Téllez Gaña Alejandro
Computer Geometry
3CM2

Terrain Reconstruction from Contour Maps

Introduction

Terrain reconstruction from contour maps is using bidimensional topographic maps defined by contour lines, known also as contour maps, to create a three-dimensional representation of such terrain, via image-processing, using techniques such as Delaunay triangulation and image sampling.

Three-dimensional terrain representation can find usages on orography, hydrography and other topographical data representation forms, for strategical and tactical purposes, like military intel; or for civilian use, such as weather prediction, fauna behavior (migrations), and distribution and availability of natural resources.

Process for terrain reconstruction from contour maps

In order to create a three-dimensional terrain reconstruction from contour maps, these steps are to be followed:

1. Read an image file, comprised by contour lines.
2. Extract contour lines (each closed curve).
3. Calculate the minimum distance between a pair of contour points from different contour lines.
4. Sample contour lines according to half the minimum distance aforementioned.
5. Calculate the Delaunay triangulation of the sampled points from all the contour lines.
6. Set height for each point of each contour line, according to some criteria (like area proportion), preserving the triangulation array.
7. Show reconstruction in 3D.

Things that you should know (a.k.a. 'Theoretical Framework')

Delaunay Triangulation

A Delaunay triangulation for a set **P** of points in a plane is a triangulation **DT**(P), such that no point in **P** is inside the circumcircle of any triangle in **DT**(P).

Properties

Let **n** be the number of points and **d** the number of dimensions.

- The union of all simplices[1] in the triangulation is the convex hull of the points.
- The Delaunay triangulation contains $\mathbf{O\left(n^{\lceil d/2 \rceil}\right)}$ simplices.
- In the plane **(d = 2)**, if there are **b** vertices on the convex hull, then any triangulation of the points has at most **2n − 2 − b** triangles, plus one exterior face.
- In the plane, each vertex has on average six surrounding triangles.
- In the plane, the Delaunay triangulation maximizes the minimum angle. Compared to any other triangulation of the points, the smallest angle in the Delaunay triangulation is at least as large as the smallest angle in any other. However, the Delaunay triangulation does not necessarily minimize the maximum angle.
- A circle circumscribing any Delaunay triangle does not contain any other input points in its interior (Delaunay condition).
- If a circle passing through two of the input points doesn't contain any other of them in its interior, then the segment connecting the two points is an edge of a Delaunay triangulation of the given points.
- Each triangle of the Delaunay triangulation of a set of points in d-dimensional spaces corresponds to a facet of convex hull of the projection of the points onto a **(d + 1)-**dimensional paraboloid, and vice versa.
- The closest neighbor **b** to any point **p** is on an edge **bp** in the Delaunay triangulation since the nearest neighbor graph is a subgraph of the Delaunay triangulation.
- For a set of points on the same line there is no Delaunay triangulation. For four or more points on the same circle the Delaunay triangulation is not unique.

[1] In geometry, a simplex (plural simplexes or simplices) is a generalization of the notion of a triangle or tetrahedron to arbitrary dimension.

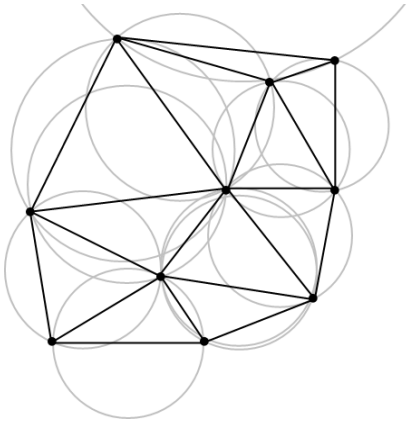

Figure 1 - A Delaunay triangulation in the plane with circumcircles shown.

Development

The development of this project was made under Java (1.6 u14) and the Netbeans IDE (7.1.1). Next, subsections will be presented according to the process mentioned before, briefly describing the development involved.

