Web-Based Demonstrations of Line Simplification Algorithms

Final Year Project Final Report

Nicholas Bernon

A thesis submitted in part fulfilment of the degree of BSc. (Hons.) in Computer Science with the supervision of Dr Michela Bertolotto.

Department of Computer Science

University College Dublin

13 March 2005

Abstract

Line simplification algorithms are used to produce a simplified version of an original polyline. Often a polyline can have a resolution far greater than is necessary. For example, a polyline can have several vertices within the vicinity of one pixel. For the sake of efficiency, line simplification algorithms have been developed and are widely used for both computer graphics and geographic information systems (GIS). This report deals with the implementation of some of these algorithms and how to demonstrate them by visual means and web technology.

Table of Contents

1 Introduction

Sometimes a linear map feature might be unnecessarily complex. Polyline simplification replaces these features with a less complex representation. For this reason line simplification algorithms have been developed with the aim of eliminating redundant or unnecessary coordinate information from line features whilst still retaining the original geometric characteristics of a line.

As figure 1 demonstrates, if successive vertices on a polyline are located too close together that a number of them become irrelevant, they can be reduced to a single vertex while still preserving the overall shape of the line therefore simplifying the line and improving the efficiency of the processing stage. The blue circles indicate the vertices to be connected after simplification.

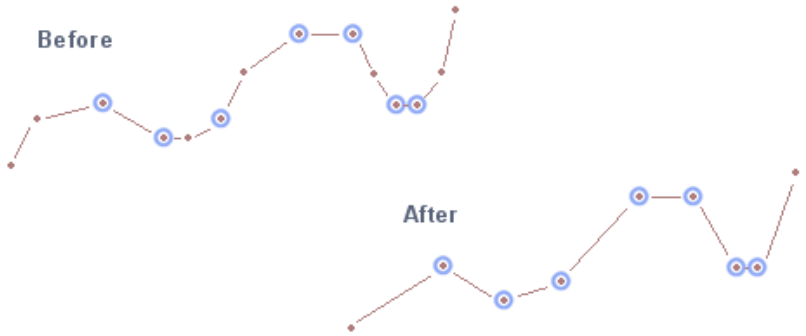


Figure 1. Before and after line simplification.

The aim of the project was to create and design and fully functional program capable of performing line simplification. This program has been designed entirely using java, line simplification algorithms and mathematical formula and has been engineered primarily using the Douglas-Peucker algorithm and recent modifications.

The project began with a clear set of initiatives and objectives. As the program progressed, however, it became clear that the objectives would have to be regularly reassessed as a number of issues and ideas were arising that demanded more immediate attention. The result is that the program became more moulded than the skeleton plans that were laid out at the start.

The program is able to analyse a line segment by drawing a polyline or inputting coordinates of the line from a file. It is then able to simplify the polyline according to a specified tolerance level. It also includes modifications of the Douglas-Peucker algorithm which enable it to eliminate intersection and self intersection. It has been customised for usability to allow line segments to be modified and adjusted. The program has also been integrated with html, allowing it to be used on the internet as an applet.

This report is concerned with the implementation of these line simplification algorithms in a java program. The report is structured as follows, the Background Research section deals with the various algorithms that have been researched and implemented in the program. It also outlines the problems with some of these algorithms and some of the modern day uses for them. The Program Functionality section deals with the functional aspects of the program and explains how they have been implemented. The Design Aspects section explains some of the interesting design issues that were encountered and some of the solutions that were considered. Detailed Design and Implementation discusses some of the interesting implementations that were created for the project. Testing/Evaluation explains the some of the testing methods that were employed to improve the application. The conclusion describes what has been achieved and what the future holds for line simplification.

2 Background Research

2.1 Line Simplification Algorithms

Simplification of lines is a "generalisation process" that has been the topic of much scientific research. Line simplification may be defined as the elimination of unwanted detail. This process is accomplished by selecting a subset of points which best represent the geometric properties of a polyline while eliminating the remaining points.

Line Simplification algorithms have become especially important in the field of cartography. Nowadays, with digital records of maps and terrains these algorithms are incorporated frequently in many areas. In general, up to 80% of the information on digital maps consists of lines. Line simplification can greatly improve the efficiency of these systems.

Reasons for Line Simplification:

- **Reduced Plotting Time:** If the plotting time is slow in relative terms, it can cause a bottleneck effect where the plotting time "clogs up" an output. After simplification, the number of vertices are reduced, therefore increasing the plotting time.

- **Reduced Storage:** Coordinate pairs take up large amounts of space in GISs. The coordinates or data can be significantly reduced by simplification, in turn decreasing the amount of storage space needed and the cost of storing it. Simplification could reduce a data set by up to 70% without perceptually changing the line.

