Building a real-time model of terrain from contour maps

Peter Gardner

Fourth Year Project Report
Computer Science
School of Informatics
University of Edinburgh
2009

Abstract

I explore various methods for recreating and rendering terrain from contour maps. After a brief look at other methods such as linking contour lines with polygons I explore the generation of height maps in detail. Then I perform image quality and performance tests on 5 different algorithms comparing the run time and output images. Finally I consider different ways to display the height map data in 3D using OpenGL.

Acknowledgements

Thanks to Edinburgh University Orienteering Club, Edinburgh Southern Orienteering Club and Scott Fraser of Scotia Maps for providing all the OCAD map files used, Bob Fisher for supervising my work and Simon Gardner for doing all the boring data entry.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Peter Gardner*)

Table of Contents

Chapter 1

Introduction

1.1 Description of the Problem

1.1.1 Contours and Maps

Contour lines are widely used to represent the height of underlying land on maps while leaving space on a two dimensional representation for the other features. They are the most commonly used method of representing height in maps in the western world today. The International Orienteering Federation specifies a contour interval of 5 meters for foot competition maps or 2.5 meters where extra detail is needed(4). However contours are a lossy representation of the shape of the ground as they only display horizontal slices through at defined heights. To create a three dimensional representation of terrain from a contour map it is necessary to attempt to reproduce this lost data somehow.

1.1.2 Motivation

There are a number of reasons to try to create a computer model of an orienteering map. GPS technology can be used to show the progress of competitors during an event and a 3D display can make the map easier to understand for people not otherwise familiar with the symbols. After an event, those involved could use the Internet to share route choices[1] and an advanced graphical display would make post-race performance analysis more thorough. In the run up to a race, competitors could use a 3D map to familiarise themselves with an area they are barred from entering by the [2]. There is also the possibility of mappers using the created model to check their map against either photographs or the real world in order to produce better maps.

1.2 Objectives Achieved

I analysed 3 techniques for creating height maps from contour maps for their quality of visual output and execution time in an attempt to find a balance between the two. I

[1]Routegadget: `http://www.routegadget.net`

[2]Catching Features: `http://www.catchingfeatures.com`

explored ways of optimising two of the techniques to increase their runtime efficiency without compromising the output. I also investigated methods of displaying the height maps in 3D, both with and without level of detail systems and discussed their suitability for displaying real-time models of terrain on a modern home computer.

1.3 Pointers into thesis

The terrain drawing library provides a number of avenues for improvement. In its current form it supports loading multiple height maps at different resolutions and scales to generate a composite tree. Further work could be done into detecting areas of high complexity on the input map file (or on the output image) and creating higher scale output images for just those areas. This would allow for other mapped features such as pits and small knolls to be added into the terrain. With some small modifications the library could load these sub-maps on the fly or even generate them procedurally to save memory and storage space. Another way to improve the visual quality of the output would be to apply different textures to the terrain based on other details read from the map (such as undergrowth type). Height map generation is also very suited to parallel processing, either on a single, multi-processor computer or a distributed system.

not really

From a purely graphical point of view, the current library suffers from popping. This is where, as the level of detail changes, bits of terrain appear to pop into their new positions. There are various ways of reducing the effect of this and all could be added to the existing code. The flexibility of the algorithm would also allow real time modification of the level of detail used in order to keep frame rate fairly constant.

On the generating of height maps there are a number of other avenues to explore. It would be possible to use Bézier patches to create smooth surfaces, dealing with the issue all the algorithms I implemented have with drawing peaks. There is also the issue of finding heights for the contours as explored in Appendix D.

Chapter 2

Background

2.1 Orienteering and Maps

For a description of Orienteering it is probably best to quote the British Orienteering website.

> "Orienteering is a challenging outdoor adventure sport that exercises both the mind and the body. The aim is to navigate in sequence between control points marked on a unique orienteering map and decide the best route to complete the course in the quickest time. It does not matter how young, old or fit you are, as you can run, walk or jog the course and progress at your own pace."

The format of an Orienteering map is dictated by the International Orienteering Federation(4). Usually for so called conventional races it will look something like the examples in Appendix C and for the purposes of this project I will assume that all maps are of a similar appearance. In order to fit as much detail in as possible, contour heights are omitted and deciphering the actual structure of the terrain is left as an exercise for the competitor.

2.2 OCAD and it's use in Orienteering

OCAD[1] is a mapping tool developed by OCAD AG and is dominant in Orienteering at the moment. Other, more general tools such as Adobe's Illustrator have been gaining in popularity recently due to pricing issues but the majority of maps in circulation currently are OCAD maps and this is unlikely to change so long as the developers continue to support the sport.

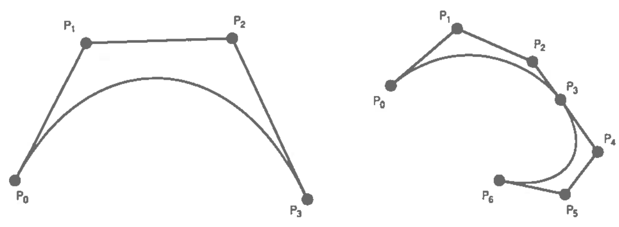
2.2.1 File Format

For a full discussion of the OCAD file format see Appendix A.

[1]OCAD Website: `http://www.ocad.com`

OCAD stores contours within its proprietary file structure[2] as Bézier splines, a chain of cubic Bézier curves (the equation for which is shown below) where P_3 of the first curve is identical to P_0 of the second one and P_2 from curve 1 is collinear with P_1 of the second, repeated along the chain. This produces a curve as shown in figure 2.1.

$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3, t \in [0,1]$$



(a) Cubic Bézier curve (b) 2 part Bézier spline

Figure 2.1: Cubic Bézier curve and spline

An OCAD map file stores all the symbol definitions within the file itself which can sometimes mean that contours are defined with a different symbol ID from one file to another. However finding the correct symbol is usually as trivial as scanning for a string.

Orienteering maps traditionally do not have heights associated with contour lines to reduce clutter, and even if a particular map does, the OCAD file format does not associate the height label with the contour line in any useful way. While a trained human being can easily determine what is up and down by looking at nearby features such as cliffs, rivers and the general shape of the land, the number of variables involved is so large that automatically determining what is up and down is beyond the scope of this project. Even if a system to do so were developed it would need heavy supervision and specific algorithms for different areas of the world. Instead it is far more practical to have a human assign values to each, or a few contours in the file before processing. A more complete discussion of the problems involved can be found in Appendix D.

Finally, an orienteering map is not always completely accurate. They are closer to an artist's representation of the real world, albeit one which is accurate enough to allow people to navigate around the area in question. As a result it is impossible to produce a one-to-one copy of the real world from such a map, however I would consider any model that an experienced orienteer can navigate using the map it was generated from a successful one.

2.3 Previous work

Gousie and Franklin(3) discussed trying to calculate intermediary contours in an attempt to rebuild a dense surface as well as drawing gradient lines from local maxima to

[2]OCAD file format description: `http://www.ocad.com/docs/OCAD9Format.txt`

minima then interpolating elevations along the gradient lines using the contour height values. Dakowicz and Gold(1) analysed various methods of generating the missing ridges and peaks from contour data and the various problems and artifacts that result.

Chapter 3

Dense surface reconstruction

In order to determine how the 3D representation will be produced we need to look at the potential uses of a solution and the hardware and software limitations they impose. Most of those discussed above require some degree of user interaction which means real time display and navigation are desirable. While modern home computers are approaching the point where they can perform real time ray tracing, most are optimised for producing polygons. While voxel terrain rendering (effectively a low detail, software ray tracing technique) gained popularity in the mid 90s, the rapid improvements in graphics hardware made it obsolete. With the decision to use polygons there are a number of techniques that can be explored.

3.1 Analysis of high level techniques

3.1.1 Triangle strips

One seemingly obvious way to produce a 3D image of a contour map is to move along adjacent contours connecting them with triangles (see figure 3.1) then storing this data and displaying it as needed. This can be done reasonably quickly and the size of data created will be small. However there are some issues. Deciding which contours are to be connected is non-trivial, especially in regions where there are multiple candidates. On top of this the algorithm must be careful to avoid artifacts where two adjacent triangle strips do not line up, causing either gaps in the terrain (figure 4.3) or overlapping polygons.

This method also requires an awareness of geographical features in order to produce the best results. For an example of this see figure 3.2. In this case the contour lines represent a spur. A naive implementation might produce the output seen in 3.2(a) resulting in what may be a very prominent feature in reality becoming a flat piece of hillside in the model. A better way to handle this would be to insert an extra contour line down the middle of the spur to create a ridge as seen in 3.2(b).

This method also produces static output. The terrain must be rendered at a fixed level of detail no matter how complex or far away it is. Sections that are out of sight can be omitted but anything visible must be drawn at the complexity that it was initially computed.

