1-1-1990

A data structural approach to the problem of contour plotting

Sivesh Pradhaan
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Part of the Higher Education Commons

Repository Citation
https://digitalscholarship.unlv.edu/rtds/102

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A data structural approach to the problem of contour plotting

Pradhaan, Sivesh, M.S.
University of Nevada, Las Vegas, 1990
A DATA STRUCTURAL APPROACH TO THE PROBLEM OF CONTOUR PLOTTING

by

Sivesh Pradhaan

A Thesis submitted in partial fulfillment of the requirements for a degree of

Master of Science

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas

July 1990

This research was partially funded by the U.S. Army Research Office
The thesis of SIVESH PRADHAAN for the degree of MASTER of
SCIENCE in COMPUTER SCIENCE is approved.

Chairman,  Evangelos Yfantis, Ph.D.
Exckning Committee Member,  Ajoy Datta, Ph.D.
Exckning Committee Member,  Laxmi Gewali, Ph.D.
Graduate Faculty Representative,  Peter Shiue, Ph.D.
Graduate Dean,  Ronald Smith, Ph.D.

University of Nevada, Las Vegas
July, 1990
ACKNOWLEDGMENT

I gratefully acknowledge the outstanding guidance and support my advisor Dr. Evangelos Yfantis provided during the course of this research. I would also like to express my appreciation to Dr. Laxmi Gewali, Dr. Ajoy Datta and Dr. Peter Shiue for serving on my examining committee and taking the time to review this work. I am thankful to the Army Research Office for partial funding of this research.
ABSTRACT

This paper describes the results of a research on the problem of contouring an estimated surface. Most of the papers published in this field have taken a mathematical approach to the contouring problem and are of "contour following" type. This paper takes a data structural approach to the problem and also uses a special logic developed to make parallelism possible. This algorithm draws all the contour curves at the same time. The data structure is developed such that the neighborhood information for each cell and all the intersection points between the corresponding face and the contour plane are stored. The data structure used by this algorithm is described in detail here. After the intersection data for all the edges of a cell has been generated, the problem of figuring out how to connect these intersection points is solved by examining the orientation of the corresponding face used to represent that patch of the estimated surface. The total number of intersections with a contour plane in a cell and the presence of horizontal lines in a face is also used to guide the plotter. The logic developed for connecting the points of intersections in a cell is also described in detail. The computer programs which implement these concepts are included in the appendix.
Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Problem Statement</td>
<td>1</td>
</tr>
<tr>
<td>The Algorithm</td>
<td>5</td>
</tr>
<tr>
<td>Data Structure</td>
<td>6</td>
</tr>
<tr>
<td>Implementation Detail</td>
<td>7</td>
</tr>
<tr>
<td>Plotting Algorithm</td>
<td>12</td>
</tr>
<tr>
<td>Conclusion</td>
<td>20</td>
</tr>
<tr>
<td>Figures</td>
<td>23</td>
</tr>
<tr>
<td>Appendix</td>
<td>43</td>
</tr>
<tr>
<td>References</td>
<td>63</td>
</tr>
</tbody>
</table>
INTRODUCTION

Data that can be displayed by a contour map is found in many engineering applications. Any piecewise continuous, single valued function of two continuous independent variables can be represented in the form of a contour map. The most common example is a contour map representing elevation as a function of position in a two dimensional geographic region. Other geographic position dependent variables that are commonly represented in the form of contour maps are temperature (isotherms) and pressure (isobars). However, the use of contour maps need not be restricted to dependent variables relating to geographic position. An example of a contour map used for a more general dependent variable is a plot of equal loudness curves drawn as a function of the intensity and frequency of an audible tone. In some applications, contour maps may be used to facilitate visualisation of the data even though an equation may exist that describes this data, for example, a plot of the equipotential lines around an electric dipole.

Problem Statement

The problem can be stated as follows: Let $f = f(x,y)$ be a two variable
function defined in a rectangular domain \( R(a \leq x \leq b; c \leq y \leq d) \). Let us suppose that \( f = f(x, y) \) is known at the nodes of a rectangular grid traced on \( R \), and that it is required to plot \( n \) contour lines:

\[
f(x, y) = l_k
\]

\[k = 1, 2, \ldots, n\]

Denoting each node by \((i, j)\), a matrix \( ||f(i, j)|| \) can be defined. It is convenient to define also, as a cell of a grid, the rectangle with four adjacent nodes of the grid as vertices and a diagonal as its fifth edge.

The solution of the problem can be achieved by two logical steps:

1. Computation of coordinates of all the intersection points between each contour plane and the edges of the face corresponding to every cell.

2. Suitable linking of these points.

There are two possible schemes that we can come up with, using steps (1) & (2). In the first scheme we process each cell of the grid, i.e. we apply step (1) to each cell and then with some criterion, to be established, the points found in step (1) are linked during the execution of step (2).

In the second scheme one point of the curve is searched and the curve is followed until it stops. So here step (1) and step (2) are performed together. Most of the algorithms developed so far for the contouring problem, follow
the second scheme. They are very useful and efficient if the output device is of mechanical nature.

In general the methods following the first scheme require the storage of all the intersection points. These methods can however avoid storing the entire matrix $||f(i,j)||$ if all the $n$ curves are treated simultaneously. The problem of labeling each contour curve can be solved by using the contour number corresponding to a contour curve as an index into the color table and using that color to plot the curve. The methods using the second scheme however require the storage of the entire matrix $||f(i,j)||$ but the storage of the intersection points is not required.

