Generating an Aquarium from AUV Log Data

Kristof Kipp
University of Bremen
Bremen, Germany
kkipp@informatik.uni-bremen.de

Nils Leusmann
University of Bremen
Bremen, Germany
leusmann@informatik.uni-bremen.de

***Abstract*—To visualize data in a simulated environment, data has to be transformed. What this implies is that real log data contains data in metric format, while the game engine that contains the simulation needs computer-readable data. This transformation and the calculation of this data is described in this paper.**

***Index Terms*—unreal engine, unreal, trajectory, travis, visualisation**

I. Introduction

Travis is a masters project operated by *University of Bremen (UoB)* and *MARUM*, which tries to re-simulate a deep dive mission in the Unreal Engine. Re-simulating in this case means that log data is provided by an Autonomous Underwater Vehicle (AUV) that needs to be read by a computer and displayed in a simulated environment. The output layer of this simulation might be a computer display or any technology involving Virtual and/or Augmented Reality.

II. Initial conditions

For the better understanding of this topic, we need to introduce some new terms. The first term will be the *aquarium*. Travis will run in an simulated environment, specifically the Unreal gaming engine. Because we do not know how large the simulated terrain will be we need to specify a part of the map in which the simulation will take place. The Box in which the simulation takes place is called *aquarium*. Inside the aquarium every point is described with x, y and z values. In the real world our reference system would you the latitude, longitude and depth values to describe an specific point where the AUV could have been. To show these different reference systems we created two different Classes. The so called *WorldAnchorPoint* will store the values of the real world (latitude, longitude, depth) and the so called *GameAnchorPoint* will hold the values of Points in the Unreal engine (x, y, z).

A. WorldAnchorPoint

A WorldAnchorPoint represents a specific point on earth. Each point on the earth can be described as an WorldAnchorPoint and each Point can be distinguished form others. A WorldAnchorPoint possesses three member variables which stands for latitude, longitude and height.

B. GameAnchorPoint

A GameAnchorPoint represents a specific point in the Unreal Engine. It is also unique, if initialized with an concert value. In contrary to an WorldAnchorPoint a GameAnchorPoint has four different member variables and in thus is able to have a specific state. The four different member variables are the following:

- x
- y
- z
- index

The index defines the state of a GameAnchorPoint. If it is *-1* the GameAnchorPoint is invalid (this can be checked by using the Valid() method). An index of *0* means that this GameAnchorPoint is uninitialized meaning it is currently not used in the bathemytry. For more information of why we are using states and how we are using them see our landscape paper.

III. Logfiles

Logfiles are written by an AUV to log the progress of a mission, it contains a lot of data (read: 500k lines in 100 columns for a mission of 6 hours). The MARUM-SEAL[1] has heaps of sensors that provide data to this log file. The following list of items is taken from a list of sensors from the official MARUM SEAL web site. The list is not complete and only contains sensors that were used in the data layer (see other paper) of the simulation. The list:

- Teledyne RDI DVL 300 kHz
- Kongsberg, Altimeter, 675 kHz (nose)
- Paro-Scientific 8B7000-I
- SBE-49 / SEABIRD FastCat CTD
- IXBLUE PHINS (dry, serial, pressure housing)

These sensors are found within the log file (again, see our data paper for a full list of all columns in the logfile) as a name infix, e.g. the first item in the list is the DVL (doppler velocity log) and can be found with the infix 'dvl_' (e.g. vcc_dvl_altitude_fb_m for the feedback (fb) data in meter (m) of the altitude sensor that lies within the dvl). There may be two different values for each set of data of a sensor: *fb*, which correlates to the feedback value of the sensor, so this is the

[1] https://www.marum.de/Infrastruktur/MARUM-SEAL.html

actually measured value, and *sp*, which is a setpoint value that bears the value in a optimal situation. This scheme resides throughout the complete log file and describes several data. Part of the complete data consists of value from the PHINS sensor, which provides us with a feedback value of an internal GPS tracker in form of geographic coordinates (latitude and longitude).