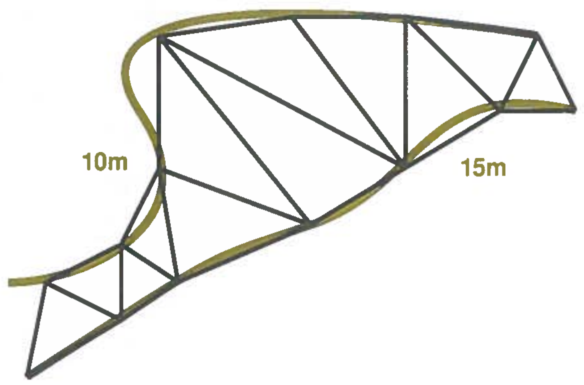
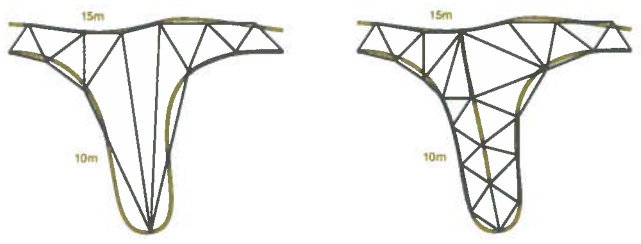


Figure 3.1: Triangle strip between two contours



(a) Undesirable behaviour (b) Possible, geographically aware behaviour

Figure 3.2: Potential triangulation of a spur

3.1.2 Height maps

A more flexible method of creating the 3D scene is to use an intermediate height map from the contour lines. This can then be used to produce either a variable or fixed level of detail render. The main disadvantage of this method is it will result in a large volume of data being created when it is not strictly necessary. For example a relatively flat, feature free area on a height map takes up the same space as a complex one. This can be mitigated somewhat by using image compression algorithms such as PNG.

As you can see from figure 3.3 even a height map may not give an exact representation of the terrain because it is limited by the resolution of the map. In this case the top of the hill is slightly off position from where the vector map says it should be. The easiest way to avoid this is to use a higher resolution height map but there are practical limits on the size mostly imposed by the memory capacity of modern computers.

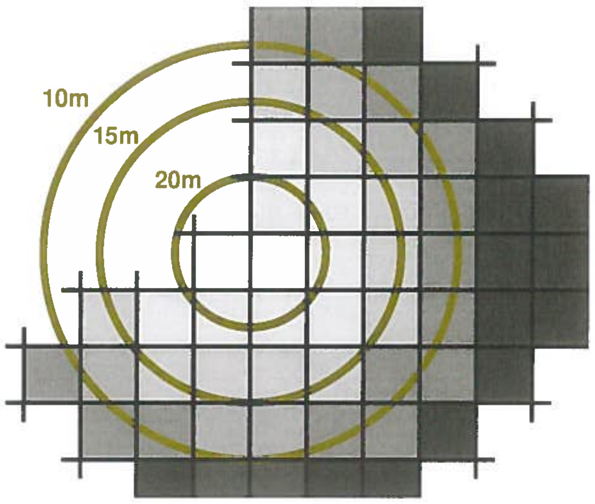


Figure 3.3: A possible height map of some concentric contour lines

3.1.3 Conclusion

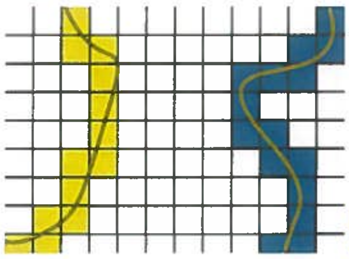
For the purposes of this project I will be concentrating on height map based methods. This is mostly due to their ability to produce varying levels of detail, but also because the output is easily comparable by subtracting one image from the other. While the correctness can only really be compared subjectively some techniques will give better results than others. It is also trivial to compare height maps by displaying the pixels that differ from one to the other, however this doesn't give an objective measure of correctness, only the difference between the compared techniques.

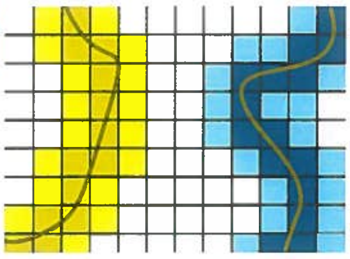
3.2 Detailed analysis of Height map generation methods

In this section I will discuss various methods of creating height maps from contour data, including the pros and cons of each.

3.2.1 Pixel expansion

In the pixel expansion method firstly the contours themselves are drawn onto a grid using the standard Bézier curve algorithm then the image is iterated over until all pixels are done. If a pixel is set then all its neighbours are checked, if they are currently unset then they are filled with the value of the pixel and the function continues. Once the finish conditions are completed the algorithm loops over the grid once more to calculate the resulting height and outputs a height map. Figure 3.4 shows a possible execution of this method over the first 2 steps. In 3.4(c) there are 3 pixels in the middle of different colour which are adjacent.


(a) Initial drawing of the contour lines


(b) Map after first expansion

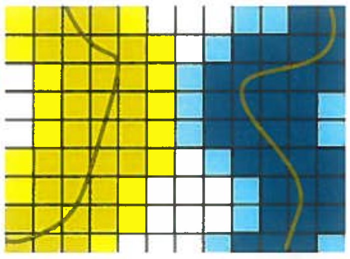

(c) Map after second expansion

Figure 3.4: Pixel expansion from contour lines

3.2.1.1 Closest neighbour

Finding the closest neighbour is the simplest form of the pixel expansion method. Expansion stops when a pixel of any other value is reached and the value written to the height map is the height of the contour we expanded from. This will produce a very stepped image and is therefore not a very good solution. To some extent this can be solved by running a blur function on the resultant map but by doing that data is lost.

3.2.1.2 Closest two

Similar to the closest neighbour method, here each spot on the grid can hold multiple entries and both the contour value and the loop number are stored. Expansion stops when it reaches any of the initial contour line pixels. To create the height map the lowest two distance values are taken and a weighted mean is used on the two heights they represent. Again this will produce a stepped output but instead of sharp corners the steps will be smoother.

3.2.2 8 way intersection

8 way intersection computes the height of each pixel by taking an average of the heights of the closest contours in each of the 8 compass directions weighted by distance. This is illustrated in figure 3.5 for a general case. If the weighting is done correctly the resultant value will be somewhere between the minimum and maximum of the 8 contours that were found.

3.2.2.1 Using precise intersection

Perhaps the most obvious way to find the lines of intersection is to find any intersections between the compass direction vectors and the contour cubics then check if the results lie within the bounds specified by the Bézier curve and the image. Because the shape of Bézier curves are invariant under Euclidean translations this can be done by rotating and translating the curve and the vector so that the vector lies along the origin. Once that is done finding the intersections is simply a case of converting the control values to a power basis by expansion and finding the roots.

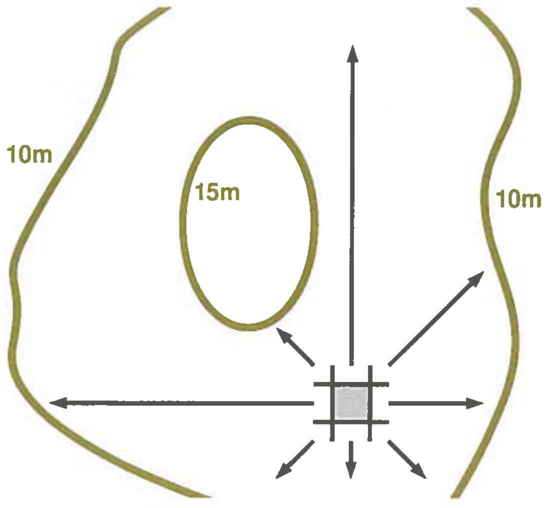


Figure 3.5: Finding the closest contour in each of the 8 compass directions

$$y(t) = At^3 + Bt^2 + Ct + d$$

Substituting in the following equations, where y_0, y_1, y_2 and y_3 are the Y component of P_0, P_1, P_2 and P_3 respectively.

$$A = -y_0 + 3y_1 - 3y_2 + y_3$$

$$B = 3y_0 - 6y_1 + 3y_2$$
$$C = -3y_0 + 3y_1$$
$$D = y_0$$

This will supply a list of all the intersections and the point along the curve they lie on. Selecting the closest can be done by placing all the intersections of all the contours into a list and sorting by distance.

A height map created using this method is reasonably accurate and smooth but it has limitations. In figure 3.6 all the pixels inside the 15 meter contour will be the same height because all the vectors will return the same height as their first hit. Calculating roots is also very computationally expensive. For example creating a 512x512 pixel image from a map with 100 contours, each with 10 Bézier segments requires over 2,000,000,000 non-trivial calculations (see section 3.2.4) and the scaling is not linear.

Fortunately there is a relatively simple optimisation that can be made. Instead of calculating 8 vector/curve intersections for each pixel it is possible to pre-calculate all the intersections needed and refer to these each time a pixel is analysed. All intersections along the vertical, horizontal and both sets of 45 degree lines are calculated and stored in arrays. When a pixel colour is calculated the appropriate 4 sets of intersections are selected and iterated through until the correct two values are found. Using the example values from above, the number of intersection calculations is reduced to just over 2,000,000.

