Progress Report

Antarctic Chasm One
16 February 2017

Client

Matt Polaine

British Antarctic Survey

Team Bravo

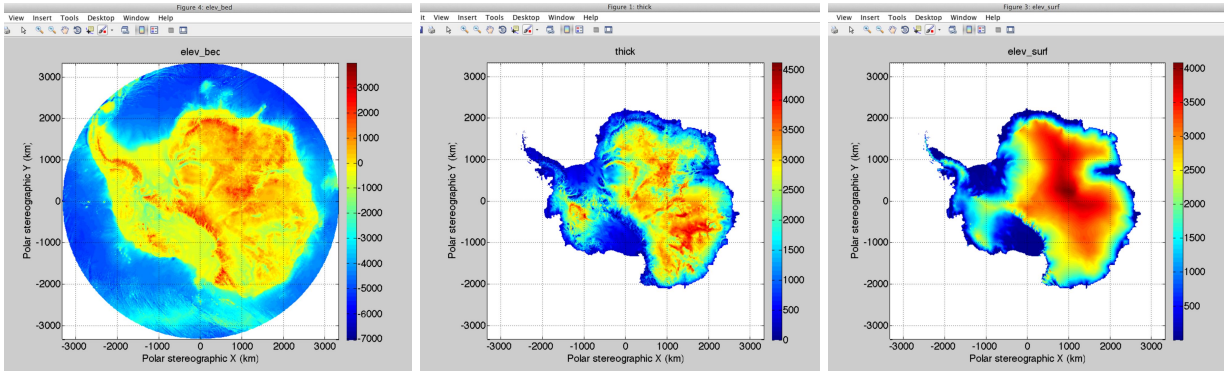
Julia Bibik, Andreea Deac, Matthew Else,

Nand Kishore, James Lomax, Valeria Staneva

So far, the project has several stages: First, we built a Preprocessor to use on the data we are given (in this case, BedMap2 data) to extract the features that are useful to the project. Then, we feed the preprocessed data into a Data Store. Then, we have an Interface between the Data Store and the Renderer, so that the Renderer can easily and promptly query the Data Store and get the information to be visualized.

1 Preprocessor: Scraping Bedmap2 Data

Although so far we have not seen the LIDAR and GPR data that the BAS has, we have been working with data from the BedMap2 project. The BedMap2 project [1] aimed to create a map of Antarctica that was generated using a lot of the available information about the continent until then. One of the formats used to represent the data about ice thickness, ice bed and surface elevation is a text file containing as an ESRI grid with resolution of 1000m, with coordinates in Antarctic Polar Stereographic projection (WGS84).



A major challenge in using the BedMap2 ESRI grid data about ice bed, thickness and elevation is that the ESRI grid contains information about the whole continent, whereas we need to isolate the area of the approximately 100 km x 100 km around the Halley Research Station. We solved that by concentrating on our area of interest that includes the Halley Research Station and Chasms One and Two.

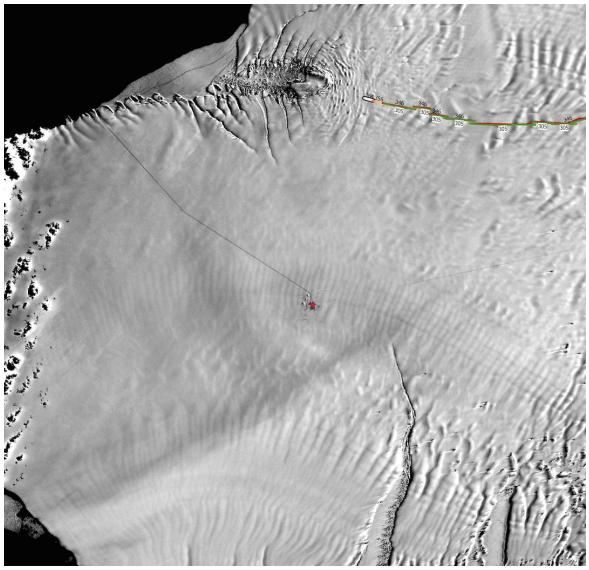
The Preprocessor is a Python script, which reads three .txt files, ESRI grids for the Surface (Elevation) of ice, ice thickness, and the ice bed (hard rock beneath the ice), as well as a configuration file. The configuration file describes what area of the grid we need to “cut” from the grid: where the station and the cracks are. The idea is that we can manipulate the Preprocessor depending on which area we want to concentrate on and achieve flexibility.

