DYNAMIC DATA STRUCTURES FOR REAL-TIME DISPLAY OF DIGITAL TERRAIN MODELS

by

Xian Chunkui

TR92-065 April 1992

This is an unaltered version of the author's
M.Sc.(CS) Thesis

Faculty of Computer Science
University of New Brunswick
P.O. Box 4400
Fredericton, N.B. E3B 5A3

Phone: (506) 453-4566
Fax: (506) 453-3566

Dynamic Data Structures For Real-Time Display Of Digital Terrain Models

by

Xian Chunkui

BSc(CS) — The East China Institute of Technology

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MSc (CS)

in the

Faculty of Computer Science

This thesis is accepted.

Dean of Graduate Studies and Research

THE UNIVERSITY OF NEW BRUNSWICK

March, 1992

Abstract

Data structures to support the rapid display of both regularly spaced and irregularly spaced digital terrain have been designed and implemented. Very large numbers of data points are handled rapidly by dynamically varying the resolution of the display. Gaze-directed viewing is used by forcing the highest resolution to be maintained where the view line intersects the data. The resolution is decreased gradually along the image as the distance from the intersection point increases. The high resolution part of the image can be moved in response to a change in the gaze direction. Using a Silicon Graphics IRIS 4D/85GT, an update speed of 10 Hz was achieved for regularly spaced data of size 800 by 650. For an irregularly spaced data set with 119845 points, an update speed of 5 Hz was achieved using 7 different resolution levels.

Contents

List of Tables

List of Figures

Acknowledgements

I wish to express thanks to my supervisor, Dr. Brad Nickerson, for his guidance and help thoughout this thesis.

Spcial thanks to my wife Tang Hong, and my parants.

Chapter 1

Introduction

Today's highly efficient data sensing and recording devices generate large numbers of very large spatial data sets. Understanding these data sets has become very important. The most natural and efficient way to help human comprehension of large data sets is by visualization [Robertson & O'Callaghan, 1987]. For this reason, many different techniques from computer graphics, image processing, computer aided design, signal processing, and user interface design are being used for large data set comprehension. All of these techniques need proper spatial data structures to efficiently represent the data.

In recent years computer graphics and spatial data structures applied in this area have been highly developed and are becoming more and more successful [Riley, 1990]. Many methods and algorithms have been implemented to generate images as realistically as possible. These effects include perspective views, specular reflection, cast shadows, and ray tracing. From these images people can immediately get a strong idea about the object or surface displayed. If a large amount of data is involved, these images usually take a long time to be generated and the size of the data sets are limited by the display buffer size. To solve these problems, we can display only a part of the data set so that the display speed can be greatly increased. If animation effects are generated, the subset wanted can be displayed immediately. In this way we can see

any part of the large image as soon as we want. This can ease the size limitation by dynamically changing the scene displayed [Smith & Paradis, 1989]. If the data set is a surface of a terrain, this effect can simulate a camera path and produce dynamic views of the terrain for enhancing the visualization.

This thesis was designed to apply computer graphics techniques and spatial data structures to solve problems in comprehending large data sets representing areas of the ocean floor. It is associated with the research work of the ongoing Ocean mapping project at UNB.

The main task for this thesis is to explore techniques and data structures for real-time display of Digital Terrain Models. The techniques include animation, viewpoint variation (variable distance and variable direction), a gaze-directed rendering model, a smoothness-directed rendering model, pseudocoloring, and a shading model. These techniques are applicable to any type of data which can be accessed as a univariate map.

In this chapter, the relevant literature concerning the problems of displaying and manipulating spatial data is discussed. Surface rendering models are reviewed along with concepts related to techniques of animation.

1.1 Digital Terrain Models

For the purpose of this thesis a Digital Terrain Model (DTM) is defined to be a computer data structure which represents the height of a surface over some finite 2D space. Applications of these models arise in digital mapping, resource management, environmental engineering, civil engineering and military operations [Fowler & Little, 1979].

A common form of DTM is the regular grid. For regularly gridded data a coordinate system (e.g. latitude and longitude or universal transverse mercator (UTM)

grid) is chosen, and the terrain is sampled at a uniform rate in both of the coordinates. The result is a series of samples at regular intervals, which is easily represented in a computer by a two dimensional array.

Irregularly spaced data sets are also often used. In some cases, this form better represents the original recorded data. Irregularly spaced data is more difficult to present and display than regularly gridded data. Usually some preprocessing is needed to construct a surface from an irregularly spaced data set.

In this thesis, the data is generated from discrete sounding data, or NED (northing, Easting, Depth) data, which describes the depth of the ocean floor at specific geographical points defined by northing and easting (x and y) coordinates.

1.2 Polygon Mesh

A polygon mesh is a collection of edges, vertices and polygons connected such that each edge is shared by at most two polygons. It can easily and naturally be used to represent a flat surface. It can also be used, although less easily, to represent curved surfaces; however the representation is only approximate. The accuracy depends on the size of the polygon used for the surface rendering. Using more polygons to represent the surface results in a more accurate portrayal of its actual shape [Foley et al, 1990].

1.3 Animation and the Flying Model

To animate is to make a feature appear lively. Although people often think of animation as synonymous with motion, it covers all changes that have a visual effect. It thus includes the time-varying position (motion dynamics), shape, color, transparency, structure, and texture of an object (update dynamics), and changes in lighting, camera position, orientation, and focus, and even changes of rendering technique

[Foley et al, 1990].

The purpose of animation in this thesis is to provide a dynamic view of a realistic terrain surface to enhance visualization for better depiction of surface characteristics. The change of view is performed according to a so-called flying model. Every frame of the animation has a new view with a slightly different viewpoint or view direction to the previous frame . The effect of the animation gives the impression of "flying" over the terrain surface. One method of flying is to give the user a six degree of freedom "bat" which allows full specification of the user's view point direction and velocity [Ware & Osborne, 1990].

1.4 Gaze-Directed Rendering

Since the display of a DTM is for human visualization, taking the property of human vision into account is very natural and reasonable. The visual system is spatially inhomogeneous in that only a small area near the center of the retina is sensitive to fine spatial detail [Zeevi et al, 1990]. The highest acuity area near the fovea centralis occupies about 4 degrees of visual arc, and falls off gradually toward the periphery of the visual field as shown in Figure 1.1. Directing one's gaze at an object consists of rotating either the eye within its socket or the entire head until the object's retinal projection falls on the fovea [Levoy & Whitaker, 1990].

By taking advantage of this variation in retinal acuity, the rendering costs can be reduced [Zeevi et al, 1990]. A high resolution inset image can be generated according to the gaze direction of the eyes. The resolution is decreased gradually along the image to roughly match the acuity of human eyes. If the high resolution area of an image is big enough and is moved quickly enough in response to the change of the gaze direction, the illusion of a full-field high resolution image is obtained. On the other hand if the surface is close enough to one's eyes it can be seen clearly but this clear area is relatively small. If one is far from the surface they can see a large area but not

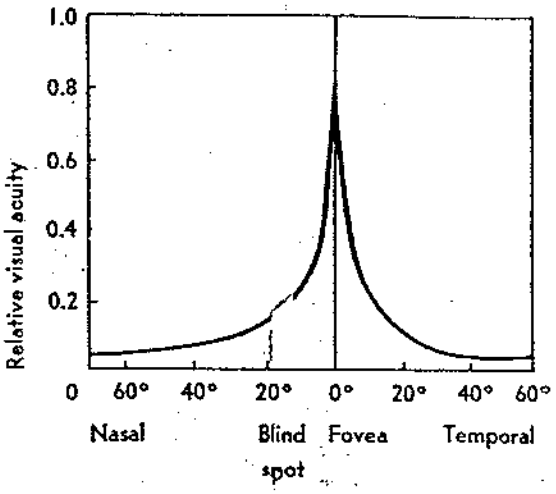

Figure 1.1: The percentages of acuity at different distances from the fovea [Hochberg, 1978].

at a high resolution. Some details are small details, less than the eye's threshold, and cannot be detected as the distance increases [Hochberg, 1978]. To take advantage of this, a more efficient approach is that the resolution chosen should correspond to the distance from the eyes to the surface.

1.5 Perspective Projections

Projections can be divided into two basic classes: perspective and parallel. The distinction is in the relation of the center of the projection to the projection plane. If the distance from one to the other is finite then the projection is perspective. If the distance is infinite the projection is parallel. Figure 1.2 illustrates these two cases.

The visual effect of a perspective projections are similar to that of photographic systems and of the human visual system.

Perspective projection is a fundamental part of computer graphics. This technique

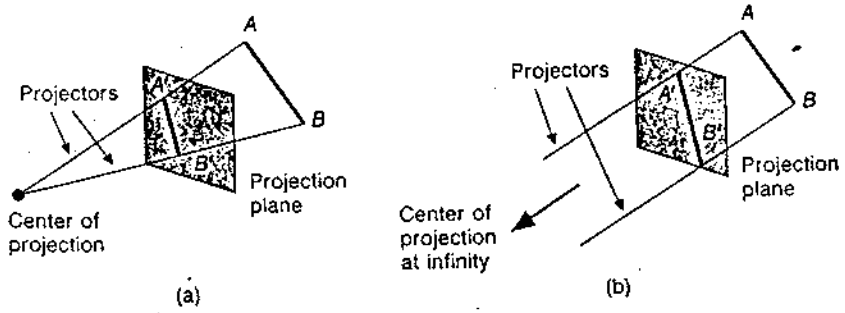

Figure 1.2: (a) Line AB and its perspective projection $A'B'$. (b) Line AB and parallel projection $A'B'$. Projectors AA' and BB' are parallel [Foley et al, 1990].

provides a solution to the mismatch between 3D objects and 2D displays, which transform 3D objects onto a 2D projection plane.

When a 3D object is transformed onto a 2D projection plane, the problem of visible-surface determination, or hidden-surface removal must be considered.

Given a set of 3D objects and a viewing specification, we wish to determine which surfaces of the objects are visible, so that we can display them. In visible-surface determination, surfaces are assumed to be opaque that may obscure other surfaces farther away from the viewer.

Many algorithms have been implemented to solve the problem of visible-surface determination [Folly et al, 1990]. Among these the Z-buffer algorithm is one of the most popular due to its simplicity and speed. Details of this algorithm can be found in Foley et al, 1990.

Chapter 2

The Gaze-Directed Approach with Regularly Spaced Data

In this chapter the gaze-directed approach is discussed. It takes advantage of the limitation of the field of view as people gaze at some place.

2.1 Definitions and Notation

In this approach, most of the important data is considered to be gathered around the point at which the user is gazing, as introduced in section 1.4. The user's eye is called the viewpoint, and the point the user is gazing at is called the view center. The line from the viewpoint to the view center is called the view line (see Figure 2.1).

The value of the highest resolution (at the view center) is decided by the distance from the viewpoint to the view center. If d_v is defined as the distance from the viewpoint to the view center, and R_{max} is defined as the highest resolution, then we have:

$$R_{max} = f_v(d_v) \tag{2.1}$$

As d_v increases, the surface is moved away from the eyes, the surface becomes

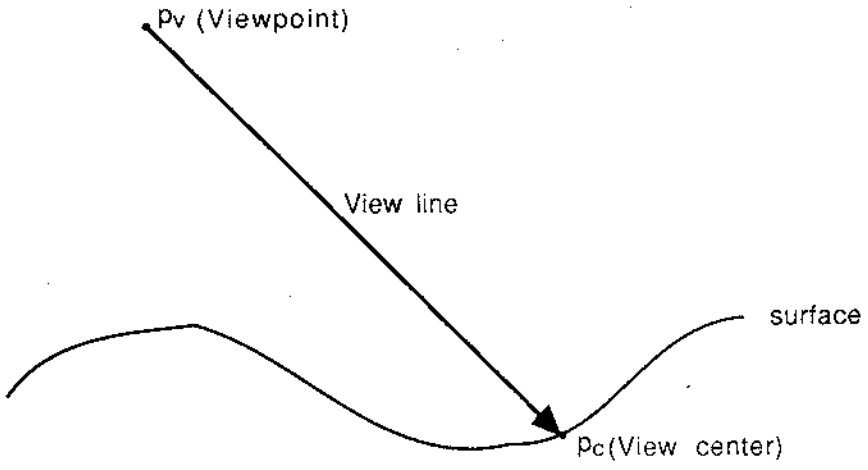

Figure 2.1: The relation between the viewpoint, view line, view center and surface.

fuzzier, and the highest resolution decreases. This function is a monotonically decreasing function.

The resolution on the surface decreases gradually from being highest around the view center to lowest at the farthest side as the distance to the view center increases. If R is defined as the resolution of a point on the surface, and d_c is the distance from the view center to this point, then the function f_c is the relation between R, d_c, and R_{max}; i.e.

$$R = f_c(d_c, f_v(d_v)) \tag{2.2}$$

As d_c (the distance from a point p to the view center) increases, the resolution of the point p decreases; that is, R decreases. On the other hand, if R_{max} increases, the resolution at every point on the entire surface increases. When $d_c = 0$, the point P is at the view center and $R = R_{max}$.

As mentioned before, the data set considered for this chapter is regularly gridded data. It can be represented as a two dimensional array as follows:

$$D = \begin{bmatrix} D_{1,1} & D_{1,2} & \cdots & D_{1,i} & \cdots & D_{1,n} \\ D_{2,1} & D_{2,2} & \cdots & D_{2,i} & \cdots & D_{2,n} \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ D_{j,1} & D_{j,2} & \cdots & D_{j,i} & \cdots & D_{j,n} \\ D_{m,1} & D_{m,2} & \cdots & D_{m,i} & \cdots & D_{m,n} \end{bmatrix} \tag{2.3}$$

Here m is the vertical size, n is the horizontal size, and i and j are indexes which specify the position of a point in the data set. The data can be any real number defined over this grid structure to represent some form of structure. Here the value of $D_{j,i}$ represents the depth of the ocean floor. If a three dimensional virtual environment is defined with x, y, and z, then i corresponds to x, j to y, and $D_{j,i}$ to z. It is assumed neighboring data points have a distance of one unit between them.

In this thesis, the surface is constructed by a triangular polygon mesh. An area of the mesh with high resolution means that this area is constructed with small polygons. If the polygons are small enough (that is, the resolution is very high), the constructed surface will be identical to the original data.

2.2 Viewpoint and View Direction Specification

In the natural world when a person observes an object or a surface they may look at it from different angles and view positions in order to get the best description of the object. In this chapter it is assumed that there is a person who is observing the surface and who can move in the three dimensional environment freely to any place. The image which would be seen is displayed on the screen.

To do this, the person's position and view direction have to be specified. The position is the viewpoint, which is easily specified. It is any point in 3D space. In order to define the view direction, another point in 3D space is defined. It is called the reference point. The view direction then can be defined as a line starting from

the viewpoint and going through the reference point. The line is called the view line. It is shown in Figure 2.2.

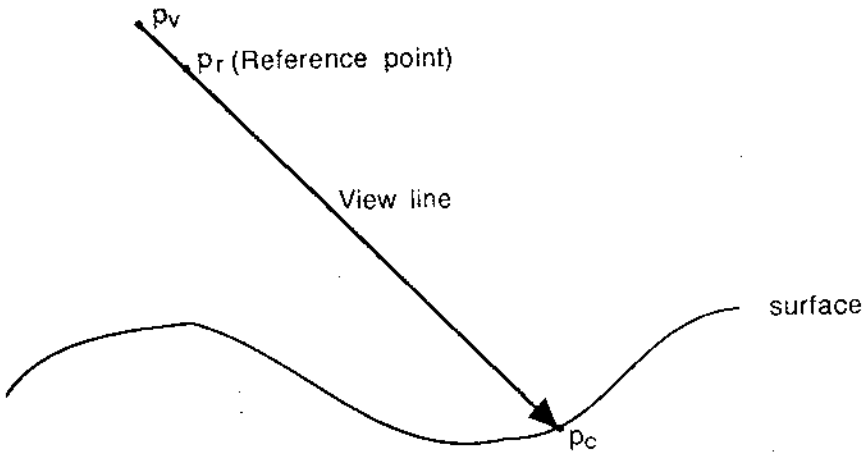

Figure 2.2: The viewpoint and the reference point define the view line.

By specifying different viewpoints and reference points, the assumed person can be located at any position and can look in any direction in 3D space.

Another concept is the field of view. When a person looks at some scene they can only see the portion which is in their field of view.

2.3 View Center Calculation

The data set used in this thesis represents depth of the ocean floor below sea level. If the view line intersects the surface represented by this data set, this intersection point is the view center as defined in section 2.1. Since the surface of the ocean floor is normally irregular, there is no direct way to obtain the view center. To find the view center, points on the view line are compared with points on the surface.

The viewpoint is defined as $P_v = (x_v, y_v, z_v)$, and the reference point as $P_r =$

(x_r, y_r, z_r). These two points determine the view line in 3D space. The line is described by the equation

$$\frac{x - x_v}{x_r - x_v} = \frac{y - y_v}{y_r - y_v} = \frac{z - z_v}{z_r - z_v} \quad or \quad P = u(P_r - P_v), \quad u\epsilon[0, \infty) \tag{2.4}$$

Here (x,y,z) is any point on the view line. If a point $(x_v, y_v, z_0^{'})$ can be found on the surface, z_v and $z_0^{'}$ are compared. If $z_v = z_0^{'}$, the viewpoint is on the surface and the point $(x_v, y_v, z_0^{'})$ is the view center. If $z_v > z_0^{'}$, the viewpoint is below (above) the surface. Otherwise the viewpoint is above (below) the surface. Here we consider the case of $z_v < z_0^{'}$ ($z_v > z_0^{'}$ is similar). Take a point from the view line which is a distance Δd away from the viewpoint (Δd is a small constant). Define this point as (x_1, y_1, z_1). If a point $(x_1, y_1, z_1^{'})$ can be found on the surface, the points (x_1, y_1, z_1) and $(x_1, y_1, z_1^{'})$ are compared. If $z_1 \geq z_1^{'}$, the point from the view line is now on or below the surface. This means that between the two points, there is a point which intersects the surface. Since these two points are very close (depending on how large Δd is), the point $(x_1, y_1, z_1^{'})$ can be considered to be the intersection point. If $z_1 < z_1^{'}$, the point from the view line is still above the surface. A point from the view line is found which has a distance $2 \times \Delta d$ away from the viewpoint, and compared with the corresponding point from the surface. By repeatedly doing the comparison, the intersection point can eventually be obtained (if there is one). If for a point (x_i, y_i, z_i) from the view line, x_i or y_i is outside this surface boundary, the point $(x_i, y_i, z_i^{'})$ cannot be found from the surface. In this case, the view line does not intersect the surface at this point.

For the viewpoint (x_v, y_v, z_v), if x_v or y_v is out of the boundary of the surface, a point $(x_v, y_v, z_0^{'})$ cannot be found from the surface. The above procedure of finding the intersection point cannot be applied directly. In this case, a point (x_k, y_k, z_k) from the view line has to be found. This point is within the boundary of the surface, but the point $(x_{k-1}, y_{k-1}, z_{k-1})$ is not. Starting with point (x_k, y_k, z_k), the above procedure of finding the intersection point can be applied.

One way to reduce comparison times is to shorten the length of the segment to be checked for the intersection point. By considering the maximum and minimum of z values Z_{max} and Z_{min} in the data set, the segment can be shortened. The intersection point can only be on the part of the view line which is between the two planes $z = Z_{max}$ and $z = Z_{min}$. That P_{max} and P_{min} correspond to the intersection of the view line with these two planes, respectively (see Figure 2.3).

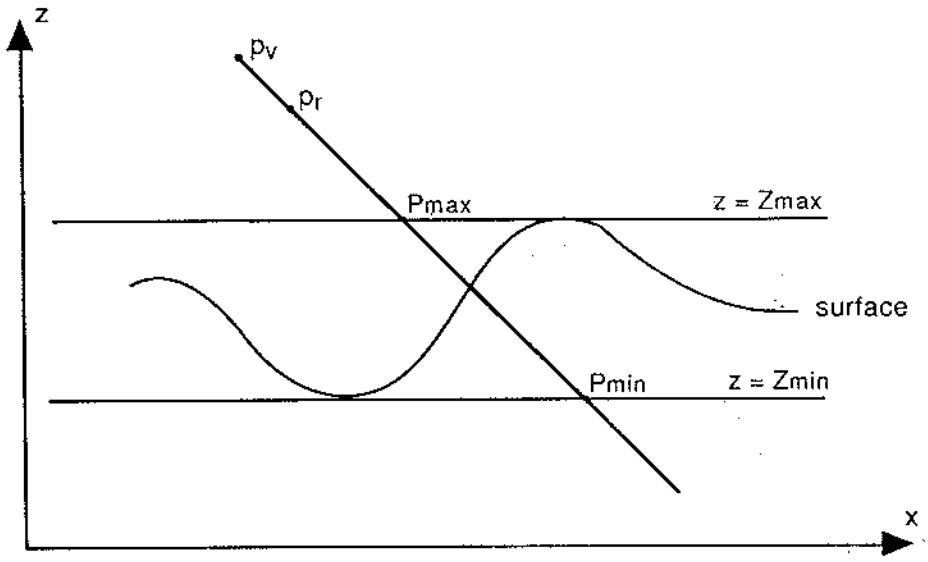

Figure 2.3: Restricting the search for the view center by considering only the z data values in range.

Only the segment $P_{max}P_{min}$ needs to be checked during the comparison. From the figure it can be seen that if the view line goes in the opposite direction, the intersection points P_{max} and P_{min} cannot be found. In this case the intersection point between the view line and the surface cannot be found either.

Another way to shorten the segment to be searched is to consider the surface boundary. Projecting the surface and the view line to the x-y plane, we can get a segment from the projected view line which is overlapped by the surface. It is obvious that the intersection point between the view line and the surface can only be on this

segment (if there is one). This segment is defined as $P_{in}P_{out}$ as illustrated in Figure 2.4.