- **Processing:** Simplification can greatly reduce the time needed for vector processing a data set. It can also speed up many types of symbol-generation techniques.

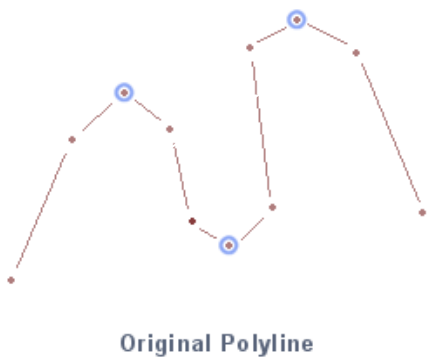
Evaluating the Simplification Process:

There are many ways to measure the simplification process, two of which are simple attribute measurements and displacement measurements. Simple attribute measurements can be applied to a single line including length, angularity and curvilinearity. Displacement measurements compare and evaluate the differences between the base line and the simplified line.

The Douglas-Peucker algorithm is one of the most effective algorithms for maintaining the critical geometric characteristics of the data.

2.2 Classical Douglas-Peucker Algorithm

The classical Douglas-Peucker algorithm was developed by David Douglas and Thomas Peucker in 1973 and still remains one of the most widely used and effective line simplification algorithms. It has a worst case complexity of $O(n^2)$, where n is the number of vertices along the polyline.



The algorithm works by calculating the point with the maximum distance from an edge segment. Given a line that needs to be simplified, the algorithm starts by joining the starting point and the end point of the line with a straight line segment. It then calculates the perpendicular distance of all vertices from this line. If the distance between each vertex and the line segment is within a specified tolerance* (distance value), then the straight line segment represents the whole line in its simplified form.

Figure 2. Polyline showing points of maximum distance.

If the tolerance condition is not met, the point with the greatest distance from the straight line segment is selected, and the straight line segment is subdivided, joining the two end points to the point of maximum distance. This process is repeated until all vertices are within the specified tolerance as this algorithm relies on a recursive decomposition of the line.

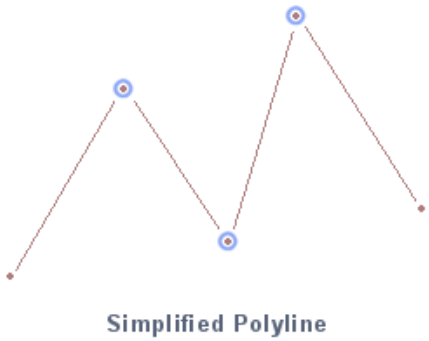


Figure 3. Douglas-Peucker: Finding the point of maximum distance.

*Tolerance: The tolerance is a pre-determined threshold distance used to define the maximum distance to be simplified. The distance is measured from each vertex along a perpendicular path to the current approximating segment.

There are, however, some problems with the algorithm in its classical form. The Douglas-Peucker algorithm does not guarantee preservation of consistency.

Problems:

- **Intersections:** This is when the simplified line intersects other polylines
- **Self Intersections:** This is where the simplified line intersects itself.

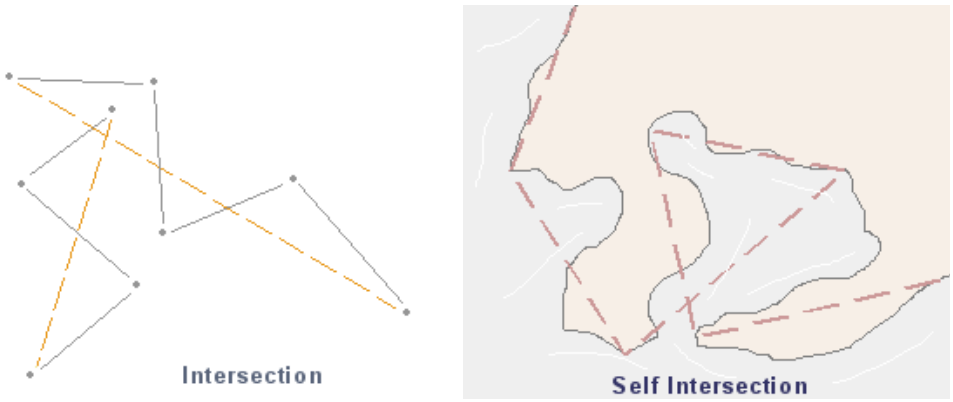


Figure 4. Example of intersection and self intersection

Figure 4 shows an example of intersection in which two separate lines intersect themselves after simplification has taken place. Self intersection is also demonstrated where the simplified line intersects itself. This occurs because the Douglas-Peucker algorithm is calculating the maximum distance from an edge segment and in this case it causes a cross-over.

- **(Point) features change sides:** This is where a feature changes sides after a polyline has been simplified.