The Preprocessor can be easily broadened to allow for heterogenous inputs and different types of processing as needed.

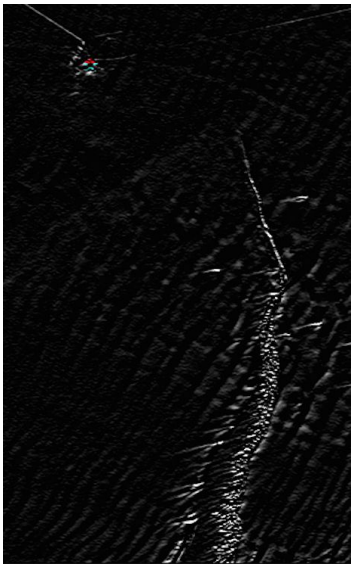
However, a big current challenge is resolution: 1000m as the unit of the ESRI grid means that the tiles we can create later on are of area 1 km^2. We need better precision if we want a

realistic and precise visualization: currently, naïve extrapolation might give very inaccurate results in areas of Chasm One.

The BAS have data that can solve both problems: a grid with 100m resolution (100 times more precise), as well as up to 40 cm resolution inside the crack, which should allow us to build a more precise visualization in the key area around the Halley Station.



Satellite image of Halley VI Research Station, Halloween Crack and Chasm 1, courtesy of BAS



Prewitt filters for horizontal and vertical edge detection. Challenge: the crack is not straight.

2 Populating the Data Store

The output from the Preprocessor was stored in a PostgreSQL with the PostGIS extension database. These were chosen due to the fact that they can support geographic objects and allow location queries to be run in SQL.

In this database (bravodb), a table (points) was created that has the following columns: col, row, surface, thickness, bed and id of type integer and geom of type geometry. In the last one, instances of type POINT are stored.

The fact that the output file from the preprocessor had the format .csv allowed copying the data from the file to the columns col, row, surface, thickness and bed with the command:

- `copy points FROM '/output.csv' DELIMITER ',' CSV HEADER;`

The id was set as the primary key in order to uniquely identify each record.

The objects of type POINT were created from the (col, row) coordinates:

- `SELECT AddGeometryColumn ('points', 'geom', 4326, 'POINT', 2);`
- `UPDATE points SET geom= ST_SetSRID(ST_MakePoint(col, row), 4326);`

A spatial index has also been added so that the renderer can make spatial queries efficiently:

- `CREATE INDEX points_gix ON points USING GIST (geom);`

The database was exported into the file dbexport.pgsql and can be imported for portability.

Tests were added to check that data has been loaded and that type checking passes.

3 Datastore Interface

The renderer needs to send spatio-temporal queries to the database, retrieve the results, and parse the results into a usable format. Since the render is in C++ and the database is Postgres/PostGIS, we need to create an interface so that the renderer can communicate with the database.

First, we define a struct in C++ to hold the data retrieved from a row in our database. This is referred to as Dbdata:

```
struct DbData {
    int id;
    int col;
    int row;
    int surface;
    int thickness;
    int bed;
    string geom;
};
```

Next, we create a Datastore class that handles all requests for connecting, sending, receiving, and querying data from the database.

The basic methods in the Datastore class for communicating with the database include:

- connect() for connecting to a certain database.
- run_query() for running an sql query against the database.
- processDbRow() for processing a row retrieved from the database.

In order to implement the connect() and run_query() methods, we use a library known as libpqxx. This is the official C++ client API for PostgreSQL[2] and is wrapper over a lower-level C library called libpq, which comes by default with PostgreSQL.

