This meant that a C++ program needed to be developed, while complying with JNI conventions, to call functions from the DLLs and read in the map.

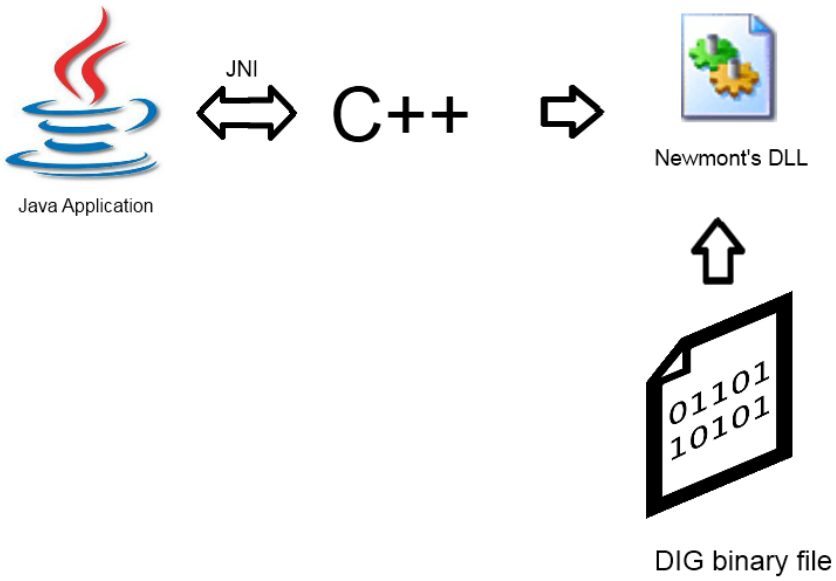


*Figure 2: System Architecture Diagram, Java application*

## The Android Tablet

The UML diagram for the Android program has three distinct sections: user interaction, data management and graphics rendering, distinguishable by red, green and blue respectively in Figure 3 below. The EventManager class, whose purpose is to receive events from the user interaction section and forward those events to either the data management or rendering sections, connects these sections. The four data types, Point, Line, OreconData and DigData, that store shape information are also shared between the three sections and can be seen in the center of Figure 3.