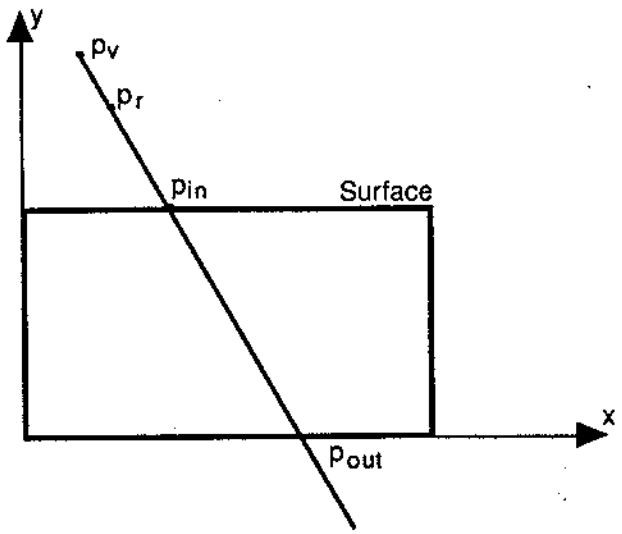

Figure 2.4: The intersection point can only be on the segment $P_{in}P_{out}$ falling inside the data boundary.

The intersection point between the view line and the surface; that is, the view center, can only fall on the line segment which is the common part of the two segments $P_{max}P_{min}$ and $P_{in}P_{out}$. If there is no common part, the intersection point between the view line and the surface does not exist.

If the view center cannot be obtained by finding the intersection point as described above, the surface is represented by the plane $z = Z_{mean}$, where Z_{mean} is the mean z value of the data set. The intersection point between the view line and this plane can be easily obtained (if there is one). This intersection point is defined as the view center. If the view center still cannot be found in this way, it is assumed that the surface is out of the field of view and the surface construction is not necessary.

Finding the existing intersection point is not guaranteed. If Δd is too big, some intersections could be missed as shown in Figure 2.5. In the figure, p_1 is above the

surface. After increasing by Δd, the point p_2 is checked. Since p_2 is still above the surface, the algorithm assumes there is no intersection point between these two points. The intersection of the view line with the surface between these two points is missed.

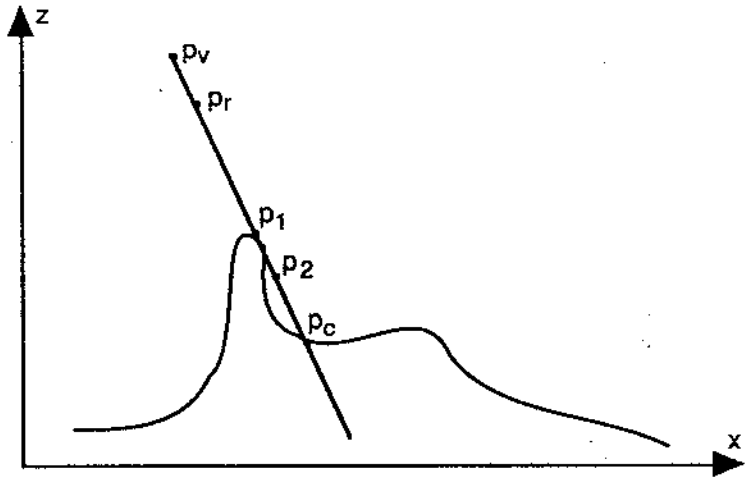

Figure 2.5: The case of missing an intersection point.

The entire algorithm for finding the view center is given in Figure 2.6.

2.4 The Linear Approach

As mentioned before, a triangular mesh is used to represent the surface. The triangular mesh must have variable resolution with the highest resolution around the view center.

In this approach, the entire surface is subdivided into grid cells. Suppose the number of grid cells is $nol \times noc$ (nol, the number of lines of grid cells; noc, the number of columns of grid cells). A triangular mesh is created for every cell. The triangular mesh inside a cell has a unique resolution. The resolution could be different

```
/* The algorithm to calculate the view center.                          */
center(Pv, Pr) /* Pv: viewpoint, Pr: reference point, Pc: view center.  */
{
        /* Get a segment inside which the intersection point falls.     */
        Seg = segment(Pv,Pr,end1,end2); /* Seg is wanted segment and has
                                           end1,end2 as 2 ends.         */

        /* s is a point on the segment which is part of the view line.
           The search is from end1. s will be updated later on till end2. */
        s = end1;

        if(s is above the surface) Above = TRUE;
        else                       Above = FALSE;

        while(Pc == NULL and s <= end2){
                s = s + STEP; /* STEP is a small constant.              */
                if(point s is above the surface and Above == FALSE)
                        Pc = s;
                else if(point s is below the surface and Above == TRUE)
                        Pc = s;
           }
   }

/* Calculate the shortened segment for finding the view center.         */
segment(Pv,Pr,end1,end2)
{
        /* Zmax and Zmin are the max and min z value in the data set.   */
        Pmax = The intersection between the view line and plane z=Zmax.
        Pmin = The intersection between the view line and plane z=Zmin.

        /* Project the view line and the surface into the x-y plane. Pin and
           Pout are two intersections between projected view line and the
           boundary of the projected surface as in Figure 2.5.          */
        Pin = the first intersection between the view line and the surface.
        Pout = the second intersection between the view line and the surface.

        /* The view line does not intersect the surface.                */
        /* supose Pin < Pout and Pmin < Pout                            */
        if(Pmax and Pmin are NULL and Pin and Pout are not NULL){
                end1 = Pin;
                end2 = Pout;
        }
        else if(Pin  and Pout are NULL) end1 = end2 = NULL;
        else{
                end1 = max(Pin, Pmin);
                end2 = min(Pout,Pmax);
        }
   }
```

Figure 2.6: The algorithm for finding the view center.

among different grid cells. If the resolution for every cell is applied properly, a triangle mesh for the entire surface with variable resolution is obtained.

Here we discuss the way to construct a triangular mesh within a grid and how to vary the resolution. The simplest way to create the triangular mesh within a grid is to connect two opposite corners of the grid and get two triangles as shown in Figure 2.7 (a). This triangular mesh has the lowest resolution. If a higher resolution is required for the grid cell, the cell is divided into four same-sized subcells; that is, 4^1. Dividing the subcells into two triangles, this cell is divided into eight triangles as shown in Figure 2.7 (b).

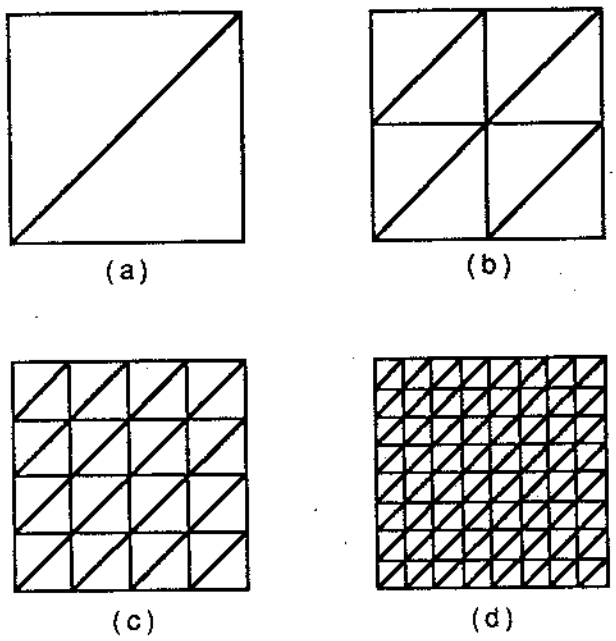

Figure 2.7: A triangle mesh in a grid, with different resolutions for (a), (b), (c), and (d).

If even higher resolution is required, every subcell is further divided into four subcells in the same manner. The total number of subcells is 16; that is, 4^2. In the same way every subgrid is divided into two triangles. The total number of triangles within the grid is 2×4^2. The same subdivision technique can be applied to get a higher resolution within the grid cell. The number of subgrids can be 4^0, 4^1, 4^2, 4^3,

$4^4, \cdots$. The number of triangles, therefore, can be 2×4^0, 2×4^1, 2×4^2, 2×4^3, 2×4^4, $\cdots$. Figure 2.7 (c) and (d) show grids with higher resolutions.

In section 2.1, it is mentioned that x and y in 3D space correspond to the subscripts i and j of the data array D. x and y can only be integers. The edge length of the subcell should be integer too, and the smallest length of an edge, therefore, is 1. To avoid non-integers during the subdivision, the edge length of a grid should be 2^k so that the subdivisions along x and y are always integers until the edge length of the subcell becomes 1. As the edge length of a grid cell is 2^k, the maximum number of subcells is $2^k \times 2^k$; that is, 2^{2k}. When this many subcells are generated, the edge length of the subcell is 1.

On the other hand, when the cell size is 2^k, the grid can only be subdivided $k+1$ times until the edge length of the subcell becomes 1. From these $k+1$ subdivisions, we can get $k+1$ different resolutions. The lowest resolution is called level 0, the second lowest is level 1, $\cdots$, the highest resolution is called level k.

One problem is that when two neighboring grids have different resolutions, cracks will be generated as shown in Figure 2.8 (a) [Tamminen, 1985].

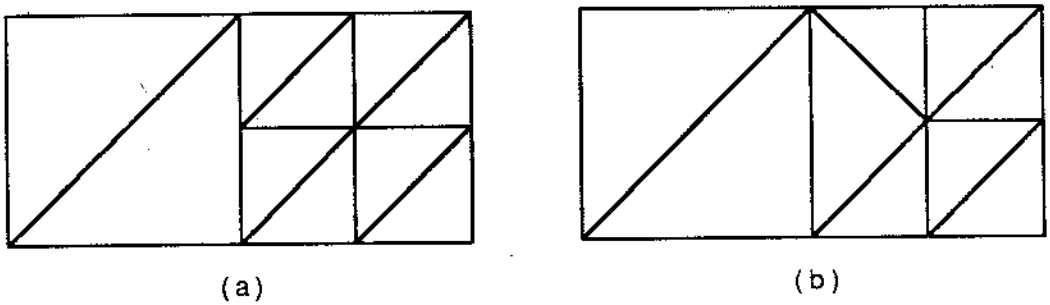

Figure 2.8: (a) Cracks are generated between neighbors with different resolutions. (b) A modified triangle mesh to avoid cracks.

To solve this problem a modification is made during the triangular mesh construction within a grid cell [Herzen, 1987]. We define that between two neighbors, the grid cell with higher resolution is modified to adapt to the lower resolution grid. The

neighboring grid cell is changed as shown in Figure 2.8 (b). From Figure 2.8 (b) it can be seen that no cracks exist between two neighbors. The transition between two grid cells is done smoothly. Here we only consider that the difference between two cells is at most one level; that is, if one cell has 4^i subcells inside it, the neighboring cell has 4^i, or 4^{i-1}, or 4^{i+1} subcells. The above figure illustrates the transition along the left side of a cell. The right, top, and bottom sides can be considered in the same way.

In some cases more than one side of the cell has to do the transition at the same time. When two sides require transition, the two sides can be considered in a similar way. A triangular mesh with two sides in transition is shown in Figure 2.9 (a). From Figure 2.9 (b) it can be seen that no cracks exist. The transition for higher resolution grid cells is similar.

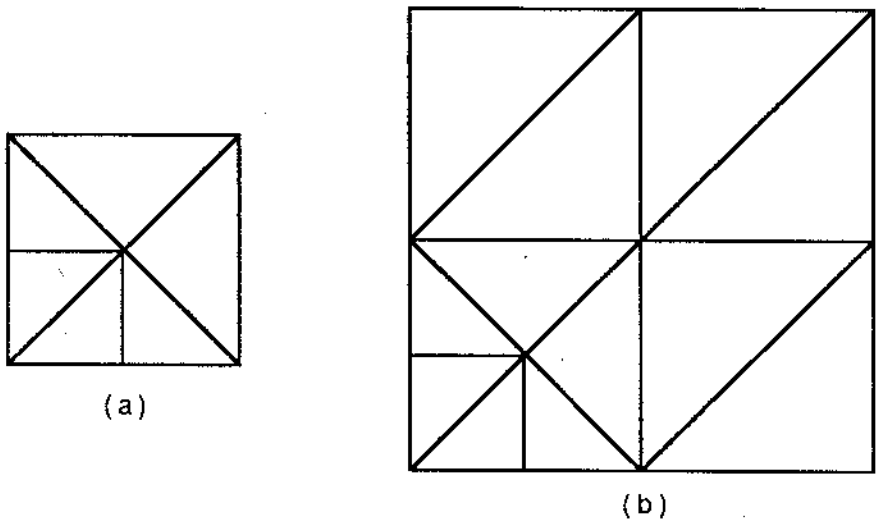

Figure 2.9: A triangular mesh with two neighbors having different resolutions.

It remains to determine the resolution for every grid cell. Firstly, the highest resolution around the view center is decided by the distance between the viewpoint

and the view center as discussed in section 2.1. This distance will decide which level of resolution is applied to the grid cell in which the view center is located. The resolution for every other grid cell is decided by the distance between this cell and the view center. If the distance is bigger than a value d_1, the resolution declines one level from the highest resolution. If bigger than d_2 ($d_2 > d_1$), the resolution declines two levels, and so on until the lowest resolution is reached.

In the real case, 5 resolution levels were used. Level 0 has the lowest resolution with 4^0 subcell in a grid cell; level 1 has 4^1 subcells; and so on. The resolution for a grid cell on the surface can be determined by a function. Here, R_{max} is defined as the resolution level for the grid cell in which the view center is located; d_v as the distance between the view point and the view center; d_c as the distance between the view center and a grid cell on the surface; R as the resolution for a grid cell on the surface. The following is a simple example of the resolution function.

$$R_{max} = \begin{cases} 4 & \text{if } d_v < 200 \\ 3 & \text{if } 200 \leq d_v < 400 \\ 2 & \text{if } 400 \leq d_v < 700 \\ 1 & \text{if } 700 \leq d_v < 1200 \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

$$R = \begin{cases} R_{max} & \text{if } d_c \leq 16 \\ R_{max} - 1 & \text{if } 16 < d_c \leq 32 \\ R_{max} - 2 & \text{if } 32 < d_c \leq 64 \\ R_{max} - 3 & \text{if } 64 < d_c \leq 128 \\ R_{max} - 4 & \text{otherwise} \end{cases} \tag{2.6}$$

If $R < 0$, R defined as 0.

Here, the value of the R_{max} and R is the resolution level, the unit for d_v and d_c is meter.

During the triangle mesh construction, the resolution of the neighboring grid cells must be considered. If a lower resolution neighboring grid cell, or cells is found, the transition has to be applied.

Figure 2.10 is the resolution function for one case of the linear approach, and Figure 2.11 shows the complete algorithm for generating a triangular mesh using the linear approach.

2.5 The Non-Linear Approach

The main objective in this approach is to construct a triangular mesh which has variable resolution due to two factors. The variation in resolution should respond to (1) change of the view center and (2) the distance between the viewpoint and the view center. It should also be very simple and easy to implement in order to generate high speed animation.

Considering all of the above requirements the pattern in Figure 2.12 was designed. Figure 2.12 is a top view of the triangle mesh which is in 3D space. From the figure it can be seen that the smallest triangles are around the view center, where the maximum resolution occurs. As the distance from the view center increases, the size of the triangles becomes bigger; that is, the resolution decreases.

In this triangle mesh every vertex of the triangles is a point from the regularly gridded data set. This triangle mesh is constructed by starting from the view center. The first step is to pick four points to form a square q_0 centered on the view center as shown in Figure 2.13 (a). From this, a triangle mesh is formed with the view center as shown in Figure 2.13 (b). The distance Δq_0 reflects the resolution within that area since it decides the size of the triangles. The second square q_1 is formed in the same way. But Δq_1 is bigger than Δq_0. The reason for this increment is that the area

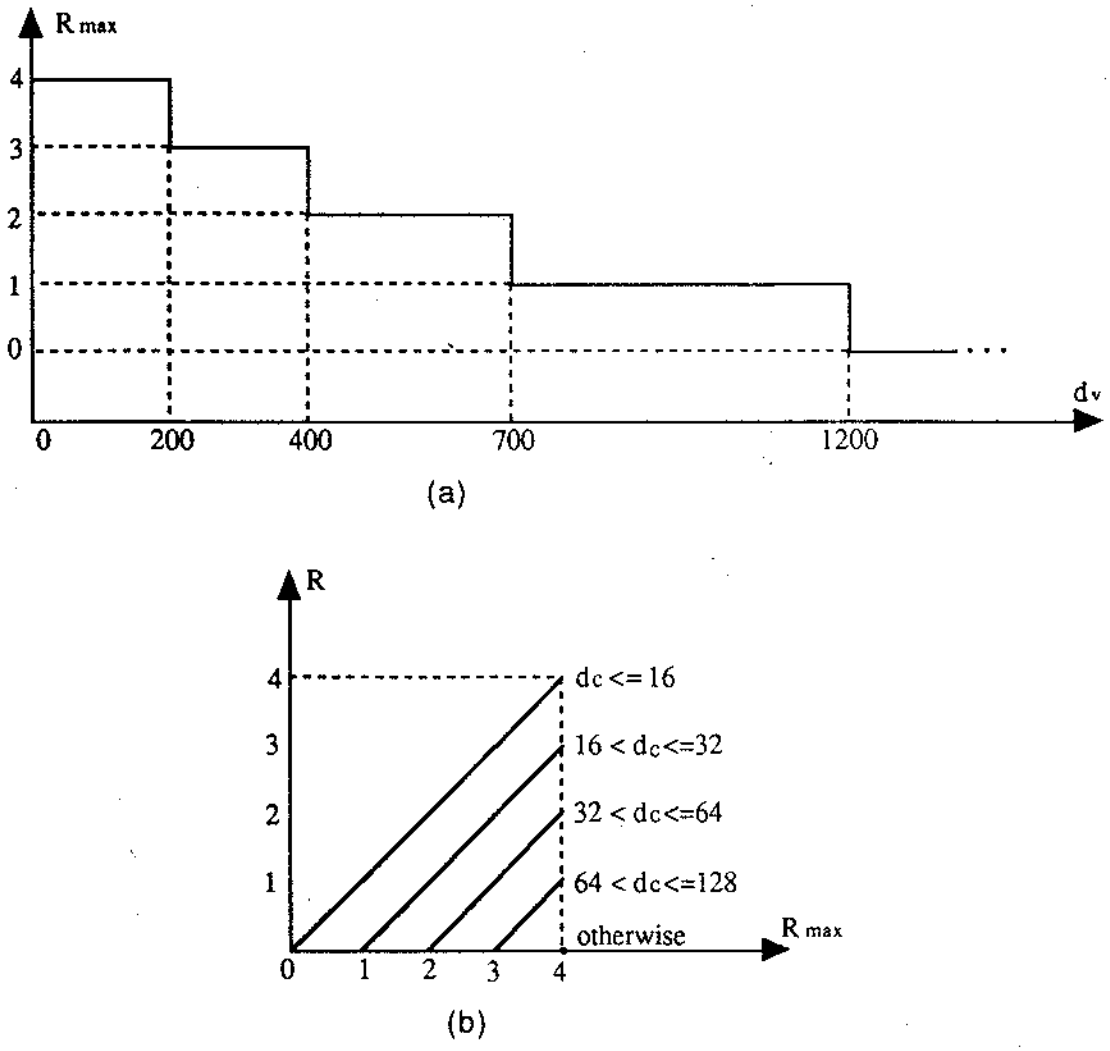

Figure 2.10: (a) The relation between R_{max} and d_v. (b) The relation between R and R_{max}.

```
/* The algorithm for constructing a triangle mesh using the linear
   approach.                                                        */

#define I 16  /* the size of a grid cells    */


mesh(Pc,noc,nol)
    /* Input includes Pc (the view center), noc (=n/I), and
       nol (=m/I) (m, n are defined in equation 2.3).    */
{
        for(i=0; i<=noc; i++){ /* Scan the columns of grid cells.*/
            for(j=0; j<=nol; j++){ /* Scan the lines of grid cells.*/
                /* The distance below means the smallest distance among
                   the distances from Pc to four corners of the grid cell. */
                dist  = distance from Pc to grid i,j;
                dist1 = distance from Pc to grid i,j-1;
                dist2 = distance from Pc to grid i,j+1;
                dist3 = distance from Pc to grid i-1,j;
                dist4 = distance from Pc to grid i+1,j;

                /* resolutions for grid i,j, and four neighbors using
                   equation (2.5)                                    */
                R  = resolution(dist);
                R1 = resolution(dist1);
                R2 = resolution(dist2);
                R3 = resolution(dist3);
                R4 = resolution(dist4);

                /* if 'diff*' is not 0, the transition is needed for grid
                   i,j in the side the correspoding neighbor is located.   */
                diff1 = R1 - R;
                diff2 = R2 - R;
                diff3 = R3 - R;
                diff4 = R4 - R;

                /* Create triangular mesh within grid i,j.                 */
                submesh(i, j, R, diff1, diff2, diff3, diff4);
             }/*for*/
          }/*for*/
}
```

Figure 2.11: (a) The algorithm for generating a triangular mesh using the linear approach.

```
/* The algorithm for constructing a triangle mesh for a grid cell.        */
submesh(i,j, R, diff1, diff2, diff3, diff4)
{
        /* Grid i,j has four corners (x0,y0),(x0,y0+I),(x0+I,y0+I),and
           (x0+I,y0). Here I is the size of the grid. Subdivid grid i, j
           into R*R sub-cells.                                             */
        sI = I/(R+1);

        /* Get subcells into an array.                                     */
        for(i=0; i<R; i++){ /* Column */
                for(j=0; j<=R; j++){ /* Line */
                        subcell[i][j][0] = x0 + sI*j;
                        subcell[i][j][1] = y0 + sI*i; } }
        /* Divide every subcell into two triangles as in Figure 2.7(a) except
           the most outside subcells along the four sides which may need
           transition.                                                     */
        for(i=1; i<R-1; i++) /* Column */
                for(j=1; j<R-1; j++) /* Line */
                        g_2_tri(subcell[i][j]); /* Generate 2 triangles.   */

        for(i=0; i<R; i++) /* The subcells along the left-most side.       */
                if(diff1 != 1) g_2_tri(subcell[i][0]); /* No transition.   */
                else{ /* Transition needed. 3 tri for 2 neighboring subcells
                         as in Figure 2.8 (b).                             */
                        left_transition(subcell[i][0],subcell[i+1][0]);
                        i = i+1; }

        for(i=0; i<L; i++) /* The subcells along the right-most side.      */
                if(diff2 != 1) g_2_tri(subcell[i][L-1]); /* No transition. */
                else{ /* Transition needed. 3 tri for 2 neighboring subcells
                         as in Figure 2.8 (b).                             */
                        right_transition(subcell[i][L-1], subcell[i+1][L-1]);
                        i = i+1; }

        for(j=0; j<L; j++) /* The subcells along the top-most side.        */
                if(diff3 != 1) g_2_tri(subcell[0][j]); /* No transition.   */
                else{ /* Transition needed. 3 tri for 2 neighboring subcells
                         as in Figure 2.8 (b).                             */
                        top_transition(subcell[0][j], subcell[0][j+1]);
                        j = j+1; }

        for(j=0; j<L; j++) /* The subcells along the bottom-most side.     */
                if(diff4 != 1) g_2_tri(subcell[L-1][j]); /* No transition. */
                else{ /* Transition needed. 3 tri for 2 neighboring subcells
                         as in Figure 2.8 (b).                             */
                        bottom_transition(subcell[L-1][j],subcell[L-1][j+1]);
                        j = j+1; }
}
```

Figure 2.11: (b) The algorithm for constructing a triangular mesh for one grid cell.