1. Read an image file, comprised by of contour lines.

Since the image files with contour lines were already proportioned, the first step in development was to extract the contour lines from an image file. Our dearest teacher ☺ provided us with a Java class [**HuellaDigital.Imagen.java**] (that was part of a digital fingerprint recognition project) with methods to read an image file as a matrix of gray-scale color pixels, as well as with other Java class to implement such methods [**HuellaDigital.Principal.java**].

```java
/**
 * Builds an in-memory representation of a grayscale image from a fi
 *
 * @param imagePath Path to the image file.
 */
public Image(String imagePath) {
    try {
        BufferedImage image = ImageIO.read(new File(imagePath));
        this.height = image.getHeight();
        this.width = image.getWidth();
        this.bits = image.getColorModel().getPixelSize();
        this.components = image.getColorModel().getNumComponents();
        this.pixels = new int[this.width][this.height];

        //Store each pixel in an array
        int pixel;
        for (int j = 0; j < this.height; j++) {
            for (int i = 0; i < this.width; i++) {
                pixel = image.getRaster().getSample(i, j, 0);
                this.pixels[i][j] = pixel;
            }
        }
    }
    catch (IOException e) {
        System.out.println(e);
    }
}
```

Figure 2 – Constructor code for creating an 'Imagen' object from an image stored on the local file system.

The constructor method receives a path to the image file, and uses the utility class 'BufferedImage' to read through the file as the method populates the object's array of pixels with the gray-scale color information. The height and width of the image are taken onto account, as well as its color model.

2. Extract Contour Lines

In order to extract the contour lines on the image file, it's necessary to read the recently populated pixel matrix, as it contains the information from the image file pixels.

Also, a new Class [XD.contour.util.Image.java] was created, in order to wrap and extend the methods provided by the Imagen class. Figure 3 shows a snippet code from the Image class method called getContourLines(), which reads the pixel matrix and stores it's values on it's corresponding ContourLine (a list of pixels), and this are stored

also onto a list (ContourLines). It keeps of how many and which different colors has read, for proper classification.

```java
/**
 * Loads the image data into pixel lists from it's pixelMatrix, to be stored
 * according to it's color as Contour Lines.
 */
public void getContourLines() {
    this.contourLines = new ContourLines();
    this.colorList = new LinkedList<Integer>();

    int currentColor;
    for (int j = 0; j < this.img.getAlto(); j++) {
        for (int i = 0; i < this.img.getAncho(); i++) {
            boolean curveExists = false;
            ListIterator<Integer> it = this.colorList.listIterator();
            //If not a white value for current pixel
            if (this.pixelMatrix[i][j] != 255) {
                if (!it.hasNext()) {
                    curveExists = false;
                } else {
                    while (it.hasNext()) {
                        currentColor = it.next();
                        if (this.pixelMatrix[i][j] == currentColor) {
                            //Color was already added
                            curveExists = true;
                            break;
                        }
                    }
                }
                //There is still no existing curve. Set non-existent. Create a
                if (!curveExists) {
                    this.colorList.add(this.pixelMatrix[i][j]);
                    this.contourLines.contourLines.add(
                            new ContourLine(
                                    this.pixelMatrix[i][j],
                                    new Pixel(i, j, this.pixelMatrix[i][j])));
                } else {
                    // If lists already exists, just add current pixel
                    this.getCurvaByColor(this.pixelMatrix[i][j]).
                            add(
                                    new Pixel(i, j, this.pixelMatrix[i][j]));
                }
            }
        }
    }
```

Figure 3 – Code for reading contour lines from the loaded pixel matrix.

3. Calculate the minimum distance

A brute force algorithm was used to calculate the minimum distance between two points from two different contour lines. The algorithm simply iterates through each point of each curve, and compares each one of them with each of the other points of the other contour lines. Figure 4 shows the implementation of this algorithm.

```
/**
 * Calculates the minimun distance that exists between two points of
 * different curves, and therefore, the samplingDistance required to sample
 * curves, as defined by half the minimum distance.
 *
 */
public void calculateMinDistance() {
    double minDist = 100000, d;
    for (ContourLine c1 : this.contourLines.contourLines) {
        for (ContourLine c2 : this.contourLines.contourLines) {
            if (c1.getIdColor() != c2.getIdColor()) {
                for (Pixel p1 : c1) {
                    for (Pixel p2 : c2) {
                        d = Image.getEuclideanDistance(p1, p2);
                        if (d < minDist) {
                            minDist = d;
                        }
                    }
                }
            }
        }
    }
    this.minDistance = minDist;
    this.samplingDistance = minDist / 2;
    //DEBUG
    System.out.println("MINDISTANCE: " + minDist);
    System.out.println("SAMPLINGDISTANCE: " + this.samplingDistance);
}
```

Figure 4 – Code for calculating minimum distance between two contour points from different contour lines.