Figure 5. The island has jumped inland after simplification

Figure 5 shows a coastline with a large indent in which there lies an island. After the coastline has been simplified, however, the island seems to move inland.

2.2.1 Hershberger and Snoeyink's modification of the Douglas-Peucker Algorithm

The most expensive step in the simplification process is finding the point of maximum distance from the current approximating segment. Hershberger and Snoeyink proposed that the maximum point must lie inside the convex hull. They developed data structures to efficiently maintain and access the current convex hull. This increases the worst case complexity of the algorithm from $O(n^2)$ to $O(n \log n)$.

2.2.2 Saalfelds Modification of the Douglas-Peucker Algorithm

Saalfeld was the first to demonstrate topological preservation in lines simplified by Douglas-Peucker subdivision. Saalfeld argued that generalisation causes coordinate displacement but that these displacements will always fall within the convex hull of the original line. The convex hull contains the two endpoints and the maximum deviation of vertices on both sides of the polyline. This essentially limits the search for these inconsistencies to within the convex hull. This is used to eliminate topological inconsistencies such as the islands jumping inland example (figure 5).

Saalfeld modified the Douglas-Peucker algorithm so that it could calculate the parity check values of these vertices that lie within the convex hull. The parity checks at each vertex by projecting a ray and counting the amount of intersections with the polyline and the segment connecting the two endpoints of the polyline. An even sum indicates the same number of polylines and the segment connecting the endpoints, which prevents an overlap. This is demonstrated in figure 4.

Saalfeld's modification guarantees:

1. **No self-intersections**: The simplified line will not intersect itself.
2. **No intersections**: The simplified line will not intersect other polylines.
3. **No topological inconsistencies**: Islands will not jump inland.

Saalfeld's algorithm has been applied to generate a sequence of simplified maps to be stored on a server and transmitted progressively upon user's request (Barbara Buttenfield, 2002). This version of the algorithm was preferred because it allowed the automatic generation of the maps while preserving consistency.

2.3 Geographical Information Systems (GIS)

Line simplification algorithms are used in many different technologies nowadays. They have given rise to new ways of media and digital technologies which can be used for depicting geography, monitoring. The usefulness of line simplification algorithms is most evident in GIS applications.

GIS is a system of computer software, hardware, data and personnel to help manipulate, analyse and present information that is tied to a spatial location. GIS is used all around the world in many different industries ranging from the petroleum industry to forest managements to phone companies. A GIS system combines various types of data about a location to provide a better understanding about that place.

A GIS could analyse for example, the distribution of mailing list subscribers in an area like the United States. Data like this is analysed and the results are used to make decisions by making future assumptions. This data could indicate where more people are likely to subscribe to a newsletter and so businesses could spend resources on focusing advertising in these areas which are likely to be more profitable.

GIS is responsible for planning, decision-making, operations management and inventory. It is trusted with the responsibility of planning and routing hundreds of miles of transmission and distribution lines, waterways, sanitary sewers and outlining boundaries for forest management practises. GIS is even used for organising traffic circulation and improving the efficiency of newspaper routes. It is incorporated in the petroleum industry letting the oil companies

GIS largely uses and stores visual records and data which occupies large amount of memory space. Line simplification algorithms can be applied to line segments which can significantly reduce the data size on disk. The fact that the size of the file has been reduced means that the plotting speed is increased improving the efficiency of the GIS.

3 Program Functionality

The project has evolved relatively close to the guidelines set down when work began. The aim was to create a program which could demonstrate the line simplification process that could also be viewed on the internet. The project went through a period of research in order to determine the appropriate means by which to construct it. It was eventually decided that the most appropriate language to design the program would be java as it could be easily implemented on the web. Once the language was decided on, it was up to the

3.1 Tolerance Level

The tolerance level specifies how much data reduction is to be applied to a polyline during the line simplification process. It works by accepting an integer from the user and simplifying every point that has a greater distance from the simplified line. The tolerance is input through a text field at present for testing purposes, but will later be implemented as a slider for easier visual demonstrations.

3.2 Mouse Events

The mouse events were set up as a way of recording and implementing details about the polyline. The mouse events are used as the indicator to record a value and modify the vectors used for storing these values accordingly. There are several of these mouse events including, `mousedReleased`, `mousePressed` and `mouseDragged`. Along with performing draw functions, they are used to return the exact location of the cursor and can store that value as a dimension in a stack. The purpose being, that if the points which make up a polyline are recorded, this data can then be manipulated using equations to come up with geometric information. The information generated is then used by the simplifying engine to produce simplified lines.