Generally, a contour curve may have more than one branch and also there may be more than one independent contour curves present for the same contour number. During the execution of step (2), some criterion must be used to guide the drawing algorithm.

The method described in this paper follows the first scheme. In this method every cell is processed. Each edge of a cell is checked for intersection point with the $n$ contour planes and the ones found are stored. So each edge is processed only once. The drawing routine also processes each cell independently and connects the intersection points of each contour number
using the logic based on the number of intersections found in that cell with that contour plane.

The fact that both the data generating algorithm and the plotter can process a cell independent of any influence from the neighboring cells, gives this algorithm the parallel capabilities. This is a very important result because to date there does not exist any parallel algorithm for the contouring problem. The details of this algorithm are given in the following sections.
THE ALGORITHM

The estimated surface is drawn on a CRT monitor using rectangular faces. The more the number of these faces, the smoother will be the appearance of the surface and smaller will be the projections of these faces on the $xy$–plane. Note that every face has a finite rectangular projection on the $xy$–plane with the lengths of edges of these projected rectangles being the increments used in the $x$ and $y$ directions. There are no vertical faces present and every rectangle -henceforth called the cell- is of same area.

The smoothness of the contour curves plotted by this algorithm is dependent upon the size of the cells and therefore on the number of faces used to draw the surface. The more the number of faces, the smoother will be the contour. Note that because of the parallel nature of this algorithm, the computational time is dependent upon the availability of networked computers as each cell can be processed on a different computer over the network.

This algorithm assumes that either the matrix $\|f(i,j)\|$ or the approximating function for the estimated surface is available. The data structure and the logic used to connect the intersection points, are the two unique features of this algorithm. The details of this algorithm follows.
Data Structure

The domain $R$ is covered by a rectangular grid and each rectangular element of this grid is called a cell (each cell is the projection of a face used to represent the estimated surface). If the matrix $||f(i,j)||$ is available then each element of this matrix is the $z$-offset of nodes of the grid, else if the approximating function is available then we can select the size of each cell and find the corresponding $z$-offsets at the nodes of the resulting grid.

Each cell consists of four edges and one diagonal. Each edge of a cell is numbered in the clockwise direction from zero to four starting from the edge closest and parallel to the $y$-axis (Fig. 1). Edge number four is the diagonal edge. For each edge we keep the coordinates of its endpoints, the type of edge, i.e. whether it is horizontal or otherwise, a pointer to the linked list containing the coordinates of the intersection points (each element of this linked list stores the coordinates of the intersection point the contour number of this point and a pointer to the next element.), and a pointer to the neighboring cell (each edge of the grid, except the outermost edges, is shared by two neighboring cells and this neighborhood information is stored). The contour number of the lowest and highest contour planes intersecting an edge
is also stored.

Thus a cell consists of five edges, each edge has a type attached to it. The endpoints of each edge, its neighborhood information with other cells and most important of all, the intersection points with the contour planes is stored in a linked list. This way all possible information, that we may want to make use of, is available.

Implementation Detail

In the data structure design, each cell has five edges and each of these five edges are processed for intersection points. But the edges zero through four are shared by the neighboring cells. Thus in order to prevent recalculation of the intersection points for these common edges as we process each cell we implement a recursive algorithm that processes the cells in the $y$ direction. After the first cell in $y$ direction has be processed, a flag is set to indicate that edge number three of the following cells should not be processed. When the recursion stops, the linked list of edge number one of the preceding cell is linked to edge number three.

The execution of this routine is shown in Fig. 2 and Fig. 3. This routine is
called by another recursive routine that processes the cells in the \( x \) direction. Even in this routine, a flag is set to prevent reprocessing of edge number zero. Fig. 4 & 5 illustrate the execution of this recursive algorithm. Fig. 6 is the final result after the complete domain has been processed.

Each edge of a face could either be horizontal or normal (i.e. non-horizontal). There will not be any vertical edge because we have assumed to be dealing with a single valued function. Thus, each face could consist of all horizontal edges, in which case the face itself will be horizontal, or could have one or more normal edges and so be a normal face. The edges of a normal face need not lie in the same plane. There will not be any vertical face because it will then consist of vertical edges only, the possibility of the existence of which we have already ruled out.

For every edge of a cell we find out which contour planes intersect this edge and then calculate the coordinates of the intersection. We have used linear interpolation methods to determine the coordinates because we assume that the surface estimation technique provides us with a very fine grid over the domain. If in any particular case this assumption does not hold true, then we can classify every edge of the cell into three categories viz.

(a) edges parallel to the \( x \) axis. To find the intersections of these edges
we keep the $y$ and $z$ fixed and vary $x$.

(b) edges parallel to the $y$ axis. To find the intersections of these edges we keep the $x$ and $z$ fixed and vary $y$.

(c) the diagonal. In this case both the $x$ and the $y$ will vary, but the $z$ will be fixed. The slope of the diagonal with respect to the $x$ and $y$ axis will determine the incrementing step size in the $x$ and $y$ direction respectively.

Fig. 7 shows the resulting inaccuracy if we use linear interpolation. Solid line represents the profile of the face approximating a segment of the actual curve -shown by broken line. If the grid that we are working with is dense, then this inaccuracy is negligibly small. Fig. 8 shows a degenerate case where the approximating face does not faithfully represent the shape of the curve. The resulting contour curve will not be accurate. But if the grid is very dense, such a case will not occur.