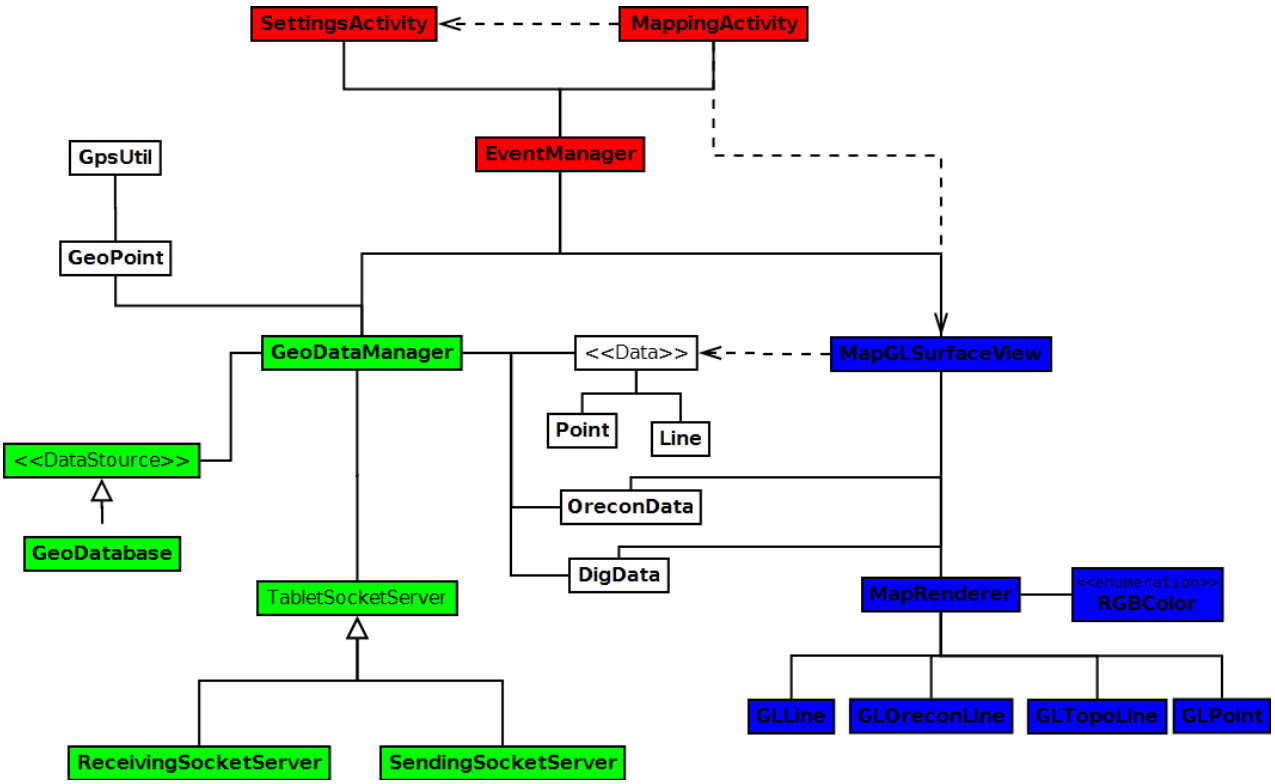


*Figure 3: Android UML diagram*

The user interaction section, shown below in Figure 4, consists of the basic screens and buttons the user sees. This section is the source of all user interaction, ranging from pressing a button to triggering the GPS by moving. This section relies heavily upon Android libraries and contains most of the Android-specific code.

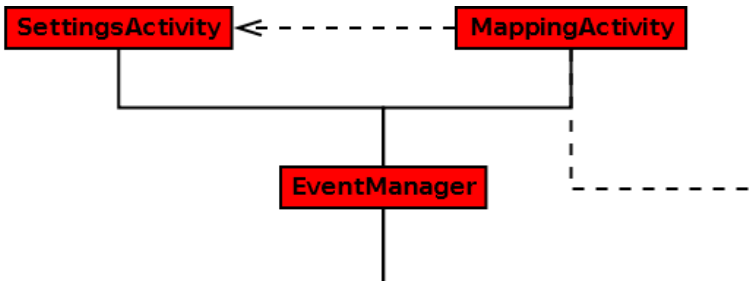


*Figure 4: Android UML diagram, User interaction section*

The graphics rendering section, shown below in Figure 5, is responsible for all OpenGL ES rendering. This includes features such as displaying point, line, and polygon data and causing the main display to zoom in and out. The physical entities (i.e., points, lines and polygons) are displayed relative to each other, meaning that a point 20 feet away from the user's current location would appear to be twice as far as a point that is only 10 feet away.