3.3 Loading Dataset Values from a File

In an earlier version, the program was only able to accept data manually by adding points each time. This is why extending the application to read in points from a file was suggested and then implemented. When the load button is pushed, a *JFileChooser* automatically appears in order to locate the file that the user wishes to be loaded. Inside the method a try block is created so that if an exception occurs, it is handled by an exception handler associated with it. A catch statement is put after the try block to associate an exception handler with it. Due to the fact that this loading feature involves file input, I/O exceptions are caught and dealt with accordingly which is in this case to exit the method and display an error message. This will greatly speed up I/O for some network conditions.

```
while there are more lines do
read line
if (line contains "newline”) then create new line
else
    if (line contains a valid x-coord) then
           readline
           if (line contains a valid y-coord) then
                  add point (x, y)
od
```

Reading in dataset values

The program then searches through the dataset looking for three strings. It searches takes each line in turn and looks for “x”. If an “x” is found it then looks for a “y” which identifies the x and y coordinates respectively. It also searches for the string “newline” which indicates that a line has ended and a new line has been created.

3.4 The Simplification Engine

The program needed to be designed in such a way as to be able to analyse a polyline and create a geometrically effective replica with a given tolerance. With this in mind, I created a JPanel with the intention of making a scribble pad to graphically demonstrate drawing these lines. The location of each vertex is stored inside vector data structures automatically based on the users mouse input. When a simplification is requested, the start point and the end point are identified and are directly connected by a line segment which is stored in a separate vector. Then each vertex is individually analysed to check the perpendicular distance between it and the line segment. When the vertex of maximum distance is found, the connecting line segment is removed from the vector and is split into two parts connecting both end points to the maximum point. The program recursively analyses the points, measuring the distance to its corresponding line segment and splitting the line in two until the furthest vertex from each line lies within the specified tolerance level. Each time a line is split in two, the old connecting line is removed from the vector and replaced by the two new lines.

```
RDPSimplify(P,i,j,ε) simplifies the polyline with vertices Vi,Vi+1,…Vj-1,Vj
Where
    P = {Vi, Vi+1, …, Vj-1, Vj} set of vertices composing the polyline
    Initialise output vertex set V to {Vi, Vj}
    RDPSimplify(P, i, j, ε);
    Output V;

    RDPSimplify(P, i, j, ε)
    {
           if (j > i + 1)
           {
                  Find the vertex Vk with maximum distance (dist) from the
segment                       linking the endpoints of P;
           If (dist > ε)
           {
                  Add Vk to output vertex set V;
                  RDPSimplify(P, i, k, ε);
                  RDPSimplify(P, k, j, ε);
           }
    }
}
```

The Classical RDP algorithm

The above algorithm outlines how to implement the Douglas-Peucker algorithm for a given polyline p looking at vetices between the indices i and j, for a given tolerance level z.

3.5 Adding Points

The *addPoint* method is the logical place to perform most vector checks. It was found that in many cases, points of the same value were consecutively being added to the vector. This level of detail is far greater than is needed. For example, there is no need to create many points for the same pixel. As a result of this, however, the efficiency was being affected and so a resolution had to be sought. As a way of resolving this, it was decided to create a check in which the *addPoint* method tests to see if there are any occurrences of points with the same value. If such a point is detected, it is automatically removed from the vector.

The *addPoint* method also calls the *checkIntersection* method each time a point is added. This is the logical place to call this method as if there is an intersection; a new point is not created.

3.6 Removing Points

Removing points involves a simple process. The program firstly checks to make sure that there are points in the working vector to remove. If there is then checks the size of the vector and subtracts one away. Essentially this removes the last added point in the vector that is currently selected.

3.7 Creating Multiple Lines

In order to create separate individual lines a new vector is created. The new vector is then added to the overall vector while also ensuring that it is now the one being added to. By creating a new vector like this, it is then possible to refer back to an individual line even after a newer line has been created. This works well for graphical selection purposes.

3.8 Checking for Intersections

One of the main modifications of the Douglas-Peucker algorithm is to check for both intersection, and self-intersection. Firstly, the program accesses the working points vector and checks to see whether any points have been added. If not, then a line can hardly intersect. It is inside the *checkIntersection* method that a line2d object (*line*) is created from the point being added to the previous point. The best way to detect an intersection is to loop through the vectors and cross reference the line2d values, and in order to do this we create a vector of vectors. In order to do this, we loop through every line set in the **total vector**, and compare it with every line in the **current line set**. While iterating, it creates a line2d object (*currentLine*), which compares itself to the line2d object (line) and checks to see if any of the coordinates match. If a matching point is found, a message dialog box appears explaining that a self-intersection has occurred. The point is therefore not added to the vector.

4 Design Aspects

4.1 Method of Deployment

The project is displayed on a web page which was constructed as a means to viewing the program online. It required constructing the project as an applet and integrating it inside a html page. The applet is very useful as the line simplification program can be run directly from a java-compatible web browser. The entire workings of the project are contained within the applet which is available online. In order to display the applet, you create a Manifest file (mymanifest.mf) with the name of the class that holds the main method. A manifest file contains information about the files packaged in a JAR file. The JAR file contains the java classes forming the applet. It means that the program can be run independently on most computers with a basic java plug-in.