While finding the intersection coordinates, horizontal and normal edges are treated differently. If the edge is horizontal, then we simply check if the $z$ coordinate of one of its endpoints is a multiple of the increment $l$ or not (if we want the contour of a surface at an interval of five meters in the $z$ direction, then the $l$ will be five meters and the contour planes will be at 5, 10, 15, ... meters). If the $z$ coordinate is a multiple of $l$, then we say that this
edge has one intersection with the contour plane and the contour number is calculated as follows:

\[
\text{contour number} = \text{int}(z/l) \times l
\]

where \(z\) = the z-coordinate of the horizontal line

\(l\) = increment in the elevation from one contour plane to the next

If the edge is normal then we first determine the lowest and the highest contour planes that would intersect this edge. Fig. 9 shows all possible way in which endpoints of an edge could be distributed with respect to the level planes. The step for calculating the lower and upper contour planes are as follows:

(a) find out the lower and upper endpoints of the edge.

(b) the contour plane intersecting this edge and closest to the lowest point is

\[
\text{IF } (z_{\text{lower}}/l) - \text{int}(z_{\text{lower}}/l) \text{ is zero THEN}
\]

\[
\text{contour number} = \text{int}(z_{\text{lower}}/l) \times l
\]

\[
\text{ELSE}
\]

\[
\text{contour number} = \{\text{int}(z_{\text{lower}}/l) + 1\} \times l
\]
This would be the lowest contour plane intersecting this edge.

(c) the highest contour plane intersecting this edge is

\[ \text{contour number} = \text{int}(z_{\text{upper}}/l) \times l \]

Fig. 9e will give a contradiction by this method i.e. the resulting lowest contour plane intersecting this edge will be above the highest contour plane intersecting this edge. This is a contradiction which can be easily detected and treated accordingly. After the detection process is done, we use linear interpolation to find out the \( x \) and \( y \) coordinates of the points of intersection with all other contour planes between the lower and upper contour planes.

This process is repeated for every edge of each cell of the grid. This way we would have generated all the intersection data. Now the plotting routine can be called. If parallel operation is desired, then after each cell is processed, the plotting routine is called. The logic used by the plotting routine guarantees correct inter-cell connections of every contour line. The next section describes the plotting algorithm.
Plotting Algorithm

During the development of the logic for connecting the intersection points, all possible orientations of a face were carefully studied, taking into account that the face could have zero or more horizontal edges. The resulting logic turned out to be strong enough to permit independent treatment of each cell while drawing all the contour lines in a cell. So this logic supports parallelism.

Take each cell and determine the lowest and the highest contour planes intersecting the corresponding face. This gives the lower and upper bound for searching a contour line in this cell. In the next step, the total number of intersection points between the face and a contour plane, within the bound, is counted from the linked lists of every edge of this cell. Since there are only five edges in each cell, there can be at most five intersection points. In general there could be 0, 1, 2, 3, 4, or 5 intersections. Let us examine each of these five cases separately.

Case 0: This implies that the face is not intersected by this contour plane and therefore its corresponding cell will not have any contour line in it corresponding to this contour plane.
Case 1: Upon close examination of this case it is found that a face cannot have only one of its edges intersected by a contour plane. In Fig. 10, let us assume that edge ab is the only edge intersected. Then the elevations of points a and b would be

\[ z_a \leq e \text{ and } z_b \geq e \]

or

\[ z_a \geq e \text{ and } z_b \leq e \]

where \( e \) = elevation of the contour plane.

Assume that \( z_a > e \) and \( z_b < e \). Then \( z_c < e \) and \( z_d < e \). So from this it is clear that ad and ac should also be intersected.

Case 2: There are 22 probable ways of getting two intersections. All of these 22 combinations can be grouped into four topologically different combinations as shown in Fig. 11 (the position of the intersected edges with respect to the diagonal decides the topological grouping). Out of these four combinations, Fig. 11 (c) & (d) are not possible. That leaves us with just two combinations. Also note that a face with only two intersected edges, will not have any horizontal edge and also its diagonal is never intersected.
Case 3: Fig. 12 shows the six different probable combinations of having three intersections. Out of these, combinations (b), (c), (d) & (e) are not possible. So for three intersections, we have two possible combinations. Again note that there will be no horizontal edges present. In this case, however, the diagonal is always intersected.

Case 4: Fig. 13 shows the seven probable combinations for having four intersections. Fig. 13 (g) has a horizontal edge. Out of these seven, Fig. (f) is not possible. So we have six possible combinations. Note that if a face has four intersections, then it can have at most one horizontal edge.

Fig. 19 shows four different orientations of a face which has four intersections and its diagonal is not intersected. These four orientations can be divided into two different groups. Fig. 19 (a) & (b) fall in one group because in both of them the diagonal of the face is above the contour plane intersecting the four edges, and Fig. 19 (c) & (d) fall in the second group because in both of them the diagonal of the face is below the level plane. It is clear that the contour line is sensitive to the orientation of the face. If the approximating function for the surface is available, then the orientation of the face can be determined by checking the function value at the center of
the cell. If the function value lies on the diagonal ac, then the orientation is as in Fig. 19 (b) or (c). If the function value lies on the diagonal bd, then the orientation is as in Fig. 19 (a) or (d).