Optimization

An optimization for this algorithm was proposed in order to reduce the number of calculations required when the number of contour lines rises significantly. On this project, an average of five contour lines per image file was assumed, so there was no need for implementing the proposed optimization. None the less, in the name of pure science, the optimization will be described next.

Introduction

Assuming there are four contour lines, each one with three, four, four and five points, respectively, giving a total of sixteen.

The brute force algorithm described early iterates through each point and compares them to each one of the contour points belonging to a different contour line; so, counting the operations needed, it will be:

$$(3\times13)+(4\times12)+(4\times12)+(5\times11)=196$$
$$39+48+48+55=196$$

Equation 1 – The number of calculations required by brute force algorithm.

But, taking into account that distances are a symmetric relation (e.g. the distance from point A to B is the same that exists from B to A, as it's not a vector, but a magnitude); so, there is no need to calculate a distance twice. In order to calculate a distance once, and only once, the following steps can be followed:

Starting from the contour line with less points on it (and on ascending order), the distance from each (three) of the points of the current contour line each to the other points of the other contour lines is calculated, giving the following number of calculations:

$$3\times(5+4+4)=39$$

Equation 2 – The number of calculations optimized for fist contour line (the one with least contour points).

So far, this is not different from the brute force algorithm, but when we move onto the next contour line a difference can be seen, as there is no need to calculate the distance from the current contour line points versus the ones on the previous contour line, as these have been already calculated. So, the number of calculations for the current contour line is reduced as:

$$4\times(5+4)=36$$

Equation 3 – The number of calculations for the second contour line (reduced compared to the ones in Equation 1).

Following this reasoning, each time our iteration moves to a new contour line, there is no need to calculate the distance to the points on the previous ones. Thus, when we reach the last contour line, there are no more needed calculations to perform, resulting the whole process on a model based on equation 4.

$$3\times(5+4+4)+4\times(5+4)+4\times(5)+5\times(0)=95$$
$$39+36+20+0=95$$

Equation 4 – The number of calculations required with the optimized approach.

As can be seen, the number of calculations is reduced significantly (one hundred, on this case). The number of calculations reduced is directly proportional to the number and difference in size of the contour lines.

Optimized algorithm

So, describing the proposed optimization as a series of steps, it will be something like:

1. Sort the contour lines on ascending order, according to the number of points contained on each one.
2. For each contour line, calculate the distance to the points from the contour lines which have not been yet iterated through, creating for each calculation, a distance element, comprising the two points and the distance value
3. Find the smallest distance element stored.

Issues

Note that it is required to store the distances values, so there will be a larger memory footprint for the program.

It is recommended to store the points within the distance element on a data structure that does not allow duplicate objects (such as HashSet on Java) to preserve the definition of distance as a magnitude and not a vector.

4. Sample contour lines

With the minimum distance value found, the sampling distance value is defined as half the minimum distance.

The method **sampleContourLines** iterates through each contour line, summoning its own method called sampleContourLine. This is useful, as for providing contour lines that define other methods for sampling.

The method s**ampleContourLine** moves along the contour line, starting at the pixel with the biggest x-value, and moving according to a mask applied to its current position. A new **Mask** object is allocated, to allow access to its methods, mask4 and mask8 in this case.

The class [**XD.contour.util.Mask.java**] was created in order to allow independent implementation for its methods, as well as allowing reimplementation or extension of this class via inheritance. The mask4 and mask8 methods implement the **8-connected neighborhood pixel search** (based on the Moore neighborhood).

On its way, the sampling algorithm deletes pixels which are not indispensable to the contour line (no significant points) to preserve the definition of 1-pixel thickness contour line.