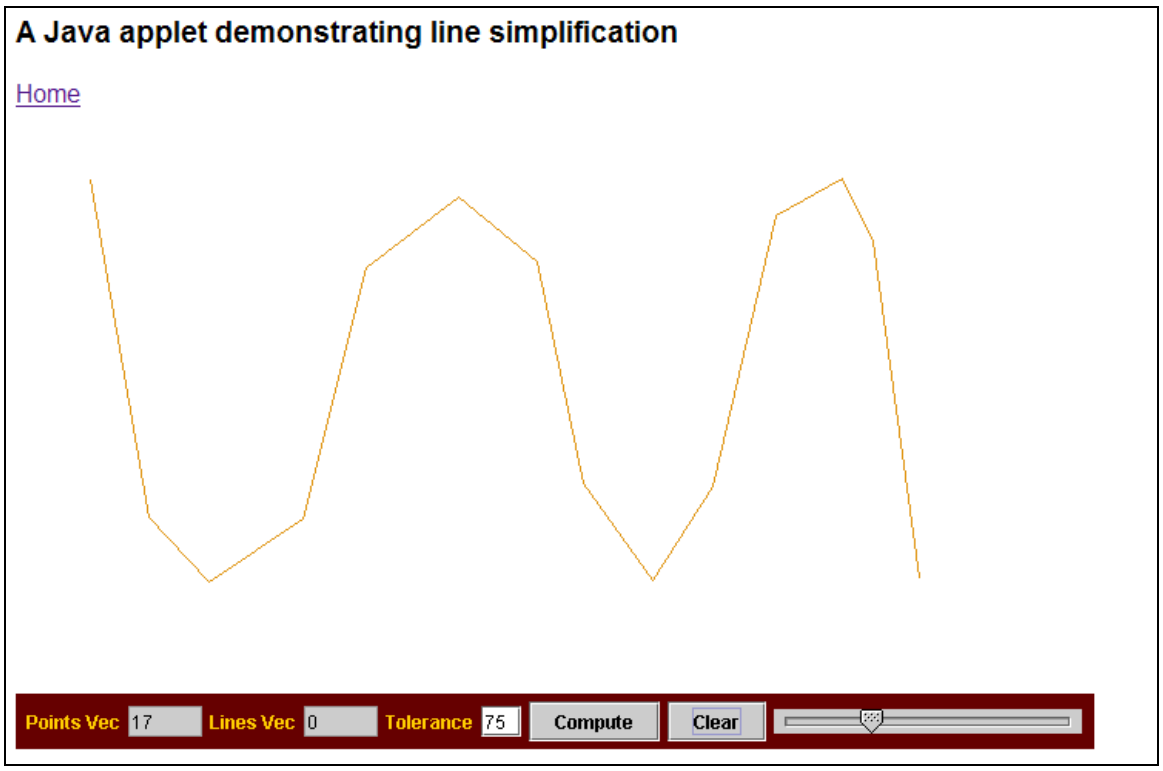


Figure 6. Line Simplification Applet

The screenshot above demonstrates the Line Simplification program as an online applet. The project has been designed purposely for ease-of-use online by making it compatible to all browsers with even the most basic versions of java. For example, when designing the interface complex swing methods that require more recent versions of java were avoided.

The web page contains instructions on how to use the program along with information on Line Simplification. It was constructed according to the theme of the project which is reflected in the design. The look of the webpage was designed using HTML (hypertext markup language) and CSS (cascading style sheets). The webpage was made using the finest web standards and is valid and accessible to all browsers.

4.2 Paint Component

A new paint component had to be designed to handle various actions. One problem that existed was that when the program was restored after being minimised, the program repainted the `JPanel`. Being that there was nothing stored in the default paint component all lines and points disappeared. This was fixed by firstly creating a custom-built paint component and secondly placing all drawing elements in the paint component to be drawn exactly as they were. Another reason for the paint component was to be able to display different variations of the simplification process for the same polyline. For this to be possible, the paint component was designed so that after every simplification the original polyline along with the simplified line (according to a specified tolerance) are drawn.

4.3 Paint Mode

With the proposed idea of allowing for graphical selection, a mechanism had to be designed to switch between drawing a point, and selecting a line. It was decided there should be a separate mode for both. This was largely for usability, as the user could simply toggle paint mode off when selecting a line, and toggle paint mode on when manipulating or creating new points.

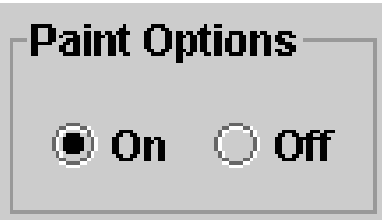


Figure 7. The Paint Components

It also avoided confusion when adding and removing points as when the paint options are off, the user is limited to graphical selection. The idea of the radio buttons indicates to the user that the paint options can be either on or off as only one can be selected at a time.