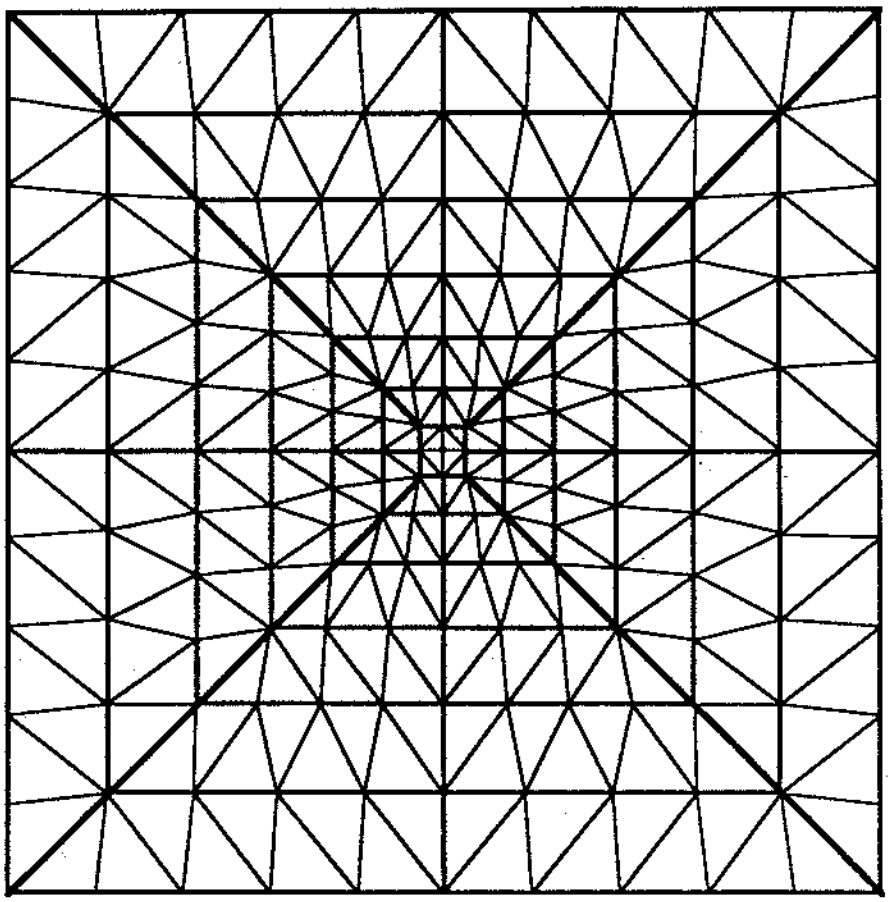

Figure 2.12: A triangle mesh for the non-linear approach.

between q_1 and q_0 is further from the view center.

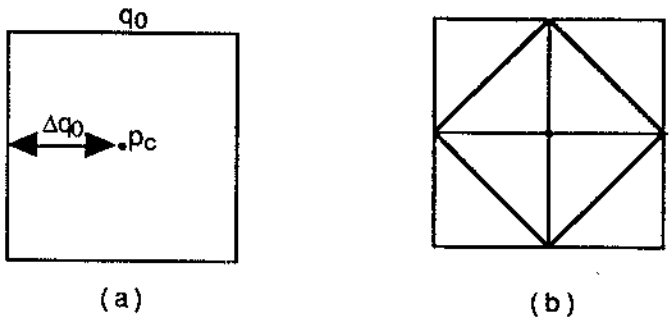

Figure 2.13: (a) The first square. (b) A triangle mesh for the first square.

If we keep generating the squares inside the surface, a sequence of squares, q_0, q_1, q_2, $\cdots$, q_{i-1}, q_i, q_{i+1}, $\cdots$, can be obtained. If Δq_i is given, a square q_i can easily be obtained by extending the previous square q_{i-1} by Δq_i on every side as shown in Figure 2.14. Later on we will discuss a function which generates the Δq_i.

From the figure it can be seen that every square forms a strip with the previous square. Every strip can be subdivided into eight areas. The division technique is shown in Figure 2.15. The boundary of every subdivided area forms a four-line polygon as shown in Figure 2.16. Geometrically the eight polygons are identical. For a polygon generated by q_i, two parallel edges are called e_{i-1} and e_i. The distance between these two edges is Δq_i.

A triangle mesh can be created for every one of these eight polygon. Two rules are followed during the triangle mesh construction. The first one is to divide the edge e_i so that every subdivision is smaller than Δq_i and all subdivisions are equally sized. The number of the subdivisions should be:

$$n_i = \lceil |e_i| / \Delta q_i \rceil \tag{2.7}$$

Here, $|e_i|$ is the length of e_i. n_i is the smallest integer which is bigger than $|e_i|/\Delta q_i$. The second rule is that the edge e_i can only be divided once. Since this edge is shared

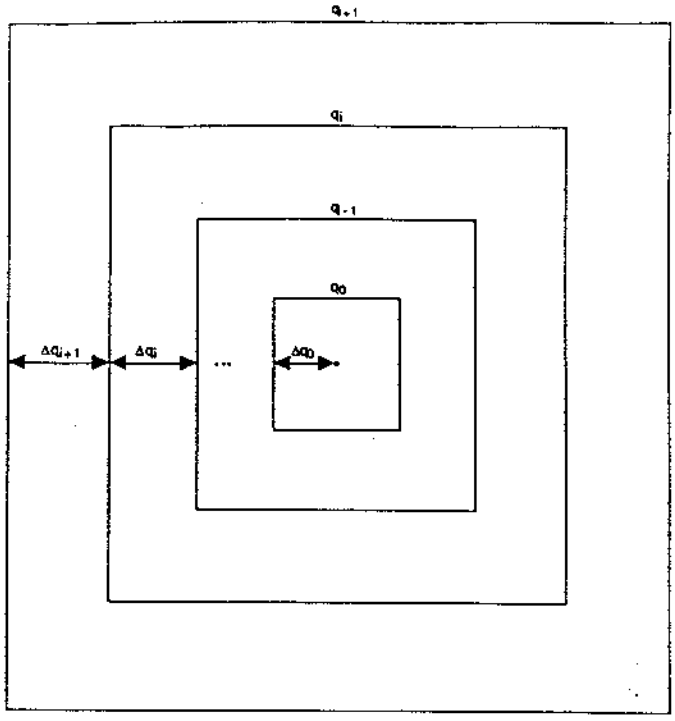

Figure 2.14: A sequence of squares are generated.

by another polygon which is from a strip generated by q_{i+1}, this subdivision will be used for triangle mesh construction in that polygon too. This rule is to prevent cracks during the triangle mesh construction for the entire surface.

Take q_1 as an example. A strip is formed by q_1 and q_0. This strip forms eight polygons. Every one of them has e_1, e_0, and Δq_1 attached as shown in Figure 2.17 (a). Here $|e_0| = \Delta q_0$, $|e_1| = \Delta q_0 + \Delta q_1$. According to the rules, the number of subdivisions for the edge e_1 should be:

$$n_1 = \lceil |e_1|/\Delta q_1 \rceil = \lceil (\Delta q_0 + \Delta q_1)/\Delta q_1 \rceil = 2 \tag{2.8}$$

A triangular mesh for the polygon can be constructed in the way shown in Figure 2.17 (b). Constructing the meshes for all the polygons in the same way, a triangle mesh for the strip formed by squares q_0 and q_1 is created. Together with the triangle mesh in Figure 2.13 (b), the result is shown in Figure 2.17 (c).

This method can be apply to every strip, which eventually leads to a triangle mesh

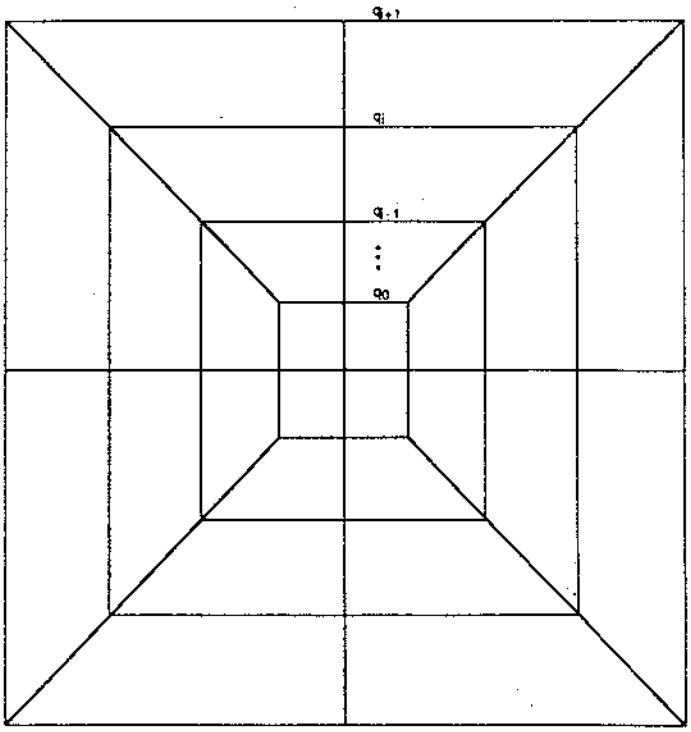

Figure 2.15: The subdivisions for every strip.

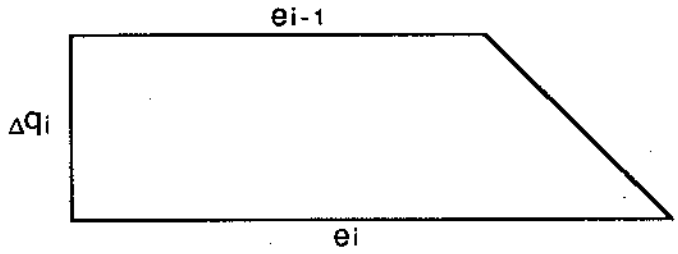

Figure 2.16: A four-line polygon from the strip subdivision.

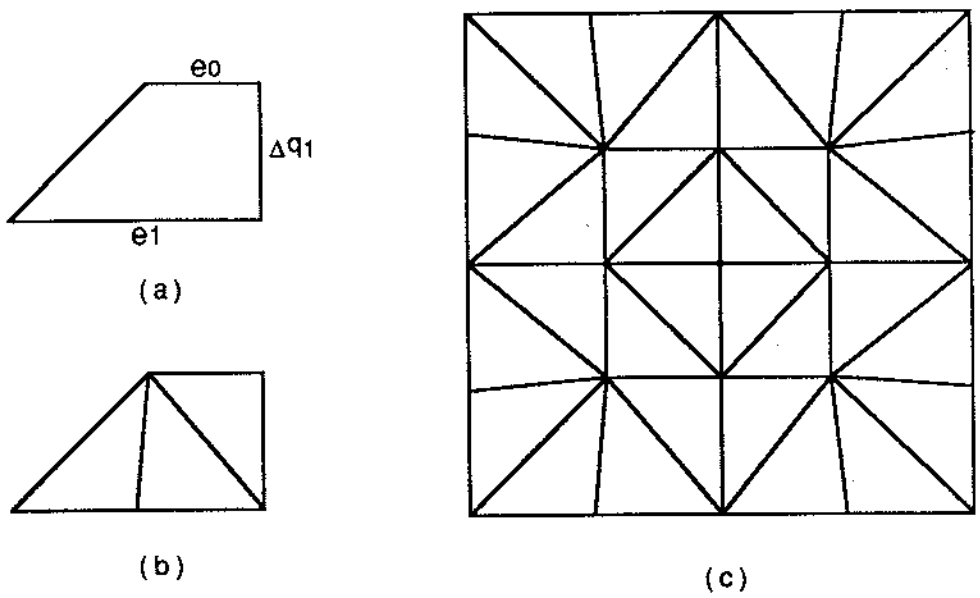

Figure 2.17: The triangle mesh for a strip formed by q_1 and q_0.

for the entire surface.

For the square q_i and q_{i-1}, we have Δq_i, e_i, and e_{i-1} associated as in Figure 2.16. The length of e_i is:

$$|e_i| = |e_{i-1}| + \Delta q_i \tag{2.9}$$

The number of subdivisions is:

$$n_i \;=\; \lceil |e_i|/\Delta q_i \rceil \tag{2.10}$$

Substituting (2.9) into (2.10), we get:

$$n_i = \lceil (|e_{i-1}| + \Delta q_i)/\Delta q_i \rceil = \lceil |e_{i-1}|/\Delta q_i \rceil + 1 \tag{2.11}$$

Suppose e_{i-1} has n_{i-1} subdivisions. We have:

$$n_{i-1} = \lceil |e_{i-1}|/\Delta q_{i-1} \rceil \tag{2.12}$$

Since $\Delta q_i > \Delta q_{i-1}$, we have:

$$\lceil |e_{i-1}|/\Delta q_i \rceil \leq \lceil |e_{i-1}|/\Delta q_{i-1} \rceil \tag{2.13}$$

Equation (2.13) can be denoted as:

$$\lceil |e_{i-1}|/\Delta q_i \rceil + 1 \leq \lceil |e_{i-1}|/\Delta q_{i-1} \rceil + 1 \tag{2.14}$$

Substituting (2.11) and (2.12) into (2.14), we get:

$$n_i \leq n_{i-1} + 1 \tag{2.15}$$

Equation (2.15) indicates that the number of subdivisions of e_i is at most one more than that of e_{i-1}. Here we make another rule during the subdivision. If the number of current subdivisions (n_i) is smaller than that of the previous subdivision (n_{i-1}), then force the current subdivisions to be the same as the number of previous subdivisions; that is, change n_i to be n_{i-1}. In this way it is guaranteed that the number of current subdivisions n_i is equal to or one more than the number of previous subdivisions n_{i-1}. If the current subdivision is equal to the previous one, the triangular mesh for the polygon is constructed in the way shown in Figure 2.18 (a). If it is one more, the triangle mesh is constructed in the way shown in Figure 2.18 (b).

As all the vertices of the triangular mesh are points in the data set, the purpose of the subdivision is to find the proper points to construct the triangular mesh. If a square q_i has some parts outside the boundary of the surface as shown in Figure 2.19 (a), the outside parts do not need to be subdivided since no points from the data correspond to the subdivisions for that part. This outside part is eliminated. The square becomes a rectangle q_i' as in Figure 2.19 (b). As discussed before, the strip area between q_{i-1} and q_i' should be subdivided. The four-line polygons are obtained from this subdivision. In this case we may not get eight polygons. Here, six polygons are obtained as shown in Figure 2.19 (c).

The technique of creating a triangle mesh within a polygon discussed before can be used to create a triangle mesh within every one of the six polygons, although the

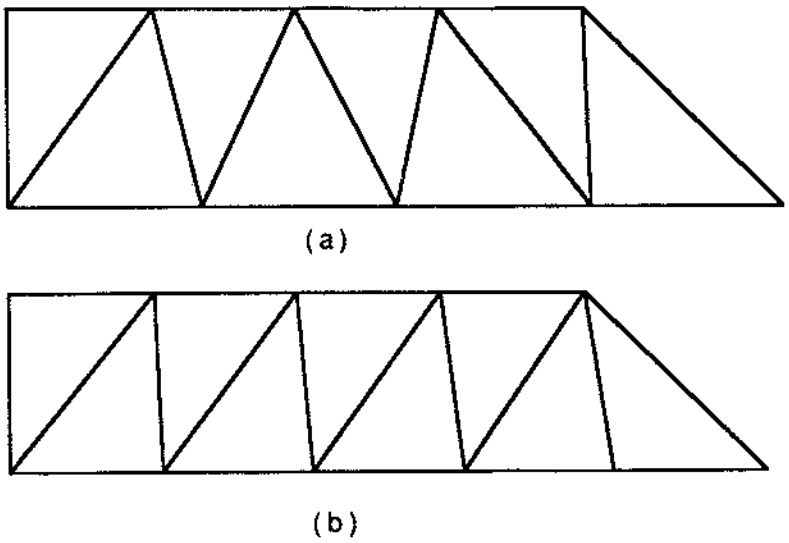

Figure 2.18: A triangle mesh for a polygon. (a) The parallel edges are equally divided. (b) The parallel edges are not equally divided.

shapes of the polygons may not be exactly the same as before. These polygons are still four-line polygons.

Until now we are only concerned about the situation in which the view center is inside the surface. If the view center is far away from the surface and the surface is out of the field of view, no part of the surface is displayed and the surface construction, therefore, is not necessary. If the view center is out of the surface, but not far away from the surface, some part or even all of the surface should be displayed. In this case, the surface is constructed similar to the way discussed before.

A sequence of squares q_0, q_1, q_2, $\cdots$, q_i is generated. All the squares are centered at the view center, as illustrated in Figure 2.20. Every square forms a strip with the previous square. Dividing the strip into eight areas as before, eight polygons are obtained. An area from a strip may not be overlapped or may be overlapped partly by the surface as shown in Figure 2.21. If it is overlapped, the boundary of the overlapped part forms a polygon. This polygon can be implemented in the same way

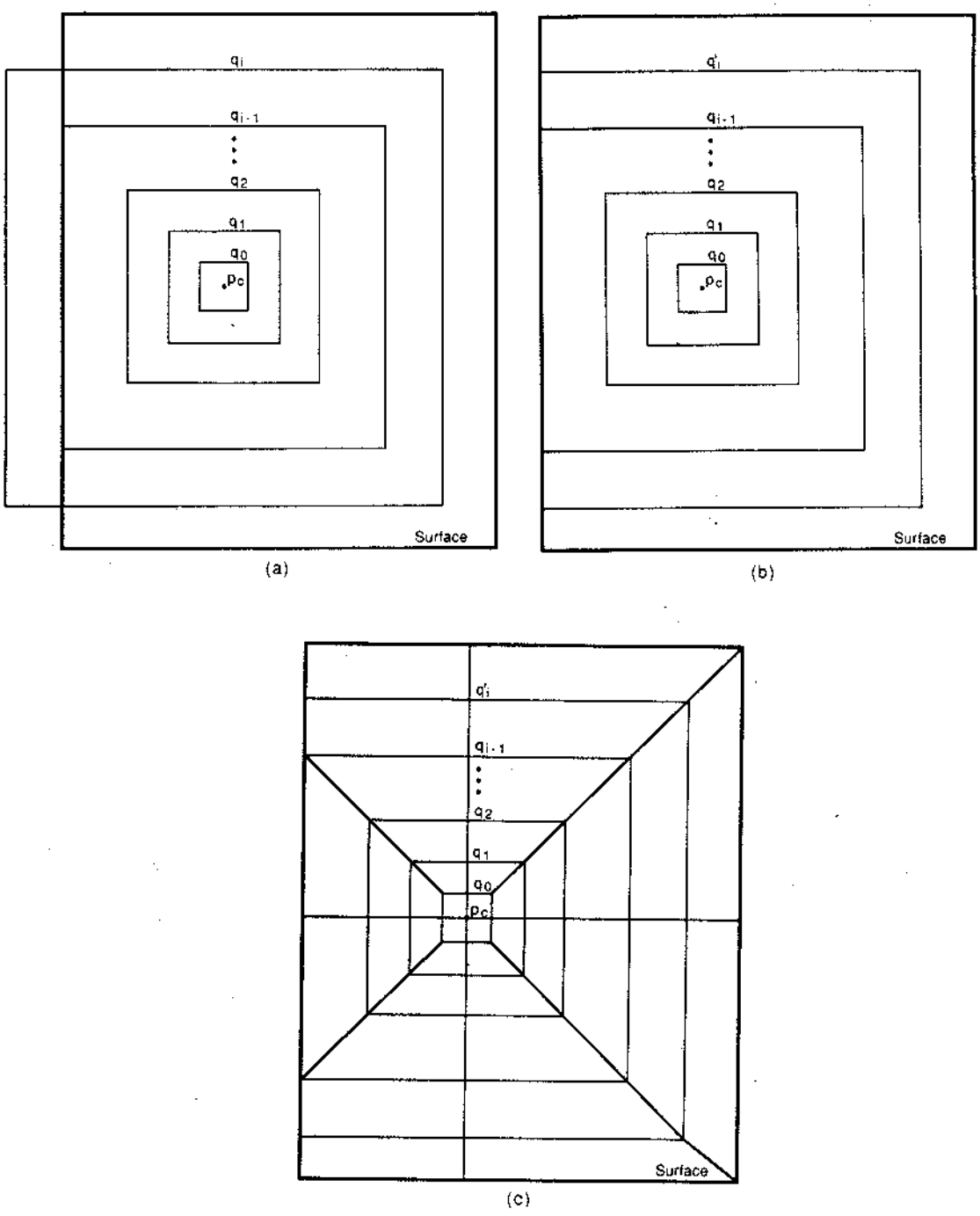

Figure 2.19: (a) Some part of the square is outside the surface boundary. (b) The outside part of the square is eliminated. (c) The subdivision for the entire surface.

as before. The edge of the polygon is subdivided and a triangle mesh is created within it. If an overlapped part is a triangle, it can be considered as a four-line polygon with one line zero sized.