IV. Aquarium

In order to calculate the width of the aquarium, we need to consider the bottom-left and the bottom-right point of the data (e.g., the lowest latitude and the highest latitude in the log files and an arbitrary but fixed longitude value) and calculate the metric distance between these points. This is done with the Haversine Formula.

$$\mathrm{hav}(\theta) = \sin^2\left(\frac{\theta}{2}\right) = \frac{1-\cos(\theta)}{2}$$

Given a static three-dimensional maximum aquarium size, the converter would theoretically stretch the given bathymetry into the desired format, e.g. a bathymetry with the dimensions 502x128:

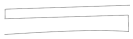

Fig. 1. Real size of a non-real bathymetry

would be stretched onto an aquarium with the maximum dimensions of 5000x5000:

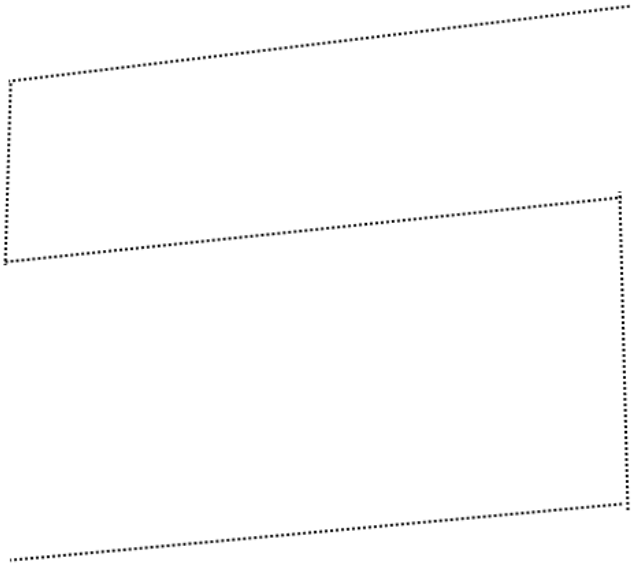

Fig. 2. distorted antisymmetrical size of a non-real bathymetry

This behaviour would distort the actual bathymetry and therefore make all data we'd show utterly useless. Thus the next problem we tackled was the aspect ratio of the data. If we'd just projected the given data on a fixed value cube the data would have been distorted as the log data would be stretched either on the x- or the y-axis. Therefore we needed to (a) calculate the aspect ratio of the log files, (b) apply this calculated ratio to the height of the real log data and (c) multiply each data point with the factor of the aspect ratio on each dimension. This implies that we needed to focus on type and unit safety throughout the whole calculation.

We make use of the following helper variables: lat_{max} is the highest value in the whole data set, lat_{min} is the lowest value, the same helper variables apply to the longitude and the depth (lng_{min}, lng_{max}) we can apply these values to the following formula for an existing logpoint pt:

$$pt_x = ((pt_{lat} - lat_{min})/(lat_{max} - lat_{min})) \times w \times x_{factor}$$
$$pt_y = ((pt_{lng} - lng_{min})/(lng_{max} - lng_{min})) \times h \times y_{factor}$$
$$pt_z = z - ((pt_{depth} + |depth_{min}|)/d_{max} \times z)/x_{meter}$$

Now a data point of the log file is transformed to a point in the game engine's aquarium and can be rendered on screen.

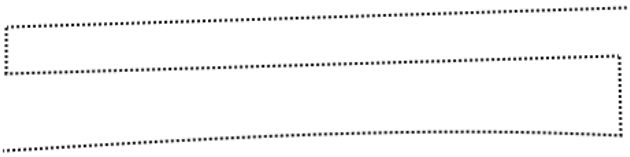

Fig. 3. correct symmetrical size of a non-real bathymetry

V. Bathymetry

After the aquarium is initialized it is possible to calculate the bathymetry, with the help of another file. Here it could be possible that the height map we want to create is bigger then the actual aquarium. For this case we created an new calculation function. Given an input vector into this function, its values will be compared with the min and max values of the currently initialized aquarium. If the value is not inside the defined boarders the function will return, an invalid game vector. Otherwise we can calculate the point as it is using the already existing formula.

