Final Report – Drill Hole Viewer

Newmont #1
C. Jhones, N. Lantz, R. Merillat

June 20, 2017

Contents

1 Introduction

Newmont Mining Corporation is a Colorado-based gold mining company. They have mines all over the world, from Colorado and Nevada to Peru and Australia. Newmont also prides itself on a very strong exploration department, something that has recently become fairly unique to this company. This department is responsible for determining possible new locations for gold mines.

A number of steps are taken in order to decide the viability of a new site. One of these steps is to take core samples at a potential site. These core samples drill hundreds of feet into the ground and show geologists what features are below the ground. The geologists then log all of this data with the goal of deciding whether a site is a good candidate for a new mine location.

Newmont currently has thousands of these drill hole data files from their claims all over the world. These files contain information about what types of geological features and minerals there are at that particular site, how deep from the surface those features are, and a variety of other pertinent values. Newmont's geologists currently access all of this data from a desktop client which must be installed on a user's Windows machine before they are able to view or change any data.

Our goal is to write a web client that will have very similar functionality to Newmont's current program but will be easier to access from the varied locations in which Newmont operates. Having a web client will also make sharing these data within the company much easier, as a coworker no longer needs to have the thick client installed in order to view the drill hole log data.

2 Requirements

This project consists of three stages, only the first of which is required. Once stage one is finished, we will move on to stage two, and so on. In order to produce the product that has been outlined above, we adhere to various functional and non-functional requirements.

2.1 Functional Requirements

The visual reference shown in Figure 1 accurately depicts several of the functional requirements of the software that we are creating. The UI will contain various utility options along the top such as open and zoom buttons. The open button will queue a server/database for a JSON file to load into the data columns. The column headers will display the core data to the user in a clear and readable manner. A ruler is necessary to line up with and mark the depth of the data on the core sample. Various colors, fonts, and imagery will be displayed in the data fields to represent different data. If stage one is completed, the new goal in stage two is to allow for the user to add comments to the data in a comments column. Additionally, any custom features we would like to be implemented would also be considered in stage two. In the event that this is completed, stage three will allow a user to append additional data to the column data.