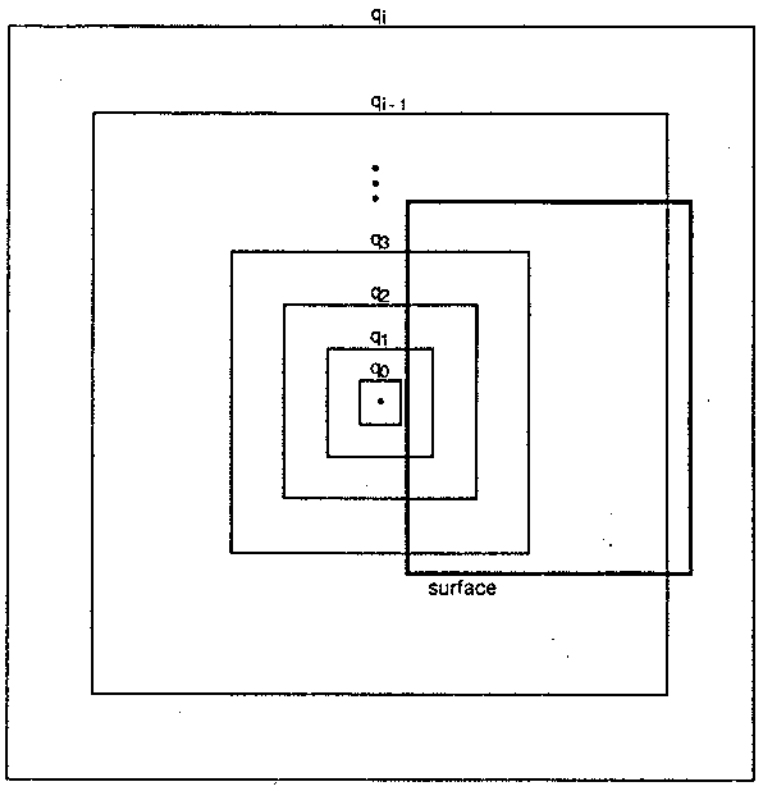

Figure 2.20: A sequence of polygons generated when the view center is outside the data boundary.

The resultant triangular mesh for the regularly gridded Louisbourg data set is shown in Figures 2.22 and 2.23.

2.6 The Resolution Function

One of the main advantages of this approach is the easy-to-define resolution function. The resolution function allow us to calculate the step value Δq_i. From the above discussion we know the step value Δq_i depends on two things; (1) the distance from the view point to the view center d_v, and (2) the distance from the view center to

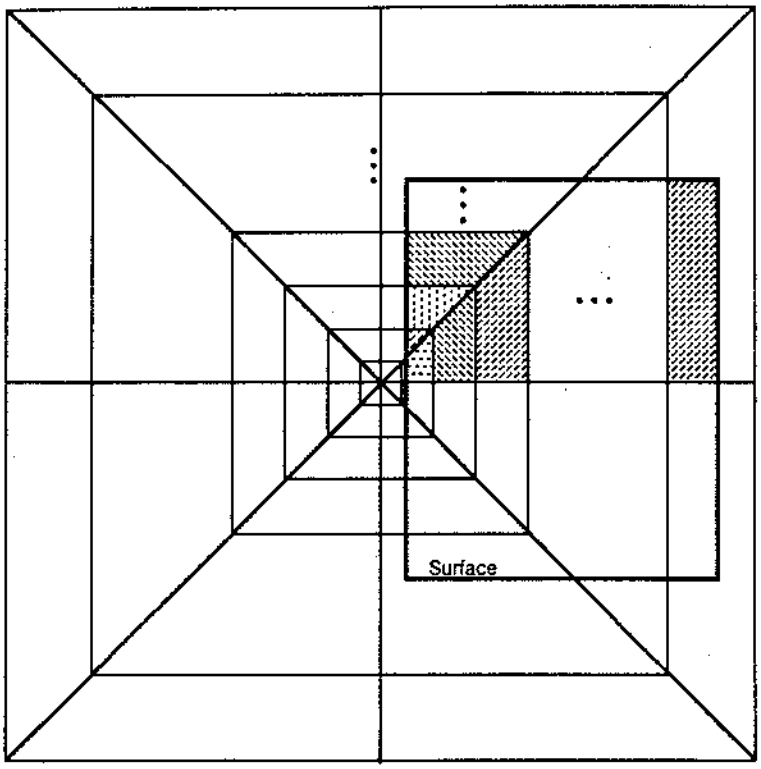

Figure 2.21: Partly overlapped strips

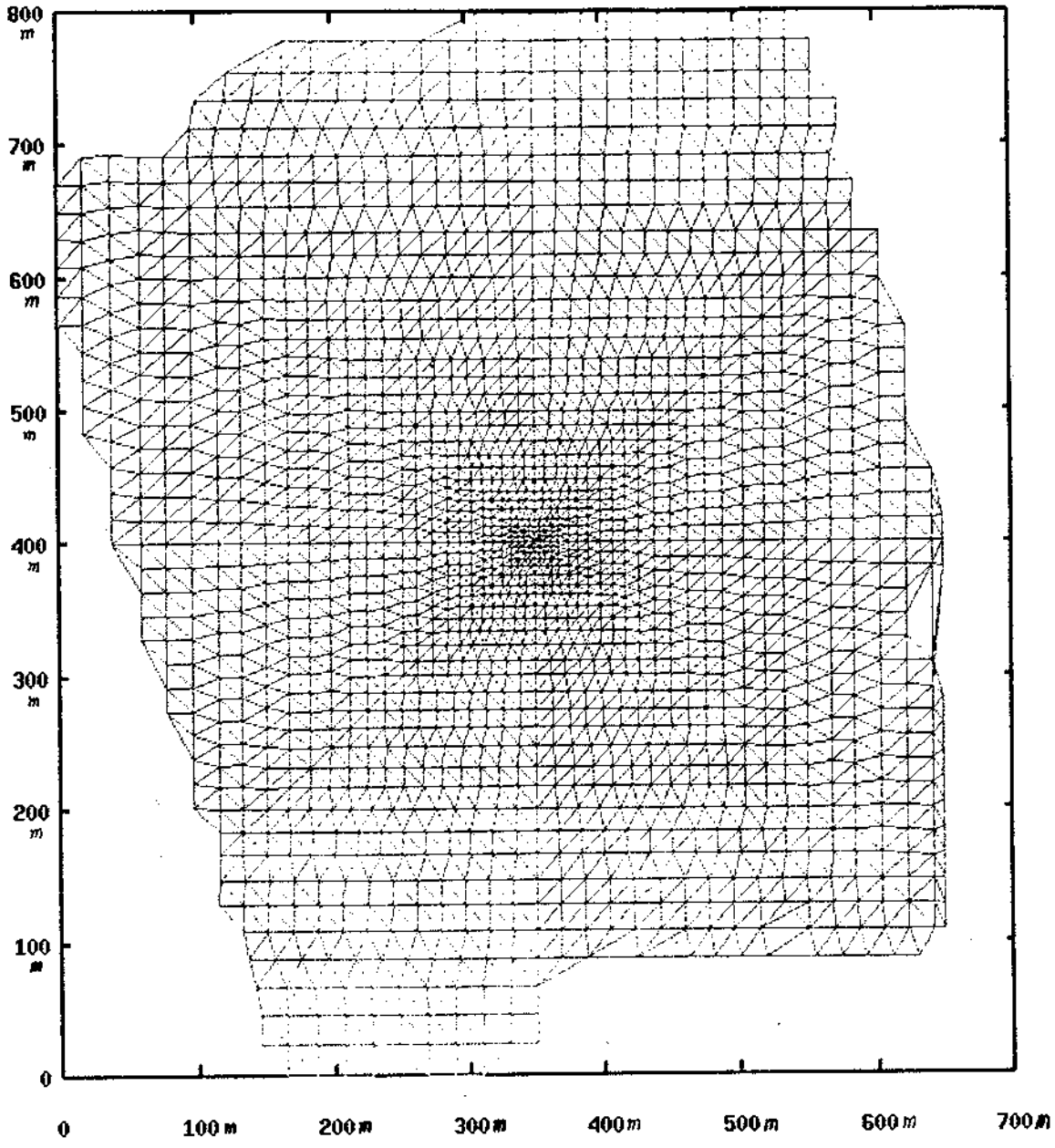

Figure 2.22: One frame of the final non-linear triangular mesh for the regularly spaced Louisbourg data set.

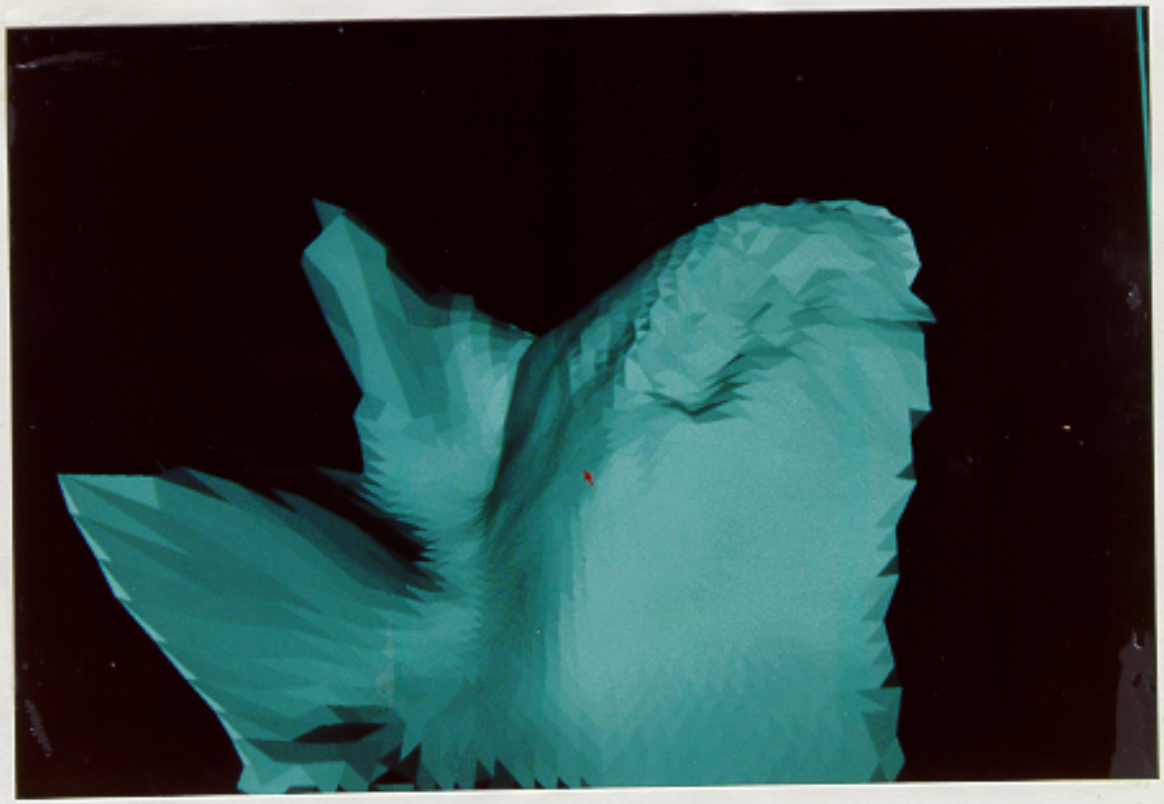

Figure 2.23: One frame of the final result with shading for the regularly spaced data set. The red arrow points to the view center.

q_i, that is, d_c. By knowing the highest resolution, we can decide the resolution of other squares according to d_c. The first part of the resolution function is to decide the highest resolution R_{max} according to d_v. This could be a linear, a non-linear or a step function. The second part of the resolution function is to compute the Δq_i according to the highest resolution R_{max} and the distance d_c.

An example of the resolution function used here is:

$$R_{max} = \begin{cases} I + d_v/30 & \text{if } d_v < 400 \\ I + d_v/25 & \text{if } 400 \leq d_v < 800 \\ I + d_v/20 & \text{if } 800 \leq d_v < 1200 \\ I + d_v/15 & \text{if } 1200 \leq d_v < 1600 \\ I + d_v/10 & \text{if } d_v \geq 1600 \end{cases} \tag{2.16}$$

Here, I is the R_{max} value defined for $d_v = 0$. For the Louisbourg data, I was chosen as 1. The units are the same as used in the data set. Now, each Δq_i can be computed as:

$$\Delta q_i = \Delta q_{i-1} + Inc \tag{2.17}$$

$$\Delta q_0 = R_{max}$$

$$\text{i} = 1,2,3,\ldots$$

Here Inc is a constant increase; in this example it is chosen as 1.

Comparing the above two approaches, the non-linear approach has several advantages. For the non-linear approach, the resolution can be increased or decreased continuously; no special transition algorithms are needed for avoiding cracks.

Chapter 3

The Smoothness Directed Approach

In Chapter 2 the triangle mesh which represents the surface is constructed only by considering the x and y grid coordinates. When an area has small triangles in 2D space, this area is considered to have high resolution. The surface, however, is in 3D space. If a triangle in 2D space is small, it does not necessarily mean that the triangle is small in 3D space. If a large height difference exists among the three vertices, the triangle could be very big. In this sense, using the methods of Chapter 2 an area which should have high resolution is not guaranteed to have high resolution. A large z difference could make the resolution in this area decrease. To solve this problem, the smoothness directed approach is considered. It is based on the gaze-directed approach. If a big z difference exists between two vertices of an edge which is generated by the gaze-directed approach, this edge is subdivided into two subdivisions. The triangle which possesses this edge is subdivided to smaller triangles so that for every smaller triangle the z difference between two vertices of an edge in 3D space is acceptable.

If only one of the three edges has a large z difference, this edge is subdivided into two equally sized edges and the triangle is divided into two as shown in Figure 3.1 (a). In this figure $\triangle ABC$ is the original triangle. Points A and B have a large z

difference. $\triangle ADC$ and $\triangle DBC$ are the subdivided triangles. Point D is in the middle of points A and B.

If two of the three edges need to be subdivided, each of two edges is subdivided into two at the middle of every edge, and the triangle is subdivided into three triangles as shown in Figure 3.1 (b). In the figure $\triangle ABC$ is the original triangle and the edges AB and AC need to be subdivided.

If all three edges in a triangle need to be subdivided, every edge is subdivided into two at the middle of the edge and the triangle is subdivided into four triangles as shown in Figure 3.1 (c). In the figure $\triangle ABC$ is the original triangle.

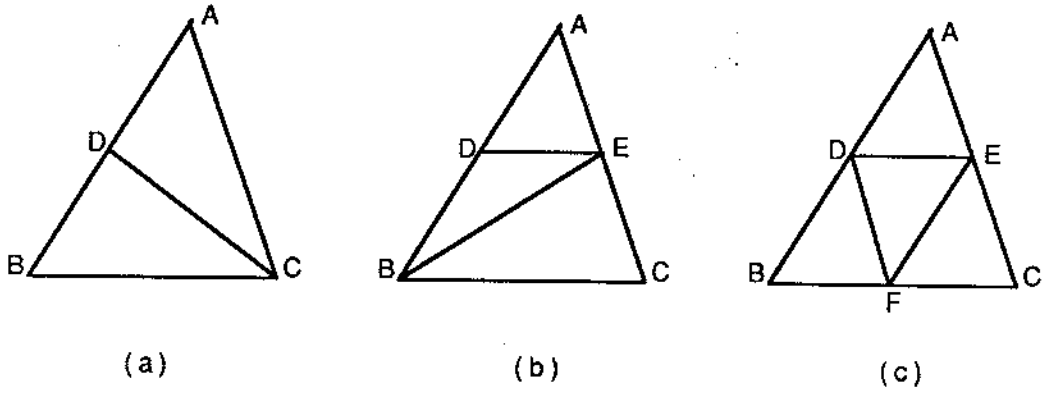

Figure 3.1: (a) One edge is subdivided. (b) Two edges are subdivided. (c) Three edges are subdivided.

All newly obtained triangles in Figure 3.1 have to be checked. If any triangle needs to be further subdivided, the same method can be used. This check is recursively performed until all the subdivided triangles are acceptable.

Since every edge is shared by two triangles, if the edge is not acceptable for one triangle, it is not acceptable for the other one either. The same subdivision is required for both triangles so that cracks can be avoided.

After trying various methods to handle the subdivision, this approach was abandoned. It was discovered that the complexity of building the triangular mesh would lead to too large a time cost for effective near real-time animated display. The principle difficulty was finding the neighboring triangles so that the mean normal vectors could be calculated for a smoothly shaded lighting model.

Chapter 4

The Gaze-Directed Approach with Irregularly Spaced Data

In the real world viewing irregularly spaced data is more important since this is the way the data is normally collected. Ocean bathymetric data collected by swath sounding systems is irregularly spaced. From this data, regularly spaced data can be produced by gridding or binning techniques. It is more accurate to construct a surface from the original irregularly spaced data. It is, however, more difficult [Chen & Guevara, 1987]. In this chapter, the gaze-directed approach with irregularly spaced data will be discussed. The gaze-directed approach with regularly spaced data has already been presented in Chapter 2.

4.1 Overplot Removal and Subsets

In this approach, several subsets are generated from the original data set and every subset has a different number of points in it for the entire surface. Each subset has a different resolution. If the displayed surface is constructed from the data from different subsets, that is, each one of them for only a part of the surface, then a variable resolution for the surface is obtained. The first problem is to get proper

subsets from the original data set.

Points in the original data set may overlap or be very close to each other. Overplot removal is applied to correct this situation [Mason, 1990]. The basic idea is as follows:

> For a picked point p, if there exists another point p' which is closer to p than a constant ϵ, then point p' will be removed and point p is put into the subset.

In order to find the points like p' one way is to search all the points in the overall data set. If there are n points in the data set, searching will require a running time of $O(n^2)$, which is very time consuming. A better strategy is to divide the data set into grid cells aligned with the x and y coordinate axes. Every point is distributed into a cell if it is inside this cell (see Figure 4.1). The z value of the point is ignored. If the size of the grid cell is bigger than ϵ, only the points in the grid cell to which the picked point p belongs and eight neighboring grid cells need to be checked. This is because all points outside of these grid cells have distances from p bigger than ϵ. A picked point and a removed point are flagged, but in different ways. When a point is removed it will never be picked. On the other hand, if a point has been picked it will never be removed. This procedure generates a subset, called P_0. The value of ϵ depends on the smallest distance allowed between points.

The same grid cell technique can be used for generating the other subsets. All points in P_0 are distributed into grid cells. For a point in P_0 the distances to points in the same grid cell and eight neighboring grid cells are calculated to find the smallest distance. If no other points are found in these grid cells, all the neighboring grid cells of the neighboring eight grid cells will be checked. This will have sixteen gird cells involved. This can be recursively performed until a smallest distance is found. This smallest distance is denoted as d. Multiply d with a ratio r ($r > 1$), and let $d' = d \times r$. Using d' as a radius and the picked point as the center, a circle is drawn. Besides the center point, at least one other point is inside the circle, that is, the point with which the smallest distance is found. If the point or points inside the circle have never

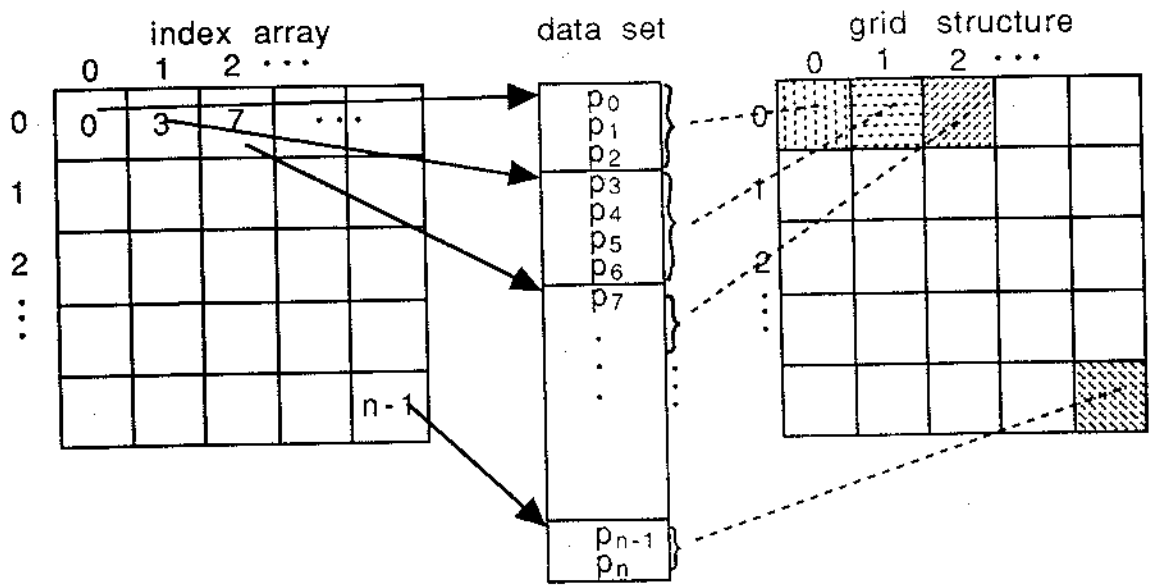

Figure 4.1: Illustration of the grid cell technique.

been picked before, they will be removed. For every point a different d and d' will be obtained, but the value of r is constant. By doing this for all points another subset is generated, called P_1. The value of r depends on how many points are expected to be removed. If r is big, more points will be removed. In the same way, other subsets P_2, P_3, P_4, P_5, P_6, and P_7 are generated and $P_7 \subset P_6 \subset P_5 \subset ... \subset P_0$. The algorithm is described in Figure 4.2.

The advantage of this method is that it can keep the general pattern of the distribution of the data set. When a point is in a high density area and it is picked, the smallest distance between this point and other points will be small, that is, the d is small, and d' $(= d \times r)$ is small as well. This indicates that the area "belonging" to this point is small in the high density area and more points will be picked to form the subset. In this way the percentage of removed points from a high density area and from a low density area is kept as similar as possible. This can be seen in Figures 4.3 and 4.4. Details of the number of points in each data set are given in Chapter 5.

```
/* Algorithm to generate a point subset.                                    */

subset(p,index)

int p[TOTAL][3];        /*The points.                                       */
int index[LINE][COLUMN];/*The grid array as in Figure 4.1. Total no. of
                        cells = LINE*COLUMN.                              */

{
        int flag[TOTAL];        /*Flag every point in the point array. The value
                                could be DELETED, PICKED, or UNTOUCHED.    */

        /* Scan every point in every grid cell. Here i is the index of line and
        j is the index of column.                                          */
        for(i=0;i<LINE;i=i+1){
          for(j=0;j<COLUMN;j=j+1){
             /* The data set has been sorted such that p[k] is a point
                within a grid indicated by i and j. The k is from index[i][j]
                to index[i][j+1] - 1.                                      */
             for(k=index[i][j];k<index[i][j+1];k=k+1){
                if(flag[k] not equal PICKED and flag[k] not equal DELETED){
                    flag[k] = PICKED;
                    /* Find the nearest point to the picked point by checking
                       the cells near the grid cell indicated by i and j.  */
                     closest_p = closest(p[k],i,j);

                     min_distance = distance between closest_p and p[k];
                     radius = min_distance*RATIO;

                     /* Check all the grid cells which are totally or partly
                        overlapped by the circle which has picked point as the
                        center and 'radius' as radius. If a  point is within
                        the circle and the related flag value is 'UNTOUCHED',
                        delete this point.                                 */
                     delete(radius,i,j,k);
                }/*if*/
             }/*for k */
          }/*for j*/
        }/*for i*/
}
```

Figure 4.2: Pseudocode for generating a point subset.