If the approximating function is not available, instead the matrix $||f(i, j)||$ is given, then the orientation of the face cannot be determined. Therefore we assume the correct orientation to be that of Fig. 19 (b) & (d). This assumption may lead to the generation of an inaccurate contour line in the cell, but if the grid is dense enough, then this inaccuracy does not seriously affect the resulting contour.

Case 5: Fig 14 shows the five probable combinations of having five intersections. Fig. 14 (a) & (c) is not possible. If the face has no horizontal edge then only Fig. 14 (b) is possible. If there is just one horizontal edge, then only Fig. 14 (d) is possible. If there are three horizontal edges, then only Fig. 14 (e) is possible. If there are five horizontal edges, then only Fig. 14 (f) is possible.

Fig. 20 shows four different orientations of a face having five intersections with not a single horizontal edge. As in the preceding case if the orientation of the face is determined then the contour line in this cell can be drawn.
faithfully. Else Fig. 20 (b) & (d) is assumed to be the correct orientation of the face. The resulting inaccuracy may not even be noticeable if the grid is dense enough.

From the discussion so far we can conclude the following:

(a) If a face is intersected by a level plane, then at least two of its edges will be intersected.

(b) If two or three edges of a face are intersected, then this face does not have any horizontal edge.

(c) If a face has two intersected edges, then its diagonal edge is not intersected. i.e. the intersected edges lie on the same side of the diagonal.

(d) If the face has three intersected edges, then its diagonal is always intersected. That is, two if the intersected edges lie on the opposite sides of the third intersected edge - the diagonal.

(e) If the face has four intersected edges, then it may have at most one horizontal edge. If a horizontal edge is present then the diagonal edge is intersected and the diagonal is not horizontal. That is, the diagonal can never
be horizontal. If there is no horizontal edge present then the diagonal is not intersected except maybe at one of its endpoints.

(f) If the face has five intersected edges, then there could be either none or 1, 3, 5 horizontal edges present. If there is at least one horizontal edge present, then the diagonal will always be horizontal. If all five edges are horizontal then the face itself is horizontal.

These conclusions will be the guiding factors when the intersection points, belonging to a particular contour line, in a cell are connected. The strategy for connecting intersection points of a contour line in a cell is based on the number of intersections with that contour plane in the cell and the number of horizontal edges present. The details are described below.

A cell with no intersection has no contour lines in it. So nothing needs to be done with it. If the cell has two intersection points, then the first intersection point found while searching in the clockwise direction, starting from edge #0, is connected to the second intersection point. In case of three intersections, the first one is connected to the diagonal intersection, the diagonal intersection is connected to the second intersection. Fig. 15 shows the cases of two and three intersections.
If a cell has four intersections then the following control structure is used:

**IF** there is a horizontal edge present **THEN**

draw the horizontal edge.

**ELSEIF** the diagonal is intersected but not at its end point **THEN**

connect first point with the second,

second point with the fourth and

fourth point with the third.

**ELSEIF** the diagonal is intersected at its end point **THEN**

connect first point with the fourth and

fourth point with the second.

**ELSE**

connect first point with the second and

third point with the fourth.

Fig. 16 illustrates this case.

If the cell has five intersections then the following control structure is used (Fig. 17):
IF no horizontal edge is present THEN

connect first point with second,
second point with fifth,
fifth point with fourth and
fourth point with third.

ELSEIF there are up to three horizontal edges present THEN

draw the diagonal edge. i.e. connect first point with the third.

If the corresponding face has all five edges horizontal, then it is a horizontal face. i.e. this face lies on the level plane. Such a face can be put in two categories viz.

(i) horizontal faces with horizontal neighboring faces only.

(ii) horizontal faces with at least one non-horizontal neighboring faces.

In this case the non-horizontal face will have at least one and at most three horizontal edges.

Looking at Fig. 18 we can conclude that when a face is horizontal, we skip it because if it falls in category (i), then there are no contour lines in it, and if it falls in category (ii), then the contour line associated with it will be
drawn when its non-horizontal neighbors are processed.

Conclusion

From the discussions so far, it is very clear that due to the parallel nature, this algorithm is an extremely fast algorithm. The speed of execution will depend mainly upon the number of computers available over the network, and the degree of detail desired. Since the neighborhood information of each cell is stored, this algorithm can be improved to reduce the memory requirement by removing those cells from the data structure, that have no contour lines passing through them. This way all the elements of the data structure will have contour lines information. Actual saving in memory space will depend on $l$ - the elevation increment between two consecutive contour planes. The bigger the $l$, the smaller will be the number of cells with contour lines passing through them and vice versa. As a matter of fact, if no transformation is to be carried out after the contour curves have been displayed, then no data need be stored. The memory requirement will then be the minimum and will be equal to the memory needed to store the intersection points of each edge of a cell. In the worst case the memory requirement for a cell could be
5 * n, where n is the total number of contour planes present. In other words, each edge of this cell is intersected by all the contour planes -this is highly improbable in practice.

Let us suppose that m is the total number of points given in the x direction, and p is the total number of points given in the y direction. m > 1, p > 1 and m > p.