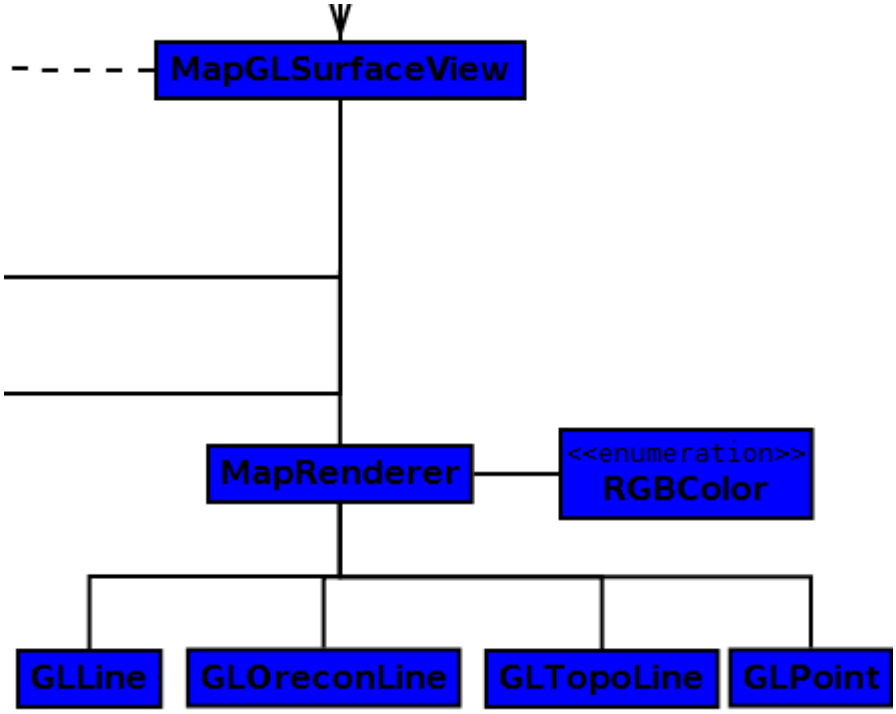


*Figure 5: Android UML diagram, Graphics rendering section*

The data management section, shown below in Figure 6, controls data persistence and synchronization. The primary functions of this section are storing information on the local database and transferring that data to the external java application for synchronization, once a network connection is established.

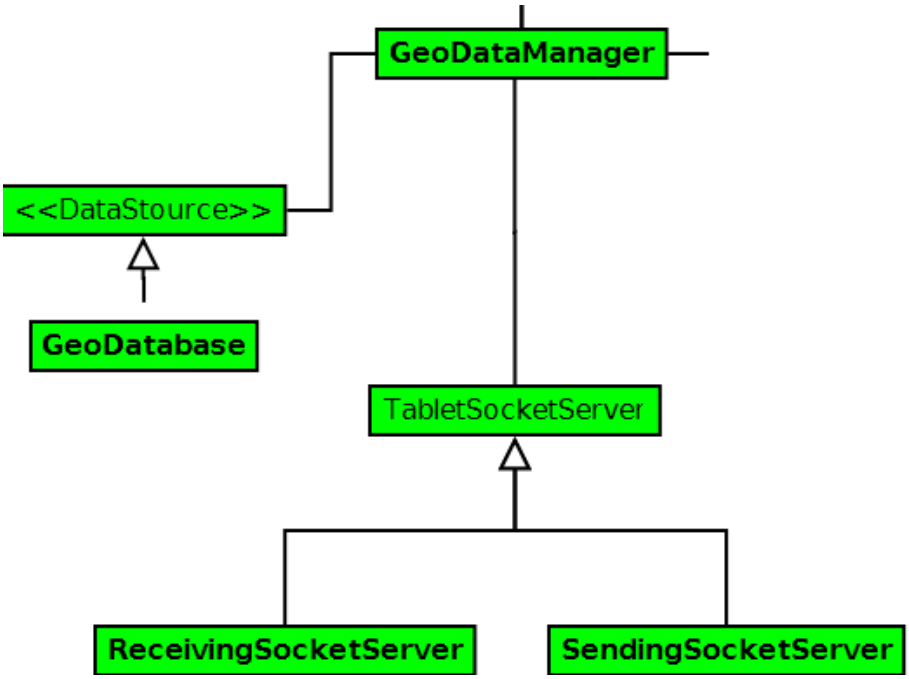


*Figure 6: Android UML diagram, Data management section*

## The External Java Application

The UML diagram for the External Java Application, shown below in Figure 7, represents the user interface and the various interactions with other components. The user interface consists of three tabs, one for sending data to the tablet, another for syncing data from the tablet with Newmont's OreconDB (database for storing geologic data), and the third for changing application settings. These panels then use the Communications class to request their various tasks; the Communications class then connects to the tablet and/or OreconDB to make the necessary changes. The Communications class is also responsible for requesting DIG file information from the native DigReader class, which then uses the JNI to connect to the C++ program and read the file into a DigData object.

