Terrain Mapping: Data Collection and ODETLAP

Jeff Frey

Michael Todd

Abstract

We face the problem of collecting data on real world terrain slices with a sparse collection system and generating complete virtual terrain slices with that data. Several papers in the past have faced the problem of terrain generation over sparse data sets[2], with varying success. Other papers in other fields have studied the issues of terrain sampling, searching for more and more accurate methods to measure terrain data[1].

In our paper we do both: creating a collection device, gathering real world data, and then performing terrain generation techniques (namely ODETLAP) to fill out our final terrains.

Data Collection

In our solution for terrain measuring we utilize a custom designed GPS device to sample altitude data at different global coordinates. This data is collected using an EM-406A SiRF III GPS receiver, accurate to 10 m, connected to an Arduino microcontroller unit. The entire unit can be powered by a 9 volt battery. This device produces terrain data in a longitude latitude altitude format approximately 10 times a second and outputs it via a serial connection.

This serial data is fed into a BlueSMiRF Bluetooth adapter where it is then sent via a Bluetooth serial connection to an Android wireless device.

On the Android wireless device the data is averaged, and then once every second uploaded via a data connection to a waiting Java Server running on Google App Engine. There the data is stored in a permanent database, where it can be retrieved later.

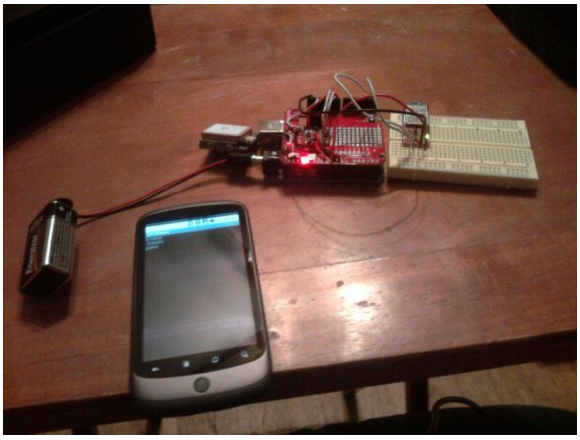

Figure 2: The collection device

Analysis

Once we have a collection of coordinates, our program takes over. The program takes up to four arguments: a text file containing coordinates, the desired grid resolution,

smoothness factor, and which axis is to be altitude.

Only the text file is really required. Grid size defaults to 10 unless otherwise specified, and smoothness factor defaults to 1. If the fourth argument does not exist, it assumes the z axis is height. These steps will be explained in greater detail below.

The program is split into several parts

1. Parse and store the coordinates
2. Translate and scale the cloud near the origin
3. Distribute the coordinates in a grid of boxes
4. Calculate the average altitude for boxes that contain coordinates
5. Loop through the grid to fill in the empty boxes
6. Scale the mesh
7. Create the obj file

Parse and store the coordinates

This simply runs through the text file, grabs each x, y, and z coordinate, and stores it in a vector for later use. While this is happening, it also determines what the max and min of x, y, and z are so that they can be used for the next step. Normally z will be considered altitude, though this can be altered via the command line.

Translate and scale the cloud near the origin

Once all of the coordinates are parsed the points are scaled so that all values lie between 0 and 1.

Distribute the coordinates in a grid of boxes

The grid is a 2D array of Boxes. The size is specified by command line. For instance, if the size is 10, then the grid will contain 10 x 10 boxes (100). Using a coordinate's x and y position they are sorted into one of these boxes with the following logic:

gx = (int) ((x / maxX) * gridSize);

gy = (int) ((y / maxY) * gridSize);

Calculate the average altitude for boxes that contain coordinates

Once the coordinates are distributed, it runs through each box and acquires the average altitude of the coordinates it contains. Boxes that contain coordinates are marked appropriately.

Loop through the grid to fill in the empty boxes

This is where ODETLAP (Overdetermined Laplacian Partial Differential Equation) comes into play. It allows us to take a grid where only a few points are known, and computes the surface over the entire grid. The equation itself just involves averaging the four altitudes of a point's neighbors. In addition to this equation, the paper talked about a Smoothness factor R, which trades off accuracy and smoothness. The higher the smoothing factor, the smoother the mesh becomes [2]. However, in our program, the smoothing factor is reversed (smaller = smoother). This is shown later.

We do this step inside a while loop. In each iteration, it goes through the grid box by box and, for each box that isn't marked, acquires the average altitude of its neighbors. It does not take into account neighbors that haven't been initialized yet. The first run-through fills in

the empty boxes. The following run-throughs help the grid of heights reach stability. It will continue to run through the while loop until the difference of its previous average and its new average is less than .000001.

By default, boxes that are marked are not altered during this process. However, if the user specifies a smoothness factor, than the following equation is applied.

neighborAverage = smoothness * grid[i][j].getOriginalZ() + (1 - smoothness) * neighborAverage;

getOriginalZ() is the box's original average z, acquired from its coordinates. The lower the smoothness factor, the more it will blend in with the rest of the grid.

Once the grid is stable, we continue.

Scale the mesh

This involves going through the grid one more time and finding the new minimum and maximum altitudes and scaling them between 0 and 1

Create the obj file

The last step involves two parts: vertices and faces.

Boxes need to be assigned a number from 1 to the resolution size so that it can be used by the faces later. This is done by making a counter that starts at 1, and then scanning through the grid, box by box. While assigning a box its number, you also output the vertex to the obj file, in the format of "v x y z", where x, z are replaced by a box's coordinates, and y is the box's average altitude (this was done so that it would display correctly in our renderer).

x = i / (float) gridSize;

z = j / (float) gridSize;

y = grid[i][j].getAverageZ();

Once all of the vertices are printed, it is time to do the faces. Two triangles are constructed for every box coordinate, except those in the last row and last column.