- total number of points in the domain = \( a = m \times p \leq O(m^2) \)
- total number of cells = \( b = (m - 1) \times (p - 1) \leq O(m^2) \)
- total number of edges = \( c = 3(m - 1) \times (p - 1) \leq O(m^2) \)
- maximum number of contour planes that can intersect each edge = n
- total number of intersection points (in the worst case) = c * n

If the matrix \( ||f(i,j)|| \) is given to us then c is a constant and thus the memory requirement is \( O(n) \).

The processing time for each cell is dependent upon the time taken to process each of its edges. The processing time for each edge is a function of the total number of contour planes that will intersect that edge. In the worst case, an edge can be intersected by all the contour planes. So its processing time, in the worst case, will be \( O(n) \). Therefore the processing time for each
cell, will be $O(n)$. In general:

\[
\text{processing time for the domain} = \text{number of edges} \times \text{time for each edge;}
\]

\[
\Rightarrow c \times n = O(n);
\]

If $n = 0$, i.e. if $l$ is so large that no face is intersected, then the time taken for processing the complete domain is constant and is equal to the total number of edges present i.e. $O(c)$.

If the approximating function is given then this algorithm always produces correct contour and is truly a parallel algorithm. But if the matrix $||f(i, j)||$ is given then it may not produce the correct contour in a cell that either has four intersected edges and whose diagonal edge is not intersected, or has five intersected edges and has no horizontal edges. This is because in these cells, the contour lines are sensitive to the orientation of the corresponding face and just by looking at the four points, we cannot determine the orientation of the face. May be if we consider the behaviors of the neighboring faces, then we can determine the orientation of this face. But in doing so, the parallel nature of the algorithm will be lost. This is one degenerate case that needs further investigation.
FIGURES
FIG # 1

Ordering of the edges of a cell
actual curve
approximation
level line

FIG # 7

FIG # 8

level line
nine possible distributions of the end points of an edge with respect to the level_lines

FIG # 9
point of intersection

FIG #10
*•* => indicates the point of intersection

FIG # 11
• => indicates the point of intersection

FIG # 12
\[ \bullet \Rightarrow \text{ indicates the point of intersection} \]

FIG # 13
(a) 

(b) 

(c) 

(d) 

(e) 

(f) 

• => indicates the point of intersection

FIG # 14
FIG #15

(a)  

(b)  

(c)  

(d)
FIG #17
FIG # 18
FIG # 19
FIG # 19
the level plane

contour of this surface

FIG #20
the level plane
contour of this surface
APPENDIX

The Source Code
#include<gl.h>

#define INFINITY 10000
#define TRUE 1
#define FALSE 0
#define NORMAL 0
#define HORIZONTAL 1
#define VERTICAL 2
#define UP 1
#define DOWN 0
#define NULL 0
#define FUNCTION 25.0 * exp( - (pow(x - 10,2) + pow(y - 10,2)) / 50.0) + \
  64.0 * exp( - (pow(x - 20,2) \ + pow(y - 20,2)) / 32.0) \n
typedef struct {
  float x;
  float y;
  float z;
} point;

typedef struct {
  int level_no;
  point p;
  struct levelinfo * next;
} levelinfo;

typedef struct {
  levelinfo * level_ptr;
  int low;
  int high;
} level_detail;
typedef struct {
    int type;
    level_detail level;
    point pt[2];
    struct edge_info * neighbour;
} edge_info;

typedef struct {
    edge_info edge[5];
} rect_info;

/*******GLOBAL VARIABLES*******
/****** (START WITH UPPERCASE LETTERS)******/

float X_inc, Z_inc, Tolerance = 0.0, Ratio;
int Level = 5, Lattice_flag = UP, Rect_flag = UP;
float X_min = 0.0, X_max = 100.0, Z_min = 0.0, Z_max = 100.0;

/******THIS FUNCTION PROCESSES HORIZONTAL LINES*******/

level_detail horizontal_line(pt)
point pt[2];
{
    level_info * ptr;
    level_detail data;
    float multiplier, deci_no;
    int whole_no;

    multiplier = pt[0].y / Level;
    whole_no = (int)multiplier;
    deci_no = multiplier - whole_no;

    data.level_ptr = NULL;
    data.low = 0;
data.high = 0;

if (deci_no — Ratio <= 0.0)
{
    ptr = (levelinfo *) malloc(sizeof(levelinfo));
    ptr->level_no = whole_no * Level;
    ptr->p.x = pt[0].x;
    ptr->p.y = ptr->level_no;
    ptr->p.z = pt[0].z;
    ptr->next = NULL;
    data.level.ptr = ptr;
    data.low = data.high = ptr->level_no;
}

return data;

LEVELDETAIL normal_line(pt)
point pt[2];
{
    levelinfo * ptr, * ptr1 = NULL;
    leveldetail data;
    float multiplier, deci_no, x_ratio, z_ratio, y_diff;
    float x1, x2, z1, z2;
    int whole_no, level[2], temp, low.pt, high.pt;
    int flag = UP, i;

    data.level.ptr = NULL;
    data.low = 0;
    data.high = 0;

    if (pt[0].y < pt[1].y)
    {
        low.pt = 0;
high_pt = 1;
}
else
{
    low_pt = 1;
    high_pt = 0;
}

for (i = 0; i < 2; i++)
{
    multiplier = pt[0].y / Level;
    whole_no = (int)multiplier;
    deci_no = multiplier - whole_no;