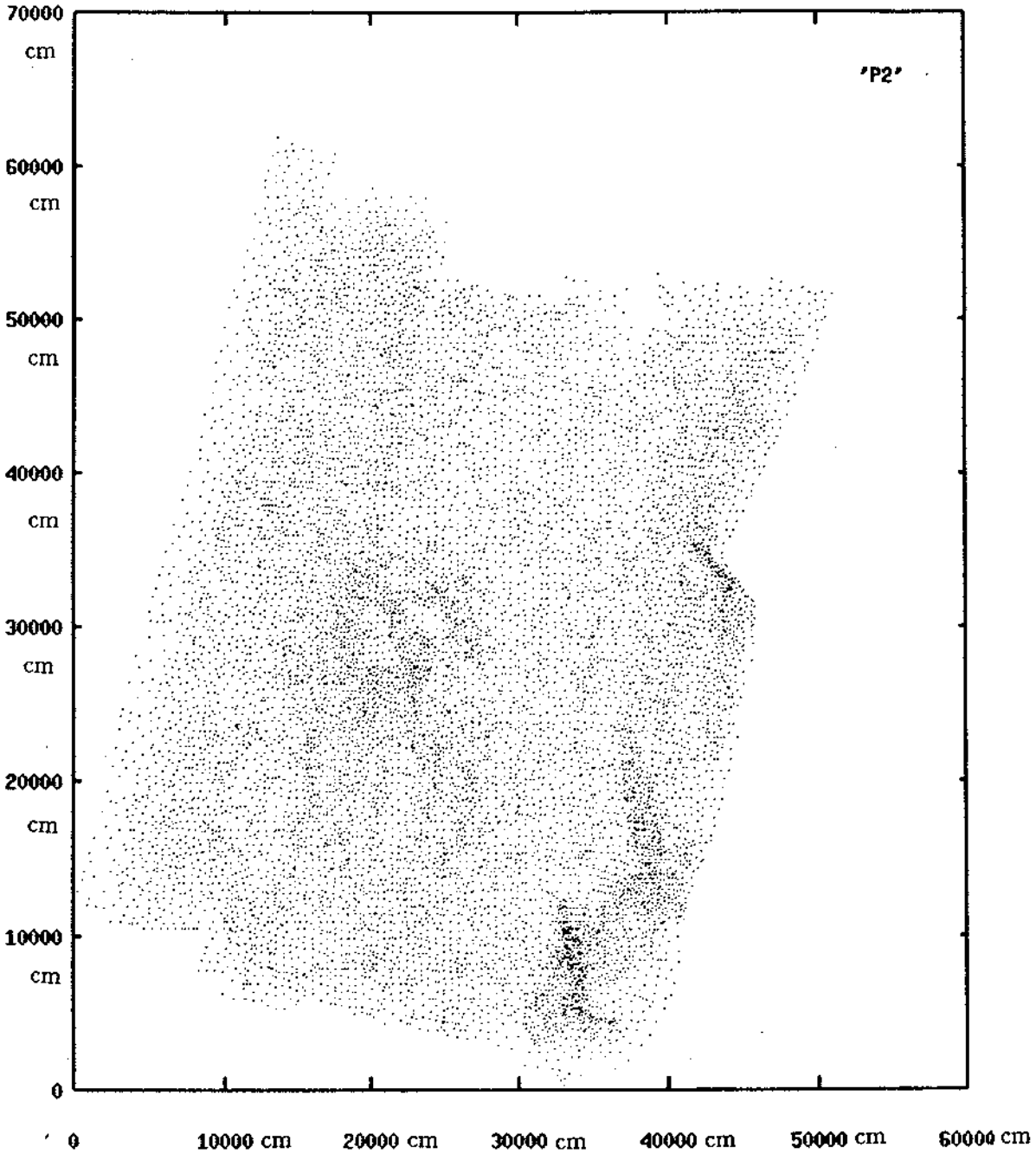

Figure 4.3: Distribution of selected points for subset P_2 with $r = 1.2$.

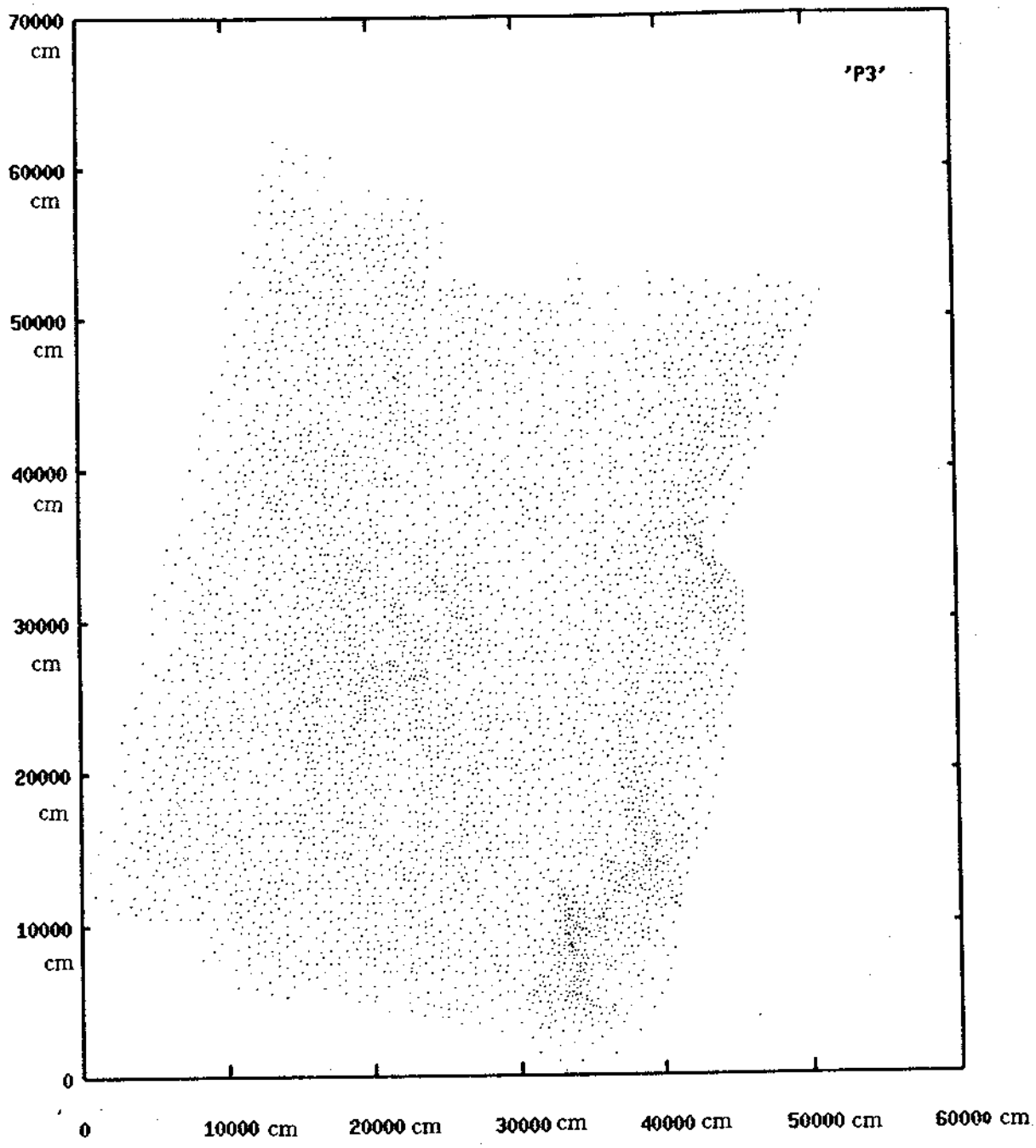

Figure 4.4: Distribution of selected points for subset P_3 with $r = 1.2$ (from P_2).

One problem arose in this process. A subset would lose data coverage on the right side of the boundary. The reason for this "shrinkage" can be seen in Figure 4.5. From the diagram it can be seen that all the data in the right-most grid cell is

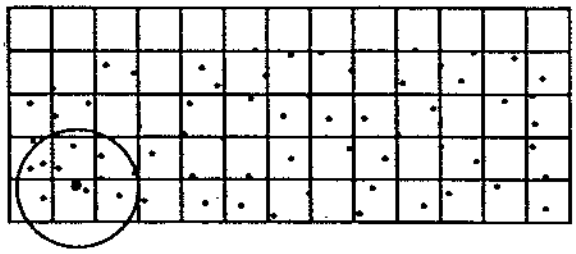

Figure 4.5: The cause of a subset coverage shrinkage.

removed, and data coverage is reduced. If this is not prevented, the subset P_7 may lose a significant amount of coverage. To solve this problem, the order of picking points can be changed. Instead of going from left to right in a row, the right-most cell and the left-most cell are considered first. In this way the coverage loss can be prevented.

4.2 Triangle Networks

In the previous section, eight irregularly spaced data sets P_0, P_1, ..., P_7 were obtained. As for the regularly spaced data, triangle networks are used to construct the surface. Since irregularly spaced data sets are used, the network is triangulated irregular network (TIN) (McKenna, 1987). From irregularly spaced data sets, there are several ways of generating a triangular mesh. Elfick [1979] mentions four different methods. The most common pattern of triangles used for DTM formation is the Delaunay triangulation. This section describes the Delaunay triangulation, and explains the algorithms used to create one from an irregularly spaced set of data points.

4.2.1 Definition of a Delaunay Triangulation

The Delaunay triangulation can be defined as:

> " ... that instead of showing the regions surrounding points it is also possible to connect the neighboring points, or Thiessen neighbors, to produce the dual of Thiessen polygons, the Delaunay triangulation." [McCullagh and Ross, 1980].

Thiessen polygons can be defined as:

> "Given a set of points P_i in a plane, its characteristics are such that each point P_i is surrounded by a convex polygon. All points within this polygon are closer to P_i than any other points in the set P_i."[Mortenson, 1985].

This is shown in Figure 4.6.

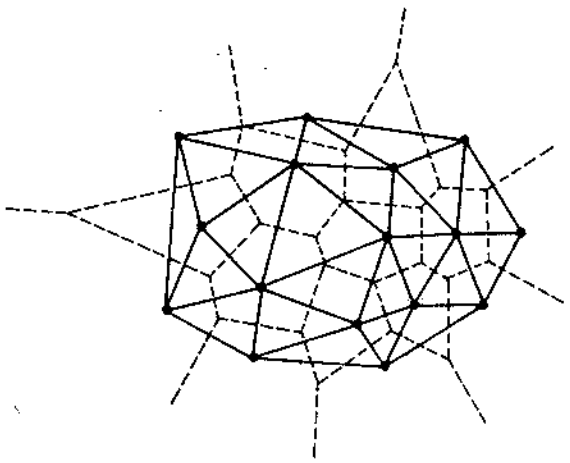

Figure 4.6: A set of points in the plane with their corresponding Delaunay triangulation [Preparata & Shamos, 1985].

The Delaunay triangulation has a number of the desired properties for use as a base for automatic terrain construction. They are:

1. The triangles form a minimal area triangulation. This means that the average size of the triangles which fit all the data points is smaller than any other

triangulation. As the area of triangles decreases, the resolution of the terrain definition increases.

2. The triangulation is unique. No matter where the process of triangulation begins, an identical pattern of triangles results.

3. A Delaunay triangulation consists of well conditioned triangles; most triangles are as close to equilateral as possible [Mason, 1990].

4.2.2 The Algorithm for Generating Delauney Triangulation

Although the above definition suggests that the Delaunay triangulation is derived from Thiessen polygons, the Delaunay triangulation can be produced directly from an irregularly spaced data set. The algorithm is based on the following theory. The third point of a triangle is defined by the circumscribing circle passing through the base line and the third point. The right most/left most neighbor of a base line is found when the right/left part of the circumscribing circle contains no other points [Bjorkee,1988]. Figure 4.7 illustrates this.

The first base line can be formed by picking any point P_1 and finding the nearest point P_2. Line P_1-P_2 is the first base line. Since the triangulation is unique, it does not matter which point is picked as P_1. For a base line A-B (not the first base line), to find the third point for a triangle, points in the data set are tested. If for a point there is no other point in the right/left part of the circumscribing circle, the point being tested is the third vertex of a triangle. This third point is called D (see Figure 4.7). Now there are three lines; that is, line A-B, A-D and B-D. All of them can be used as a base line. Actually, all edges of triangles will be used in turn. Edges become base lines if they are not shared by two triangles and are not along the boundary. Repeating in this way the entire triangulation can be formed.

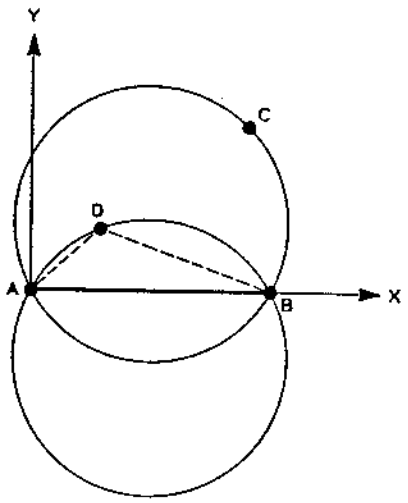

Figure 4.7: Search for the third point of a triangle [Bjorkee, 1988].

The search for the third point is a critical step for the efficiency of the algorithm. For the same reason as generating the subset of P_0, P_1, ..., the data set is divided into grid cells. Every grid cell contains zero or more points. Every time the search only needs to cover the grid cells which are overlapped or partly overlapped by the circumscribing circle. In this way the search time can be significantly reduced. The data structure for the grid cell technique is shown in Figure 4.1.

In the algorithm, the data structure for a triangle, t, includes (1) three vertices, (2) three pointers to three neighboring triangles, and (3) a unit normal vector of the triangle for screen display purposes.

When a third point is needed for a base line to form a triangle, a circle with the base line as diameter is drawn. If the right/left part of the circle contains only one point, it is the third point. If the right/left part of the circle contains no point the radius of the circumscribing circle should be increased, and the center of the circle is in the same side in which the third point is located, as shown in Figure 4.8 (a).

The center is always on line A-B, which is perpendicular to the base line. The radius is increased until one or more points are included, or the radius becomes larger than the maximum radius allowed. The radius is increased by the same increment

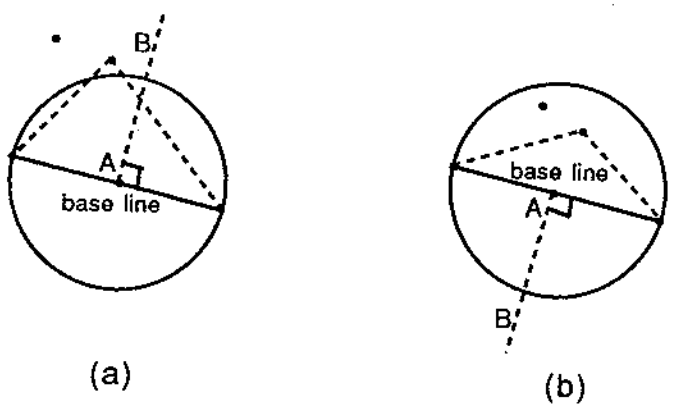

Figure 4.8: (a) The searched point is on the same side as the center of a circle. (b) The searched point is on the opposite side from the center of a circle.

(called STEP) every time. The value of STEP is decided according to the density of the data set. If the radius becomes bigger than the maximum radius, this base line is a boundary of the surface. If one point is found, this point is the third point. Otherwise more than one point is found.

If more than one point is found, the radius has to be decreased. The decrement size is half of the value of STEP. This decrement could lead to the third point or, could be too large such that no point is included, or too small such that more than one point is still found. If the decrement is too large, the radius is increased by half of the previous decrement. If the decrement is too small, the radius has to be further decreased by half of the previous decrement. The radius is repeatedly adjusted in this way until the third point is found. The value of every adjustment is always a half of the value of the previous adjustment.

When the base line acts as diameter and more than one point is found in the right/left part of the circle on the opposite side from line A-B (see Figure 4.8 (b)), the radius must be increased. From now on, the center is always on the line A-B as shown in Figure 4.8 (b). The radius is increased until one or no point is included. The size of the increment is a gain denoted as STEP. If one point is found, this point is the third point. Otherwise no point is found and the previous increment was too

big. A radius decrement is necessary. The size of the decrement is half of STEP. The decrement could lead to the third point, or could be too big such that more than one point is found, or too small such that no point is included. The radius is increased or decreased following the mothed outlined in the paragraph above until the third point is found.

The algorithms described above are shown in Figures 4.9 (a), (b), and (c).

One special case is when two or more points are on the circumscribing circle besides those of the base line as shown in Figure 4.10 (a) [Lee & Schachter, 1980]. In this case, any one of these points can be considered as the third point. In the algorithm it is handled as shown in Figure 4.10 (b). The points on the circumscribing circle are sorted in counterclockwise order. The end points of the base line are at the head. In Figure 4.10 (a) the order is as A,B,C,D,E. The base line picks the next point as the third point, that is, C. Update the base line as A-C, and pick the next point D as the third point. This is repeated until all points in the list are used. Figure 4.10(b) shows the final result. This prevents possible errors as shown in Figure 4.10 (c).

The pattern in Figure 4.10 (c) is caused by picking point C as the third point and ignoring other points. Later on, points C and D could form a line C-D. As the line C-D is used as a base line, the point A could be picked as the third point. This is not a Delauney triangulation and will result in "cracks" if these triangles are displayed as the surface.

4.2.3 Results of Delauney Triangulation

A total of eight triangulation networks T_0, T_1, T_2, ..., T_7 are produced from point sets P_0, P_1, P_2, ..., P_7, respectively. Figure 4.11 illustrates the result of the triangulation for point set P_4. The original point set is the Louisbourg data set observed in Louisbourg Harbour in 1989. The size of the surface is 600 $\times$ 510m in northing and easting, respectively. The depths are between 33cm and 1620cm. Since the depth

```
/* Algorithm to construct Delauney triangulation.                          */

struct T_LIST{              /* A triangle in the incomplete triangle list.  */
        int index;          /* The index of this triangle.                  */
        int p[3];           /* Three points of a triangle                   */
        int neighbor[3];    /* Three neighoring triangle indices.           */
        float norm_vector; /* Unit normal vector of the triangle.           */
        int done;           /* If all above values are found, done.         */
        struct T_LIST *next_t; /* point to next triangle (next node of a list) */
}
struct L_LIST{              /* A base line in the base line list.           */
        int p[2];           /* The two points of a base line.               */
        struct T_LIST *ptr;/* A triangle to which the base line belongs.    */
        int side;           /* The side the third point is on.              */
        struct L_LIST *next_l; /*point to next base line ((next node of a list) */
}
main() {
        extern struct T_LIST * head_T, * tail_T;
                /*Point to the head and tail of the incomplete triangle list.  */
        extern struct L_LIST * head_L, * tail_L;
                          /* Point to the head and tail of the base line list. */
        extern int index=0; /* Indicate the index of a triangle.            */
        struct L_LIST    *tmp_L;  /* temp pointer of base line list          */

        /* Allocate space for triangle list and for base line list.       */
        head_L = malloc(sizeof(struct L_LIST));
        head_L -> next_L = tail_L = malloc(sizeof(struct L_LIST));
        tail_T = malloc(sizeof(struct T_LIST));
        /* Find the first base line near the center of the surface. p0 and p1 are
           the two points of the first base line.                          */
        first_base_line(&p0,&p1);
        /* Put the first base line into the line list twice. One is for a triangle
           on its right side, other for the left side.                     */
        head_L -> p[0] = p0;    head_L -> p[1] = p1;
        head_L -> ptr = -1;     /* The pointer is empty.                    */
        head_L -> side = RIGHT; /* Indicate on which side third point should be.*/
        tail_L -> p[0] = p0;    tail_L -> p[1] = p1;
        tail_L -> ptr = -1;     /* The pointer is empty.                    */
        tail_L -> side = LEFT;  /* Indicate on which side third point should be.*/

        triangle(); /* construct a triangle in right side of the first base line*/
        head_T = tail_T; /* first triangle is created, and is pointed by tail_T */
        triangle(); /* construct a triangle in left side of the first base line */
        /* The first two triangles are neighbors */
        head_T -> neighbor[0] = tail_T -> index;
        tail_T -> neighbor[0] = head_T -> index;

        while(head_T != tail_T) triangle();   /* main loop */
   }
```

Figure 4.9: (a) Pseudocode for for creating a Delaunay triangle from a point set.

```
/* Find the third point for a base line in the head of the base line list and put
   the newly formed triangle into the incomplete triangle list.                  */
triangle()
{
        extern struct T_LIST * head_T, * tail_T;
        extern struct L_LIST * head_L, * tail_L;
        extern int index;
        struct T_LIST *tmp_T;
        struct L_LIST *tmp_L;
        extern int  p3;     /* Store the third vertex of a triangle.             */
        int  status;

        /* The base line used is pointed to by head_L.                           */

        status = 1;
        third_point(status); /* Find the third point and put in p3. */

        /* The base line and the third point form a triangle. It is put into the
           triangle list.                                                        */
        /* allocate memory for new triangle in the triangle list */
        tail_T -> next_T = malloc(sizeof(struct LIST_T));
        tail_T = tail_T -> next_T; /* tail_T points to this new node */
        tail_T -> index = index; /* The index of the new triangle.  */
        tail_T -> p[0] = head_L -> p[0]; /* first point of the new triangle */
        tail_T -> p[1] = head_L -> p[1]; /* first point of the new triangle */
        tail_T -> p[2] = p3; /* first point of the new triangle */
        tail_T -> norm_vector = vector(tail_T); /* the unit normal vector */

        /* This new triangle and the triangle to which the base line belongs are
           neighbors.                                                            */
        tail_T -> neighbor[0] = (head_L -> ptr) -> index;
        if((head_L -> ptr) -> neighbor[1] == EMPTY)
                (head_L -> ptr) -> neighbor[1] = index;
        else{           /* (head_L -> ptr) -> neighbor[2] == EMPTY               */
                        /* This triangle is completed.                           */
                (head_L -> ptr) -> neighbor[2] = index;
                (head_L -> ptr) -> done = DONE;
           }

        /* The two new edges of the triangle are two new base lines. Put them
           into base line list.                                                  */
        /* The first base line of two.                                           */
        tail_L -> next_L = malloc(sizeof(struct LIST_L));
        tail_L = tail_L -> next_L; /* tail_T points to this new node */
        tail_L -> p[0] = head_L -> p[0]; /* The first vertex of the base line.  */
        tail_L -> p[1] = p3;             /*The second vertex of the base line.  */
        tail_L -> ptr = tail_T; /* To the triangle to which this line belongs.  */
```