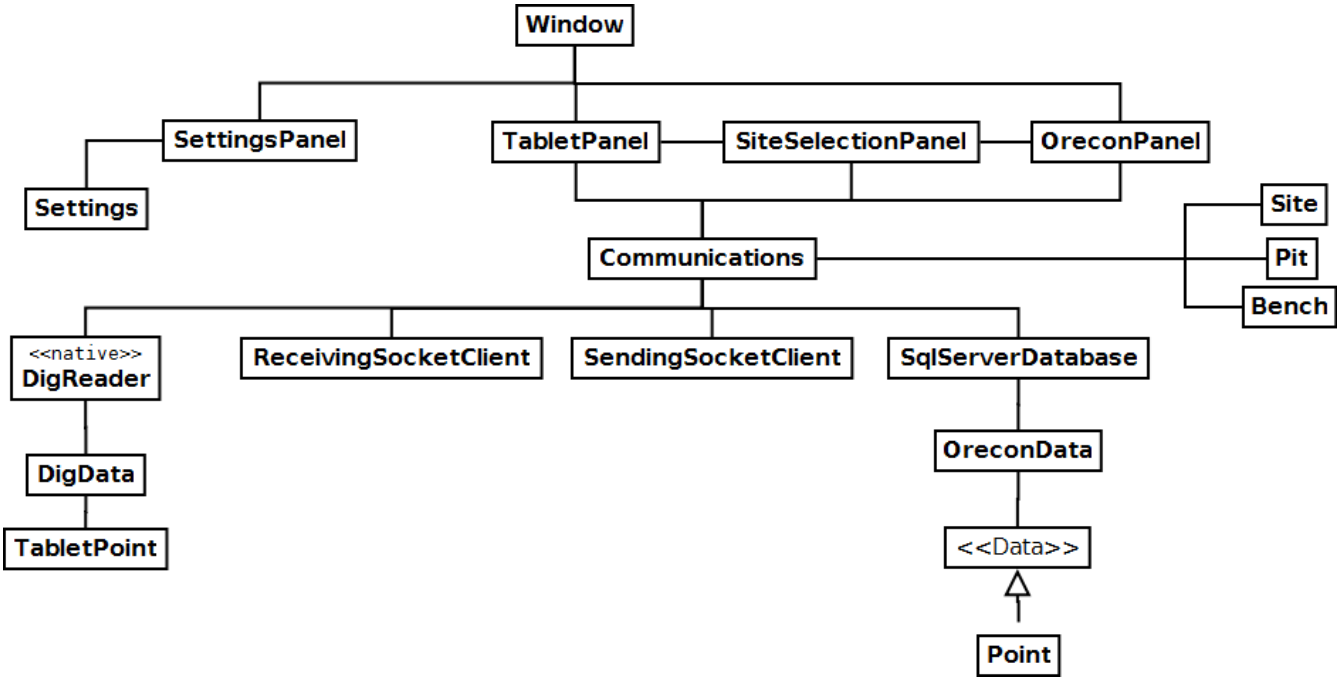


*Figure 7: Java application UML diagram*

## Database Schema

### On-board SQLite database

The on-board SQLite database is used to store point and polygon data for the particular area that is being studied. The database has tables to store points, lines and polygons, as well as an origin point that is used to calculate the distance, in feet, of the other points in relation to this origin.

- The Origin table has one entry for each set of data that has been loaded onto the tablet. This entry contains two doubles representing the latitude and longitude of the origin point.
- For each point in the points table a unique id and two doubles, that represent the x and y distances from the origin in feet, are stored.

- The lines table stores a unique id for each line, the id of the starting and ending points of the line, and the id of a polygon that the line belongs to, or null if the line does not belong to a polygon.
- The polygon table stores a unique id for each polygon, the polygon type, and any notes that are associated with the polygon.