VI. Source Code

The source code can be found in the project-relative path */aquarium/aquarium.*{*c,h*}. The most important methods are:

- void CalculateAquarium();
- GameAnchorPoint *calculatePoint(float lat, float lng, float depth);
- GameAnchorPoint *calculateBathymetry(double lat, double lng, double depth, double invalid_data_value);

We will now take a closer look at these methods and what impact they have on the calculation.

CalculateAquarium

```
if (this->WorldPoints.size() > 0)
   {
      for (WorldAnchorPoint pt : this->WorldPoints)
      {
         // made this to increase readability
    instead of performance, I'm aware that a bunch
    of if statements would be better
         this->maxLat = pt.lat > this->maxLat ? pt.
    lat : this->maxLat;
         this->minLat = pt.lat < this->minLat ? pt.
    lat : this->minLat;
         this->maxLng = pt.lng > this->maxLng ? pt.
    lng : this->maxLng;
         this->minLng = pt.lng < this->minLng ? pt.
    lng : this->minLng;
         this->maxDepth = pt.height > this->maxDepth
     ? pt.height : this->maxDepth;
         this->minDepth = pt.height < this->minDepth
     ? pt.height : this->minDepth;
      }
   }

   this->maxDepth = this->maxDepth + abs(this->
    minDepth);
   double a = (this->maxLat - this->minLat);
   double b = (this->maxLng - this->minLng);
   this->superFaktor = (1 / ((1 / (a > b ? (a / b) :
     1)) == 1 ? (1 / (a > b ? (a / b) : 1)) : (b / a
    )));

   this->xMeter = distance_in_meters(this->minLat,
    this->minLng, this->maxLat, this->minLng) / (
    this->maxX * this->superFaktor);
   this->yMeter = this->xMeter;
   this->zMeter = this->maxZ / (this->maxDepth - abs
    (this->minDepth));

   // update the game points
   for (WorldAnchorPoint pt : this->WorldPoints)
   {
      GameAnchorPoint *foo = this->calculatePoint(pt
    .lat, pt.lng, pt.height);
      this->GamePoints.push_back(foo);
   }
```

The CalculateAquarium method does exactly what the name of the method makes you believe it does. Some insights of the logic behind it: first the highest and lowest latitude and longitude are checked. There is sadly no better way than to iterate through all elements, thus making it run in $\mathcal{O}(n)$. After having projections to the left and rightmost top and bottom points, it is now possible to upsample the data while maintaining the right aspect ratio.

calculatePoint

```
   float myX = (((lat - this->minLat) / (this->
    maxLat - this->minLat)) * (this->maxX)) * this->
    superFaktor;
   float myY = (((lng - this->minLng) / (this->
    maxLng - this->minLng)) * (this->maxY)) * this->
    superFaktor;
   float myZ = this->maxZ - (abs(depth) * this->
    superFaktor);
   return new GameAnchorPoint{ myX, myY, myZ };
```

This function does exactly what it says: it converts a wordanchorpoint to a gameanchorpoint, thus projecting a point in the real bathymetric data to a point inside the simulation while keeping the aspect ratio.

calculateBathymetry

```
   if (lat > (this->maxLat * (1 + diff)) || lat < (
    this->minLat*(1 - diff)) || lng >(this->maxLng *
     (1 + diff)) || lng < (this->minLng * (1 - diff)
    ) || depth == invalid_data_value) {
      auto gap = new GameAnchorPoint(-1, -1, -1);
      gap->id = -1;
      return gap;
   }
   auto p = calculatePoint(lat, lng , depth );
   p->z = p->z * -1;
   return p;
```

While being rather similar to the calculatePoint method the calculateBathymetry method is used to *draw* the deep sea landscape, which is discussed in our landscape paper. Trough the usage of the *diff* helper variable it is possible to set an small offset and draw more parts of the landscape.