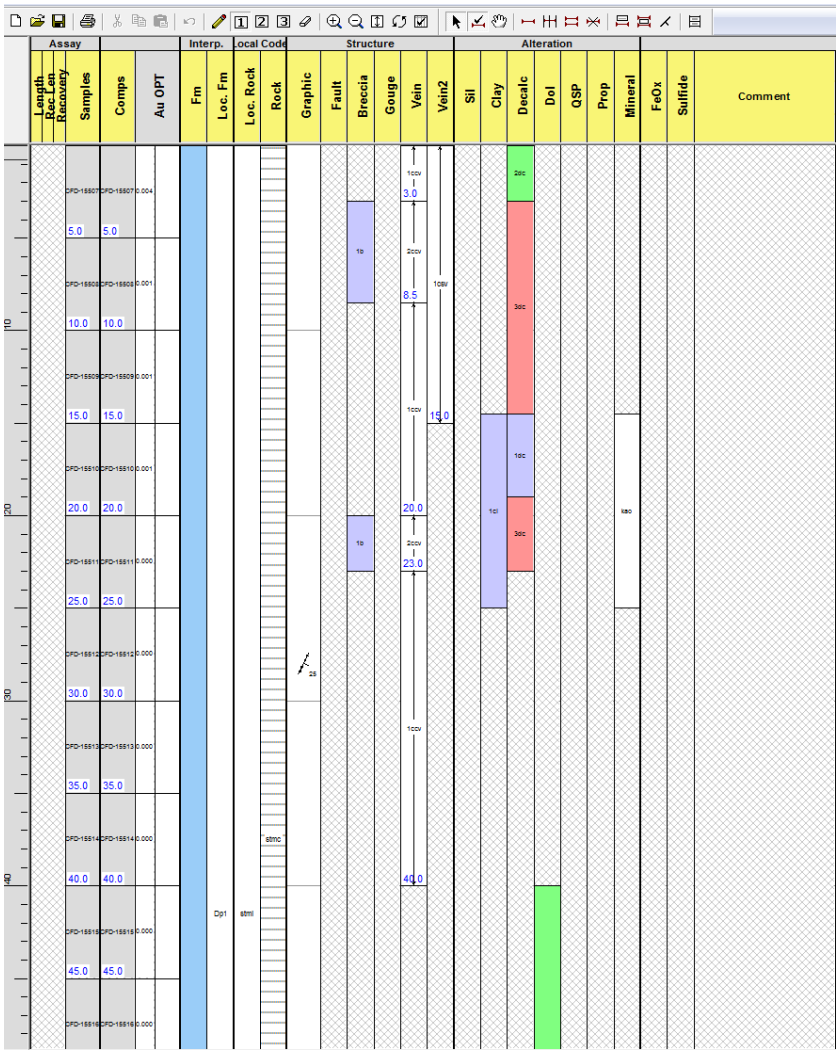

Figure 1: Screen Capture of Newmont's current drill hole viewer

2.2 Non-Functional Requirements

We will be using the Visual Studio Code IDE for developing our HTML, TypeScript, and JavaScript. Our code will be synchronized between team members and our contractor via GitHub. This software must operate without issue on Google Chrome. Unit testing will be performed using a combination of Mocha and Chai run through a Node.js build scripting language called Gulp.

3 System Architecture

The architecture diagram in Figure 2 depicts the flow of data throughout our software and server. Newmont has given us two JSON files. One of these contains the format, colors, and id's for each data column, and ensure only valid data is allowed into those columns. The second file contains a record of data for our website to display. Within an array resides data representative of a single column of data. We have a TypeScript parser read the data from these JSON files, and then populate a number of objects that were created by the initial JSON template file.

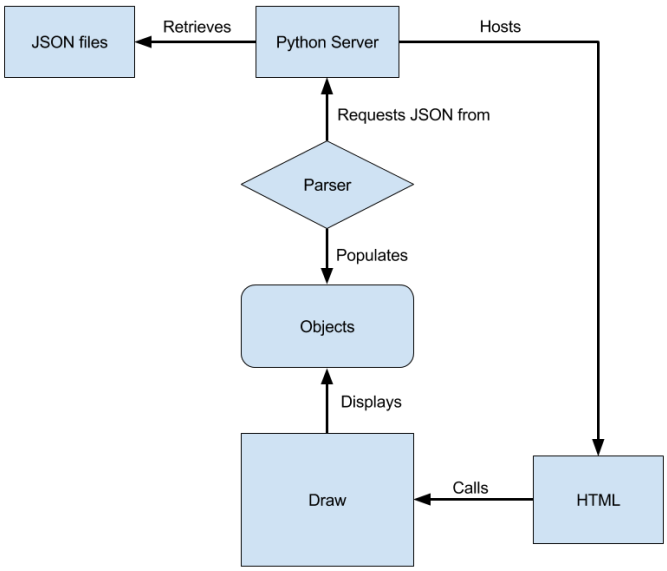

Figure 2: Architecture Diagram of DrillVis

Separate from the data ingestion, we have an HTML file which acts as a sort of controller for our website. It is responsible for drawing all of the "pretty" aspects of our website. It also calls our compiled TypeScript, which will pull the data values from our various objects, discussed above, and then display values in their appropriate places within the visualization (see Figure 1). This TypeScript code is also responsible for formatting each column with colors, patterns, and labels according to the JSON data.

4 Technical Design

4.1 Python Server and Loading Data

As with most web applications, our code requires a server of some sort to be running in order to work properly. We chose to use a Python `http.server` for testing, as it is easy to use and comes with every installation of Python. We also needed a way to enable Cross-Origin Resource Sharing (CORS) on our server in order to properly retrieve and parse our JSON data. Using Python made this very easy, as we were able to just add the `'Access-Control-Allow-Origin'` header to our server script.

We also extended our Python server to index the JSON files that are stored on the server. Using the `os` module, our script puts all of the JSON file names into an array. Then it uses the `json` module to put the filenames into a new JSON file designed to index the possible data files on the server.

4.2 Drawing the Data

The key feature of our application is the viewing of drilling data. There were several approaches to this problem such as HTML tables, divs, or using a canvas. We settled on the canvas approach because it seemed to be the easiest way when we began the project.

The HTML Canvas is simply a rectangle of pixels that can be placed in an HTML document. Web browsers provide a series of primitive operations to manipulate the canvas' pixels. Some of these operations include setting a foreground and background color, drawing a rectangle, tiling an image on top of a shape, etc. Further, if a pixel has been written to in the past and it is written again, it takes on the new value (which is to be expected). This turns the problem into drawing a series of layers of shapes to the screen. A general drawing algorithm is given in Figure 3.