For sampling purposes, it keeps a pixel as a pivot, moving only every samplingDistance interval, and adding itself as a significant point to the sampled version of the contour line.

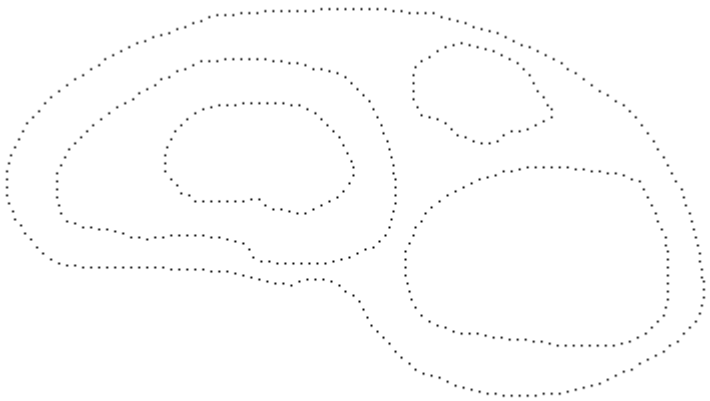

Figure 4 – The output of the sampling process.

```
/**
 * Moves along the contour line, starting at the pixel with the biggest
 * x-value, and moving according to a mask applied to it's current position
 * (4 and 8 neighborgs).
 *
 * @param width            The total width of the image, for bounds
 *                         detection
 * @param height           The total height of the image, for bounds
 *                         detection
 * @param samplingDistance The sampling distance, as calculated and defined
 *                         as half the minimum distance.
 */
void sampleContourLine(int width, int height, double samplingDistance) {
    Mask m = new Mask(width, height);
    Pixel next, current, pivot, past;
    LinkedList<Pixel> iterated;
    boolean flag1 = false;
    this.sampledCurva = new ContourLine(this.getIdColor());
    pivot = this.getMaxXPixel();
    this.sampledCurva.add(pivot);
    //DEGUG
    System.out.println("MaxXPixel: " + pivot.toString());
    iterated = new LinkedList<Pixel>();
    past = null;
    do {
        current = pivot;
        do {
            next = m.mask4(current, this, past, iterated);
            if (next == null) {
                next = m.mask8(current, this, past, iterated);
            }
            if (next != null) {
                if (iterated.contains(next)) {
                    flag1 = true;
                    break;
                }
                else {
                    iterated.add(current);
                    double dist = Image.getEuclideanDistance(pivot, next);
                    if (dist >= samplingDistance) {
                        this.sampledCurva.add(next);

                        past = current;
                        current = next;
                        pivot = current;
                    }
                    else {
                        past = current;
                        current = next;
                    }
                }
            }
            else {
                flag1 = true;
                break;
            }
        }
        while (true);
        if (flag1) {
            break;
        }
    }
    while (true);
}
```

Figure 5 – Code for sampling the curve, using the 8-connected neighborhood search.

5. Calculate the Delaunay Triangulation

For calculating the Delaunay Triangulation, we recurred to the Java Delaunay Triangulation API (jdt), which core functionality is embedded in a few classes. Besides the actual core classes, it also includes a GUI, which we partially implemented for previewing the result of the Delaunay Triangulation.

The API provided methods for reading a file (a .TSIN file) containing the coordinates of the points to be triangulated, as well as methods for saving the output on a file (a .SFM file), with an OFF like format, containing the points to be triangulated (converted to its own local coordinates) as well as the faces or triangles, defined by three points

Next, the format for the files will be described. Details for the actual implementation of the triangulation by the jdt API are not on the scope of the present work.

TSIN file

The original TSIN file format stores the X, Y and Z coordinates of the points to be triangulated. In order for us to preserve the color information of the pixels along the triangulation (since the original coordinates get lost when the API parses them to their local coordinates and, therefore, can't be retrieved from the original image's matrix of pixels) we added a four component (a color component).

The first line of the file tells the number of points to be triangulated. For us to know the number of points, it's gotten by simply adding the size of the contour lines stored in memory.

```
1  504
2  371 170 300.0 79
3  371 174 300.0 79
4  371 178 300.0 79
```

Figure 6 – Snippet of our custom TSIN file.