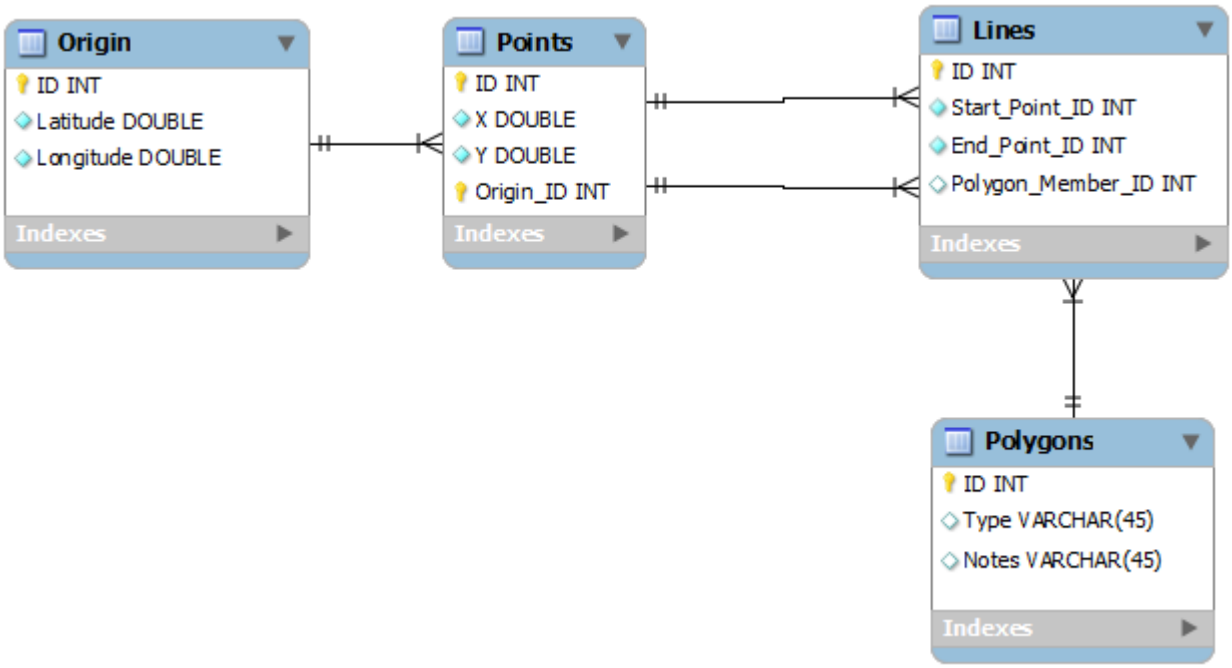


*Figure 8: SQLite database schema*

### Newmont's OreconDB

Newmont's OreconDB is a fairly complex database, however, much of the stored information is unnecessary for our purposes. The relative portion of the schema can be seen below in Figure 9. The schema consists of three tables that define where data is located and an additional table for the data itself. A site is the broadest definition of an area and typically defines what part of the globe you are in. Pits are more specific, as they indicate the part of a site that is being examined, and each site can have multiple pits. The pit bench is the last step in locating where to place the data and refers to the elevation inside of a pit. The polygon table itself has fields for what bench it belongs to, point data, creation date, creator and the geologic zone it is classified as.

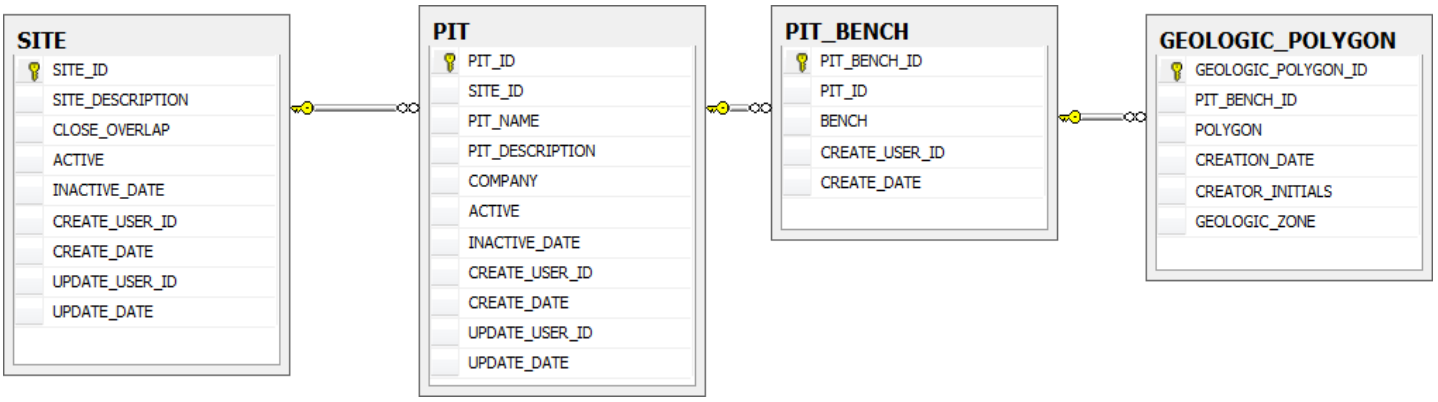


*Figure 9: OreconDB SQL Server database schema*

# Description of Modules

## External Java Application

This application is the portion of the project that is responsible for communication with the tablet device. The application loads the initial data before collection and reads the collected data after collection is complete.

