Vector to raster data structure conversion in DX software and related issue

Ping Mi
The University of Montana
The University of MONTANA

Permission is granted by the author to reproduce this material in its entirety, provided that this material is used for scholarly purposes and is properly cited in published works and reports.

** Please check "Yes" or "No" and provide signature **

Yes, I grant permission  
No, I do not grant permission  

Author's Signature  

Date  

Any copying for commercial purposes or financial gain may be undertaken only with the author's explicit consent.
Vector to Raster Data Structure Conversion in DX Software and Related Issue

by

Ping Mi
B.S. Tongji University, China, 1992
presented in partial fulfilment of the requirements for the degree of Master of Science 1995

Approved by:

Chairperson

Dean, Graduate School

May 25, 1995
Date
## Contents

1. Introduction ...........................................................................1

2. GIS Spatial Analysis: Raster & Vector in DX ............3

3. Vector to Raster Conversion .........................................8

4. Implementation and Algorithm ........................................11

5. Summary ...........................................................................24

Appendix A ...........................................................................25

Appendix B ...........................................................................26

Appendix C ...........................................................................27
List of Figures

Regular Grid Example (Figure 1) .............................................6
Faces/Edges/Loops Example (Figure 2) .................................7
Vector vs. Raster Representation of a triangle (Figure 3) ...8
Polygon and Scan Line (Figure 4) .........................................11
Original Odd-Parity Rule (Figure 5) .................................12
Scan-Line Odd-Parity Rule (Figure 6) .................................13
Scan Line Through Vertices (Figure 7) .................................14
Horizontal Edges in Polygon (Figure 8) .............................15
Zoom in Processing (Figure 9) ...........................................16
First Scan (Figure 10) .......................................................17
Second Scan (Figure 11) .....................................................18
DX Field Example (Figure 12) ..........................................21
Complete Algorithm (Figure 13) ........................................23
List of Tables

Vector vs. Raster Representations (Table 1) .................. 4

Standard DX Field Component (Table 2) ..................... 20
1 Introduction

The purpose of this project is to examine the issues involved in generalizing vector to raster conversion from two to three dimensions. The work is motivated by use of vector data structures to represent spatial objects, and use of raster structures to allow analysis and display of these objects. Such structures, and conversion from vector to raster representation, have long been used in a 2-D context. For example, in geographic information system, areas on the earth's surface have traditionally been represented by the lines (vectors) that form the area boundaries; however, for purpose of analysis or display, this vector format is often converted to a comparable raster which approximates the area with a set of contiguous square pixels.

An important assumption in traditional GIS is that all vectors lie in the same plane, i.e., in 2-D. This assumption essentially means that the surface of the earth is assumed to be flat, and prevents the description of objects, such as mines or geological formations, that have 3-D spatial characteristics. In this project, we assume that objects can have true 3-D vector descriptions and look at the result of extending existing 2-D vector to raster conversion algorithms to cover 3-D vector descriptions of spatial objects.

To illustrate the 2-D and 3-D applications, we use the Data Explorer (DX) visualization environment. DX supports both general purpose vector and raster objects. We use DX formats as the source and target for our conversion process, and we use DX display capabilities to illustrate the results of the conversion.

In this project, I implemented a standard 2-D vector to raster conversion for DX data structures. I also extended the algorithm to cover 3-D vector to raster conversion, which
involves the conversion of point, line and face data. A second version of the 3-D conversion is also implemented which deduces volumetric properties which are described only implicitly in the 3-D vector geometry.
2 GIS Spatial Analysis: 2-D Vector to Raster Conversion in DX

2.1 GIS Spatial Analysis and Vector vs. Raster

A Geographic Information System is a special type of database which supports the analysis and manipulation of data objects that represent spatially referenced entities and attributes. It is designed for the collection, storage and analysis of objects and phenomena where geographic location is an important characteristic of the analysis.

A spatial object is an abstraction of an entity referenced to the (2-D) surface of the earth. The exact spatial reference system is usually based on one of several types of geographic coordinate systems. For a small study area, it can be any convenient grid. There are two valid methods of representing spatial data in a GIS: vector and raster. In raster format, the spatial data is associated with a set of contiguous pixels of the grid that cover the area representing the object. In vector format, the spatial data is associated with a set of connected lines that outline the area representing an object. In both cases, data attributes can be associated with an object. Spatial data attributes could refer to landuse, the height of a forest canopy, the population of a city, the vegetation type and so on. In the raster representation, attribute values are copied and assigned to each pixel. In the vector representation, all values for an object are collected in one record, which is then linked to a record which contains the line description.

A typical GIS groups sets of logically related geographic features and their attributes to form a data layer. Database operations perform certain geographical analysis functions, such as allowing the user to ask questions like: “What is data value of A at location B?”, “What is the path of the least cost, resistance or distance along the ground from X to Y?”, “Reclassify objects having certain combinations of attributes!” and “How many occurrences of type A are there within a distance D of B?”. 
Depending on the display and analysis tasks of the GIS, both vector and raster data representation of spatial data have their advantages and disadvantages, as illustrated in Table 1.