The first triangle is "f num num+1 num+gridSize" where num is a box's assigned number, and gridSize is the resolution.

The second triangle is "f num+1 num+gridSize+1 num+gridSize"

After the output file is closed, it then runs a system command that launches our renderer, using the obj file we just created. Inside of the renderer, it can be viewed, subdivided, and given basic Gouraud shading using existing algorithms [3,4].

Result

The run time of the program is dependent on the resolution of the grid, as well as the cut-off point for when the grid is stable (.000001). There is a noticeable spike in execution time when running with a resolution of 50 (2500 boxes), and a resolution of 100 (10000 boxes). Reducing the cut-off point fixes this problem, but results in a slightly less accurate mesh.

Figure 2: Varying smoothness factors on an image of the 'Approach' in Troy, NY From the top left, smoothness of 1, .5, .1, .01, 0

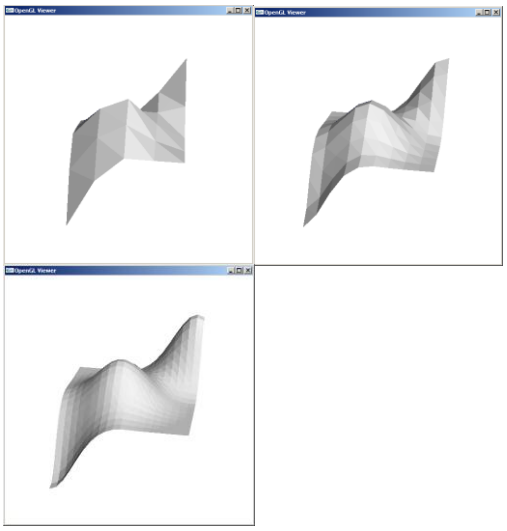

Figure 3: Three different resolutions of a contrived set of data, 5x5, 10x10 and 20x20

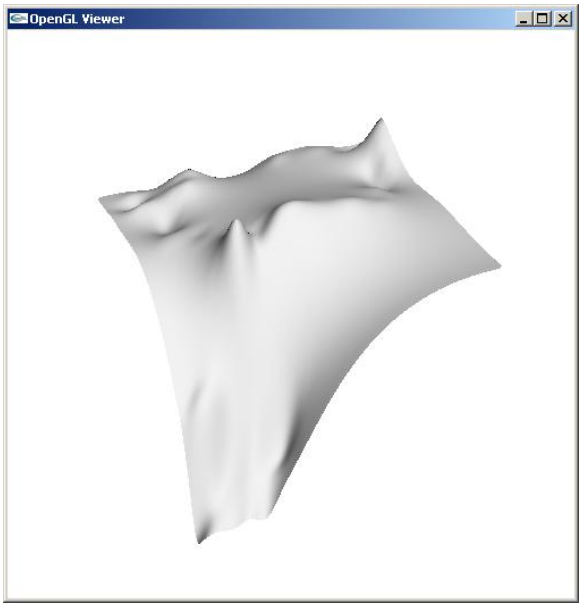

Figure 4: Example showing generated terrain of the greater RPI Campus in Troy, NY

Conclusion

Our techniques produced viable terrain samples of multiple real world terrain pieces and allowed us a great ability to understand the connections between original data and results. By examining both our data collection methods and our data processing methods we were able to gain insights on how to improve the overall technique. In our test cases accuracy of data is important, but quantity of data is even more so. GPS sampling techniques can yield errors that drift during different times of day, as the GPS unit communicates with different GPS satellites. These errors can be reduced by having more accurate GPS systems, but can even more accurately be accounted for by simply taking many samples at many different times of day. As the density of the point cloud grows, so does its accuracy. Focusing on quantity of data rather than quality also allows for a greater number of cheap GPS units to be used, greatly reducing the cost of individual modules.

Since data is averaged and smoothed while being redefined to specified resolutions in the post processing phase, the final results are also always of specified data size, regardless of the quantity of input data utilized.

Future Work

In the future our module would be extended upon to be able to utilize data from more precise GPS units, as well as switch to using a more robust data transfer medium, to allow for sampling rates greater than 1 a second over http, for instance UDP transmissions being beamed to a personally established server. Different collection methods could also be integrated into the same framework, allowing data transmitted from satellite or range finder to be uploaded and utilized in the same server framework.

Our program would be extended upon to be able to generate non-square output, as well as make it so the weight of the heights inside a box are dependent on the distance from the center of the box (the closer it is, the more it contributes).

References

[1]U.S. Department of the Interior. "Measuring and Mapping the Topography of the Florida Everglades for Ecosystem Restoration." *Http://egsc.usgs.gov*. Web. 22 Apr. 2010. <http://egsc.usgs.gov/isb/pubs/factsheets/fs02103.pdf>.

[2]Lau, Tsz-Yam, You Li, Zhongyi Xie, and Randolph W. Franklin. "Sea Floor Bathymetry Trackline Surface Fitting Without Visible Artifacts Using ODETLAP." Web. <http://www.ecse.rpi.edu/~wrf/p/129-acmgis2009-lau.pdf>.

[3]DeRose, Tony, Michael Kass, Tien Truong, and Pixar Animation Studios. "Subdivision Surfaces in Character Animation. " Web. <http://portal.acm.org/ft_gateway.cfm?id=280826&type=pdf&coll=GUIDE&dl=GUIDE&CFID=87864502&CFTOKEN=20721222>

[4]Hoppe, Hugues, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle. "*Piecewise Smooth Surface Reconstruction*. " Web. <http://research.microsoft.com/en-us/um/people/hoppe/psrecon.pdf>.