### Uploading Existing Data to Tablet

The application will read a topological area map from an existing Newmont file and then load the area map into the Android application. The application will also load existing point, line and polygon data from Newmont's OreconDB and store this data into the SQLite database on the Android device.

### Syncing Collected Data from Tablet

The application will also read point and polygon data from the SQLite database on the Android device and upload it to Newmont's OreconDB, while leaving any unchanged data intact. Table 1 shows the SQL command needed to insert data into OreconDB as well as the SQL statements needed to present the user with the proper choices of where to store the data to.

OreconDB SQL statements

| Action | Data | SQL |
|---|---|---|
| List available sites | none | Select SITE_ID, SITE_DESCRIPTION from SITE |
| List available pits | int site_id | Select PIT_ID, PIT_DESCRIPTION from PIT where SITE_ID=site_id |
| List available benches | int pit_id | Select PIT_BENCH_ID, BENCH from PIT_BENCH where PIT_ID=pit_id |

| | | |
|---|---|---|
| Insert polygon | int bench_id, string points, string creator, string zone | Insert into GEOLOGIC_POLYGON (PIT_BENCH_ID, POLYGON, CREATION_DATE, CREATOR_INITIALS, GEOLOGIC_ZONE) values(bench_id, points, CURRENT_TIMESTAMP, creator, zone) |

*Table 1: SQL Server statements*

**On-board SQLite database**

The on-board SQLite database will be used to store point and polygon data for the particular area that is being studied. Table 2 shows a list of actions that the SQLite database is responsible for, what data is needed in order to make that action, and the SQL call(s) that will result.

SQL statements

| Action | Data | SQL |
|---|---|---|
| Add point | double x, y | Insert into Points (X, Y) values(x, y) |
| Load point | int point ID | Select (X, Y) from Points where _ID=ID |
| Add single Line | point ID start, end | Insert into Lines (Start_ID, End_ID) values(start, end) |
| Add a polygon | Set of line IDs from 1 to n, Polygon ID, String type and notes | Insert into Polygons (Types, Notes) values(type,notes)<br>Insert into Lines (X, Y, Poly_ID) values($x_1$, $y_1$, poly_id)<br>Insert into Lines (X, Y, Poly_ID) values($x_2$, $y_2$, poly_id)<br>…...<br>Insert into Lines (X, Y, Poly_ID) values($x_n$, $y_n$, poly_id) |
| Load polygon | int Polygon ID | Select (Start_ID, End_ID) from Lines where poly_id=ID |
| Set Origin | double longitude, latitude | Insert into Origin (Latitude, Longitude) values(latitude, longitude) |
| Read Origin | none | Select (Latitude, Longitude) from Origin |

Table 2: SQLite statements

**TabletMapper Application**

This is the application that will run on the Android device. It has three key functions: the user interface, the interaction and storage of data on the database and the interaction with the on-board GPS module.

**User Interface**

The user interface of the application was implemented using OpenGL ES, in addition to Android's layout XML, due to instruction from our client. The main screen has a topological map as its background, as well as several buttons, including buttons to pan the screen, zoom in and out, and add or remove points, lines and polygons. For the OpenGL ES portions, we implemented a subclass of GLSurfaceView, called MapGLSurfaceView, which handles most of the user interface on screen. The MapGLSurfaceView is associated with a renderer that is responsible for creating each point and rendering it on the MapGLSurfaceView object. This occurs whenever the user specifies to add a point, by pressing the add point button. This renderer is also responsible for adding lines and polygons, as well as removing points, lines and polygons, when the appropriate buttons are pressed by the user.

**GPS interaction**

The GPS module is accessed whenever the map screen is opened by the MappingActivity class, which is responsible for creating the map. The MappingActivity class requests updates from the GPS module and creates a listener that handles any future changes in location. When the user presses the add point button, the click listener in the MappingActivity class requests the current location from the location listener, creates a point, and then adds it to the database. The MapGLSurfaceViewer renderer then retrieves a new list of total points from the database and refreshes the screen with these points the next time the screen repaints.