Figure 4.9 (b) Pseudocode for forming the triangle list and the base-line list.

```
/* If the point other than the two points in the base line is in the
   right side, then LEFT is assigned to this base line, otherwise, RIGHT*/
if(head_L -> p[1] is in right side of tail_L) tail_L -> side = LEFT;
else    tail_L -> side = RIGHT;

/* The second base line of two.                                        */
tail_L -> next_L = malloc(sizeof(struct LIST_L));
tail_L = tail_L -> next_L; /* tail_T points to this new node */
tail_L -> p[0] = head_L -> p[1];
tail_L -> p[1] = p3;
tail_L -> ptr = tail_T;

if(head_L -> p[1] is in right side of tail_L) tail_L -> side = LEFT;
else    tail_L -> side = RIGHT;

/* remove the base line just used from the base line list */
tmp_L = head_L;
head_L = head_L -> next_L;
free(tmp_L);

/* If any triangles on the head of the triangle list are done, write into
   data file.                                                          */
while(head_T -> done == DONE){
        write   head_T -> p, head_T -> neighbor,
                head_T -> norm_vector  into data file;
        /* remove this triangle from the triangle list */
        tmp_T = head_T;
        head_T = head_T -> next_T;
        free(tmp_T);
        if(head_T == tail_T) exit();
   }

 index++; /* one more triangle has been created */
}
```

Figure 4.9 (b) (continued) Pseudocode for forming the triangle list and the base-line list.

```
/* Algorithm to find the third point for a base line. If the third point cannot
   be found, the base line is the boundary.                                    */
third_point(int status)
{
        extern double center[2];          /*The center of circumscribing circle.*/
        extern double radius, pre_radius; /*The radius of the circle.           */
        extern double adjust_size;        /*The size used to adjust the radius. */
        extern struct L_LIST *head_L;
        extern int   p3; /* the third point */
        int   num_of_p; /* the third point, and the number of points found */

        if(status == 1){
                /* The base line acts as the diameter of the circumcircle.     */
                radius = (The length of the base line) / 2.0;
                center = find_center(status,head_L,radius);
                /* Find all points on the correct side of the base line
                   and within the circle. If only one point is found, put it
                   in the external valuable p3.        */
                num_of_p = points_in_circle(head_L,center,radius);
                if(num_of_p == 1) /* found third point, terminate recursive.   */
                else if(num_of_p == 0) third_point(status = 2);
                else third_point(status = 4); /* More than one point is found. */
           }
        else if(status == 2){ /* Radius increases a step. See Figure 4.8 (a).  */
                adjust_size = STEP;
                radius = radius + adjust_size;
                center = find_center(status,head_L,radius);
                num_of_p = points_in_circle(head_L,center,radius);
                if(num_of_p == 1) /* found third point, terminate recursive.   */
                else if(num_of_p == 0)  third_point(status = 2);
                else                    third_point(status = 3);
           }
        else if(status == 3){
                /* The last increment of the radius is too big, or, the last
                   decrement is not big enough. Need decrement. The decrement is
                   the half size of the last adjustment (Figure 4.8(a)).       */
                adjust_size = adjust_size/2.0;
                radius = radius - adjust_size;
                center = find_center(status,head_L,radius);
                num_of_p = points_in_circle(head_L,center,radius);
                if(num_of_p == 1) /* found third point, terminate recursive.   */
                else if(num_of_p == 0)       third_point(status = 33);
                else                         third_point(status = 3);
           }
        else if(status == 33){
                /* The last decrement of the radius is too big, or, the last
                   increment is not big enough. Need increment. The increment is
                   half the size of the last adjustment (Figure 4.8(a)).       */
```

Figure 4.9 (c) Pseudocode for for finding the third point of a Delaunay triangle.

```
            adjust_size = adjust_size/2.0;
            radius = radius + adjust_size;
            center = find_center(status,head_L,radius);
            num_of_p = points_in_circle(head_L,center,radius);
            if(num_of_p == 1) /* found third point, terminate recursive.    */
            else if(num_of_p == 0)       third_point(status = 33);
            else                         third_point(status = 3);
        }
    else if(status == 4){
            /* A radius increment is needed. The center is on the opposite
               side. See Figure 4.8 (b).                                     */
            adjust_size = STEP;
            radius = radius + adjust_size;
            center = find_center(status,head_L,radius);
            num_of_p = points_in_circle(head_L,center,radius);
            if(num_of_p == 1) /* found third point, terminate recursive.    */
            else if(num_of_p == 0)   third_point(status = 5);
            else                     third_point(status = 4);
        }
    else if(status == 5){
            /* The last increment of the radius is too big, or, the last
               decrement is not big enough. Need decrement. The decrement is
               the half size of the last adjustment. See Figure 4.8 (b).    */
            adjust_size = adjust_size/2.0;
            radius = radius - adjust_size;
            center = find_center(status,head_L,radius);
            num_of_p = points_in_circle(head_L,center,radius);
            if(num_of_p == 1) /* found third point, terminate recursive.    */
            else if(num_of_p == 0)       third_point(status = 5);
            else                         third_point(status = 55);
        }
    else if(status == 55){
            /* The last decrement of the radius is too big, or, the last
               increment is not big enough. Need increment. The increment is
               half the size of the last adjustment. See Figure 4.8 (b).    */
            adjust_size = adjust_size/2.0;
            radius = radius + adjust_size;
            center = find_center(status,head_L,radius);
            num_of_p = points_in_circle(head_L,center,radius);
            if(num_of_p == 1) /* found third point, terminate recursive.    */
            else if(num_of_p == 0)       third_point(status = 5);
            else                         third_point(status = 55);
        }
}
```

Figure 4.9 (c) (continued) Pseudocode for for finding the third point of a Delaunay triangle.

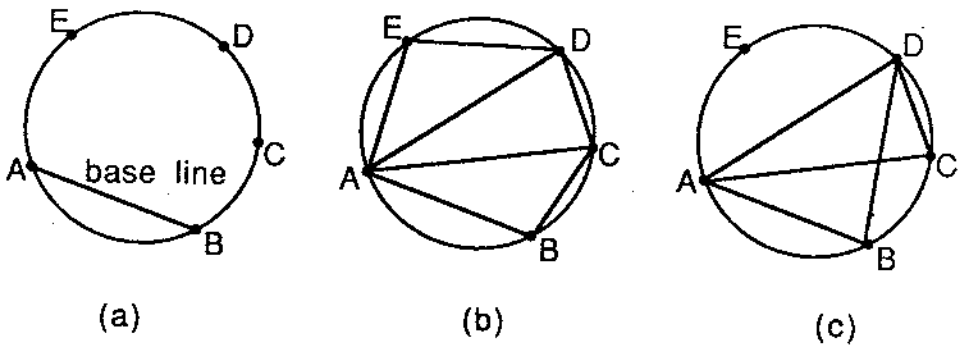

Figure 4.10: A special case in the Delaunay triangulation creation.

difference between points is small compared with the northing and easting, the depth is exaggerated by a factor of 16. This factor was chosen after experimenting with the look of several other factors.

4.3 Surface Construction

The main purpose of this research is to construct a surface with variable resolution controlled by gaze direction as discussed in section 2.1. In this chapter the concepts of the view point, reference point, view line, and view center are the same as introduced in section 2.2. The triangle set T_7 is used to calculate the view center as it has the least number of triangles in it, which makes the calculation faster.

4.3.1 View Center Calculation

If the view line intersects a triangle in 3D space, this intersection point is the view center. When the line and the triangle are projected to a 2D plane they overlap. The reverse theory is not true; that is, if the projected view line and a triangle overlap in 2D space it does not necessarily mean they intersect in 3D space. By getting all triangles overlapped by the view line in x-y space, the most likely intersected triangles are obtained. These triangles are put into a list T_{xy}. Projecting T_{xy} and the view

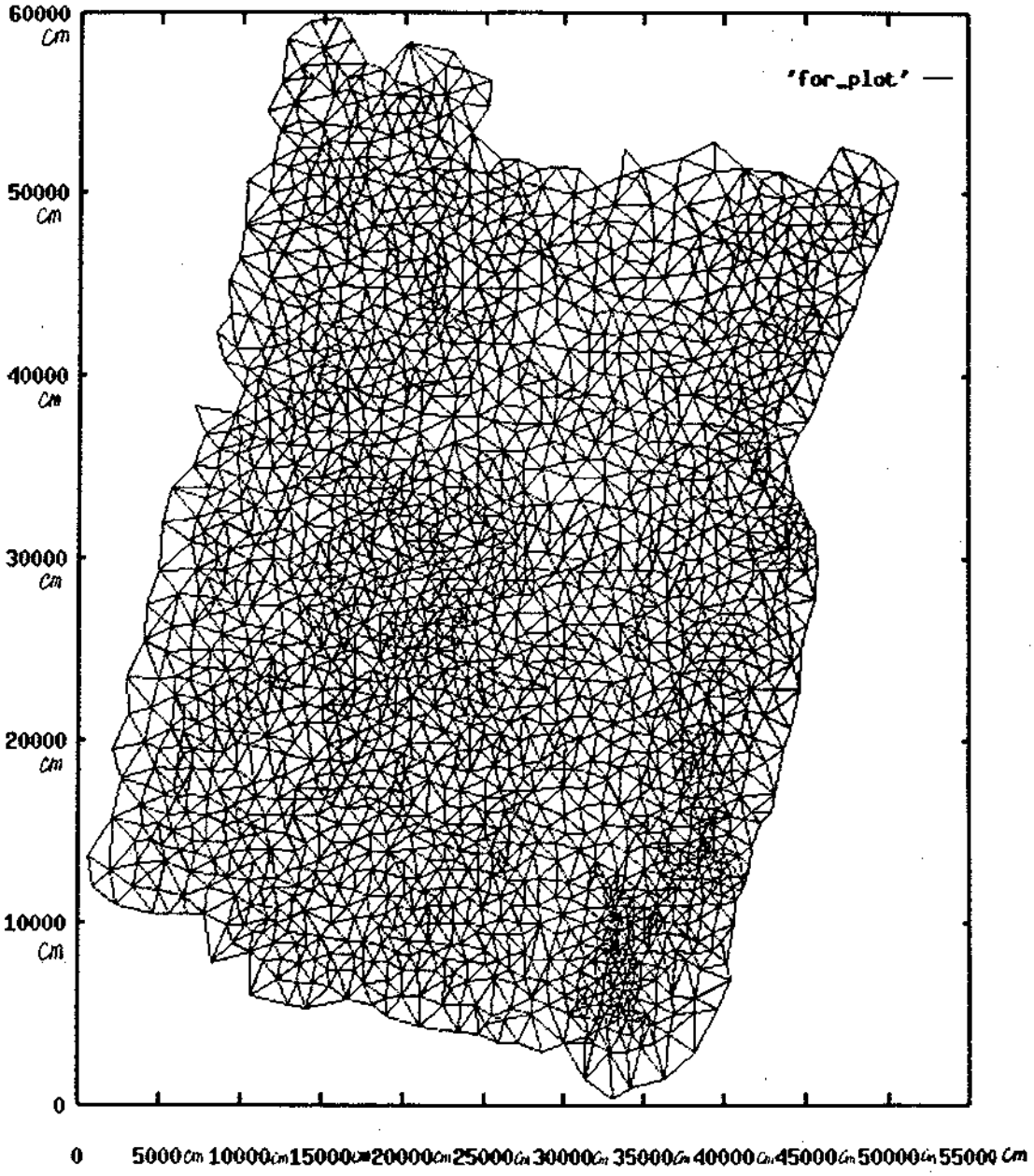

Figure 4.11: Delaunay triangular mesh for point set P_4 (4198 triangles)

line to the x-z plane and getting overlapped triangles, the number of triangles can be further reduced. These triangles form a list T_{xz}. The same method is used with the y-z plane. The third list is T_{yz}. Here, $T_{yz} \in T_{xz} \in T_{xy}$. From the triangles in T_{yz} the intersected triangle and further the intersection point can be found.

Now the problem is how to generate T_{xy} from all the triangles in the data set T_7. If the view line intersects two edges or overlaps one edge of a triangle, this triangle is considered as overlapped by the view line, as shown in Figure 4.12.

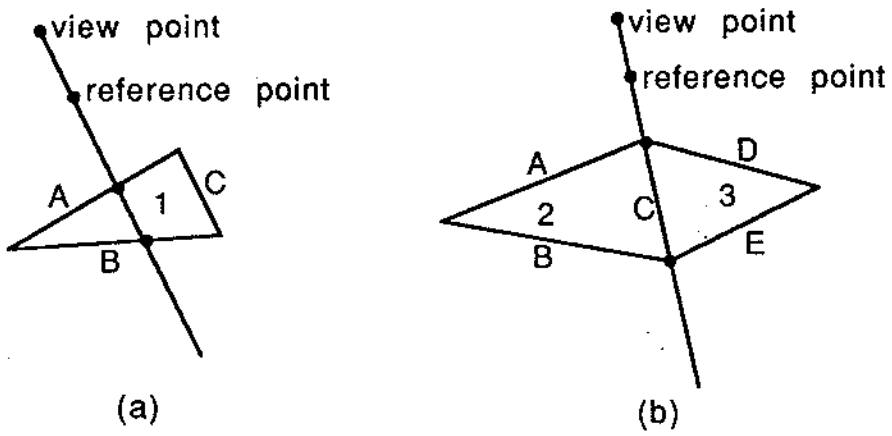

Figure 4.12: The view line intersecting triangles in the x-y plane.

In Figure 4.12, there are three triangles numbered 1, 2, and 3. All three of these triangles are overlapped by the view line. For finding all overlapped triangles, one way is to check all triangles against the view line. This will take too much time. A better way is first to find a triangle which is overlapped by the view line. By checking the neighboring triangles of this triangle, for example, the triangles sharing the edge A and B in Figure 4.12 (a), or the triangles sharing the edges of A,B,D, and E in Figure 4.12 (b), more overlapped triangle can be found. By recursively performing the above process, all overlapped triangles can be found.

One way to find the first triangle is to check the triangles in the data set one by one. The worst case is when all data in the data set must be checked. To avoid this

time consuming process, a better way is to divide the surface into grid cells. Every grid cell could overlap zero, one, or more triangles. Only one overlapped triangle is assigned to a grid cell if more than one triangle overlaps the grid cell. This one is picked randomly.

Projecting the view line into x-y space (z=0), it will at least intersect one grid cell (if an intersection point can be found). If this grid cell has one triangle associated with it, this triangle is checked. If this triangle intersects the view line in 2D space, it is the first triangle. If it does not, its three neighbors (t_1, t_2, t_3) are checked. If none of these three triangles can be considered as the first triangle, the neighbors of t_1, t_2, and t_3 are checked. Repeatedly doing this for all triangles overlapped by the grid cell, eventually the first triangle can be found. The grid cells are sized small enough so that only a few triangles are overlapped, so the first triangle can be easily found. Experiments show that most times the repeat is not necessary. If the grid cell does not have a triangle attached, another grid cell is found which overlap the projected view line. If none of the overlapped grid cells have associated triangles it means that there is no intersection point between the view line and surface.

After projecting the view line and triangles in T_{xy} to the x-z plane and checking every triangle, the data set T_{xz} can be generated.

In the same way, projecting the view line and triangles in T_{xz} to the y-z plane and checking every triangle, the data set T_{yz} can be produced.

By using 3D geometry the intersection point of a triangle and the view line can be easily calculated [Mortenson, 1985]. When an intersection point is obtained, the procedure is terminated and other possible intersection points are ignored. This point is the view center.

On the other hand, if there is no intersection point between the view line and the surface, the surface will be extended infinitely as a plane of $z = Z_{mean}$. The Z_{mean} is the mean z value of all points in point set P_7. The intersection point between the view line and the plane is considered as the view center which is outside of the surface

boundary.

If the view center cannot be found even by extending the surface, the view center is defined as far away from the surface so that the surface cannot be found within the view field.

4.3.2 The Surface Division Function

To construct a surface with variable resolution the different triangle sets T_0, T_1, ..., T_7 are used at the same time. Every set is used to construct part of the area of the entire surface. It is possible that some triangle sets will not contribute to the final surface. The principle is that the area close to the view center has high resolution; that is, the set with high resolution will be employed to construct the area. The resolution also depends on the distance between the view point and the view center. If the distance is small, the resolution is relatively high for the entire surface. For an area on the surface, the resolution depends on the distance between the view point and the view center and the distance between the view center and the area. As either one of these distances decreases, the resolution increases.

If the view center is inside the boundary of the surface, the surface is divided as shown in Figure 4.13.

Here, β_i is the distance between neighboring boundaries. The first square (the most inside one) is the area for data set T_0. The area between the first square and the second one is for data set T_1, and so on. Normally the distance between inside squares and outside ones is the same on all four sides. If one or more edges reaches the surface boundary, the distances between the four sides are no longer the same. Once the distance between the viewpoint and the view center is decided, the division function is known, and the areas for subsets (as shown in Figure 4.13) is determined. If the view center is inside and near the boundary of the surface, it is possible that the area boundary for T_7 cannot reach the surface boundary in some side or sides. In this case the boundary for T_7 is extended to the surface boundary for this side.

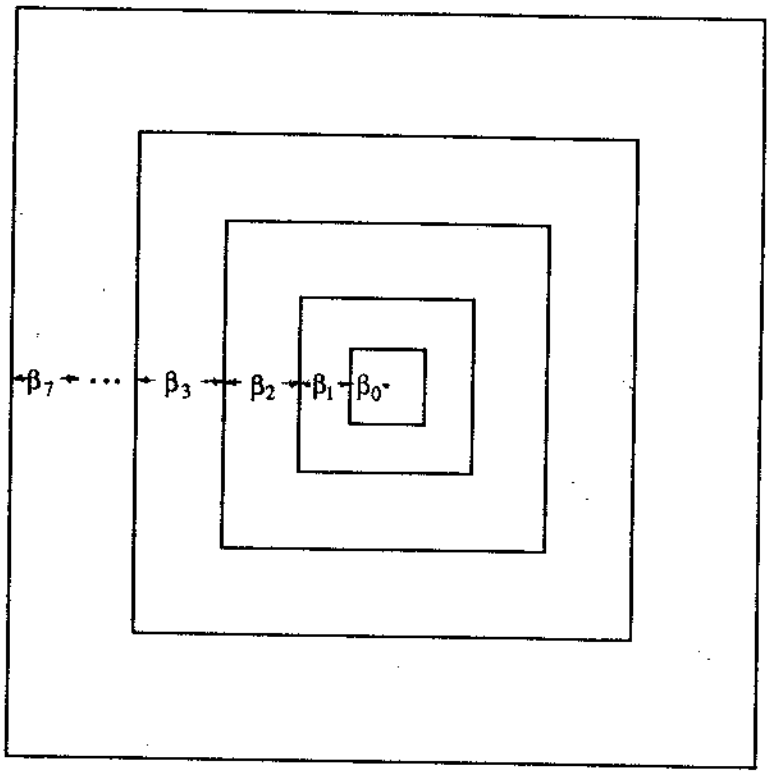

Figure 4.13: The surface division for different data sets with the view center inside the surface boundary.

If the view center is outside of the boundary, the surface is divided as shown in Figure 4.14.

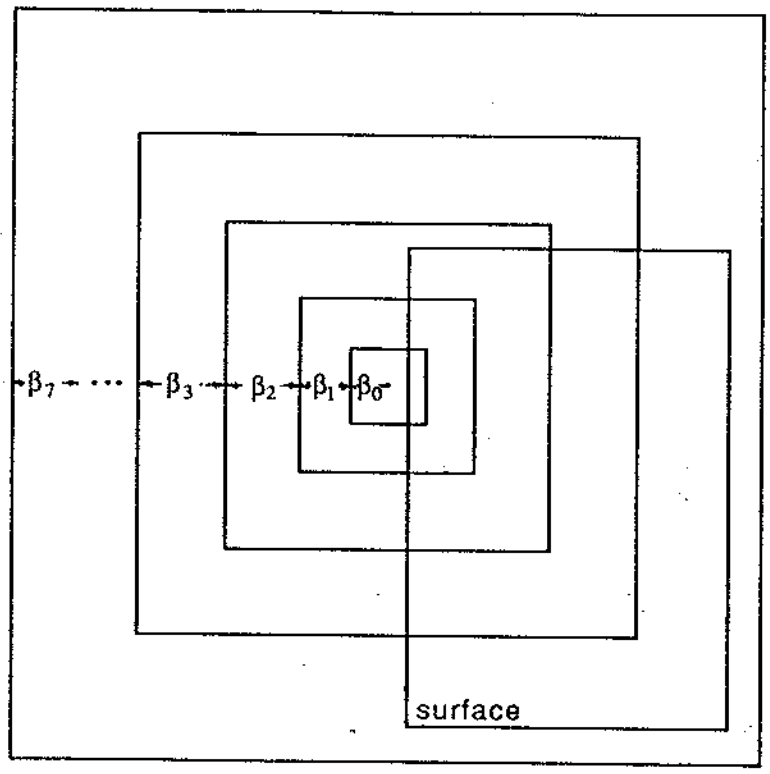

Figure 4.14: The surface division for different data sets. The view center is outside of the surface.

As the distance, denoted as d_v, between the view point and the view center decreases, the values of β_0, β_1, ..., and β_7 increase. If d_v is small enough, the triangle sets with the lowest resolution may not be used as the high resolution areas have reached the boundary. On the other hand, as d_v gets larger, the high resolution triangle sets may not be used because of its high resolution, and β_i (i=0,1, ..., 6) may be zero. If β_i is zero, β_0 to β_{i-1} are zero too as the corresponding triangle sets T_0 to T_{i-1} have even higher resolutions and require smaller d_v values. Figure 4.15 shows a configuration where the data sets T_0, T_1, and T_2 are not used due to their high resolutions, and T_7

is not used as the T_6 triangle set has reached the boundary of the surface. To decide whether β_i should be zero or not, a constant ρ_i is associated with β_i. If the distance d_v (between view point and view center) is bigger than ρ_i, then β_i is zero.