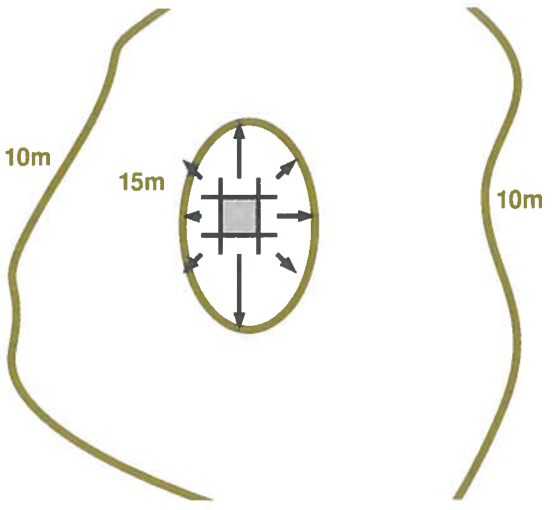


Figure 3.6: The 8 way expansion method can not create peaks

3.2.2.2 Approximation by pixel

A lot of the heavy calculation of the above method can be avoided by working at a pixel level rather than one of cubics and vectors. If the contour lines are first drawn onto a grid this can be used to come to a reasonable approximation of the height map created by precise intersection. When calculating the value of a pixel, the algorithm iterates over the grid of drawn contours in each of the eight directions until it reaches a filled square or the edge of the grid. Once all eight values are returned the height is then calculated in the same way as before.

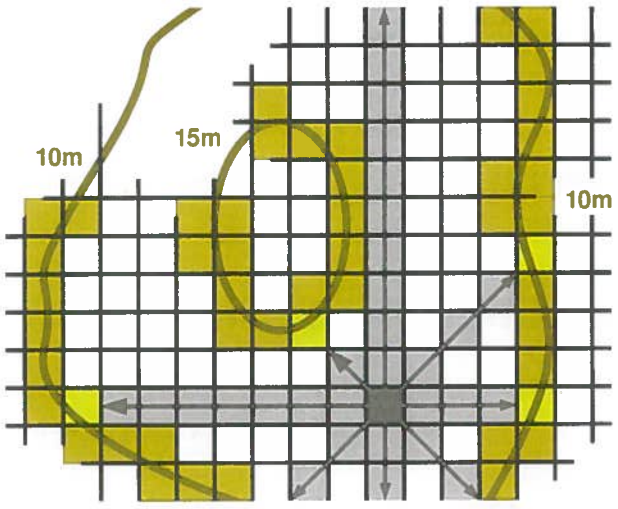


Figure 3.7: 8 way expansion using pixel intersection

While this solves the speed issues presented by the above method, it does nothing to help the other problems such as creating peaks and adds another issue on top. If the algorithm that draws the contours onto the grid leaves gaps or one of the diagonal expansion lines crosses a contour line at a diagonal it is possible that it may not detect a pixel. How this affects the overall result depends on the resolution of the height map and the positioning of the contours themselves.

This method can be optimised by using the same scan line method as the precise calculations, however the algorithm is even simpler. Each scan line is followed in reverse, incrementing a distance value and setting the current height as the height of that pixel until a contour line is found. At this point the height value is updated to the height of the newly found contour and the distance is reset to zero. Doing this for all 8 directions will produce 8 values per pixel which can then be averaged as usual to create the composite value.

3.2.3 Unexplored methods

One method of producing height maps that I have not explored is to create new Bézier curves along the horizontal or vertical scan lines, using points of intersection with contours as the base points of the curve and extrapolating control points, either from the previous intersections or using another method. On the face of things this would appear to solve the issue of peaks. However, as can be seen from figure 3.8 where the control points are set to be roughly 1/3 of the way between the base points of the curve, in line with the previous points, there is substantial terracing. It may be possible in future work to attempt to detect peaks in one of the other methods and apply this technique just for that area of the map.

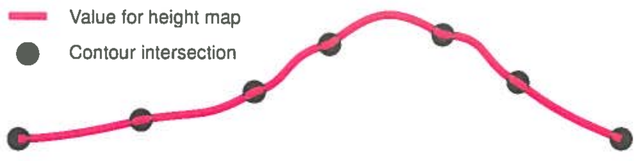


Figure 3.8: A scan line with Bézier interpolation of contour intersections

3.2.4 Computational complexity

Table 3.1: Key for table 3.2

Symbol	Meaning	Example 1	Example 2	Example 3
x	Size of the image (X and Y dimensions)	1024	128	2048
n	Number of contours in the map	1000	1000	200
s	Number of segments per contour	10	10	10

All calculations in table 3.2 are assuming the worst case and no further optimisations than discussed above. For the equation intersections it is assumed that each line intersects every segment of every contour, making the sorting of intersections non-trivial. For the expansion algorithms it is assumed that the number of iterations is equal to the number of 45 degree diagonal scan lines in the image (the absolute worst case being the contour drawing phase leaves a single pixel in one corner of the image). However these values do not tell the full story, as the actual intersection calculations are far more complex in both the precise equation methods than in any of the other three. For Pseudo code for all 5 techniques see Appendix E.

Table 3.2: Computational complexity of the proposed height map generation algorithms

Algorithm	Complexity
Closest neighbour	$f(x,n,s) = O(x^2)$
Closest two	$f(x,n,s) = O(x^2)$
Precise intersection by equation	$f(x,n,s) = O(x^2.n.s)$
Precise intersection by equation (optimised)	$f(x,n,s) = O(x.n.s)$
Precise intersection by pixel	$f(x,n,s) = O(x^3)$
Precise intersection by pixel (optimised)	$f(x,n,s) = O(8x)$

Table 3.3: Example numbers for the proposed height map generation algorithms

Algorithm	Example 1	Example 2	Example 3
Closest neighbour	1048576	16384	4194304
Closest two	1048576	16384	4194304
Precise intersection by equation	10485760000	163840000	8388608000
Precise intersection by equation (optimised)	10240000	1280000	4096000
Precise intersection by pixel	1073741824	2097152	8589934592
Precise intersection by pixel (optimised)	8192	1024	16384

Based on these numbers, the optimised Precise intersection by pixel method should be by far the fastest.

3.2.5 Other optimisations and speed increases

One very simple optimisation that could be made to the Bézier intersection algorithm above is a box test. The line of the curve always lies within a box created by joining the 4 points used to create the curve. this means that instead of doing a full intersection for each contour segment, those out of range can be discarded by intersecting those 4 lines with the direction vector. The intersection of two lines is far more computationally efficient than a full curve intersection so this would potentially save a lot of time in real world applications (although not in the worst case situation).

Dividing the contours on the map into grid squares would also provide some advantages. In figure 3.9 the black pixel would check for intersections in its own grid square, satisfying 2 of the directions. When the third returned nothing, it would check the next square and find an intersection. This way only a small subset of the contours need be checked for each pixel. The challenge would be finding a grid size where the overhead of sorting each contour segment into a grid does not outweigh the benefits.

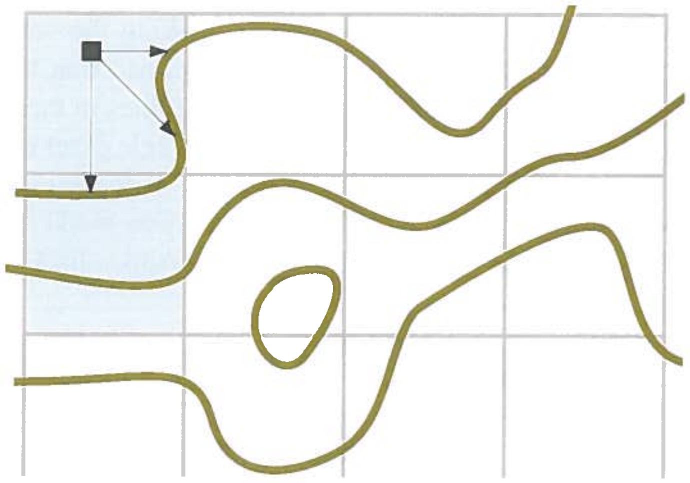


Figure 3.9: A contour map divided into a grid

A final area that could be considered is parallel processing. In almost every method each pixel is calculated in isolation meaning it would be very easy to assign chunks of the image to discrete processing units for computation. Again, the overhead cost must be considered but with modern personal computers increasingly having more than one core on their CPU this is something that could provide a substantial speed increase.

Chapter 4

3D Render & User Interface

4.1 3D Rendering

4.1.1 Basic display

The simplest way to display a height map in 3 dimensions is to plot a point in space for each pixel of the height map. However, even modern computers have issues with displaying a 1024x1024 image in this way and there is no real sense of viewing a 3D image as all that can be seen is a pure white pixel. Some of this can be solved by stepping over the image in increments of greater than 1. Figure 4.1 shows this technique used on a partial height map of Holyrood Park.