4.4 Choice of Data Structures

A vector was the data structure of choice because the project demanded the functionality of an array, but couldn't be tied to a specific length. Continuous lines joining points to points are contained in the overall vector, whilst the complete lines from endpoint to endpoint are vectors themselves. The vectors containing the complete lines are stored as points for good reason. For example, there are two connecting lines that need to be stored and they can be stored as either points or lines. If the two lines were being stored as points, then it would take three points to represent the two lines, however, if the two lines were stored as individual lines, it would require storing four endpoints, two of which are the same.

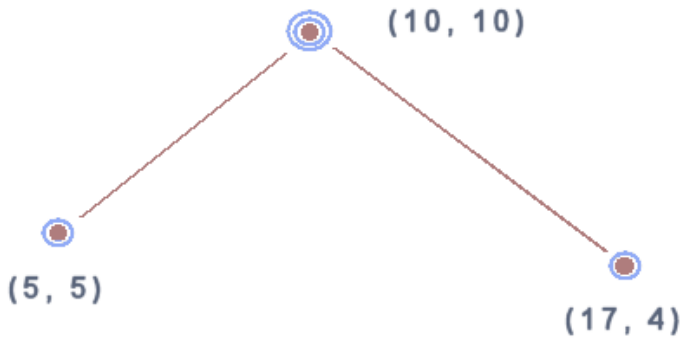


Figure 8. Two Lines made up of Three Points

The data above can be stored as either lines or points:

- Lines stored as lines: (5, 5) (10, 10), (10, 10) (17, 4)
- Lines stored as points: (5, 5) (10, 10) (17, 4)

If there were complex individual lines dealing with three dimensions then it might be practical to save the data as lines, but in the case of this program, it is more efficient to save the data as points. The main reason for this is that when the coordinates are recorded as data, consecutive points connect to each other and so there is no need to distinguish between lines unless they are segregated.

4.5 Controls

A control system also was set up for the program in order to perform certain tasks. The program needed to be able to display some information in the applet to be viewed by the user, for example, the amount of points added to the vector. It also required the user be able to modify certain parts of the code, namely the tolerance. The final element in the control panel specified that users also needed to be able to execute commands.

Figure 9. Tolerance Control System

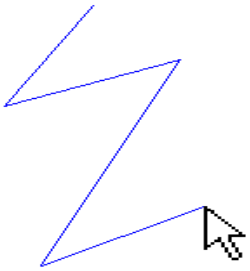
The most effective way of displaying and modifying the data turned out to be using `Text Fields and a slider`. This is because the data values could be easily inserted into the `Text Fields` to be either displayed or modified. I had initially used various mouse events to execute commands, but found using buttons to be far more effective and user friendly.

There are a number of buttons available to the user with the following uses:

- The `compute` button controls when the polyline simplification is initialised.
- The `clear` button actually resets the program, removing all elements from the vectors and repainting the `JPanel`.
- The `remove` button removes the last entered point in the currently selected vector
- The `new line` button gives the option to create multiple lines.
- The `paint` option toggles the paint mode on and off.
- The `load` button loads a dataset file of coordinates into the program.
- The `save` button allows the contents of the points vectors to be stored as a dataset file.
- The about button prompts a dialog box to pop up displaying some information about the project and it's designer.

4.6 Graphical Selection

Graphical selection works by reconstructing every line that's been drawn and finding if the mouse cursor has been clicked within a certain proximity. If a line is selected, the points that make up the line are assigned to the workingPointsVector. This enables the user to modify that line and indeed any line that has previously been drawn. When a line is selected, the line's colour changes to indicate that it has been selected, and likewise the previously selected line changes colour to indicate that it has been deselected.



Step 1: Demonstrates a simple line being drawn. The colour of the line is blue, indicating that it is the currently selected line.

Figure 10. Step 1: Drawing a Line

***Step 2**: Shows a new line being created. Note: the colour of the first line changes as it is deselected.*

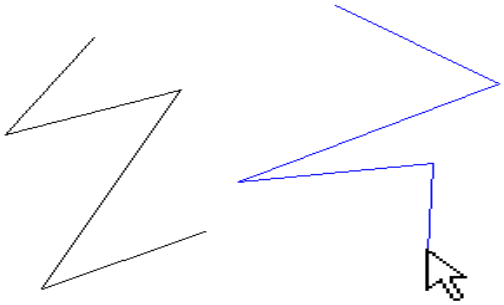
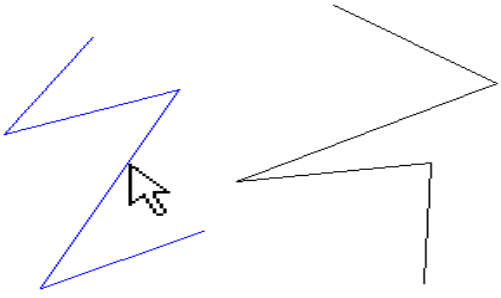