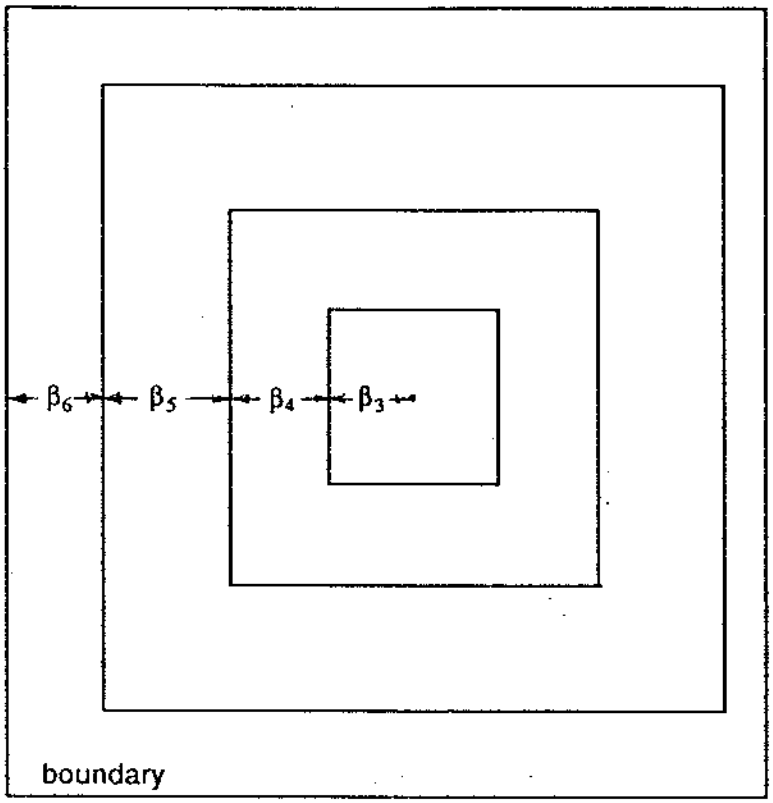

Figure 4.15: The surface division for different data sets. The high density data sets are not required because of the large distance from the viewpoint to the view center.

If β_i is too small, very narrow strips will be produced and these strips may not be wide enough to hold any triangle from its corresponding T_i. This should be prevented. To achieve this, each β_i is defined so that if β_i is not zero it will be equal to or bigger than a certain value δ_i.

If β_i is defined as a linear function of d_v, the function can be formed as

$$\beta_i = \begin{cases} 0 & \text{if } d_v > \rho_i \\ \delta_i + (\rho_i - d_v) & \text{otherwise} \end{cases} \tag{4.1}$$

The first part of the function indicates that if the distance d_v between the viewpoint and the view center is too big ($d_v > \rho_i$), the data set T_i has too high a resolution to construct the surface, and $\beta_i = 0$. From the second part of the function, it can be seen that the β_i jumps from 0 to δ_i as d_v changes from $d_v > \rho_i$ to $d_v = \rho_i$. This is to avoid narrow strips. When d_v decreases, (ρ_i - d_v) increases and so does β_i. This function satisfies all the previous requirements. If the linear relation between β_i and d_v needs to be adjusted, then (ρ_i-d_v) can be multiplied by a ratio α_i, that is, $(\rho_i - d_v) \times \alpha_i$. The function becomes:

$$\beta_i = \begin{cases} 0 & \text{if } d_v > \rho_i \\ \delta_i + (\rho_i - d_v) \times \alpha_i & \text{otherwise} \end{cases} \tag{4.2}$$

This function is the one used in the algorithms developed here. The constants α_i, δ_i and ρ_i are determined by experimenting with real data sets.

The functions used in this approach are

$$\beta_0 = \begin{cases} 0 & \text{if } d_v > 2000 \\ 1000 + (2000 - d_v) \times 0.5 & \text{otherwise} \end{cases} \tag{4.3}$$

$$\beta_1 = \begin{cases} 0 & \text{if } d_v > 4000 \\ 2000 + (4000 - d_v) \times 0.5 & \text{otherwise} \end{cases} \tag{4.4}$$

$$\beta_2 = \begin{cases} 0 & \text{if } d_v > 6000 \\ 3500 + (6000 - d_v) \times 0.5 & \text{otherwise} \end{cases} \tag{4.5}$$

$$\beta_3 = \begin{cases} 0 & \text{if } d_v > 10000 \\ 4000 + (10000 - d_v) \times 0.4 & \text{otherwise} \end{cases} \tag{4.6}$$

$$\beta_4 = \begin{cases} 0 & \text{if } d_v > 16000 \\ 5000 + (16000 - d_v) \times 0.25 & \text{otherwise} \end{cases} \tag{4.7}$$

$$\beta_5 = \begin{cases} 0 & \text{if } d_v > 24000 \\ 6000 + (24000 - d_v) \times 0.25 & \text{otherwise} \end{cases} \tag{4.8}$$

$$\beta_6 = \begin{cases} 0 & \text{if } d_v > 32000 \\ 7000 + (32000 - d_v) \times 0.25 & \text{otherwise} \end{cases} \tag{4.9}$$

$$\beta_7 = \begin{cases} 0 & \text{if } d_v > 42000 \\ 8000 + (42000 - d_v) \times 0.25 & \text{otherwise} \end{cases} \tag{4.10}$$

The unit for these equations is cm (the same as the unit used in the data set). A graph of these functions is shown in Figure 4.16.

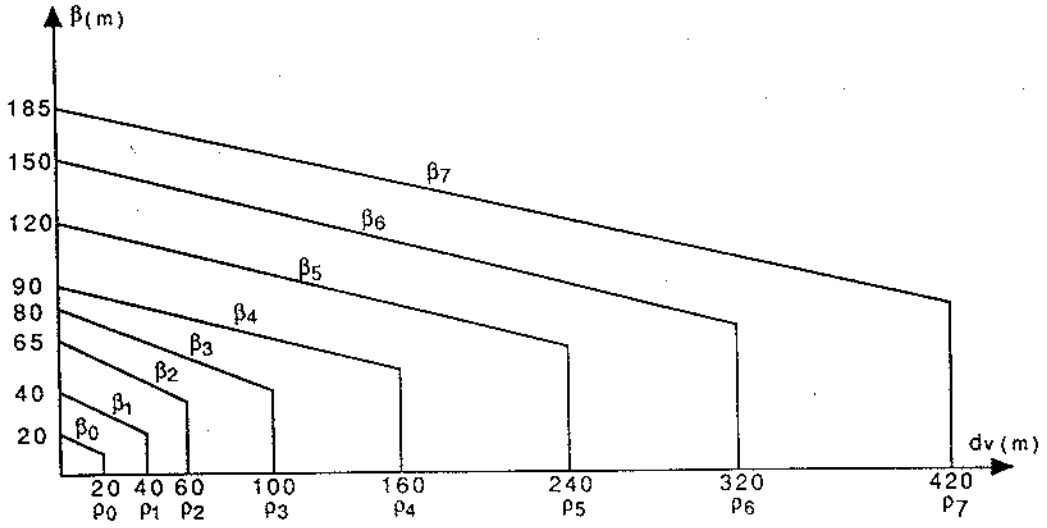

Figure 4.16: The surface division functions for data sets 0 to 7.

4.3.3 Area Construction

As the entire surface has been divided into several areas, the problem becomes how to use different data sets to construct their corresponding areas.

As provided by the data structure, if one triangle is found inside the area, or at least one vertex of a triangle is inside the area, the whole area can be constructed by recursively checking the neighboring triangles of the inside triangle. Here, an inside triangle means that it has all three vertices inside the area. If a triangle has been found inside the area it is flagged to prevent it from being repeatedly picked during the recurrence and only the neighboring triangles which have not been flagged are picked. If two of three vertices are inside the area this triangle crosses the boundary of the area. The edge between two inside vertices is put into a list. As the area is constructed, two lists are created. One is for the edges along the inside boundary of the area and another is for the outside. The next section will discuss how to make use of this list and another to connect two separately constructed, disconnected areas.

For finding a first triangle to construct an area, the grid cell technique is used. The entire surface is divided into small grid cells. For data set T_i every grid cell could be overlapped by zero, one, or several triangles as shown in Figure 4.17. If a grid is overlapped by one or more triangles, one triangle is assigned to the grid cell. When an area is defined, we can easily find a grid cell which is totally inside the area. If a triangle has been assigned to the grid cell, this triangle is the first triangle.

When all areas are constructed one after another, the areas are disconnected as can be seen in Figure 4.18.

4.3.4 Transition Between Levels

As mentioned in the previous section, edge lists are generated along the boundaries of the areas. For a boundary which separates the area for T_i and the area for T_{i-1}, two edge lists are along it. One list is from T_i and another is from T_{i-1}. To connect

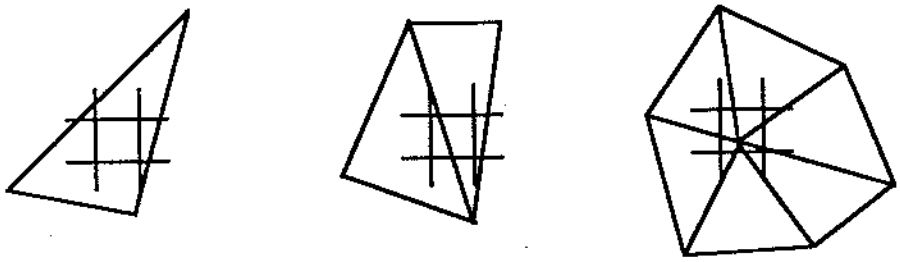

Figure 4.17: A grid cell is overlapped by one triangle or several triangles.

these two areas, a triangle mesh is drawn between these two lists. A boundary can be divided into four parts; that is, left, right, top, and bottom as can be seen in Figure 4.19. The edge lists are also divided into the same four parts. For every list four sub-lists are generated. Then, the sub-lists are sorted so that the connected edges are neighbors. Take the top sub-list as an example (the others are the same). From two lists, two sub-lists are obtained for the top. To draw a triangle mesh between these two sub-lists the first step is to connect two first points in two sub-lists as the points a and a' in Figure 4.20. The line $a - a'$ is called the base line. Next points b and b' are compared. The one on the left will be selected to form a triangle with the base line $a - a'$. Here b' is picked. Now the base line becomes $a - b'$. The two next points are b and c'. Since b is on the left side of c', another triangle $a - b' - b$ is formed. The base line is updated as $b - b'$ and the next points are c and c'. By repeating this procedure until all points in the two sublists are used, a triangle mesh is produced.

The same method is applied on the left, right, and bottom. The two areas are connected and the transition is done for this boundary. This algorithm is summarized in Figure 4.21. Applying the transition procedure for all area boundaries results in a completed surface as can be seen in Figure 4.22. Figure 4.22 shows clearly the variable resolution for irregularly spaced data. The resultant triangular mesh is shown in Figure 4.23.

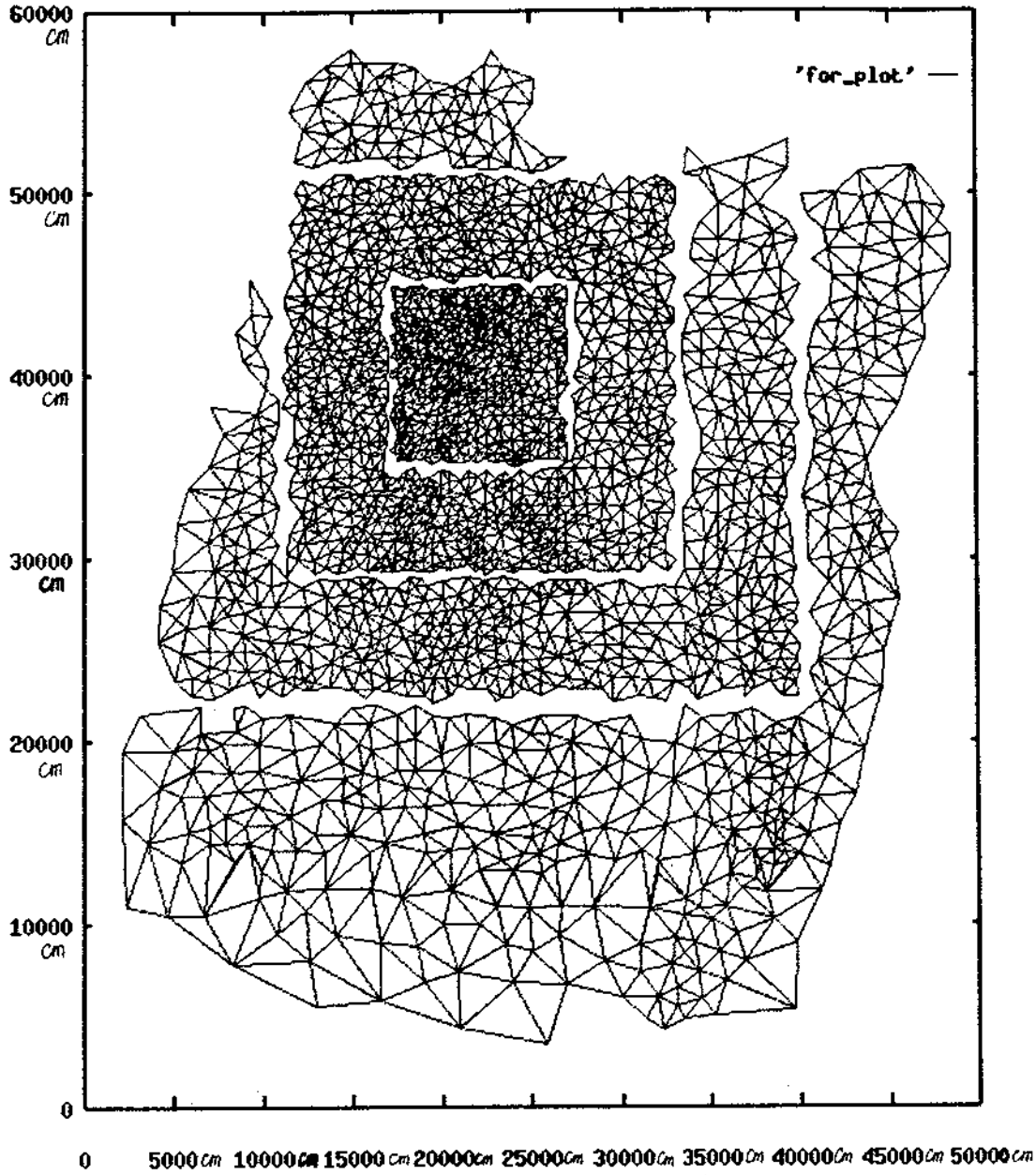

Figure 4.18: Disconnected triangle sets for T_3, T_4, T_5, and T_6 for the Louisburg data set.

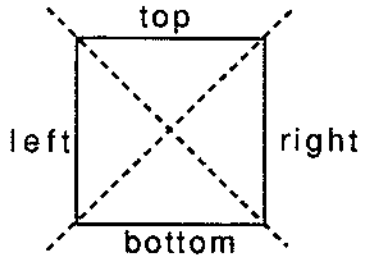

Figure 4.19: The data sets boundary division and the edge lists division.

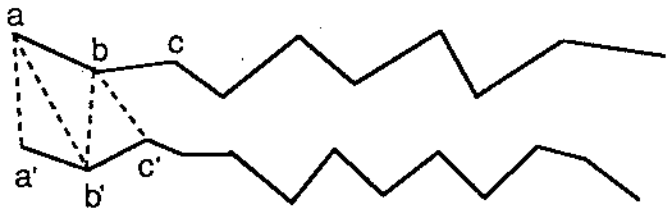

Figure 4.20: Connection of two edge lists between two neighbouring triangle sets.

```
/* The algorithm for transition between gaps.                          */
transit(L1, L2)
  /* L1 and L2 are edge lists. The transition is from edge lists L1 to L2. */
{

        if(An edge e from L1 is along north side.) e -> L1n;
               /* Put e into sublist L1n.                               */
        else if(An edge e from L1 is along south side.) e -> L1s;
        else if(An edge e from L1 is along east side.) e -> L1e;
        else if(An edge e from L1 is along west side.) e -> L1w;

        if(An edge e from L2 is along north side.) e -> L2n;
               /* Put e into sublist L2n.                               */
        else if(An edge e from L2 is along south side.) e -> L2s;
        else if(An edge e from L2 is along east side.) e -> L2e;
        else if(An edge e from L2 is along west side.) e -> L2w;

        /* Sort all the sublists so that for northen and southen sides the
           edges are from left to right and for eastern and western they are
           from top to bottom. Put the results back.                     */
        sort(L1n, L1s, L1e, L1w, L2n, L2s, L2e, L2w);

        /* Transition function transits from the first sublist to the second
           sublist.                                                      */
        transition1(L1n, L2n);
        transition1(L1s, L2s);
        transition2(L1e, L2e);
        transition2(L1w, L2w);
}
```

Figure 4.21: (a) The main function for transition.

```
/* The algorithm to transit from sublists La to Lb. This is for the northern
   and southern sides. The eastern and western sides are similar.            */
transition1(La, Lb)
{
        /* Take the first vertice of both sublists as the ends of the base
           line. It will be updated later on.                                */
        Base_line[0] = La[0];
        Base_line[1] = Lb[0];

        /* Take the second vertice of both sublists as the next points.
           They will be updated later on.                                    */
        a = b = 1;

        while(La or Lb is not empty yet){ /* Empty: has reached the end.   */
                if((La[a] on left of Lb[b]) or (Lb is empty)){
                        Base_line[0], Base_line[1], La[a] form a triangle;
                        Base_line[0] = La[a];
                        a = a + 1;
                   }
                else if((Lb[b] on left of La[a]) or (La is empty)){
                        Base_line[0], Base_line[1], Lb[b] form a triangle;
                        Base_line[1] = Lb[b];
                        b = b + 1;
                   }
           }
}
```

Figure 4.21: (b) The algorithm for triangle transition along the north and south sides.

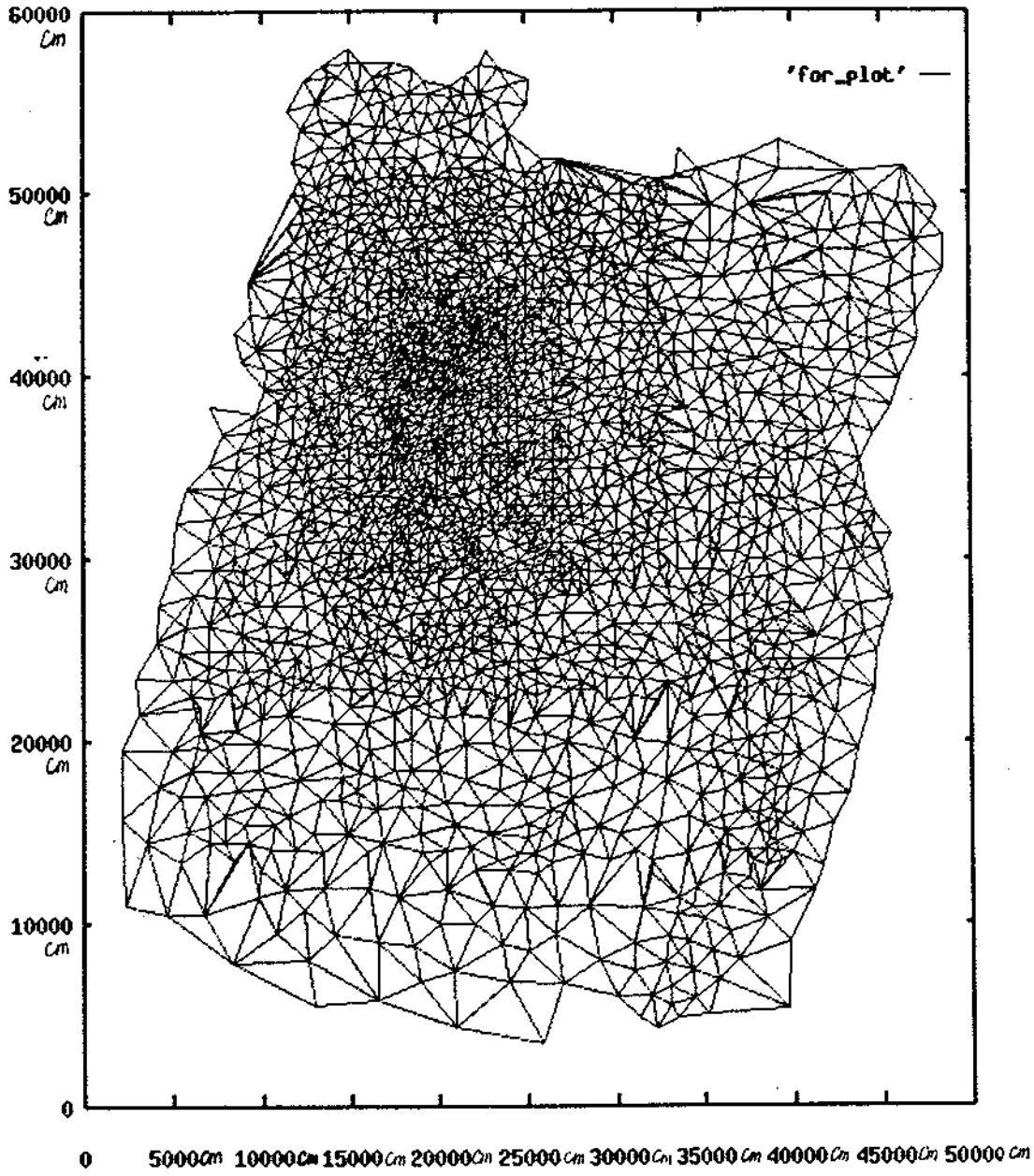

Figure 4.22: Complete variable resolution display with transitions defined (from Figure 4.18, about 3700 triangles).