VII. Divergence between logdata and bathymetric data

In the later time of the project the consortium together with Ralf Bachmayer from MARUM pointed out that the trajectory does not *naturally* overlap with the bathymetry in our simulation. Since we ran out of implementation time we could only narrow down the problem and document our results.

The main problem at this point is the data and structure behind the bathymetric ascii file that is not well documented. The data found in the log files are rather easily interpretable, so the min and max values for each the latitude and the longitude are as easily found: iterate through all the entries, find the lowest and highest values for each field. this yields the following result (coming from *RStudio*) with:

```
log16_adjusted <- read.csv("C:/Users/stud6/Downloads
    /MV_Ridge/MV_Ridge/log16_adjusted.csv", dec=",",
     quote="'", stringsAsFactors=FALSE)
> min(log16_adjusted$vcc_phins_latitude_fb_deg)
[1] "38.30261153333334"
> max(log16_adjusted$vcc_phins_latitude_fb_deg)
[1] "38.33708008333334"
> max(log16_adjusted$vcc_phins_longitude_fb_deg)
[1] "17.78288008333333"
> min(log16_adjusted$vcc_phins_longitude_fb_deg)
[1] "17.71249268333334"
```

Giving the following results **for the log file**:

- Max Latitude: 38.33708008333334
- Min Latitude: 38.30261153333334
- Max Longitude: 17.78288008333333
- Min Longitude: 17.71249268333334

Now the bathymetry file (short: **bat**) has a whole different layout of the presented data (see document regarding the batfile). The header structure deliver the following values to work with:

```
ncols 10123
nrows 5097
xllcenter 17.6763952465
yllcenter 38.2872007428
cellsize 0.0000100000
nodata_value -99999.000000
```

Therefor, it's obvious that xllcenter is the arithmetic center of the longitude, as the dead center and the amount of values to each direction (*ncols* / 2) are known, the max and min values for both directions can be calulated as well,

$$lng_{min,max} = 17.6763952465 \pm (10123/2) * 0.0000100000$$
$$lat_{min,max} = 38.2872007428 \pm (5097/2) * 0.0000100000$$

Thus giving the values **for the bat-file**:

- Max Latitude: 38.3381707428
- Min Latitude: 38.2362307428
- Max Longitude: 17.7776252465
- Min Longitude: 17.5751652465

Due to the structure of these files the following natural constraints are set:

1) The maximum latitude of the bat **MUST BE** $\geq$ max lat o
2) The maximum longitude of the bat **MUST BE** $\geq$ max lng of the logfiles
3) The minimal latitude of the bat **MUST BE** $\leq$ min lat of the logfiles
4) The minimal longitude of the bat **MUST BE** $\leq$ min lng of the logfiles

Inserting the numbers found in the observations above, we get the following propositions:

1) $38.3381707428 \geq 38.33708008333334$
2) $17.7776252465 \geq 17.78288008333333$
3) $38.2362307428 \leq 38.30261153333334$
4) $17.5751652465 \leq 17.71249268333334$

Check if these are true (not using human logic but rather *Wolfram Alpha* due to the faultiness of human logic):

1) $38.3381707428 \geq 38.33708008333334 =$ **TRUE**
2) $17.7776252465 \geq 17.78288008333333 =$ **FALSE**
3) $38.2362307428 \leq 38.30261153333334 =$ **TRUE**
4) $17.5751652465 \leq 17.71249268333334 =$ **TRUE**

As seen above the data given to us does not fulfill our validity checks, therefore a manual reposition has to be added to the calculated position by the aquarium.