Figure 11. Step 2: Create a new line



***Step 3**: This involves switching the paint mode off and graphically selecting the previous line. We can see that the colour of the first line changes to blue as it is reselected.*

Figure 12. Step 3: Reselecting the first line

---

5 Detailed design and Implementation

5.1 Architecture of the Program

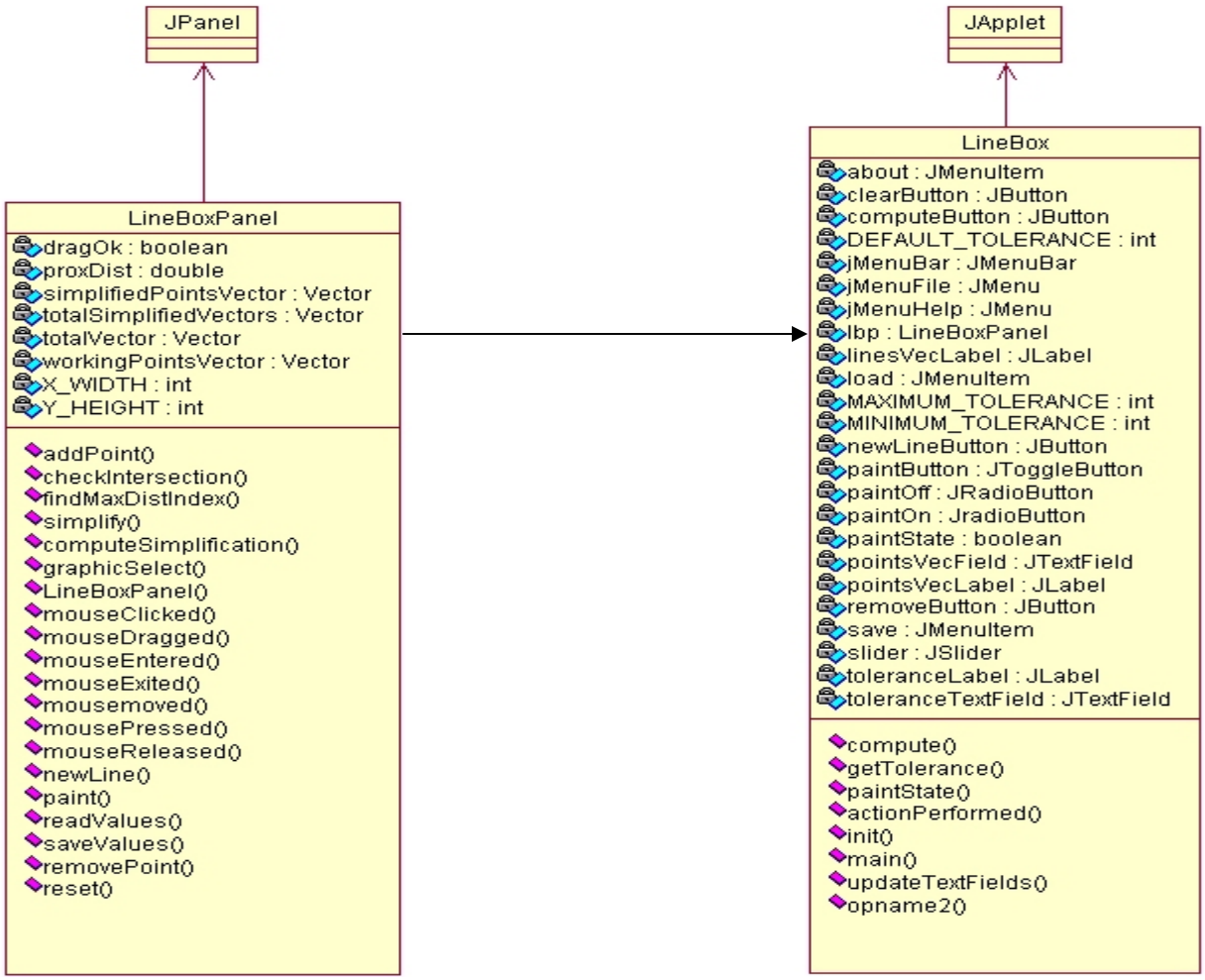


Figure 13. Class Diagram

The class diagram shows the architecture of the system. It's a visual aid as to the workings of the program.