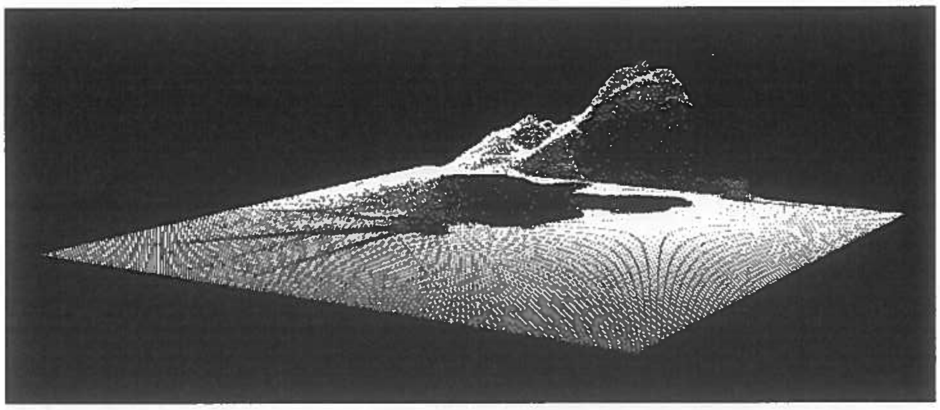


Figure 4.1: 3D point representation from a 1024x1024 source with 10px steps

Obviously this leaves a lot to be desired. The next logical step is to join the dots with shaded and lit polygons to create a triangle strip as seen in figure 4.2, in this case with a 9 pixel step between values. This produces a surface rather than discrete points which can then be coloured in a number of different way. Simply overlaying the height map as a texture projected from above is one option. An emboss texture can be applied as described in section 4.1.3 or OpenGL lighting can be used to shade the vertices.

A full discussion of OpenGL lighting is beyond the scope of this report. Simply, it works by calculating the angle between a vertex or face normal and shading as appropriate. Using surface normals gives faceted shading whereas when vertex normals are used, OpenGL blends the shading between each vertex of a face to create smooth shading. To calculate a face normal, the cross product of two of the face's edge vectors is taken then normalised. A vertex normal is just the mean of all the face normals that use said vertex.

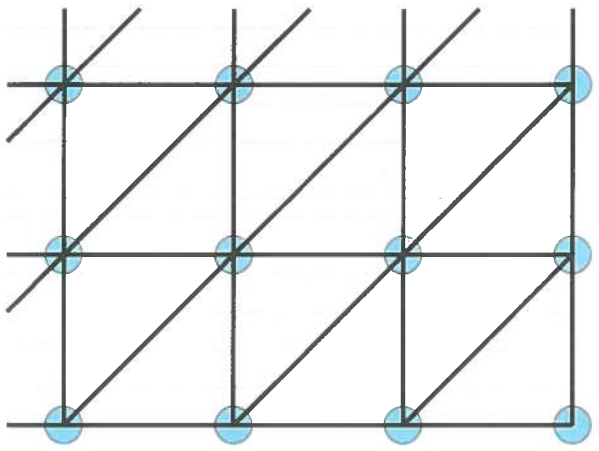
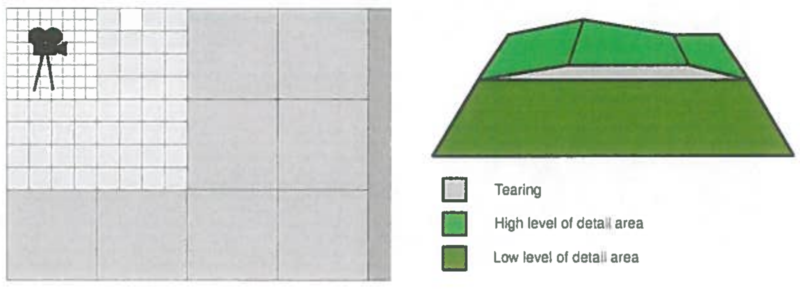


Figure 4.2: Joining height map points to create a triangle strip

It is possible to implement a very simple level of detail system in the above described method. Areas closer to the camera can be drawn with a low pixel step and those further away with a higher step (figure 4.3(a)). However this can cause tearing problems at the edges of a detail level as shown in figure 4.3(b).



(a) Level of detail (b) Tearing

Figure 4.3: Level of detail with simple polygon strips and associated tearing issues

4.1.2 Advanced display

The ROAM algorithm(2) describes a method of drawing terrain that updates relative to camera position and angle, as well as the complexity of the terrain to be rendered, all in real time. It achieves this by splitting the area to be rendered into pairs of triangles that form a square when placed base to base. The triangles are split down the middle until a desired level of accuracy is achieved then the whole structure is rendered. Tearing is avoided by recursing into the triangle structure when the base neighbour of a split triangle is courser than the resultant pair. This recursion forces a split on all the triangles it encounters until a base pair is found with the same level of detail.

For the purposes of this project I have chosen to use an existing library which is based very loosely on the ROAM algorithm(5). Instead of using triangles this method divides squares into 4 each time more detail is required. It expands on ROAM by actually storing the height values inside the quadtree to save memory and enabling and disabling quads based on camera position. This means that the height map data can be unloaded once it has been fully processed. It also allows extra detail to be added, either from higher scale height maps or generated procedurally, to the base height map. Initially I had been planning to implement the methods discussed myself, however with the source code provided with the article being freely available for use it seemed rather foolish so instead I have integrated the provided libraries into my work.

For examples of this library at work see figure 4.4. Note that in the wireframe view areas of higher detail use more polygons, and areas of similar detail but which are further away from the camera use fewer polygons. This is an example of how this method differs from ROAM. The maximum level of terrain detail is pre-computed when the height map is loaded but how much detail is shown is dependant on the range from the camera.

4.1.3 Texturing

In order to emphasise the terrain features on the 3D display a simple emboss filter is applied to a copy of the height map. When used as a texture this gives the illusion of lighting in the scene without any of the complicated normal calculations that OpenGL lighting requires. To apply the emboss filter, each pixel is run through the filter matrix below once. The resulting value is then multiplied by $factor$, added to $bias$ and the mean of the red, green and blue values is taken. To make a green texture (representative of grass) the calculated value is assigned to a RGB triplet with the red and blue values being halved.

$$\begin{pmatrix} 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 0.0 & -1.0 \\ 1.0 & 1.0 & 0.0 & -1.0 & -1.0 \\ 1.0 & 0.0 & -1.0 & -1.0 & -1.0 \\ 0.0 & -1.0 & -1.0 & -1.0 & -1.0 \end{pmatrix}$$

$$factor = 1.0, bias = 128.0$$

The matrix and values above represent an emboss filter. The intensity of the filter is determined by the size of the matrix used, so a 3x3 matrix would produce a less pronounced 3D effect and a 10x10 matrix would make it more pronounced.

To add a grass effect, a tiled image of grass is loaded and combined with the generated texture with a bias of 0.25 to the grass and 0.75 to the emboss map. Examples of all 3 stages on the Holyrood Park map can be seen in figure 4.5. However care must be take not to dilute the lighting effect a little too much to the point where it no longer highlights the shape of the terrain. Figure 4.5 shows the 3 stages of creating a texture map.

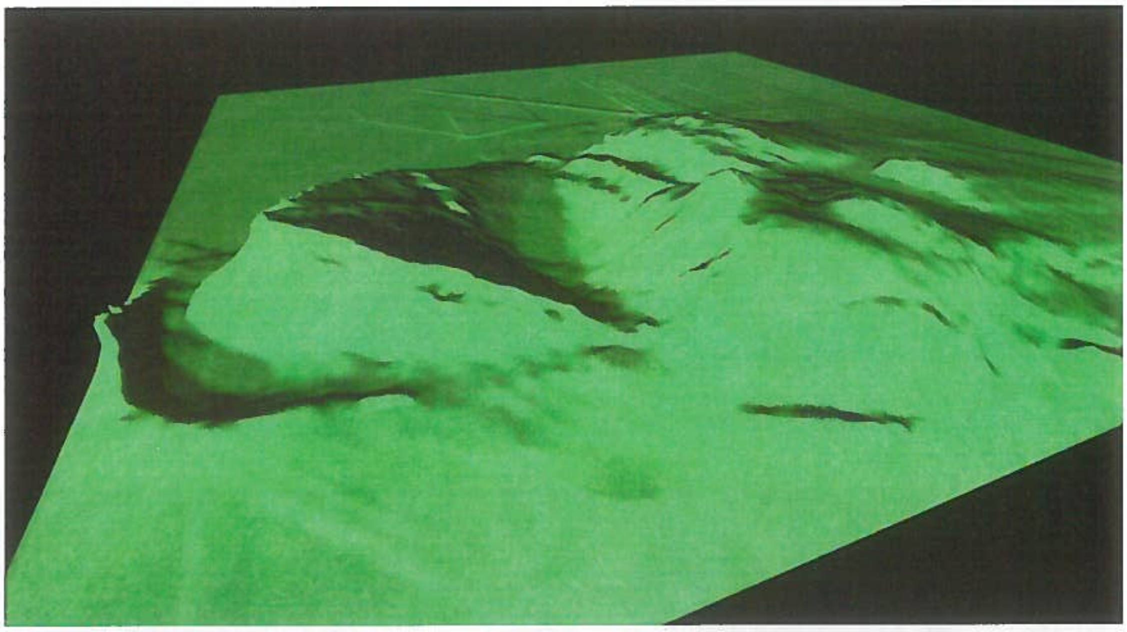
4.2 User Interface

It quickly became obvious during development that having a user assign all the height values individually for a map would be very time consuming. Add to that the fact that contours are often ordered in the map file in an unintuitive way that relates more to the process the mapper used than their actual position and the need for a graphical interface is quite clear. The interface was built using Qt[1] due to it being available for free under an LGPL and for cross platform development. There are 3 discrete components combined under an MDI interface for ease of comparison, a contour view, a height map view and a 3D view.

