Terrain Generation Using Delaunay Triangulation and Image Analysis

Brian Truhlar

Abstract

When artists want to create a 3d mesh that is influenced by a height map, it is usually done through a manual process that is time consuming, inaccurate, and results in inefficient mesh density. This paper describes our method for solving these problems with an automated process with little user variables to set. Fewer options to manage, however, can lead to inexact results.

1 Introduction

One of the common ways to generate a terrain mesh that reflects the values of a height map is to use a greyscale image as a displacement map for the geometry. The main problem with this is the fact that in order to obtain a detailed result, one must use a rather dense mesh. Not only is this a potential waste of memory if there are areas of low detail in the displacement texture, but it also results in uniformly sized geometric detail.

This paper attempts to solve these problems by analyzing the image with subdivided quadrants for areas of high detail, applying Delaunay Triangulation to the resulting data, and then printing the results so a 3d application can import the mesh. Other groups have addressed similar problems where they have a constrained area that needs to be triangulated as well as weight masks for areas that need high geometric detail [Raman et al. 2008]. Lee and Schachter [1980] discussed ways of improving the speed of Delaunay Triangulation; however, this paper does not focus as much on the computation time since this technique is not intended for real-time use.

2 Related Works

While not completely related to this project, the mesh editing from Homework 1 served as inspiration for this project. In addition, many files from the homeworks were used

as templates and the Mersenne Twister header file was included directly.

3 Implementation

In this section the custom classes, libraries, and the algorithm itself will be described.

3.1 Classes

Due to the fact that Delaunay Triangulation becomes a much larger challenge in three dimensions, our data structures are designed to utilize only two dimensions while storing the third.

3.1.1 Point

Instead of storing an x, y, and z value, we only store x and y, while z is the 0 – 255 value of the pixel. The pixel's value in relation to the image will be discussed in the algorithm section.

Based on these facts, x and y are floats and z is an integer value. The Point class also has all the standard set() and get() functions.

3.2 CImg Library

This algorithm also makes use of the CImg library. CImg is a self-contained header file that only needs to be linked to the code; from there, one can perform image based operations on their file input. For this algorithm, the image() function was used to access a pixel's red channel value, 0 – 255. Although this algorithm utilizes greyscale images for the height maps, the images themselves are saved in RGB color space. This allows for the algorithm to potentially expand beyond greyscale images for input.

3.3 Algorithm

3.1 Image Analysis

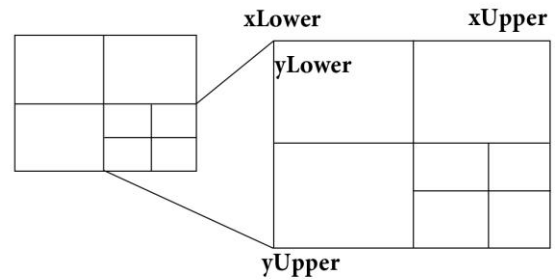

Figure 1: Diagram of a quadrant

Throughout analyzing the image, the concept of a quadrant will be used frequently. A quadrant can be defined as ¼ the area that is currently being considered, with points xLower, xUpper, yLower, and yUpper designating the top left (xLower, yLower) and bottom right (xUpper, yUpper) corners of the area. A second important note is that this algorithm currently only works with images that have equal width and height with an even number of pixels.

First, housekeeping is done to ensure that all images have at least a minimum amount of points assigned to them. To do this, points are placed at each corner of the image, one in the "center" of the image, and one point at the center of each quadrant (the image is initially split into 4 quadrants).

Next we call the function analyzeQuadrant() and pass in the 4 main quadrants of the image. analyzeQuadrant() is a recursive function which takes the image, the 4 corners of the current quadrant, a count, and a maxdepth as input.

analyzeQuadrant() will first check that count is not greater than max depth. This prevents the quadrants from being subdivided too much. Next, it will call

computeStandardDeviation() on the pixels that it covers, for now, we only need to know that computeStandardDeviation() returns true if the pixels that the quadrant covers vary enough in their values to be considered "detailed".

There are four conditions that can end the recursive function:

1. Max depth has been reached, and the quadrant's pixels are considered "detailed" :

   Add the quadrant's center pixel's average value as a point (the surrounding 3x3 grid), return.

2. The next quadrant subdivision will make the quadrant's size less than 3x3, and the quadrant's pixels are considered "detailed" :

   Add points for all the pixels in the quadrant, return.

3. The next quadrant subdivision will make the quadrant's size less than 3x3, and the quadrant's pixels are *not* considered "detailed" :

   Add the quadrant's center pixel's average value as a point (the surrounding 3x3 grid), return.

4. The quadrant's pixels are *not* considered "detailed":

   Add the quadrant's center pixel's average value as a point (the surrounding 3x3 grid), return.

Finally, if none of those stopping criteria are met, call analyzeQuadrants() four times, once for each sub quadrant of the current quadrant.

The computeStandardDeviation() function analyses all the pixels that are in a quadrant and calculates their standard deviation (1). It then checks how many of those pixels lie outside of 1 standard deviation from the mean and calculates the percentage out of the whole that are the "outliers". This percentage can then be compared to a user defined threshold. If the percentage is larger than the user defined amount, the pixels in that quadrant can be considered varied enough to be "detailed".

$$\sigma = \sqrt{1/N \sum_{i=1}^{N} (xi - \mu)2} \quad (1)$$

N is the number of pixels, xi is the value of the current pixel, and μ is the mean.