**Data interactions**

All external interactions with the on board database are handled inside the TabletMapper application by either the SendingSocketServer or ReceivingSocketSecketServer classes. These classes allow the external Java application to retrieve data from the tablet as well as upload topographical maps and existing polygon data.

All internal interactions with the on-board database, such as storing data while it is being collected, will be handled by the GeoDatabase. The GeoDataManager class will contain an instance of a GeoDatabase object. Whenever a point, line or polygon is created or deleted, the information is sent to the GeoDatabase from the GeoDataManager, using this GeoDatabase object. Once that information reaches the GeoDatabase, it will be used in the manner described above in the On-board SQLite module section.

# Implementation and Results

## Implementation Details

1. For this project, Newmont provided us source code for accessing their custom data types in the form of C++ libraries and a wrapper written in C#, which was the only way for us to access the binary files that needed to be transferred to the tablet. However, in order to access this code using Java, we were required to utilize the Java Native

Interface, which made calling the functions in the wrapper impractical. Therefore, we were forced to utilize Newmont's C++ libraries directly and ignore the wrapper functions. The code for this interaction was implemented in the external Java application, as we determined that learning the native interface in Android would be an inefficient use of our time at that stage of the project.

2. Due to driver issues and limitations of the tablet hardware we were provided for testing, the communication between the tablet and the external Java application is achieved using a network connection, instead of through universal serial bus (USB). This is accomplished by opening a ServerSocket on the Android device using a specific port, and then connecting to that same port using the external Java application. In this situation, the Android device acts as the server, and the Java application acts as the client, due to limitations of the Android operating system.

3. When reading in data from Newmont's OreconDB, a problem arose when attempting to render the downloaded data in the correct position on the screen. This is due to the fact that the data is not stored in an absolute coordinate system, but rather in a relative system that depends on the map that the data is being drawn upon. In order to solve this, the map for the data that is being used is required, in order to calculate the correct offsets necessary to center and display the data. However, this requires that the data must be sent with a topological map, or else the data will not display properly.

4. The data that in Newmont's OreconDB is stored in a specific data type known as a Geometry object, specific to Microsoft SQL Server. However, it is possible to query the database to return the data in file format, similar to that of XML, inside of a Geometry Markup Language (GML) file. This file can then be parsed by Java in the same manner as an XML file, which allows us to read the data that is stored in the file. The data itself is nothing more than a series or multiple series of two dimensional x-y points and a description of how they are connected. Each series of points describes a single polygon, and the description of how they are connected is almost always "Linear Ring," which means to connect the points with a line.

5. In order to store and render the topographical map and the OreconDB data, while keeping it separate from collected data, we decided to store the data in a file and allow the render to read the data from these files. This not only keeps our data in the on board database separated from the already existing data, but it also allows us to not use SQL queries when receiving data, which results in a much faster transfer of data to the Android device.

## Results

In this project, we were successful in meeting all of the functional and non-functional requirements that our client gave to us. However, we were not able to meet any of the stretch goals for the project, which are now included in future work for this project.

**Functional Requirements**

Utilizing native support in Android for accessing the GPS module, we were able to query the module to extract the user's current GPS location, and we used OpenGL ES to display a point on the screen, in relation to an origin, that represents the user's GPS coordinate. After multiple points were collected, we were able to use OpenGL ES and Android to recognize touch events on the screen, and select the points that the user was pressing nearest, in order to construct a polygon. This polygon could then be sent to a computer and received with our external Java application, which would then upload this data to OreconDB using SQL statements. The external Java application was also able to read a DIG binary file using the JNI and Newmont's DLLs to extract the topological map and then send this data to the Android application

**Non-Functional Requirements**

As the program was successfully deployed onto Android devices, we were able to use the Android SDK successfully, as well as Java and XML in order to program for Android. We were also able to successfully use a SQLite database to store our data and utilize the JNI to read Newmont's existing libraries.