SMF file

The SMF file contains the points that were triangulated, as well as the triangles or faces, defined by three points. As before, we customized our SMF file format by adding a color component to the points. These are defined by a letter 'v' on the start of the line. Triangles or faces are defined by a latter 'f' on the start of the line. It is to be noted that the triangles are not composed of each of the coordinates of the three points that define them, but as three index numbers for identifying three points within the file (the index for the points starts at 1).

```
503  v 371.0 170.0 300.0 79
504  v 371.0 174.0 300.0 79
505  v 371.0 178.0 300.0 79
506  f 474 475 471
507  f 474 496 475
508  f 475 476 471
```

Figure 7 – Snippet of our custom SMF file

So, in order to communicate with the API, first it was needed to write a TSIN file with the points of each sampled contour line to be triangulated. Then, use the jdt methods to read it and calculate the triangulation, and then writing the triangulation result on a SMF file, then again, reading the SMF file to process the triangles.

6. Set height for each contour line

The heights were set using an algorithm based on the relative area of each contour line to the max area found along all contour lines; since it is assumed that all the contour lines belong to **terrain elevations such as mountains** (orographic map) the contour line with the biggest area is the lowest height.

```java
/**
 * Calculates the heights for each contour line, setting a max height as the
 * minimum value between the height and width of the 2-D original image;
 * assigning to each contour line a height proportional to the area of its
 * bounding box. The height for each contour line is proportional to the
 * area it occupies.
 *
 */
public void calculateHeights() {
    this.maxHeight = Math.min(this.img.getWidth(), this.img.getHeight());
    int maxArea = 0;
    //Finds biggest area (lowest contour line)
    for (ContourLine c : this.contourLines.contourLines) {
        c.calculateBoxArea();
        if (c.getBoundingBox() > maxArea) {
            maxArea = c.getBoundingBox();
        }
    }
    System.out.println("Max Area is " + maxArea);
    //The height for each contour line is proportional to the area
    //it occupies
    for (ContourLine c : this.contourLines.contourLines) {
        c.height = (c.getBoundingBox() * this.maxHeight) / maxArea;
        c.risePixels();
    }
}

public void imprimeAlturas() {
    for (ContourLine c : this.contourLines.contourLines) {
        System.out.println("Curva " + c.getIdColor()
                + " @(" + c.height + ")");
    }
}
```

Figure 8 – Code for setting the height of each contour line.

Or put in simpler words, the relation between the height of each contour line and the max height (min value of height and width of the 2-D original image) is directly proportional to the relation between the area of each contour line and the max area (the biggest area found along contour lines)

$$\frac{contourLineHeight}{\max Height} : \frac{contourLineArea}{\max Area}$$

Figure 9 – The proportion used for determining the height of each contour line.

7. Show reconstruction in 3D

To draw in 3D, we used the Java 3D API (J3D). We proceed to create triangle objects (as defined by the class XD.contour.util.j3d.Triangle_d.java).

```
while (s != null && s.charAt(0) == 'f') {
    StringTokenizer st = new StringTokenizer(s);
    //Jump the first 'f' character identifier
    st.nextToken();
    //Add to the triangle array, from 3 to 3 coordinates
    Triangle_d t = new Triangle_d();
    int i1 = new Integer(Integer.parseInt(st.nextToken()));
    int i2 = new Integer(Integer.parseInt(st.nextToken()));
    int i3 = new Integer(Integer.parseInt(st.nextToken()));

    t.setP1(puntos.get(i1 - 1));
    t.setP2(puntos.get(i2 - 1));
    t.setP3(puntos.get(i3 - 1));

    triangulos.add(t);
    s = is.readLine();
}
```

Figure 10 – Code snippet for triangle creation while reading the SMF file, setting its points from the indexes read at each iteration, indicating the position of the stored points in the list 'puntos'.