    if (((deci_no - Ratio <= 0.0) || (i != low_pt))
        level[i] = whole_no * Level;
    else
        if (i == low_pt)
            level[i] = (++whole_no) * Level;
}

if (level[low_pt] > level[high_pt])
    flag = DOWN;

if (level[0] > level[1])
{
    temp = level[0];
    level[0] = level[1];
    level[1] = temp;
}

if (flag == UP)
{
    if ((pt[high_pt].x - pt[low_pt].x) < 0.0)
    {
        x1 = pt[high_pt].x;
        x2 = pt[low_pt].x;
    }
else
{
    x2 = pt[high_pt].x;
    x1 = pt[low_pt].x;
}

if ((pt[high_pt].z - pt[low_pt].z) < 0.0)
{
    z1 = pt[high_pt].z;
    z2 = pt[low_pt].z;
}
else
{
    z2 = pt[high_pt].z;
    z1 = pt[low_pt].z;
}

y_diff = pt[high_pt].y - pt[low_pt].y;
x_ratio = (x2 - x1) / y_diff;
z_ratio = (z2 - z1) / y_diff;

for (i = level[1]; i >= level[0]; i -= Level)
{
    ptr = (level.info *) malloc(sizeof(level.info));
    ptr->level_no = i;
    ptr->p.x = x1 + ((Level - pt[low_pt].y) * x_ratio);
    ptr->p.y = ptr->level_no;
    ptr->p.z = z1 + ((Level - pt[low_pt].y) * z_ratio);
    ptr->next = NULL;
    if (ptr1 != NULL)
    {
        ptr->next = ptr1;
        ptr1 = ptr;
    }
    data.level.ptr = ptr;
    data.low = level[0];
    data.high = level[1];
}
return data;
}

/******THIS FUNCTION PROCESSES***********/
/**********LINES FOR INTERSECTION POINTS**********/

level_detail level_processor(edge_type, edge_no, pt)
int edge_type, edge_no;
point pt[2];
{
    level_detail data;

    data.level_ptr = NULL;
    data.low = data.high = 0;

    switch (edge_type)
    {
        case HORIZONTAL : data = horizontal_line(pt);
                          break;
        case VERTICAL : data = vertical_line(pt);
                        break;
        case NORMAL : data = normal_line(pt);
                      break;
    }
    return data;
}

/******THIS FUNCTION PROCESSES EACH SUBLATTICE***********/

rect_info *rect_processor(a)
point a;
{

rect_info *ptr, *ptr1 = NULL;
point b, pt[2];
float x,z,xmin,zmin,xmax,zmax;
int edge_no, edge_type;

xmin = X_min;
xmax = X_max - X_inc;
zmin = Z_min;
zmax = Z_max - Z_inc;
x = a.x;
z = a.z;
ptr = (rect_info *) malloc(sizeof(rect_info));

/******* PROCESSING EDGE # 0 ***********/
ptr->edge[0].pt[0].x = pt[0].x = x;
ptr->edge[0].pt[0].z = pt[0].z = z;
ptr->edge[0].pt[0].y = pt[0].y = FUNCTION;
ptr->edge[0].pt[1].x = b.x = pt[1].x = x;
ptr->edge[0].pt[1].z = b.z = pt[1].z = z + Z_inc;
ptr->edge[0].pt[1].y = b.y = pt[1].y = FUNCTION;
z += Z_inc;
edge_no = 0;
if ((abs(pt[0].y - pt[1].y) - Tolerance) == 0.0)
    ptr->edge[edge_no].type = edge_type = HORIZONTAL;
else
    if ((abs(pt[0].x - pt[1].x) - Tolerance) == 0.0)
        ptr->edge[edge_no].type = edge_type = VERTICAL;
else
    ptr->edge[edge_no].type = edge_type = NORMAL;
if (Lattice_flag == UP)
    ptr->edge[0].level = level_processor(edge_type,edge_no,pt);

/******* PROCESSING EDGE # 1 ***********/
ptr->edge[1].pt[0].x = pt[0].x = ptr->edge[0].pt[1].x;
ptr->edge[1].pt[0].z = pt[0].z = ptr->edge[0].pt[1].z;
ptr->edge[1].pt[0].y = pt[0].y = ptr->edge[0].pt[1].y;
ptr->edge[1].pt[1].x = pt[1].x = x + X_inc;
ptr->edge[1].pt[1].z = pt[1].z = z;
ptr->edge[1].pt[1].y = pt[1].y = FUNCTION;
x + = X/inc;
edge_no = 1;
if ((abs(pt[0].y — pt[1].y) — Tolerance) == 0.0)
  ptr->edge[edge_no].type = HORIZONTAL;
else
  if ((abs(pt[0].x — pt[1].x) — Tolerance) == 0.0)
    ptr->edge[edge_no].type = VERTICAL;
else
  ptr->edge[edge_no].type = NORMAL;
ptr->edge[1].level = level_processor(edge_type,edge_no,pt);

/***** PROCESSING EDGE # 2 ************/
ptr->edge[2].pt[0].x = pt[0].x = ptr->edge[1].pt[1].x;
ptr->edge[2].pt[0].z = pt[0].z = ptr->edge[1].pt[1].z;
ptr->edge[2].pt[0].y = pt[0].y = ptr->edge[1].pt[1].y;
ptr->edge[2].pt[1].x = pt[1].x = x;
ptr->edge[2].pt[1].z = pt[1].z = z — Zinc;
edge_no = 2;
if ((abs(pt[0].y — pt[1].y) — Tolerance) == 0.0)
  ptr->edge[edge_no].type = HORIZONTAL;
else
  if ((abs(pt[0].x — pt[1].x) — Tolerance) == 0.0)
    ptr->edge[edge_no].type = VERTICAL;
else
  ptr->edge[edge_no].type = NORMAL;
ptr->edge[2].level = level_processor(edge_type,edge_no,pt);