The contour view (figure 4.6(a)) displays the contours from a loaded OCAD file with the contour ID and height (read from a separate file) along side. The main purpose of this view is to allow the user to input contour height values. This is done by dragging the mouse across the map to draw a vector. On releasing the mouse the vector is intersected with all the contours in the map and the intersections are sorted in order of distance. The height value of the first intersection is taken as the base and each subsequent intersection in the ordered list has its height value set to $base + (5.position)$. This allows the height values in a map to be rapidly assigned.

The height map view (left side of figure 4.6(b)) is mostly for the user to check accuracy but is also useful for comparing the output of two height map generating algorithms. It can be useful when compared with the contour view for finding incorrectly assigned height values. The 3D view (right side of figure 4.6(b)) allows basic navigation of the 3D world using the mouse to look and move around.

[1]Qt: `http://www.qtsoftware.com`


(a) Textured

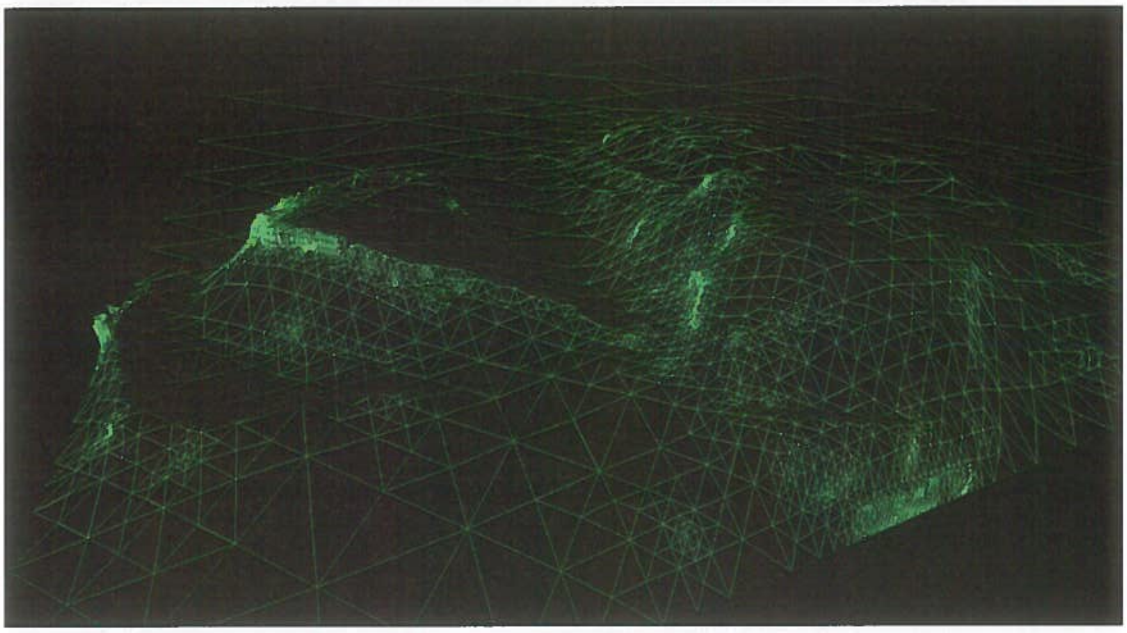
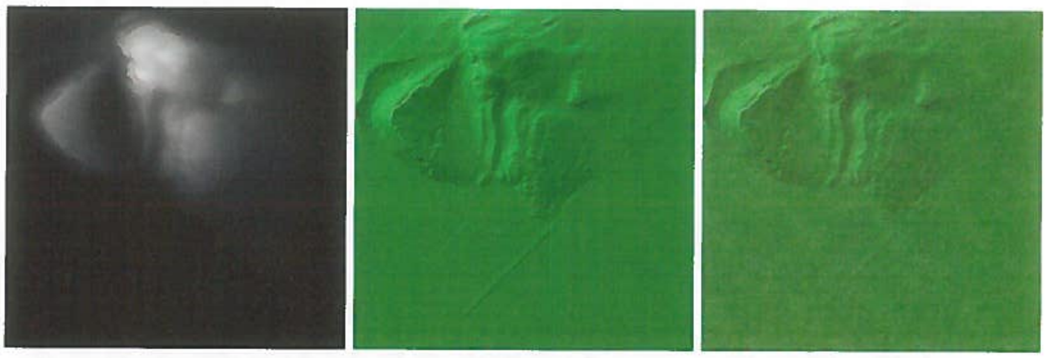
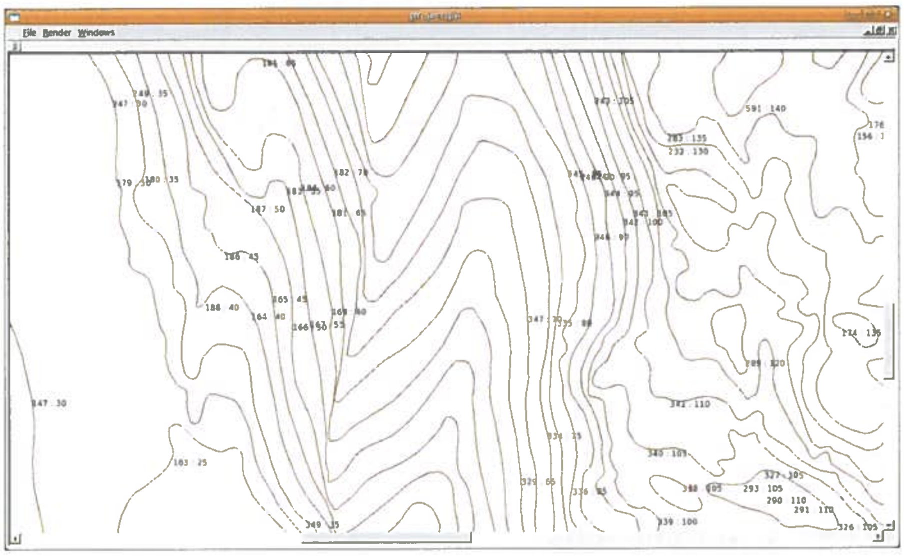

(b) Wireframe

Figure 4.4: Advanced level of detail rendering

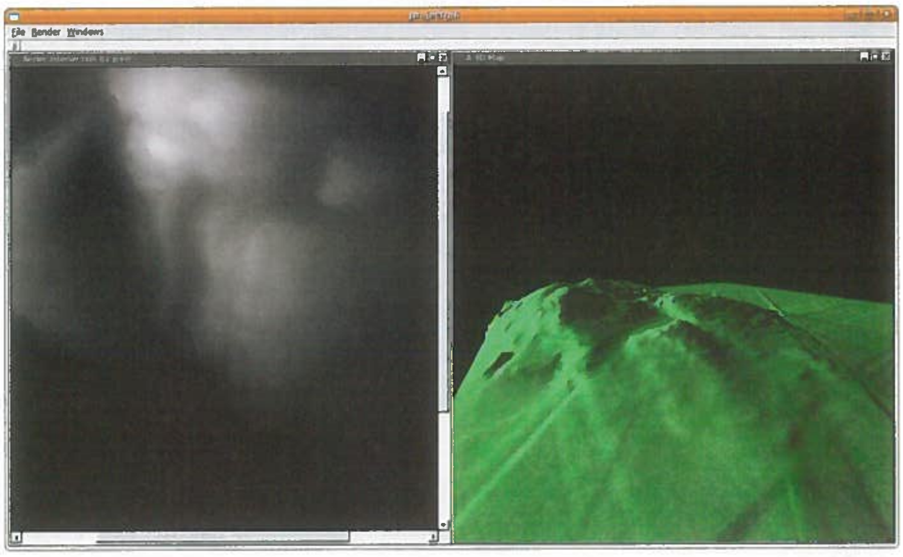


(a) Height map (b) Emboss map (c) Texture map

Figure 4.5: Generating textures from a height map



(a) Contour view



(b) Height map and 3D views

Figure 4.6: User Interface

Chapter 5

Evaluation

5.1 Performance evaluation

Tables 5.1 and 5.2 show the execution times of all 5 implemented algorithms on a 512x512 image. The tests were performed on an Intel Core 2 Duo running at 3GHz with 4GB of RAM on the Ubuntu Linux distribution.