**Integration Test**

The integration test was performed to ensure that all the different modules operate correctly together and the results indicate that our product is functioning successfully. The steps to our test are as follows:

1. Host a local SQL Server database designed to mimic Newmont's OreconDB.
2. Install the Android program and Java application onto a test tablet and test computer.
3. Connect the tablet to the computer and download data from the SQL Server database onto the tablet.
4. Edit the transferred data and collect new data using the tablet.
5. Sync the tablet with the SQL Server database using the Java application.
6. Verify both the tablet and database data.

# Conclusion and Future Direction

## Future Work

Due to time constraints and unforeseen issues with using certain tools and languages, we were not able to meet many of the stretch goals supplied by our client. These goals, and several ideas discovered while writing the project, constitute future work that can be done to continue the project, and are as follows:

- Add support for working with multiple sites.
- Add support for dragging shapes to correct GPS positions.
- Add support for reading topographical maps from a SD card.
- Add functionality to automatically close polygons while drawing.
- Add functionality to upload certain shapes to different benches in OreconDB.

- Add support for recording elevations at each point.
- Add functionality for continuously collecting points while walking
- Devise a method for storing line segments in OreconDB.
- Add support for the creation polygons pre-existing lines.
- Allow for modification of existing data in OreconDB.
- Add functionality for displaying the user's position relative to the rest of the map.

## Lessons Learned

- Time Estimates- Due to our lack of knowledge about several tools that we needed to use, we spent a large amount of time installing, configuring and learning these tools, and languages, while our time using them for a useful purpose was lessened

- Microsoft Visual Studio 2010 Ultimate- In order to view and write C++ and C Sharp code, we needed to use Visual Studio. However, many of the settings we had to enable were not located in the actual code, but in libraries and build paths, which required us to seek the help of developers at Newmont, and consumed a very large amount of time. We spent more time configuring Visual Studio than writing code with Visual Studio.

- Microsoft SQL Server and Microsoft SQL Server R2- In order to view the database data, as well as to view any changes we made once the database was setup, we were required to install Microsoft SQL Server. However, the database file we were provided was created using SQL Server R2, which is incompatible with SQL Server. Unfortunately, the installation of SQL Server takes more than an hour, and uses so much of the system's resources that code cannot be written at the same time. As we were required to be at Newmont's offices in order to receive assistance with the database setup, we lost several hours installing SQL Server and SQL Server R2, and importing the 10-gigabyte database file into SQL Server R2.

- OpenGL ES- Due to a recommendation from our client, we used OpenGL rendering in order to draw all of the graphics displayed on our mapping activity. However, we had no prior knowledge about OpenGL, so we spent hours researching OpenGL, then even more hours implemented before we were able to achieve a workable result.

- Android Emulator- While developing for this project, we made much use of the Android emulator provided with the Android SDK in order to test our application. However, it soon became apparent that the emulator was very inefficient and not a true representation of an Android device. It became much more efficient to create an Android application and deploy it to an Android device than to use the emulator.

# Glossary

| | |
|---|---|
| **Android** | A software package for mobile applications that includes an operating system, and is commonly used in smart phones and tablet computers. |
| **DLL** | Dynamically linked library - a C/C++ file that is used for linking code at run-time. |
| **DIG Files** | Binary flat files used by Newmont to store topographical maps and other geologic data. |
| **GPS** | Global Positioning System - a series of 24 satellites that can be used to calculate your location anywhere on earth. |
| **GML** | Geography Markup Language- a variation of XML that is used to format geographic information. |
| **JNI** | Java native interface - Java's support for calling code that is written in another programming language. |
| **OreconDB** | A SQL Server R2 database used to store geological information. |
| **Relational Database** | A set of software that stores information and relations between that information. The links are stored inside of tables and the relations are represented by the rows of those tables. |
| **SDK** | Software Development Kit - a set of tools that is used when creating applications. |
| **SQL** | Structured Query Language - a standard language used to manipulate relational databases. |
| **SQLite** | A less functional form of SQL that is used to store data locally. |
| **SQL Server** | Microsoft's version of a SQL database. |
| **Tablet** | A complete mobile computer integrated into hardware that usually incorporates a touch screen, also known as a tablet computer. |
| **USB** | Universal Serial Bus - a common interface for connecting peripheral devices to a host computer. |

| | |
|---|---|
| **XML** | Extensible Markup Language - a document format that is used to specify layout formats in Android. |