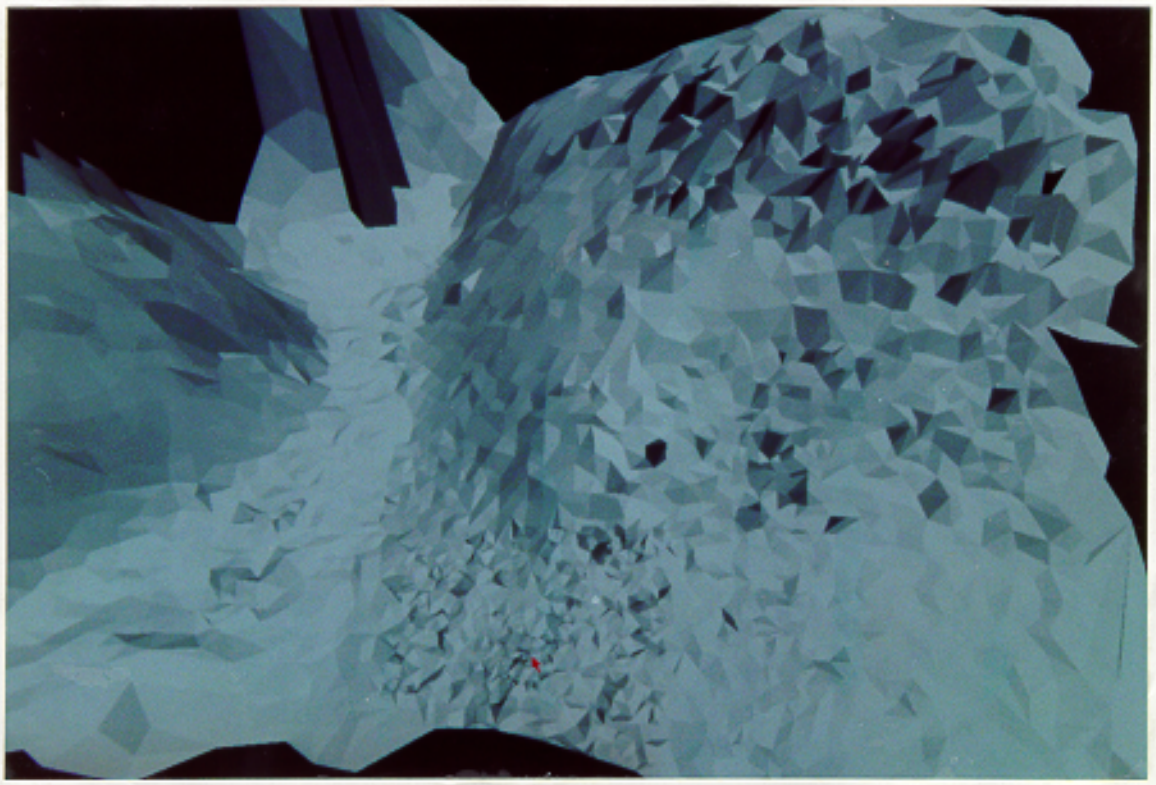

Figure 4.23: One frame of the final result with shading for the irregularly spaced data set. The red arrow points to the view center.

Chapter 5

Implementation

In this chapter an examination of aspects of the implementation of the software is made. Before examining the software itself, the hardware and the software environments are introduced.

5.1 The Hardware and Software Environments

The workstations used for this work included a Silicon Graphics IRIS 4D/85GT and a SUN Workstation SUN 4/150 (for preprocessing). The operating system is UNIX and the Silicon Graphics Function Library was used extensively. The C language was used for the implementation.

Some graphical capabilities provided by the system are very important for the display. Examples are double buffering and the lighting model. The double buffering technique provides the possibility for animation. The lighting model greatly enhances the quality of the display.

The technique of double buffering allows the creation of graphics that appear to change smoothly. The system's standard bit planes are divided into two halves; one is displayed while the other half is rendering. When the drawing is complete, the system swaps buffers; the previously invisible buffer (now containing the next frame) becomes

visible, and the previous visible buffer becomes invisible and becomes available for drawing the next frame [IRIS-4D Series, 1990].

The appearance of the surface on the display depends on three lighting components; the material, the lighting sources, and the lighting model. The material represents a set of properties that determines how it behaves under illumination. Each light source has a position and a color. The lighting model defines the characteristics of the lighting environment. In this environment, both the light and the material with which polygons are drawn, along with other primitives can be controlled. In the lighting model, the surface normal vectors for vertices control how the lighting of the final surface appears.

5.2 The Software Description

The software developed for this project includes the dynamic data structure implementation and real-time display for both regularly spaced data sets and irregularly spaced data sets, along with the preprocessing for irregularly spaced data sets. The display software includes the shaded surface and wire-frame display for both regularly spaced data sets and irregularly spaced data sets. Preprocessing for irregularly spaced data sets is to generate subsets and to create the Delauney triangulation for the subsets.

For the regularly spaced data set programs, there are approximately 10,000 lines of C code for shaded surfaces. For the irregularly spaced data set, the preprocessing software contains about 3,800 lines of C code, and the shaded surface construction and display software has about 5,300 lines of code.

5.2.1 The Implementation for Regularly Spaced Data Set

The data structure created for this approach has been discussed in chapter 2. The software for display includes the lighting model, color mapping, and double buffering

techniques. In this approach three versions of the display are generated. They are (1) a wire-frame display where the surface is represented by the outline of the triangles; (2) a smoothly shaded surface; and 3. a non-smoothly shaded surface. In the second and third cases the lighting model is applied. In the second case, the entire surface is smoothly shaded. In the third case, the lighting is smoothly shaded within a triangle, but not smoothly shaded between triangles. For the smoothly shaded surface, since the lighting model greatly depends on the normal vector specified for a vertex, the normal vector specified for a vertex which is shared by several triangles should be able to represent all of these triangles. Here a mean normal vector is calculated. This mean vector is specified for the shared vertex. By doing this, the lighting for a triangle can be smoothly transferred to its neighboring triangles. No individual triangle can be identified from the surface. For the non-smoothly shaded surface, the smooth transfer of the lighting is not applied. Every single triangle can be identified from its neighboring triangles.

The architecture of the algorithm for all the cases is shown in Figure 5.1.

The data set used here is a regularly spaced digital terrain model derived from the Louisburg data. It has 650×800 points. When about 4000 triangles are used to construct a triangular mesh, the update speed is approximately 10 times/second for wire-frame, and 5 times/second for shaded surface. Near real-time is achieved and the image of the high resolution area has very good quality.

5.2.2 Preprocessing for the Irregularly Spaced Data Set

The irregularly spaced data set preprocessing includes subset generation and Delauney triangulation creation for the subsets. The algorithms used for the preprocessing have been discussed in chapter 4. The architecture for the subset generation algorithm is shown in Figure 5.2.

Eight subsets are generated. The number of points and triangles in each subset is shown in Table 5.1. The times shown are for running on a SUN Workstation SUN

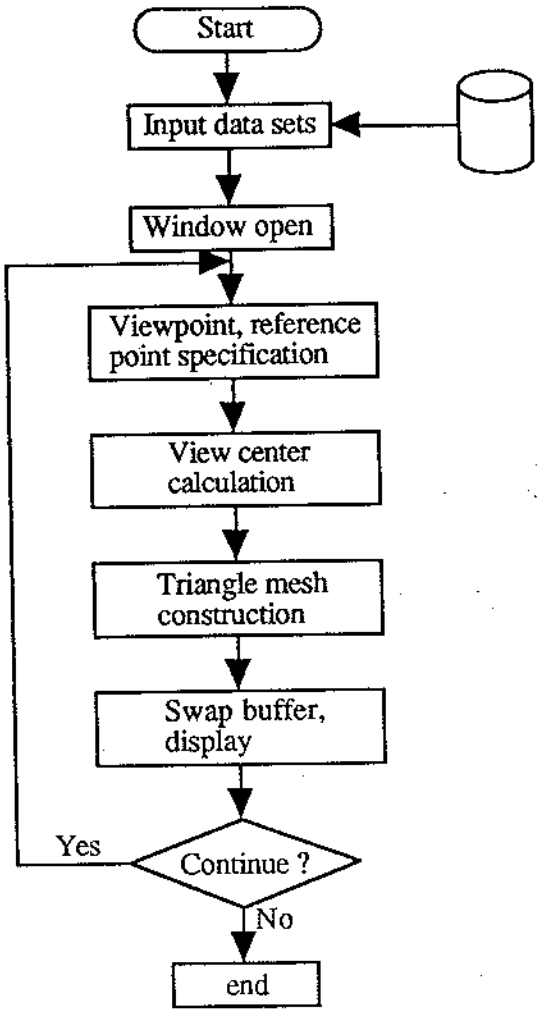

Figure 5.1: The algorithm for surface display of the regularly spaced data sets.

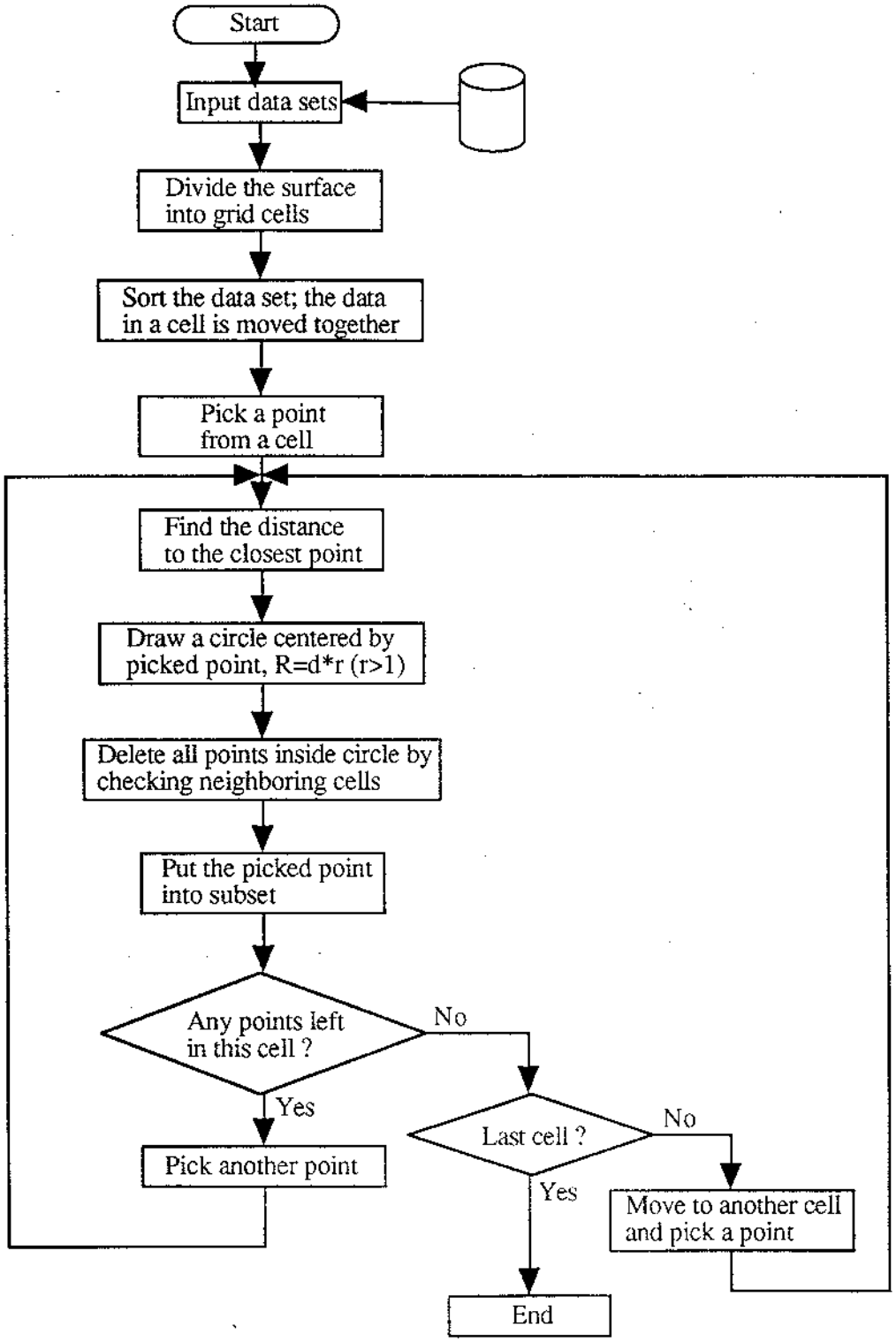

Figure 5.2: The algorithm for subset generation for irregularly spaced data.

4/150 with 32 MB RAM and 650 MB harddisk.

Set	The number of points	Time for generating the point subsets	The number of triangles	User time for creating the triangle sets
0	119845	600.00s *	239067	7h 3m 50.43s
1	35225	343.00s	70178	31m 12.22s
2	10739	29.90s	21297	2m 50.55s
3	4702	28.18s	9249	47.86s
4	2160	10.81s	4198	11.23s
5	982	4.83s	1891	6.10s
6	441	2.70s	825	2.95s
7	187	1.92s	325	1.88s

* time for the overplot removal algorithm

Table 5.1: The number of points, triangles and processing time for the subsets of the Louisbourg data set.

5.2.3 The Implementation for Irregularly Spaced Data Sets

The data structure and the method for constructing the surface have been discussed in chapter 4. The display includes a lighting model, color mapping, and double buffering techniques. Due to the complexity of getting mean normal vectors for a vertex which is shared by several triangles, the smoothly shaded surface was not implemented. The wire-frame and non-smoothly shaded surface (same color within one triangle) are generated. The architecture of the algorithm is shown in Figure 5.3.

In this approach the update speed is about 5 times/second with about 3700 triangles representing the surface.

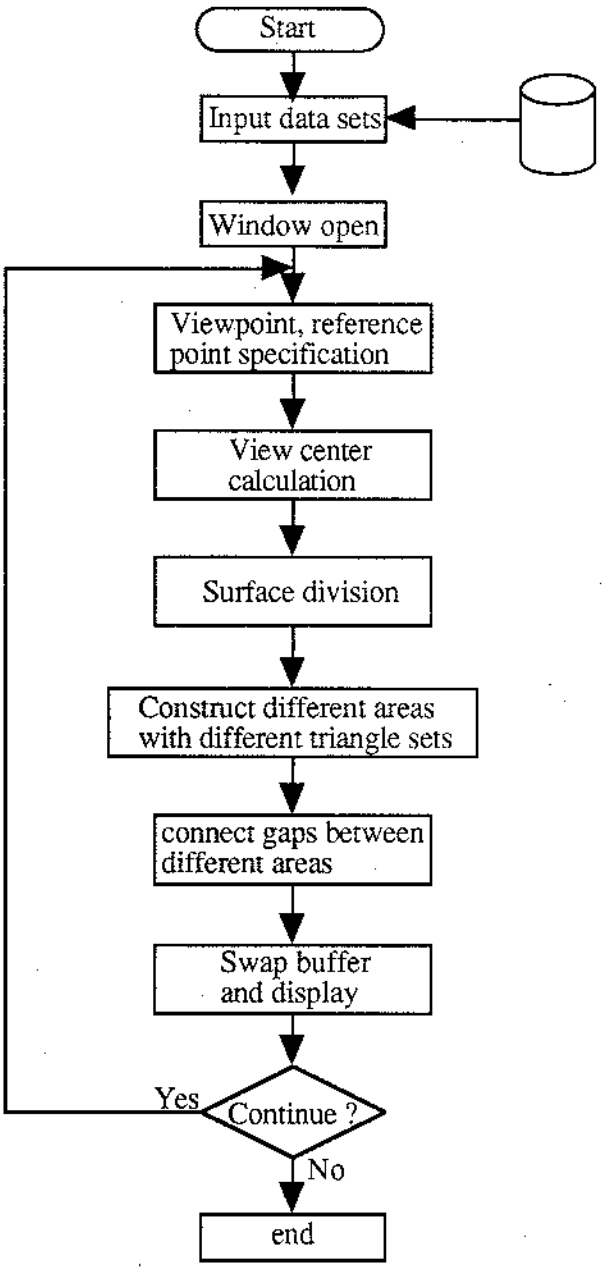

Figure 5.3: The algorithm for display of irregularly spaced data sets.

Chapter 6

Conclusions

The major achievement of this thesis is the creation of dynamic spatial data structures for displaying large terrain surfaces with variable resolution in near real-time. The software works for both regularly spaced data and irregularly spaced data. The constructed terrain surface has a variable resolution, with the gazed area having a higher resolution than other areas. As the update of the display is near real-time, the high resolution area can be moved quickly enough in response to a change in the gaze direction and the illusion of a full-field high resolution image is obtained.

6.1 Variable Resolution Approaches

In the gaze-directed approach with regularly spaced data, an algorithm for efficiently finding the view center was developed. From the view center a triangular mesh is constructed with variable resolution. The resolution for an area depends on the distance from the area to the view center. The highest resolution is for small distances. The resolution also depends on the distance from the viewpoint to the view center. If the distance is small, the resolution for the entire surface is high. The resolution is controlled by a function which is related to these two distances. This function is designed to be easily adjusted. The size of the triangles on the surface are varied

continuously in response to changing the distances. Six degrees of freedom are allowed when viewing the surface. They are specified by giving the positions of two points in 3D space which defines the view line. The software produced for the approach is able to display a 650 by 800 digital terrain with both lighting and coloring models. The update speed is about 10 times/second for wire-frame, and 5 times/second for shaded surface with approximately 4000 triangles in each display.

The smoothness directed approach with regularly spaced data was designed to ease the problem caused by large differences in the z direction. If the difference in z was too big, the resolution could be reduced significantly. To solve this problem, triangles with large differences in z are detected and subdivided so that the z difference is acceptable. This approach was abandoned due to its complexity and research time limitation.

In the gaze-directed approach with irregularly spaced data, several subsets are generated. From these subsets, triangular meshes are created by using Delauney triangulation. Each triangular mesh has a different resolution. After the view center is determined, the terrain is divided into several areas. The resolution for every area is decided by the distance from the area to the view center and the distance from the view center to the viewpoint. As soon as the resolution is decided for an area, a triangular mesh with the required resolution is employed to fill the area. Different meshes are "stitched" together along their boundaries. This approach can display a variable resolution surface of about 4000 triangles at a rate of approximately 5 Hz. It is particularly valuable due to the fact that the original data points are preserved and displayed.

6.2 Further Research

Possible items for further research include (1) better selection of subsets which may take z values into consideration during a subset selection, (2) adding a user interface

for changing the function controlling the variable resolution (see equation 4.2), and (3) adding an interface with the bat for flying which should allow start/stop update; i.e. when the bat motion stops, the resolution function is changed dynamically to give more resolution. Can the smoothness directed approach (described in chapter 3) be combined with the irregularly spaced data display methods?

References

[1] IRIS-4D Series, "Graphics Library Programming Guide", Silicon Graphics Incorporated, 1990.

[2] Bjorke, J.T., "Quadtrees and Triangulation in Digital Elevation Models", *International Archives of Photogrammetry and Remote Sensing*, Vol. 27, B4, 1988.

[3] Chen, Z.T. and Guevara, J.A., "Systematic Selection of Very Important Points (VIP) from Digital Terrain Model for Constructing Triangular Irregular Networks", *Auto-Carto 8*, 1987, pp. 50-67.

[4] Elfick, M.h., "Contouring by Use of a Triangular Mesh", *The Cartographic Journal*, Vol. 16, No. 1, 1979, pp. 24-29.

[5] Foley, J.D., vanDam, A., Feiner, S.K. and Hughes, J.F., *Compter Graphics*, Addision-Wesley Publishing Company, Inc., 1990.

[6] Fowler, R.J. and Little, J.J., "Automatic Extraction of Irregular Network Digital Terrain Models", *Computer Graphics*, Vol. 13, No. 2, 1979, pp. 199-207.

[7] Herzen, B.V. and Barr, A.H., "Accurate Triangulations of Deformed, Intersecting Surfaces", *Computer Graphics*, Vol. 21, No. 4, July 1987, pp. 103-110.

[8] Hochberg, J.E., *Perception*, Prentice-Hall, Inc., 1978, pp. 24-47.

[9] Lee, D.T. and Schachter, B.J, "Two Algorithms for Constructing a Delaunay Triangulation", *International Journal of Computer and Information Sciences*, Vol. 9, No. 3, 1980, pp. 219-242.

[10] Levoy, M. and Whitaker, R., "Gaze-Directed Volume Rendering", *Computer Graphics*, Vol. 24, No. 2, March, 1990, pp. 217-223.

[11] Mason, M.W., "Automation of Chart Compilation from Hydrographic Depth Information", MScE Thesis, Department of Surveying Engineering, University of New Brunswick, November, 1990.

[12] McCullagh, M.J., Ross, C.G., "Delaunay Triangulation of a Random Data Set for Isarithmic Mapping", *The Cartographic Journal*, Vol. 17, No. 2, December 1980, pp. 93-99.

[13] McKenna, D.G., "The Inward Spiral Method: An Improved TIN Generation Technique and Data Structure for Land Planning Applications", *Auto-Carto 8*, 1987, pp. 670-679.

[14] Mortenson, M.E., *Geometric Modeling*, John Wiley & Sons, Inc., 1985.

[15] Preparata, F.P., Shamos, M.I., *Computational Geometry*, Springer-Verlag New York Inc., 1985, pp. 198-217.

[16] Riley, S.R., "Three-Dimensional Visualization Tools for High Volume Bathymetric Data", MSc(CS) Thesis, Faculty of Computer Science, University of New Brunswick, May, 1990.

[17] Robertson, P.K., O'Callaghan, J.F., "Colour Graphics and Three-Dimensional Scene Synthesis in Image Display", *Computers & Graphics*, Vol. 11, No. 4, 1987, pp. 496-473.

[18] Smith, D.R., Paradis, A.R., "Three-Dimensional GIS for the Earth Sciences", *Auto-Carto 9*, 1989, pp. 324-335.

[19] Tamminen, M., "An Integrity Filter for Recursive Subdivision Meshes", *Computers & Graphics*, Vol. 9, No. 4, 1985, pp. 351-363.

[20] Tarvydas, A., "Terrain Approximation by Triangular Facets", *Auto-Carto 6*, 1983, pp. 1-10.

[21] Ware, C. and Osborne, S., "Exporation and Virtual Camera Control in Virtual Three Dimensional Environments", *Computer Graphics*, Vol. 24, No. 2, March, 1990, pp. 175-183.

[22] Zeevi, Y.Y., Porat, M. and Geri, G.A., "Computer Image Generation for Flight Simulators: the Gabor Approach", *The Visual Computer*, Vol. 6, No. 2, 1990, pp. 93-105.

VITA

Candidate's Full Name:	Xian Chunkui
Place and Date of Birth:	Jianhu, Jiangsu, China
Perminant Address:	Xinhu Street, Jianhu Jiangsu, China, 224700
School Attended: (with dates)	Jianhu High School Jianhu, Jiangsu, China 1973-1978
Universities Attended: (with dates and degrees held)	East China Inistitue of Technology Nanjing, Jaingsu, China 1978-1982, Bsc(CS) University of New Brunswick Fredericton, NB, Canada 1990-1992