5.1.1 Synthetic data

All 3 synthetic contour patterns are made up of 50 contour lines at 5m intervals. The Cone and Ripple patterns have 4 Bézier segments per line and the slope has 1.

Table 5.1: Execution times for synthetic data sets

Algorithm	Execution time in seconds		
	Cone	Ripples	Slope
Closest neighbour	3	3	1
Precise intersection by equation	482	412	147
Precise intersection by equation (optimised)	No result	No result	1
Precise intersection by pixel	4	5	4
Precise intersection by pixel (optimised)	3	3	1

The unoptimised equation methods failed to draw the cone and ripples correctly, instead producing an odd, shaded square and the optimised method would not complete.

5.1.2 Real world data

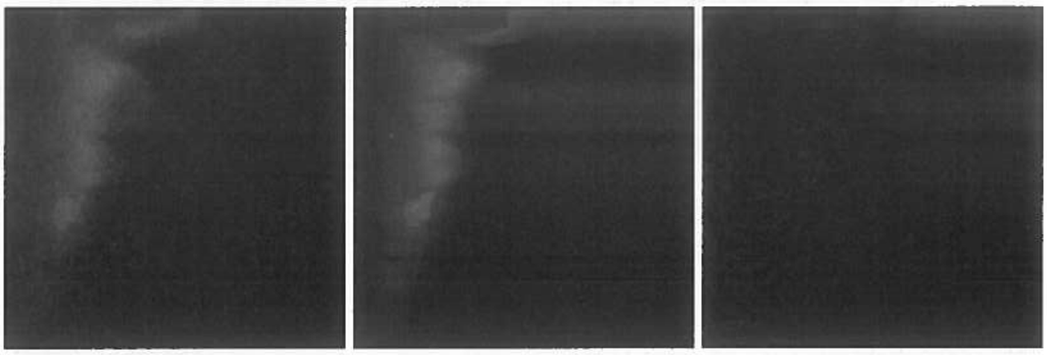
As well as demonstrating the algorithms with landmarks local to Edinburgh, these three real world maps provide a decent spectrum of map styles. Holyrood has 781 contour lines, Corstorphine has 107 and Blackford has 318. Holyrood is spread out evenly over a large area of the image while Corstorphine is long and thin, and Blackford is L-shaped. This means they provide radically different cross-sections for intersection lines.

Table 5.2: Execution times for real world data sets

Algorithm	Execution time in seconds		
	Holyrood	Blackford	Corstorphine
Closest neighbour	73	20	20
Precise intersection by equation	11264	3584	3072
Precise intersection by equation (optimised)	11	4	3
Precise intersection by pixel	90	27	22
Precise intersection by pixel (optimised)	90	21	21

5.2 Image comparison

Image comparisons were done by loading both output files into the same window in The GIMP and setting the upper layer's overlay type to "Difference". All comparisons were done at 512x512 pixels using the Corstorphine map due to it's low number of contours. Both optimised methods produced identical height maps to the associated unoptimised algorithm. This means the optimisations were well designed from an image quality point of view.



(a) Corstorphine by equation (b) Corstorphine by expansion (c) Difference

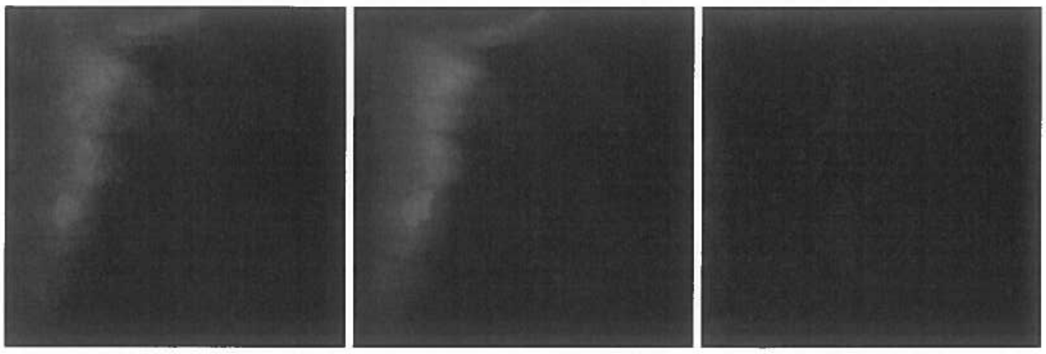
Figure 5.1: Differences in equation and expansion output

Figures 5.1, 5.2 and 5.3 show the results of the difference test on all 3 combinations of algorithm. The images don't tell the full story as the stepping seen in the expansion method only registers as a very small difference due to each contour being 5m apart. It is obvious that the expansion method has far worse edge artifacts than the other two methods, and that there are a number of large gaps in the "true" intersection method's output.

5.3 Subjective evaluation

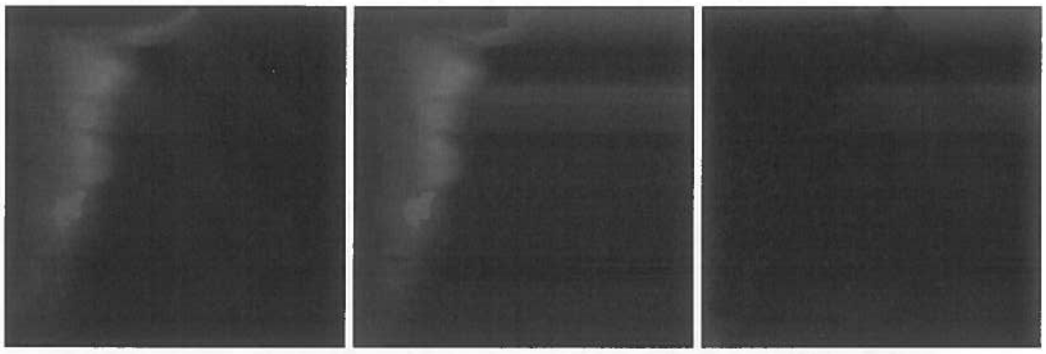
5.3.1 Height maps

The closest neighbour method, as expected, gives a very stepped image, the other two methods (assuming the optimised algorithms produce similar enough results to



(a) Corstorphine by equation (b) Corstorphine by pixel (c) Difference

Figure 5.2: Differences in equation and pixel output



(a) Corstorphine by pixel (b) Corstorphine by expansion (c) Difference

Figure 5.3: Differences in pixel and expansion output

their unoptimised version to ignore) create what appears to be a very smooth height map. However, on closer inspection there are still steps in the slopes which become increasingly obvious when an emboss filter is applied. This could be fixed to some extent by applying a blur to the image but that would result in a loss of detail. I would consider the stepping to be acceptable though, as it only really shows on shallow slopes. There are a lot of visual artifacts around the edges of all the height maps. These are caused by the last contour not being at height zero.

Interestingly the precise intersection method produced less accurate results than the one that used pixel level intersection. After some investigation I concluded that this was due to gaps in the contour file causing some vectors to miss contours entirely. When the mapper is creating the file they often make one contour out of several contour lines and the ends may not quite meet. On the pixel intersection method these are mostly eliminated by the process of drawing the contours onto the initial grid.

5.3.2 3D terrain

The maps I used are on the small side for orienteering maps, only covering a couple of square kilometres at the most. Neither of the 3D rendering methods I looked at could really provide an adequate level of detail for someone to move around at foot level and not notice the polygons. However, from a distance features were easily identifiable as their real world counterparts. The advanced method suffered from difficulties rendering vertical areas of a height map, creating one polygon per height map pixel along the vertical edge. This can produce a severe FPS hit, especially as many of the height map generating techniques produce artifacts in the empty areas of the map. These usually manifest in the 3D view as a straight line with vertical edges (figure 5.4). Performance when these artifacts are off screen is smooth.

The level of detail issues could be resolved by creating larger scale height maps of higher detail areas and adding them to the quadtree at load time, either detecting the areas manually or automatically. If these two problems were rectified I would be fully satisfied with the performance and image quality of the advanced 3D method.

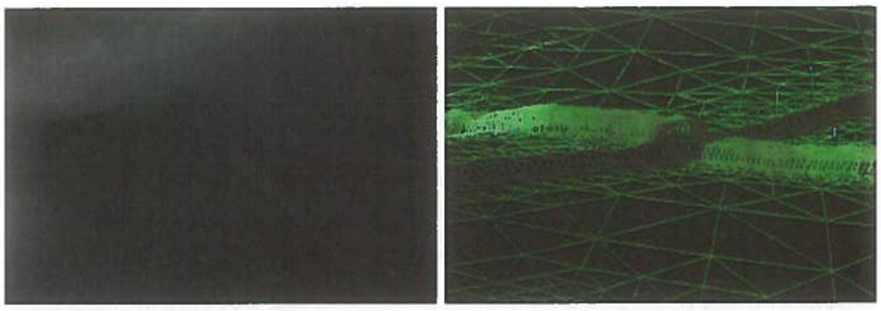

(a) Height map (b) 3D view

Figure 5.4: Artifacts in the height map of Corstorphine Hill and their effect on the LoD view

Chapter 6

Conclusions

Out of all the dense surface reconstruction methods analysed, the optimised pixel intersection method provided both the best image quality and a reasonable speed. Unfortunately it still suffers from artifacts around the edges of the map where the last contour is not at a height of 0m. These slow down the 3D rendering algorithm and reduce immersion. For the 3D display, neither of the methods looked at are really adequate on their own. Even a height map of 2048x2048 pixels does not contain enough data to make a realistic looking landscape when viewed from ground level, and using higher resolution maps is impractical for reasons of both memory space and computational time. In order to improve the quality of the 3D view more work is needed in the height map generation stage to identify areas of high detail and remap them at a larger scale.