/***** PROCESSING EDGE # 3 ************/
ptr->edge[3].pt[0].x = pt[0].x = ptr->edge[2].pt[1].x;
ptr->edge[3].pt[0].z = pt[0].z = ptr->edge[2].pt[1].z;
ptr->edge[3].pt[0].y = pt[0].y = ptr->edge[2].pt[1].y;
ptr->edge[3].pt[1].x = pt[1].x = ptr->edge[0].pt[0].x;
ptr->edge[3].pt[1].z = pt[1].z = ptr->edge[0].pt[0].z;
ptr->edge[3].pt[1].y = pt[1].y = ptr->edge[0].pt[0].y;
edge_no = 3;
if ((abs(pt[0].y - pt[1].y) - Tolerance) == 0.0)
    ptr->edge[edge_no].type = HORIZONTAL;
else
    if ((abs(pt[0].x - pt[1].x) - Tolerance) == 0.0)
        ptr->edge[edge_no].type = VERTICAL;
else
    ptr->edge[edge_no].type = NORMAL;
if (Rect_flag == UP)
{
    ptr->edge[3].level = level_processor(edge_type, edge_no, pt);
    Rect_flag = DOWN;
}

/**** PROCESSING EDGE # 4 (THE DIAGONAL)************/
ptr->edge[4].pt[0].x = pt[0].x = ptr->edge[0].pt[0].x;
ptr->edge[4].pt[0].z = pt[0].z = ptr->edge[0].pt[0].z;
ptr->edge[4].pt[0].y = pt[0].y = ptr->edge[0].pt[0].y;
edge_no = 4;
if ((abs(pt[0].y - pt[1].y) - Tolerance) == 0.0)
    ptr->edge[edge_no].type = HORIZONTAL;
else
    if ((abs(pt[0].x - pt[1].x) - Tolerance) == 0.0)
        ptr->edge[edge_no].type = VERTICAL;
else
    ptr->edge[edge_no].type = NORMAL;
ptr->edge[4].level = level_processor(edge_type, edge_no, pt);

if (a.x == xmin)
    ptr->edge[0].neighbour = NULL;
if (a.x == xmax)
    ptr->edge[2].neighbour = NULL;
if (a.z == zmin)
    ptr->edge[3].neighbour = NULL;
if (a.z == zmax)
    ptr->edge[1].neighbour = NULL;

if (a.z != Z_max)
    ptr1 = rect_processor(b);
if (ptr1 != NULL)
    {
        ptr1->edge[3].neighbour = ptr;
        ptr->edge[1].neighbour = ptr1;
        ptr1->edge[3].level = ptr->edge[1].level;
    }
return ptr;

/*******THIS FUNCTION PROCESSES THE COMPLETE LATTICE***********/

rect_info *lattice_processor(a)
point a;
{
    rect_info *ptr, *ptr1 = NULL;

    Rect_flag = UP;
    ptr = rect_processor(a);
    a.x += X_inc;
    if (a.x != X_max)
    {
        Lattice_flag = DOWN;
        ptr1 = lattice_processor(a);
    }
    if (ptr1 != NULL)
    {
        while ((ptr != NULL) && (ptr1 != NULL))
            {
                ptr->edge[2].neighbour = ptr1;
                ptr1->edge[0].neighbour = ptr;
            }
ptr1->edge[0].level = ptr->edge[2].level;
ptr = ptr->edge[1].neighbour;
ptr1 = ptr1->edge[1].neighbour;
}
if ((ptr != NULL) || (ptr1 != NULL))
    printf("ERROR : Pointer misalignment.\n");
return ptr;
}

******THE CONNECTING SUBROUTINE*******/

void connect(ptr,count,level)
rect_info *ptr;
int count, level;
{
    level_info *ptr1;
    point pt[5];
    int cut = 0, i = 0, low, high, edge_cut[5], horizontal_flag = DOWN;
    int no_of_hori_lines = 0, hori_line[5], j = 0;
    float vert[3];
    pt[0].x = pt[0].y = pt[0].z = INFINITY;
    pt[1].x = pt[1].y = pt[1].z = INFINITY;
    hori_line[0] = hori_line[1] = hori_line[2] = 0;
    while ((cut != count) && (i < count))
    {

low = ptr->edge[i].level.low;
high = ptr->edge[i].level.high;
if (ptr->edge[i].type == HORIZONTAL)
{
    horizontal_flag = UP;
    no_of_hor_lines++;
    hori_line[j++] = i;
}
if ((level >= low) && (level <= high))
{
    edge_cut[i] = TRUE;
    ptrl = ptr->edge[i].level.level_ptr;
    while ((ptrl != NULL) && (ptrl->level_no != level))
        ptrl = ptrl->next;
    if ((ptrl != NULL) && (ptrl->level_no == level))
    {
        pt[cut].x = ptrl->p.x;
        pt[cut].y = ptrl->p.y;
        pt[cut].z = ptrl->p.z;
        cut++;
    }
    else
        printf("ERROR \n");
}
i++;
}
if ((cut != count) && (i == count))
    printf("ERROR \n");
color(level);