To use the Datastore, the connect() method is first called by the user to attempt connecting to the database. If the connection fails, an error message is thrown and handled to fail gracefully.

If the connection is successful, then sql queries can be run against the database by calling run_query(). Again, if a run_query() call fails then an error is thrown and handled.

Finally, the run_query() method returns a series of database rows. These rows can be processed by processDbRow(), which turns each row into a DbData object that can then be used by the rest of the program.

We also provide a higher-level api in the Datastore class that exposes useful methods for running commonly used requests by the render. These methods run pre-built queries and automatically process the results into a format easily useable by the renderer. Currently, these methods include:

- getAllPoints()
- getPointsInRectangle()

These methods internally run lower level (spatio-temporal) sql queries by calling run_query(). An example of the sql query for getAllPoints() is:

```
SELECT id, col, row, surface, thickness, bed, ST_AsText(geom) FROM
points
```

Most of the Datastore api has been implemented. Users can successfully connect to a database, run specific sql queries by calling run_query(), and use the higher level methods getAllPoints() and getPointsInRectangle() for easily retrieving the data.

The code has been thoroughly tested by connecting to a test database, and then calling the Datastore methods including run_query(), getAllPoints(), and getPointsInRectangle() on sample data in the database.

The only aspect of the Datastore remaining is to add more higher level methods for retrieving data that might be useful for the render. Possible examples of such methods include the data inside a polygon, data in a certain radius, data in location over certain time, etc. These methods will be developed and added in progression with the renderer, as they depend on the types of data and queries demanded by the renderer.

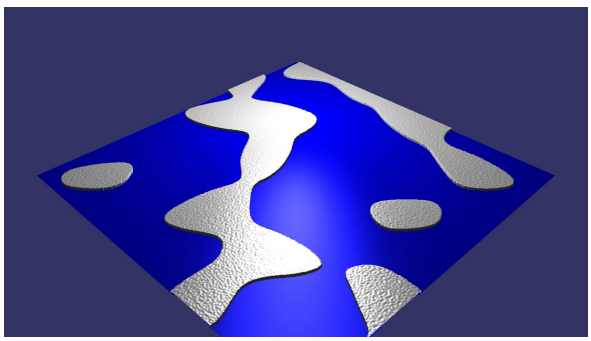
4 Renderer

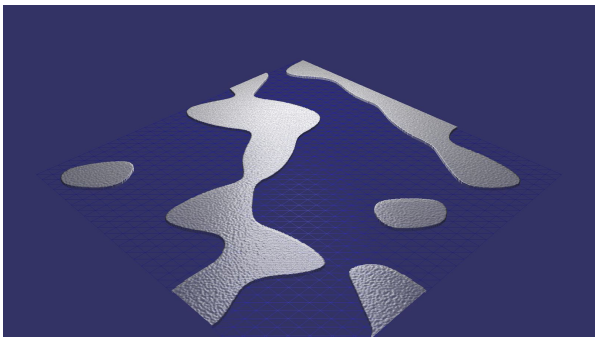
Implemented using C++ and OpenSceneGraph, the renderer component (currently quite basic) renders tiles of the terrain. Tiles of the ice surface are rendered by turning a heightmap into a vertex buffer. The tiles are a way of dividing up the terrain, allowing the renderer to load only the bits of terrain it needs. Each tile represents a leaf node in the scene graph, which allows the renderer to show only the correct LOD (level of detail) of this tile in the scene. There are several different types of tile, corresponding to what is drawn within the tile.