The failure of various algorithms when presented with synthetic data is cause for concern, but is most likely faulty implementation rather than an error of the actual method itself. Performance was roughly around that expected by analysing the Pseudo code. The performance of the optimised equation based intersection algorithm was very good, and if the image quality issues could be resolved this would be a very good candidate for future use as the edge artifacts were generally less severe.

The height maps output from the synthetic data were almost all defective. Every algorithm had at least one data set where there were large gaps in the height map result. It appears that the surface reconstruction methods work better with data that isn't regular.

Appendix A

OCAD File Format

The OCAD file format is documented at (?). I will refer to structure names from the linked file throughout this appendix.

A.1 Reading

The file consists of a number of different types of block of 255 items, each referenced from the header and with a reference to the next block. The blocks that are of note to this report are the symbol and object blocks. An OCAD file is interesting in that there are no pre-defined map symbols in the application. Instead they are defined in individual map files as a series of vectors and colours. This allows mappers to create their own symbol sets and easily distribute them with their map files, however it makes it harder than it would otherwise be to find contours, as the symbol may be in a different place in each file. However, to find the contours in a file it is not necessary to read the whole symbol information in.

Because contours are always line symbols, and all we are interested in is the ID and symbol description the more complex types of symbol can be ignored. When iterating through the symbol blocks it is only necessary to read a TBaseSym as this gives us all the data we need. However some files have multiple symbol definitions for both "Contour" and "Index Contour" so it is important to take that into account. Future work should read in form lines as well.

Once the symbol IDs for the desired contour types are found the file reader iterates over the object index blocks, reading in TElements until one of the recorded symbol IDs is found. The actual Bézier patch points are stored within a TDPoly array at the end of the TElement.

A.2 Previous versions

The file format described above is valid for OCAD9 and OCAD10 files (although untested on OCAD10). Previous versions are incompatible and the reader I implemented will not load them although it will make an attempt at OCAD8 files.

Appendix B

A Brief Analysis of Catching Features

B.1 What is Catching Features

Catching Features is a game designed to simulate the sport of Orienteering on a personal computer, either via online competition with other people or against an AI. The developer has made available tools to import OCAD maps into the game's level format so it covers some of the same areas as this project.

B.2 Terrain Rendering Method

Catching Features takes a different approach to rendering terrain than I have for this project. All the polygons in the terrain are pre-calculated then a BSP tree is built to determine what can be seen from where. The BSP tree is then saved to a file along with texture and other model information. This allows the engine to only render the area of ground that can be seen from any given point. Slight distortions such as paths and pits are baked into the structure of the terrain. An example of the process used can be seen in figure B.1.

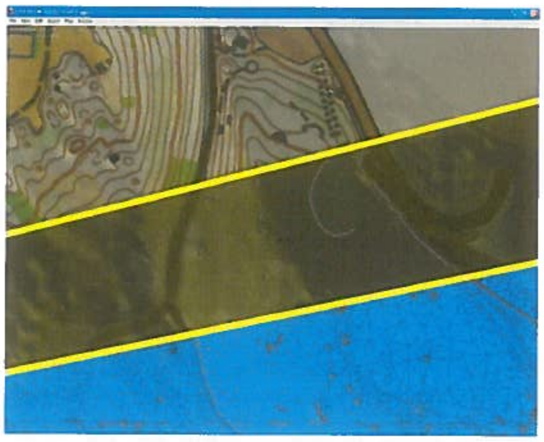


Figure B.1: Catching Features terrain generation, taken from the game's website

B.3 Pros and Cons

The BSP tree method allows for a lot of the terrain processing to be done at build time rather than in real time during rendering, offloading a lot of the work from the user's computer. It also handles occlusion so that terrain hidden behind other terrain is not visible. The main downside is a lack of flexibility after the initial generation. There is no varying level of detail other than that added at creation, and the heights can not be modified without starting from scratch.