5.2 Extra Functionality

When dealing with polyline simplification, it is common practice to include mouse clicked events which simply create a vertex each time the mouse is clicked. This line simplification application also includes a drag function which gives the user the option of making more complex polylines. For example, a map image can be loaded into the program, the contents of which can then be manually replicated by the user by simply dragging the mouse over the image and creating the desired lines.

5.3 Finishing Touches

The finishing touches were general improvements that polished the functionality and look of the program. The program needed to be able to draw a series of lines within the limits of a *JPanel*. If the mouse cursor travelled outside these boundaries then its location could not be recorded. In order to combat this error, a boundary limit was specified in the *JPanel* to stop the mouse cursor overrunning its limits.

Various colours are used in the program and during its lifespan the program has seen many different changes in the designs. Colours were introduced to enhance the polylines. Layouts were carefully tweaked to control the location of the buttons and Text Fields. The finished version looks drastically different to the original model.

6 Testing/Evaluation

With every new addition, the program went through extensive testing to ensure that the implementation did not create errors in other aspects of the program. It is very important to detect these problems at this stage in order to limit the possible knock-on effect. You can never be guaranteed, however, that all the problems will be detected. It is for this reason that the line simplification program has undergone extensive testing by numerous different people. A number of participants agreed to take part in a session at various times in which all functional aspects of the program were explored and tested. A testcase was developed as a means of testing all functionality whilst also as a means of reference for the participants. It included all functional methods and required the participants to test all aspects and report any irregularities. The testcase proved to be very helpful in detecting errors that may have gone unnoticed. For example, a toggle button was initially created which changed text to “off” as it was selected and “on” as it was deselected. As the length of the first string is longer than the latter, the text was stretching one of the panels to accommodate the change and as a result part of the text was becoming truncated. This was missed in the initial test as it was barely noticeable, but was identified by one of the volunteers while following the testcase. The feedback of this was used to develop, shape and improve the program as the final version approached. It proved to be a very helpful exercise with regards to testing the program, but also for generating ideas to improve it.

7 Conclusions and Future Work

The focus of this program has been to develop a functioning program demonstrating line simplification processes using visualisation and web technology. From the beginning of the project there were certain objectives that were set down as a goal with the intention to complete. One of these objectives that did not get completed was to be able to calculate the boundaries of the convex hull to eliminate all topological inconsistencies. The main reason for this was that as the project continued there were many other issues to take into account that had previously been planned for. With this in mind, the focus changed slightly from developing the convex hull to more immediate matters. When reading in data from a file, the data has to be scaled to proportionally demonstrate it in this particular program. The idea of graphical selection was born mid-project providing the user with much greater control and functionality then was previously the case. As a result the project has become far more rounded and operative then the objectives had initially laid out.

Line simplification is a major factor in developing GIS software packages which have a significant role in today's large scale decision making processes. It is capable of improving the computational process in graphic development. It opens up huge opportunities for further research in cartography and countless other fields. The pace of change is accelerating and the need for line simplification is growing all the time.

8 Acknowledgements

The author would like to thanks Dr. Michela Bertolotto of University College Dublin for her continuous help, guidance and advice throughout the project. Also thanks to Min Zhou for her correspondence. Special thanks to Fintan Fairmichael for his creative suggestions and advice.

9 References

9.1 Web-Sites

http://www.sli.unimelb.edu.au/gisweb/

http://www.cs.sunysb.edu/~algrith/

http://www.dca.fee.unicamp.br/~ting/Publications/

http://geometryalgorithms.com/Archive/

http://www.codeproject.com/cpp/dphull.asp

http://www.geog.ubc.ca/~brian/

http://www.esri.com/

http://www.gisnet.com/notebook/inetsize/maps.html

http://www2.dcs.hull.ac.uk/CISRG/publications/DPs/DP4/DP4.html

http://csdl.computer.org/comp/proceedings/sibgrapi/2003/2032/00/20320060abs.htm

http://www.geovista.psu.edu/sites/geocomp99/Gc99/020/gc_020.htm

9.2 Related Articles

David Douglas & Thomas Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature", The Canadian Cartographer 10(2), 112-122 (1973)

Saalfeld, A. (1999) topologically consistent line simplification with the Douglas-Peucker algorithm. Cartography and GIS, 26 (1).

M. Zhou, M. Bertolotto, Implementation of Progressive Vector Transmission Using a New Data Structure and Modified RDP Algorithm, Proceedings GISRUK 2004, Norwich, UK, April 2004, pp. 69-72.

K. Chang, "Geographical Information Systems", McGraw-Hill, 2002.

Buttenfield, B.P. 2002. “Transmitting Vector Geospatial Data across the Internet”

M. Zhou, M. Bertolotto, Efficient detection of self-intersection in line simplification

P. Longley, D.J. Maguire, M.F. Goodchild, D.W. Rhind (Eds.), "Geographic Information Systems and Science", Wiley, 2002.