switch (count)
{
case 1 : printf("ERROR : only one intersection found\n");
    break;
case 2 : bgnline();
    vert[0] = pt[0].x;
    vert[1] = 0.0;
    vert[2] = pt[0].z;
    v3f(vert);
    vert[0] = pt[1].x;
    vert[1] = 0.0;
    v3f(vert);
endline();
break;
case 3 : bgnline();
    vert[0] = pt[0].x;
    vert[1] = 0.0;
    vert[2] = pt[0].z;
    v3f(vert);
    vert[0] = pt[2].x;
    vert[1] = 0.0;
    v3f(vert);
    vert[0] = pt[1].x;
    vert[1] = 0.0;
    v3f(vert);
endline();
break;
case 4 : if (edge_cut[4])
{
    bgnline();
    vert[0] = pt[0].x;
    vert[1] = 0.0;
    vert[2] = pt[0].z;
    v3f(vert);
    vert[0] = pt[1].x;
    vert[1] = 0.0;
    v3f(vert);
    vert[0] = pt[3].x;
    vert[1] = 0.0;
    v3f(vert);
    vert[0] = pt[2].x;
    vert[1] = 0.0;
    v3f(vert);
    endpoint();
}
else
{
    bgnline();
    vert[0] = pt[0].x;
    vert[1] = 0.0;
    vert[2] = pt[0].z;
    v3f(vert);
    vert[0] = pt[1].x;
    vert[1] = 0.0;
    v3f(vert);
    endpoint();
}
bgnline();
    vert[0] = pt[3].x;
    vert[1] = 0.0;
    v3f(vert);
    vert[0] = pt[2].x;
    vert[1] = 0.0;
    v3f(vert);
endline();
}
case 5 : switch (no_of_hor_lines)
{
    case 0 :
        bgnline();
        vert[0] = pt[0].x;
        vert[1] = 0.0;
        vert[2] = pt[0].z;
        v3f(vert);
        vert[0] = pt[1].x;
        vert[1] = 0.0;
        v3f(vert);
        vert[0] = pt[4].x;
        vert[1] = 0.0;
        v3f(vert);
        vert[0] = pt[3].x;
        vert[1] = 0.0;
        v3f(vert);
        vert[0] = pt[2].x;
        vert[1] = 0.0;
        v3f(vert);
        endline();
        break;
}
case 1 : ;
case 3 : bgnline();
    vert[0] = pt[0].x;
    vert[1] = 0.0;
    vert[2] = pt[0].z;
    v3f(vert);
    vert[0] = pt[2].x;
    vert[1] = 0.0;
    v3f(vert);
endline();
break;
case 5 : for (i = 0; i < 5; i++)
    {
        if (ptr->edge[i].neighbour == NULL)
            {
                bgnline();
                vert[0] = pt[i].x;
                vert[1] = 0.0;
                vert[2] = pt[i].z;
                v3f(vert);
                vert[0] = pt[i+1].x;
                vert[1] = 0.0;
                vert[2] = pt[i+1].z;
                v3f(vert);
                endline();
            }
    }
break;
default : printf("ERROR\n");
break;
}
/****** THE PLOTTING SUBROUTINE ******/

void plotter(ptr)
rect_info *ptr;
{
    rect_info *ptr1;
    int cut_count, draw_level = 0, low_level, high_level, i, low, high;

    ptr1 = ptr;

    /****** GRAPHICS WINDOW IS OPENED ******/
    prerposition(0,1279,0,1023);
    winopen(""");
    color(BLACK);
    clear();

    /****** EACH RECTANGLE IS BEING PROCESSED ******/
    while (ptr1 != NULL)
    {
        low_level = 0;
        high_level = INFINITY;

        /****** FINDS THE LOWEST AND THE HIGHEST LEVELS IN THIS RECT. ******/
        for (i = 0; i < 5; i++)
        {
            low = ptr1->edge[i].level.low;
            high = ptr1->edge[i].level.high;
            if (low_level > low)
            {
                low_level = low;
            }
            if ((high_level < high) || (high_level == INFINITY))
            {
                high_level = high;
            }
        }
    }
}
draw_level = low_level;

/*****EACH LEVEL IS DRAWN IN THIS RECT.******/
while ((draw_level >= low_level) && (draw_level <= high_level))
{
    cut_count = 0;

    for (i = 0; i < 5; i++)
    {
        low = ptr1->edge[i].level.low;
        high = ptr1->edge[i].level.high;
        if ((draw_level >= low) && (draw_level <= high))
            cut_count++;
    }
    connect(ptr1,cut_count,draw_level);
    draw_level += Level;
}
ptr1 = ptr1->edge[1].neighbour;
}

/*****THE MAIN SUBROUTINE******/
main()
{
    point a;
    rect_info *ptr;
    int x_segments = 50, z_segments = 50;

    Ratio = Tolerance / Level;
    X_inc = (X_max - X_min) / x_segments;
    Z_inc = (Z_max - Z_min) / z_segments;
    a.x = X_min;
    a.y = 0.0;
    a.z = Z_min;
Lattice_flag = UP;
ptr = lattice_processor(a);
plotter(ptr);
}
REFERENCES
REFERENCES