<table>
<thead>
<tr>
<th>Vector Method</th>
<th>Raster Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advantage 1. Accurate positional information and accurate area calculations</td>
<td>1. Easy to perform some types of spatial analyses</td>
</tr>
<tr>
<td>2. Pleasing visuals due to retention of details</td>
<td>2. Simple data structure</td>
</tr>
<tr>
<td>3. Compact data structure with small storage requirements</td>
<td>3. Can import, display and manipulate remotely sensed data</td>
</tr>
<tr>
<td>Diadvantage 1. Data structure can be complex to manipulate</td>
<td>1. Approximate positional information and area calculations</td>
</tr>
<tr>
<td>2. Spatial analysis can be hard or impossible to perform</td>
<td>2. Blocky display appearance with loss of details as pixel size increase</td>
</tr>
<tr>
<td>3. Poor match with remotely sensed data (which are organized as rasters)</td>
<td>3. Large storage requirements</td>
</tr>
<tr>
<td></td>
<td>4. Projections and some other transformations are time consuming</td>
</tr>
</tbody>
</table>

Table 1
2.2 2-D vs. 3-D support

GIS have traditionally imposed the limit both in vector and raster models, that spatial objects be described only in 2 dimensions. The only form of 3-dimentional support (generally called “2 1/2-D”) has been special support for an “elevation” attribute that can be attached to 2-D coordinates. Though this allows landscapes to be depicted via special “rubbersheet” operations, it does not allow GIS to treat real 3-D structures, such as mines, subway systems, etc. As a result, 3-D spatial structures such as mine descriptions have traditionally been constructed using other more general software, such as line drawing packages. For analysis or display, the line drawings and GIS (2-D surface) data typically have to be painstakingly combined from GIS and other sources.

2.3 Data Explorer Software

Data Explorer is a visualization system running on a workstation environment used in many application areas, including GIS. It has a variety of data representations and limited spatial analysis capability. The whole system is designed using an object-oriented approach to data description.

DX allows end users to perform tasks at three levels of sophistication. A user can apply data and adjust input values for an existing visualization process. A user can create a visualization process by connecting a network of DX modules. A user can create a new module based on his/her own C or FORTRAN code.

DX supports general purpose structures for n-dimentional grids (raster) and n-dimentional vectors (faces/loops/edges). A grid structure is specified by origin, delta and counts defining the extent of the grid. As illustrated in Figure 1, for a regular grid, the origin is at [0 0 0], the deltas are 2 in the x dimension, 4 in the y dimension and 1 in the z dimension, and the counts would be specified as 3 1 2.
The n-dimensional vector structure in DX is faces/loops/edges, where faces describe the surfaces that enclose or bound objects, loops define any “holes” in a face, and edges define the lines that outline the faces.

In the DX object description, the “faces” component is a set of faces. Each face is described as a set of loops. That is, each entry in the face array is a single integer index into the “loops” array identifying the first of a consecutive set of loops for this face. Each “loop” component references a set of “edges” in the same way. Finally, each “edge” component references a set of “positions”. For example, in Figure 2, face entry 0 identifies the first of a consecutive set of loops: loop entry 0. Face entry 1 identifies the first of next consecutive set of loops: loop entry 3. So face 0 is made of loops 0, 1, 2. Loop entry 0 identifies the first of a consecutive set of edges: edge 0. Loop entry 1 identifies the first of next consecutive set of edges: edge 4. So loop 0 is constructed from edges 0, 1, 2, 3, which give the coordinates of four vertices of the loop.
2.4 Problem Definition

Given the DX representation for both 2-D and 3-D vector and raster structures, the problem is to implement conversion from 2-D or 3-D vector (faces/loops/edges) to 2-D or 3-D grids. In this project, I implement the conversion as a set of modules that are added to the DX environment, and illustrate the result of the conversion with appropriate examples.
3 Vector to Raster Conversion

3.1 Example Picture and Explanation

In raster format, the plane is divided into an n x n grid with one or more data items associated with each grid cell. In vector format, the data is associated with the face constructed from a sequence of edges which define starting and ending positions. For example, in Figure 3, the outlined set of pixels represents a raster object that approximate a triangle by assigning a key data value to each pixel inside the triangle area. In contrast, the object description in vector format represents a triangle by organizing three vertices into edges/loops/faces format and associating the data with whole area (face). By selecting appropriate vertices the vector format can exactly represent any area that is bounded by a collection of straight line segments. In contrast, the raster format can only approximate the area as a set of contiguous cubes (although by increasing the resolution, i.e., making the squares smaller, the approximation error can be minimized).