All that's left is to determine what kind of data to draw on screen. The application is provided two data files with JSON-formatted data. One of these files contains *template* data and the other contains *depth* data. The template data defines the appearance of the data column while the depth data file defines the findings from the rock core samples. For instance, the template data says that "1b" breccia should use an Arial font, should be a light purple, and should not have any pattern behind it. However, the data file will say that there's breccia from 3 to 8.5 feet as seen in figure 1. This distinction was made so that the user can define their preferences for how a particular feature should appear. The depth data will not vary with users, but the template data can.[1]

1. Draw the ruler
2. Draw the header
3. For each column of data...
   (a) For each range[2] of information
      i. Draw a rectangle with given size filled with the given pattern
      ii. Draw the code in the middle of the rectangle
      iii. Draw arrows if the template file says to

Figure 3: General algorithm to draw the data columns to the HTML Canvas

The approach to drawing the data was straightforward. There are functions to draw text to the screen. The arrows can be implemented by drawing lines to the screen. The only tricky part was the display of the patterns as shown in Figures 1 and 5. Newmont supplied a font that contains dozens of patterns that could be displayed by the software. For a few examples, see Figure 4. The conversion of this data from a font to a tiling pattern on the screen was difficult.

Figure 4: DejaVu Sans font (left) vs. Newmont Patterns font (right)

One feature of shapes in the HTML Canvas library is that they can be filled with patterns. Typically these patterns are filled with an image stored on the web server. However, another Canvas can be used as the source for the pattern. This feature allowed us to create a new Canvas object on the page and fill it with a single character from the font file. We then use this canvas to tile across the particular range being drawn. The canvas created for this is never attached to the HTML page itself, so it is never visible to the user of the software.

5 Design Decisions

5.1 Drawing Strategy – HTML Canvas

One of the first decisions made was on how to draw the data columns of the project. We settled on using HTML Canvas because it seemed the easiest from a beginner web design perspective. It's just manipulating pixels which is done in a lot of languages. So, this is a more familiar approach for experienced programmers who are new to web development. This is the principle reason we chose this approach.

[1] We did not implement any method to modify preferences within the software itself, but the data architecture allows this upgrade to be made in the future.

[2] A range is a box with arrows, a code, and a color. For an example see Figure 5.

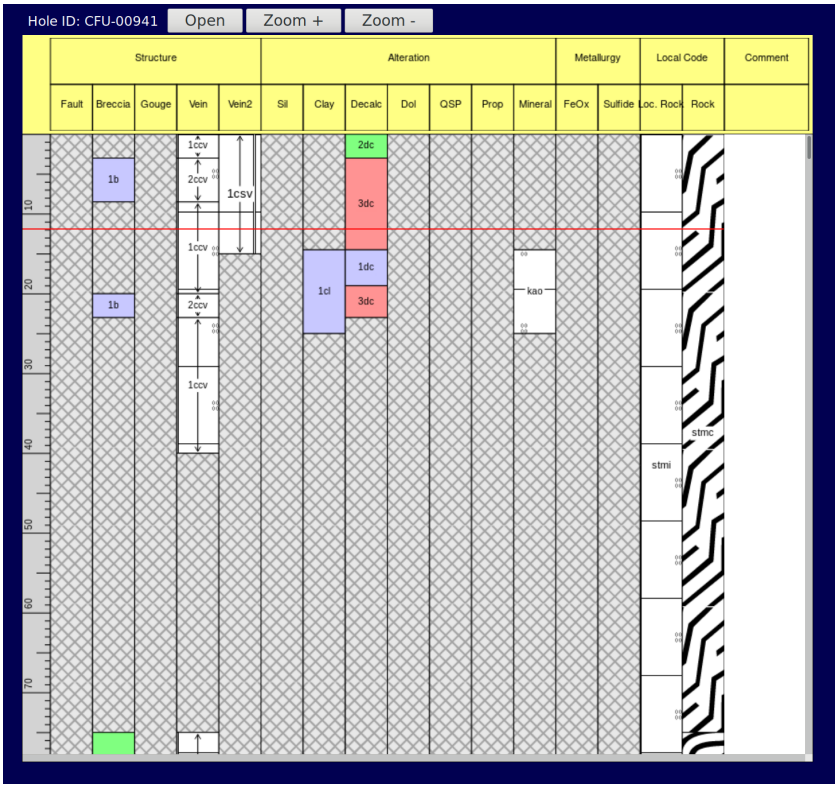

Figure 5: Screenshot of the new web-based Drill Hole Viewer

5.2 Unit Testing – Mocha and Chai

For the purpose of unit testing, our client recommended using Mocha and Chai as libraries for testing. Mocha is a test runner that runs JavaScript on Node.js. Mocha is quite minimal with its assertion library, so Chai provides more assertions to provide more expressive assertions. Notably, this was a poor decision for this project because Mocha is not very compatible with TypeScript. See the Results section for more information.

5.3 Documentation – TypeDoc

TypeDoc is a documentation system for TypeScript that generates a website with documentation on a TypeScript project. It uses the JavaDoc documentation system which is an industry standard. This makes it very easy to use and a good choice for this kind of project.