Appendix C

Sample maps and output images

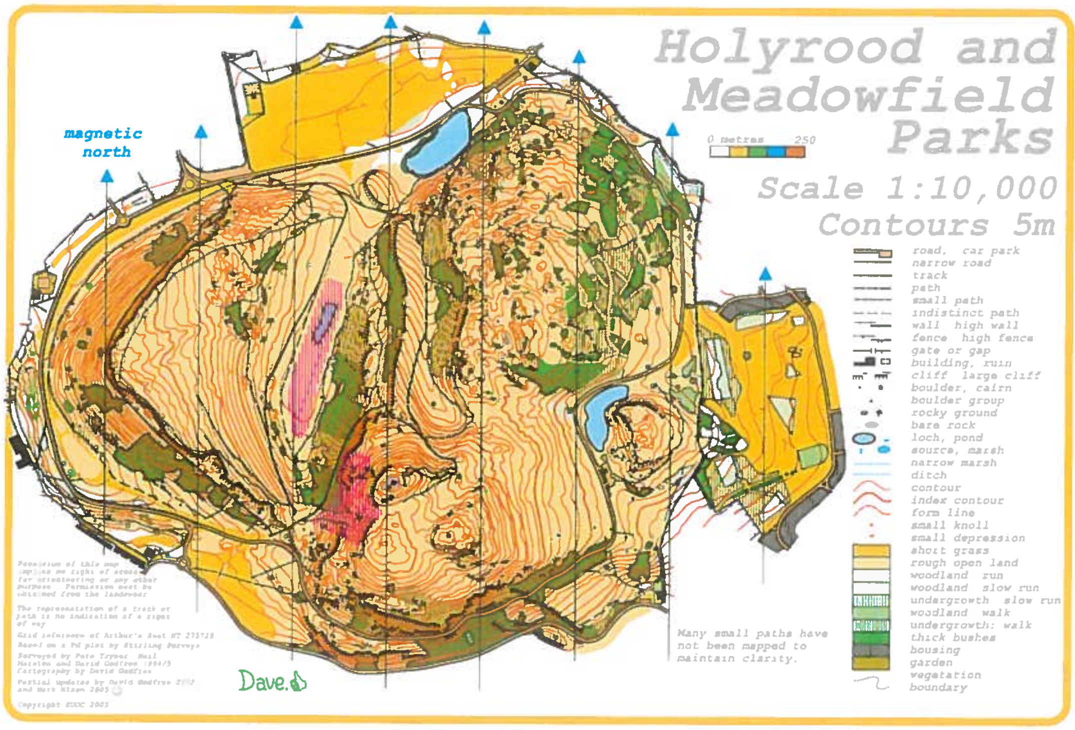


Figure C.1: Holyrood Park orienteering map

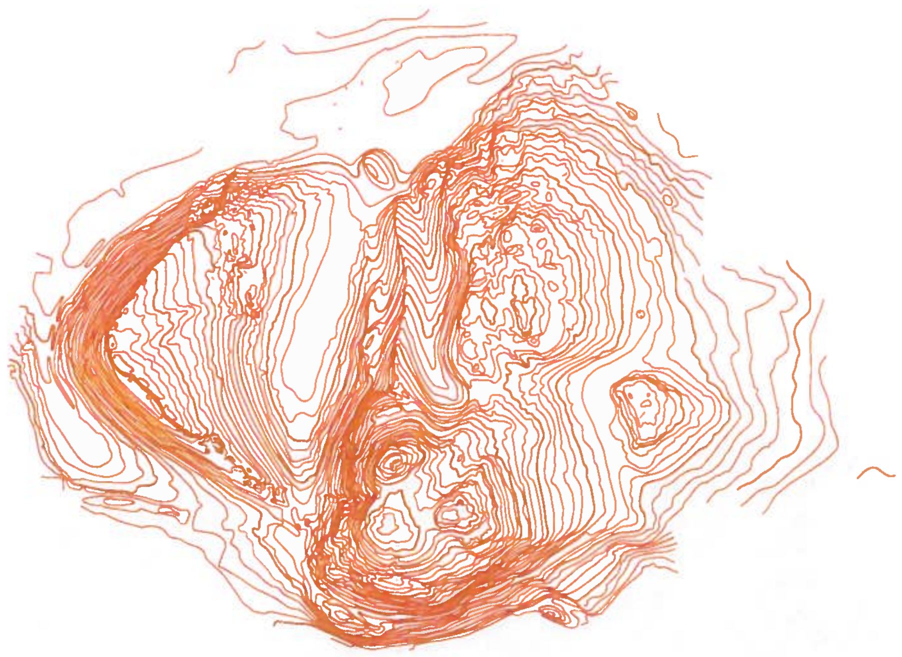


Figure C.2: Holyrood Park orienteering map showing just contours

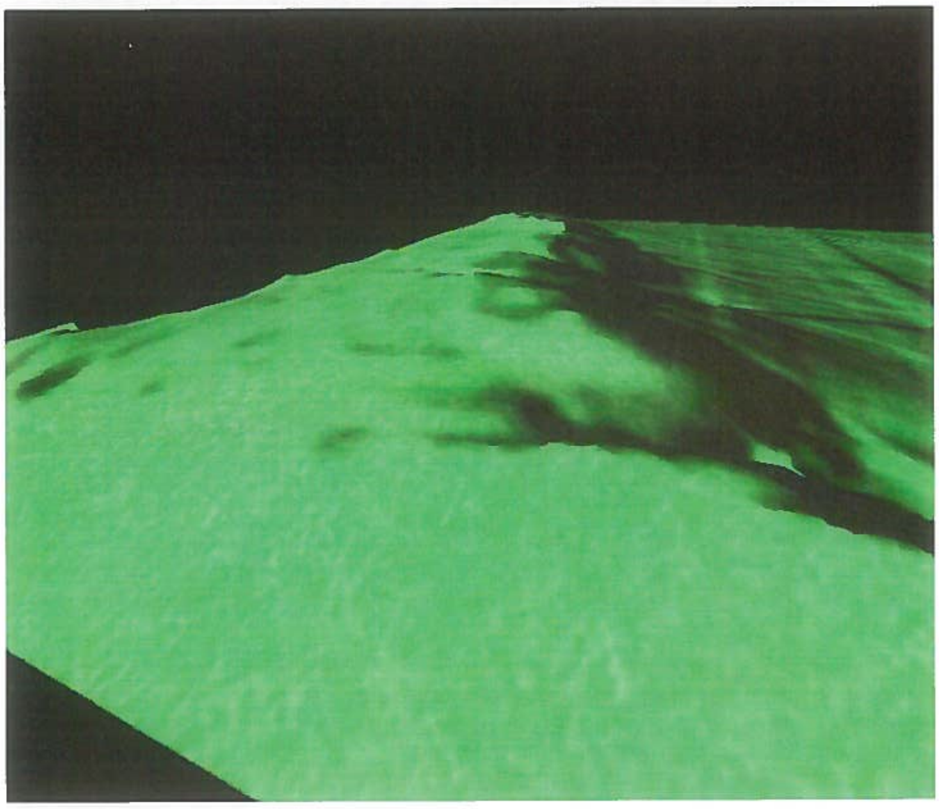


Figure C.3: 3D view of Corstorphine

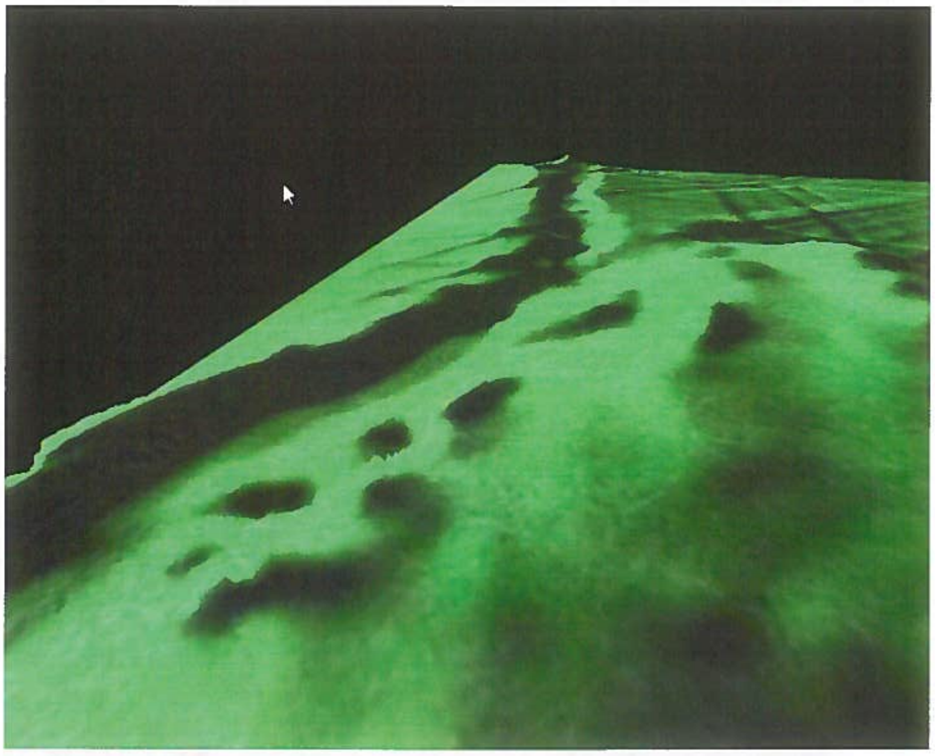


Figure C.4: 3D view of Blackford

Appendix D

Issues with automatically assigning contour heights

To create height maps from contour maps it is first necessary to determine the height of the contours. OCAD maps do not provide this data so it must either be entered by a user or computed from the provided data. In this appendix I will discuss some of the possible methods of doing so and the difficulties involved.

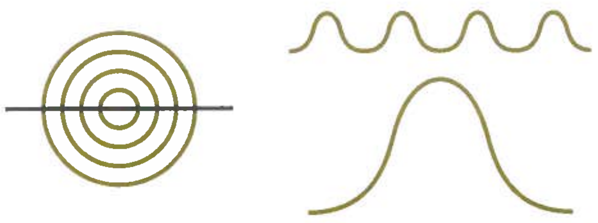


Figure D.1: Ambiguous contours

Figure D.1 shows a basic issue encountered when trying to automatically determine which way is up on a contour map. The cross section represented by the black line could produce either of the two patterns on the right. A human might realise that it is more likely to be the single curve, but the wave pattern is equally valid. In isolation it is only possible to assign a probability that one pattern or the other is correct. In order to come to a more certain decision it is necessary to look at both the surrounding contour patterns and the other map features. Water usually lies at a local minima so by extending gradient lines outwards from water features such as lakes, marshes or rivers a good idea of what is up can be determined. It is also possible to look at how features usually sit next to each other on maps and attempt to use those patterns to determine height.

However building the rules outlined above is non-trivial, and requires either a very good knowledge of geographical features or a learning algorithm and large library of maps with their height values already assigned. On top of all this, different parts of the world will require different rule sets. For example in Southern Sweden it is very common to find marshes in depresssions on top of small hills which would confuse a system trained on or designed for steep Scottish hills.

Even if all the above problems were solved, the result would still be a combination of probabilities, and would need checking by a human being for correctness. This process may well take longer than just having a human assign all or some of the values by hand. Either way, implementing such a system is way outside the scope of this project.

Appendix E

Pseudo code for height map generation algorithms

Listing E.1: Closest neighbour

```
For each contour
  For each contour segment
    Draw curve to temporary array with the height value of the
        contour
  End
End

While array isn't full
  For each pixel
    If pixel is set
      Fill in all the unset neighbours with this pixel's value
    End
  End
End

For each pixel
  Copy array value to image
End
```

Listing E.2: Closest two

```
For each contour
  For each contour segment
    Draw curve to temporary array with the height value of the
        contour
    Assign a distance value of 0 to each set height value
  End
End

While any pixel has less than 2 height values
  For each pixel
    If pixel has one or more height values
      For each neighbour
        If neighbour has one or less height value
          Copy height value to neighbour
          Increment the distance value by 1 for each height value
              just set
        End
      End
    End
  End
End

For each pixel
  Calculate a value for the pixel by using a weighted average of the
      two height values
  Copy the calculated value to the output image
End
```

Listing E.3: Precise intersection by equation

```
For each pixel
  For each of the 8 compass directions
    Project vector V from the pixel to the edge of the image
    For each contour
      For each segment S
        Intersect V with S
      End
    End

    Sort the intersections by distance
    Save the closest intersection
  End

  Calculate a value for the pixel by using a weighted average of the
      eight height values
  Write the value to the image
End
```

Listing E.4: Precise intersection by equation (optimised)

```
For each image row
  Create vector V along the row
  For each Contour
    For each segment S
      Intersect V with S
    End

    Sort the intersections by distance
    Save all the intersections
  End
End

For each image column
  As above
  ...
End

For each image diagonal (SW/NE)
  As above
  ...
End

For each image diagonal (SE/NW)
  As above
  ...
End

For each pixel
  For each of the 4 above data sets
    Find the closest value less than the pixel position
    Find the closest value greater than the pixel position
  End

  Calculate a value for the pixel by using a weighted average of the
      eight height values
  Write the value to the image
End
```

Listing E.5: Precise intersection by pixel

```
For each contour
  For each contour segment
    Draw curve to temporary array with the height value of the
        contour
  End
End

For each pixel
  For each of the 8 compass directions D
    Traverse the array in the direction D until a value is found
  End

  Calculate a value for the pixel by using a weighted average of the
      eight height values
  Write the value to the image
End
```