Having read the points and built the triangles as a logical structure, they need to be created as a 3D shape. This is done via the 'TriangleArray' class, which stores every triplet of points as a single triangle shape. The color for each vertex is preserved, and gotten via the p.getColor3f() method (a method defined in the class XD.contour.util.Pixel.java). Triangles are draw by both sides (obverse and reverse) to ensure visibility from all sides.

```
//Draw triangles to ensure visibility from obverse and reverse
triangles = new TriangleArray(
        triangulos.size() * 3 * 2,
        TriangleArray.COORDINATES | TriangleArray.COLOR_3);
int contador = 0;
for (Triangle_d t : triangulos) {
    MyPoint3d p;

    //obverse
    p = (MyPoint3d) t.p1;
    triangles.setCoordinate(contador, t.p1);
    triangles.setColor(contador, p.getColor3f());
    contador++;

    p = (MyPoint3d) t.p2;
    triangles.setCoordinate(contador, t.p2);
    triangles.setColor(contador, p.getColor3f());
    contador++;

    p = (MyPoint3d) t.p3;
    triangles.setCoordinate(contador, t.p3);
    triangles.setColor(contador, p.getColor3f());
    contador++;

    //Reverse
    p = (MyPoint3d) t.p3;
    triangles.setCoordinate(contador, t.p3);
    triangles.setColor(contador, p.getColor3f());
    contador++;

    p = (MyPoint3d) t.p2;
    triangles.setCoordinate(contador, t.p2);
    triangles.setColor(contador, p.getColor3f());
    contador++;

    p = (MyPoint3d) t.p1;
    triangles.setCoordinate(contador, t.p1);
    triangles.setColor(contador, p.getColor3f());
    contador++;
}
```

Figure 11 – Code snippet for triangle representation using ‘TriangleArray’.

8. Putting all together (Or the ‘main’ method)

Putting al together was really pretty straightforward and in a few lines of code.

```
1  package XD.contour;
2
3  import XD.contour.util.Image;
4  import XD.contour.util.j3d.Reconstruction;
5  import XD.contour.util.j3d.SemiReconstruction;
6  import jdt.gui.DelaunayPreview;
7
8  /**
9   * Class created to test functionality of objects and such.
10  *
11  * @author Alejandro Téllez G. <java.util.fck@hotmail.com>
12  */
13 public class Test1 {
14
15     public static void main(String args[]) {
16         Image i = new Image("curvas1.bmp");
17         i.getContourLines();
18         i.calculateMinDistance();
19         i.sampleContourLines();
20         i.writeImage();
21         i.calculateHeights();
22         i.write_tsin();
23         i.calculateSampledMatrix();
24         Image sampledImage = new Image(i.getSampledMatrix(), 8,
25                                        i.img.getAncho(), i.img.getAlto());
26         sampledImage.writeImage("sampledCurvas");
27         DelaunayPreview f = new DelaunayPreview("salida.tsin");
28         f.setSize(i.img.getAncho(), i.img.getAlto());
29         f.setVisible(true);
30         f.myWriteSmfFile("triangulacion.smf");
31         Reconstruction.main2(i.pixelMatrix);
32         //Reconstruction using random colors A.K.A. Rainbow :)
33         SemiReconstruction.main2(i.pixelMatrix);
34         i.imprimeAlturas();
35     }
36 }
37
```

Figure 12 – The ‘main’ method.

Results

The program seemed to respond well overall, except where the contour lines were not continuous and where there were more than 3 neighbors to follow (using the 8-connected neighborhood pixel search algorithm).

Next, screenshots from the program will be presented.