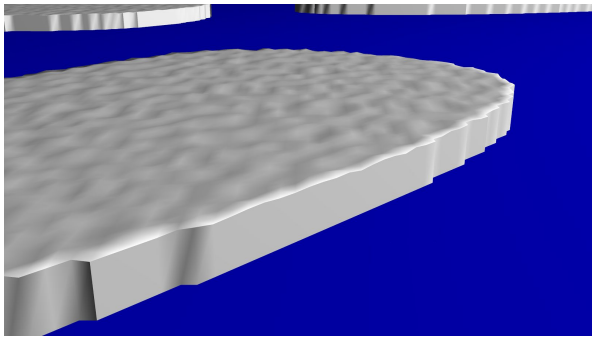
- Water tiles - are tiles which contain just water, and are drawn at the moment as a flat blue surface.
- Static ice tiles - height maps loaded from the data store are rendered as a surface
- Boundary ice tiles - At the boundary of sea and ice, an edge needs to be rendered. These tiles contain information about where that edge is.

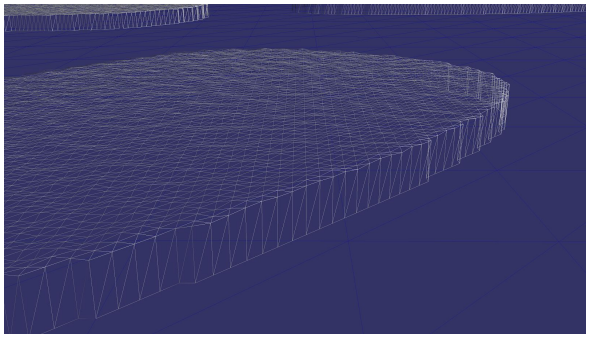
One problem we came across was how we plan to map the actual crack data. Crack data could be modelled as a height map of vertices, although, as it is visible in the below images, mapping edges as a height map gives them a rough appearance which is not desirable. This issue could be solved by increasing the number of vertices in the height map and then smoothing the edges, this method would be simplest, however, it may require a lot of data, and transitioning between time slices as the user controls the progression of the crack will be difficult. Another way of modelling the crack itself is for the data store to provide the scene data as a set of vertices, and the faces that make up the vertices. This method would provide nicer looking scenery in the renderer. It will, however, require significantly more preprocessing, as suitable vertices are difficult to find, and deciding which faces to use is a similarly difficult problem.

The images below are using sample data generating use the GLM GTC simplex noise function. We are initially using randomly generated data because of a lack of data provided by the BAS.



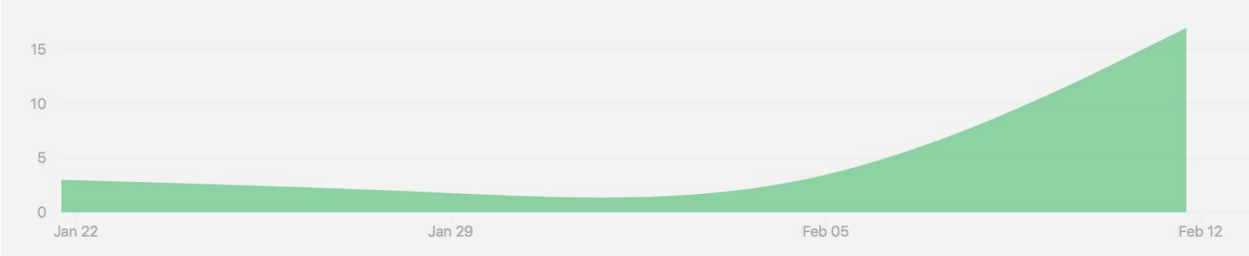






5 GitHub repository

Working collaboratively on code has been enabled through the use of the Git, and GitHub (repository at https://github.com/nandck/Bravo). This has enabled us to have a common repository of code that we can all refer back to, as well as giving the opportunity in the future to add features such as issues and pull requests to our workflow.



The GitHub graph of contributions for the Bravo repository

This enables us to each make local changes to the code base, and merge them back together with changes made by other team members at a later date. Furthermore, the GitHub user interface makes it easy to browse through code that has been written by other people to help you understand how it works.

References

1. Fretwell, P., et al. "Bedmap2: improved ice bed, surface and thickness datasets for Antarctica." The Cryosphere 7.1 (2013).

2. http://pqxx.org/development/libpqxx/