5.4 Language – TypeScript

JavaScript is a very poor language for developing large software projects. Thus, we decided to instead write the project in TypeScript which compiles to JavaScript. TypeScript has two major advantages over JavaScript: (1) it supports strong typing and (2) it is object-oriented. The strong typing allows the compiler to check types at compile-time. This helps alleviate the nasty runtime bugs JavaScript is prone to producing. Second, object-oriented systems are much more organized and thus more scalable than functional/scripting systems like JavaScript. Thus, it should be easier to expand the application in the future if there are ever more requirements to meet.

5.5 Libraries

jQuery

jQuery is a library that simplifies performing common tasks in JavaScript. We used it for loading and JSON formatted data from files and into the application. In particular, we used the `$.getJSON()` function.

Python http.server

A web server is required to host files on the server computer for a web application to use. We didn't need a substantial server, so we used a slightly modified version of Python's `HTTPServer`. As discussed earlier, we modified it to support CORS and to find JSON data files for our web application to load.

6 Results

6.1 Testing

Compatibility

Our application has been tested on Google Chrome, Mozilla Firefox, and Safari. We did not test on Internet Explorer or Opera, or any lesser-known browsers. One important restriction the application has is that the browser it runs on must support HTML5.

Unit Testing

We have successfully set up a framework to support unit tests, so it will be simple to add unit tests as future developers see fit. There are currently no true unit tests in our code, as we ran into significant trouble getting unit testing set up using TypeScript, Mocha, and Chai. The primary reason for these problems is the fact that Mocha and Chai are designed to be used with pure JavaScript, and we were primarily using TypeScript instead. In order to reconcile the two languages, we used Gulp. Gulp runs a sequence of JavaScript functions called gulp tasks in order to build and test our software.

Performance Testing

Our site runs quite quickly. The most time-intensive function is the `drawDataColumns` function which draws all of the drill hole data onto the web page. We were able to use the `console.time()` and `console.timeEnd()` functions to time the execution of this function. It finishes execution after about 50 ms on average.

6.2 Unimplemented Features

We have come up with a number of features which were either too difficult or time consuming to implement during field session. However, all of these features would improve the quality and usability of our product.

Clickable data fields

In the original desktop version of our software, each data field is clickable. When clicked, metadata about that specific field is shown in a pop-up window. Our goal was to implement this functionality in the website we created; however, we made a poor design decision early in our design cycle that caused this to be a very difficult goal. The data fields are simply drawn onto an HTML canvas, meaning it is difficult to set up mouse event listeners in each individual data field. If we had instead implemented each of these fields as separate `div`s, making mouse event listeners would have been trivial.

Editing Data Fields

In the final iteration of the website, each data field should be able to be edited. We spoke to Newmont about implementing this in our current code, and they informed us that the JSON format they use does not support replacing current data with new data; the only operation allowed on the JSON by their servers is appending data. In light of this, we purposefully avoided implementing editable data fields so that we would not cause problems with this requirement.

Comment Column

Looking at our website, an odd omission is the lack of information in the column titled "Comments". The main reason for this is that none of our sample data files have any comments in them, so we did not prioritize this aspect of the website. This omission is very simple to fix if one knows the format the data would take on.

6.3 Lessons Learned

The entirety of the CS field session is a learning experience. For many students, this is their first experience being thrust into a workplace and begin working on a real project that isn't just for homework. The project that we tackled over the last several weeks required us to work with new languages with and systems that we were likewise unfamiliar with. The following is a list of experiences that we learned from

TypeScript

The primary language that we used throughout this project was TypeScript. Beyond the fact that Newmont suggested we use this for our project, it seemed like a good way to write cleaner code than JavaScript would allow for. However, TypeScript is a relatively new language, and as such, there is a severe lack of documentation and relevent resources. What documentation we could find was often outdated, which meant we had to either figure things out for ourselves or ask our client to explain the finer points for us. While this was not an issue in the long run, it did slow us down considerably.

Unit Tests

As mentioned previously, we used the Mocha library and the Chai library for unit testing. However, these are both designed to be used in JavaScript projects, and ours was a TypeScript project. In order to reconcile the two languages, we used a number of libraries in conjunction with the Gulp scripting language.

Asynchronous jQuery

In addition to our TypeScript, we often made use of raw JavaScript in the form of jQuery functions. These functions allowed us to read and parse our JSON data. However, as novice web developers, we were originally unaware that jQuery reads and parses JSON asynchronously. This led to a number of odd bugs that took quite some time to reconcile.