General Output

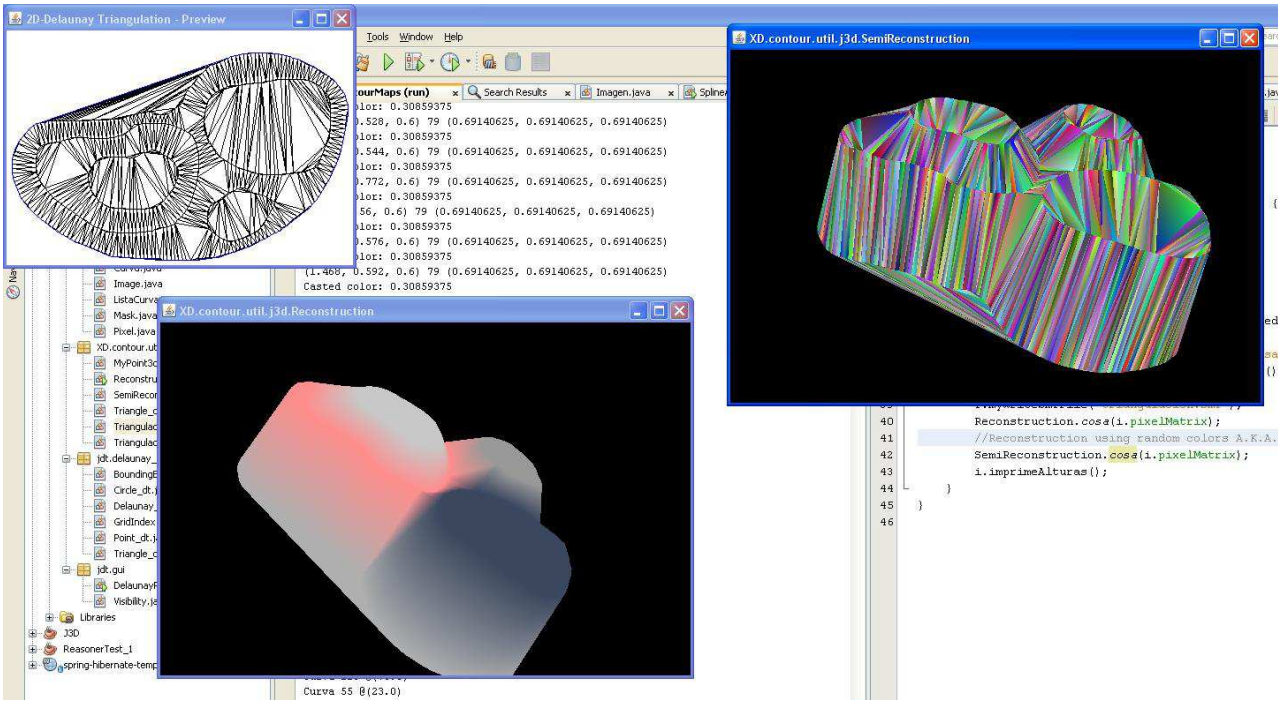

Figure 13 – The output of the full program running. The 2D – Delaunay Triangulation, as well as 3D reconstruction, with the original color restored (mapped), as well as one showing triangles in random colors (to appreciate the triangulation in 3D).

Conclusions

- There is a need to optimize the 8-connected neighborhood pixel search algorithm version used in this project, as it tries to smooth the image to preserve the 1-pixel thickness of the contour line, but only when there are at most three neighbors (including the past visited and the next to visit). So, sampling process might stop when facing more than three neighbors, assuming than the next pixel is the last visited, and therefore, assuming that all the pixels on the curve have been visited, getting to falsely normal termination.

- The optimization for the calculation of the minimum distance between two points of different contour lines was not implemented, but it might be interesting to develop it as future work, assuming there could be a very high number of contour lines to work, while there would be much memory disposition and low processing power.

- The triangles had to be drawn in obverse and reverse, to avoid the common problem of 'objects suddenly disappearing when rotating' or 'one-faced-polygons'.

- As far as I can see, there might be no problem on assigning the heights of the contour lines first, and then calculating the Delaunay triangulation; remembering that the jdt API supports three-dimensional triangulation; but I did not tested it, as there was no need or benefit from inverting the order of both steps.

- The jdt interface (as managing files for input and output parameters) was of great aid, as it released us from getting into the guts of its implementation, and therefore, fulfilling its API purpose.

- As much as I love programming on Java, and as easy as J3D might see, I really don't like programming for J3D that much ☹.

- The three-dimensional reconstruction should preserve the original color of the contour lines, as it may prove to be very useful to determine the proportion of the heights between the contour lines when viewing the 3D model. With that in mind, a mapping function was implemented on this project; and, as random as it may seem, it fulfills its function fairly well.

Credits

Boaz Ben Moshe - Providing the jdt API

Angel Pérez - Providing the Imagen class, used for reading and writing image files, as well as turning them onto a matrix of pixels.

Alejandro Téllez G. – All contents on the XD.contour package and subpackages.