3.2 Delaunay Triangulation

Once the image has been analyzed and the points representing detail areas have been added, we can triangulate the points. The pseudo-code is as follows:

```
for(i=0; i<numPoints; i++)
   for(j=i+1; j<numPoints-1; j++)
      for(k=j+1; k<numPoints-2;k++)
         triangulate(i,j,k);

Triangulate(i,j,k){
```

```
    computeCircumcenter(i,j,k)
    for(points 0… numPoints){
        //reject
        if(point is in circumcenter) return;
    }
    Add triangle (i,j,k)
}
```

The function computeCircumcenter() determines the tightest circle which passes through all three points of the triangle and returns the center point of the circle (x,y coordinates) and it's radius. The rest of the above code goes through all the points and tests whether the circumcenter computed for those three points has any other points lying within it. If it does, it's not a valid triangle, if it doesn't add the triangle to the final mesh.

Finally, the printed data is sent through a custom script to be imported into 3D software (Blender).

4. Results

For our implementation of the algorithm, while everything works, it doesn't quite reach a useable state.

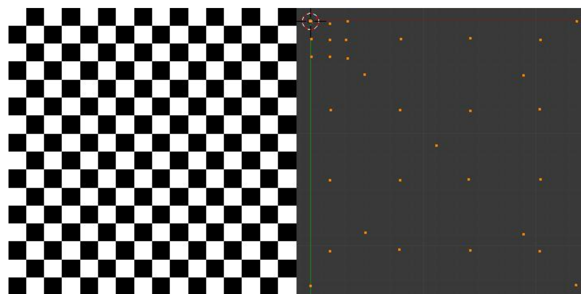

Figure 2: On the left is the input file (16x16px) and on the right is the point output.

One problem can be seen in Figure 2. When you have two extremes, such as pure black and pure white (values 0, 255) the formula to determine outliers reaches a corner case where the values and their standard deviations perfectly equal the threshold to be considered an outlier. As a result, most of the points seen on the right of Figure 1 are the default points set up at the beginning of the algorithm.

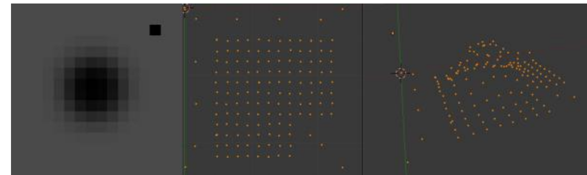

Figure 3: Left is 16x16px input, center is a top down view of generated points, right is a perspective view of the points.

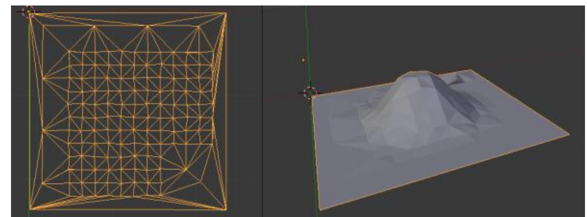

Figure 4: Delaunay Triangulation of the points from Figure 3.

Figure 4 illustrates an unfortunate feature of our implementation of Delaunay Triangulation. Because of the run-time of the triangulation is $O(n^4)$, anything much larger than 32x32 points becomes too large to run in a reasonable amount of time.

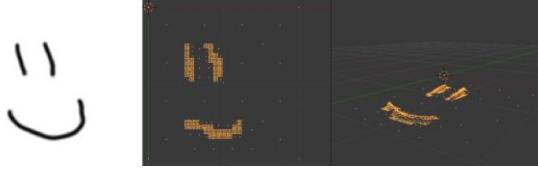

Figure 5: 128x128px image. Notice the sparse points in areas of pure white.

Figure 5 demonstrates how if a larger quadrant determines that there are no "interesting" pixels in it, parts of the image can get "cut off" from the point assignment. An example in the image would be the top right of the smile line.

Figure 6: South America, 512x512px

Figure 6 once again shows the sparse points in areas of low detail, such as the ocean; however, it also shows some faults in what we believe to be our implementation of the recursive calls to analyzeQuadrants().

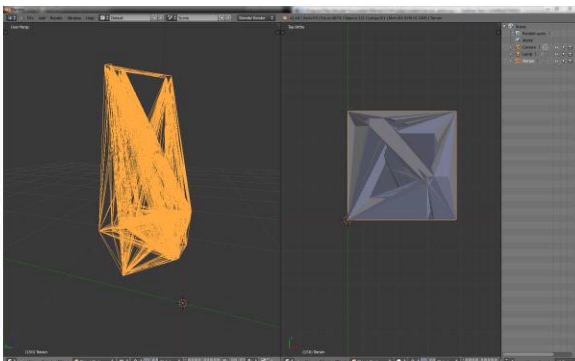

Figure 7: Incorrect triangulation

Figure 7 demonstrates what happens when you don't add a small amount (~0.05) of variation to the coordinates of the points. Since the points lay directly on a grid, the triangulation process has to deal with two plausible triangles that have the same values, even though there can only be one.

5. Conclusion and Future Work

In this paper we presented an automatic way to generate a mesh based off of a height map. The important features are the limited number of input commands needed to be given by the user (only the standard deviation threshold), smart detail placement in areas of high visual interest, and the automatic mesh generation. There are multiple areas that can be improved though, such as fixing known implementation bugs, handling non-square images, handling images with widths and heights that are not even, and most importantly, improving the Delaunay Triangulation runtime.

References

Raman, S., & Jianmin, Z. (2008). Efficient Terrain Triangulation And Modification Algorithms For Game Applications. *International Journal of Computer Games Technology, 2008*, 1-5. Retrieved April 20, 2013, from http://dx.doi.org/10.1155/2008/316790

Lee, D. T., & Schachter, B. J. (1980). Two Algorithms For Constructing A Delaunay Triangulation. *International Journal of Computer & Information Sciences*, *9*(3), 219-242.

Shewchuk, J. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator

Nasa Visible Earth http://visibleearth.nasa.gov/view_cat.php?categoryID=1484