Figure 3 Vector vs. Raster Representation of a triangle
3.2 Two Dimensional Vector to Raster Conversion

In my conversion process, I start with the well known 2-D scan line algorithm \cite{3}. This algorithm matches object vertices to the position of the pixels inside the face area, then assigns the associated data value to each pixel. Figure 3 illustrates the result of this type of conversion, with the shaded area in the raster illustrating the “error” inherent in the conversion.

In general, 2-D areas may contain one or more “holes” which must be accounted for in both vector and raster representations. In the DX faces/loops/edges representation, a hole is represented by a loop. I use the “odd-parity rule” \cite{3} to fill the face area. A hole in a raster object is simply represented by assigning no key data value to the pixels that fall inside the hole.

Whereas a vector system can use straight lines between any two points, a raster system can only approximate vector lines with the boundary lines of pixels on grid. This approximation causes the error shown in Figure 3, which has the effect called “jaggies” or “staircasing”. Antialiasing \cite{3} is a standard technique used to minimize the error in conversion without increasing grid resolution.

3.3 Two Approaches for Three Dimensional Vector to Raster Conversion

The format of 3-D raster and vector data in DX is very similar to that used for 2-D. In 3-D raster format, the grid is a 3-D cube. In 3-D vector format, the position of every ending point is simply a point in 3-D coordinate space. In 3-D grids, data can be attached to points, lines, surfaces of the cubes composing the grid, or the cubes (volumes) themselves.
DX's faces/loops/edges structure allows data on points, lines and faces, but not volumes. This means DX can only attach data to a set of points, lines or surfaces in 3-D space, but not to volumes enclosed by surfaces. Thus in DX, a 3-D vector object based on edges/loops/faces can only describe surface attributes, not attributes for the whole object. In DX, a 3-D raster object can describe volumetric properties. With 3-D rasters, we can thus analyze and display volumetric properties of the object with slicing, slabling and other operations, because the 3-D object has data values associated with both its boundary and its volume. This approach becomes very meaningful when we consider using DX to represent data from other 3-D drawing applications, such as AutoCad data structures, in which volume data is extremely important.

I implement two different types of vector to raster conversion and demonstrate both time/space and display resolution differences. The first type of conversion converts only face data. The second type of conversion both converts surface data and deduces the volumetric properties implied by the 3-D geometry.
4 Implementation and Algorithm

4.1 2-D Vector to Raster Conversion

Conversion starts with the use of a scan-line algorithms [3] that does the rasterization. Scan-line algorithms operate by computing spans that lie between left and right edges of the polygon, then creating an image one scan line at a time.

Figure 3 (from [3], Figure 3.14) shows a polygon and one scan line (line S) passing through it. The intersection of scan line S with edge AF, FE, ED and DC (a,b,c,d) are what we need to determine which pixels on scan line S are within the polygon. Because the scan lines are horizontal lines with a specified interval height, it isn’t difficult to calculate the intersections with each scan line for each edge. For each edge the algorithm goes from the starting point all the way to the ending point to calculate the intersections with every scan line and stores them in a 2-D array. After computing the intersections for all edges in one face, the algorithm sorts the intersections on each scan line.

Figure 4 Polygon and Scan Line (From [3], Figure 3.14)

Next, given a sorted set of intersections, the problem is to determine which area between
which two sections is inside the polygon. The odd-parity rule [3] to determine which elements are in the interior of an object. This simple algorithm can handle both convex and concave polygons and any enclosed area of the primitives with holes.

The original midpoint algorithm of odd-parity rule is: To determine whether a region lies inside or outside a given polygon, choose as a test point any point inside the particular region. Next, choose a ray that starts at the test point, extends infinitely in any direction, and does not pass through any vertices. If this ray intersects the polygon outline an odd number of times, the region is considered to be interior[3]. Figure 5 shows the odd-parity rule applied to a simple 2-D structure.

![Figure 5 Original Odd-Parity Rule](From [3], Figure 2.9)

An improvement can be made when we apply this odd-parity rule to the scan-line algorithm: For each scan line(except the ones which pass vertices), start from the left most point of the image which is outside the object. All points between odd number intersection and the
next intersection are inside the object. All others are outside the object. Figure 6 illustrates this approach.

![Figure 6 Scan-Line Odd-Parity Rule](image)

There are several special cases that the algorithm must deal with. First is how to deal with the scan lines which pass the vertices. There are four kinds of vertices, as shown in Figure 7. (a) and (b) are vertices connecting monotonically increasing/decreasing edges. For these vertices, the odd-parity algorithm for scan lines works just as they go through non-vertices intersections. The program must be sure to calculate these intersections once, even though the scan line crosses two edges.

(c) and (d) are vertices connecting non-monotonically increasing/decreasing edges. In order
to make odd-parity rule work on the scan lines going through these vertices, one delicate transformation is made in this way: Edges on both sides of the vertex will shrink one scan line interval, so in some sense, we get rid of this vertex to get rid of the problem. As illustrated in Figure 7, in (a) and (b), the vertex is counted once though it is on two edges, in (c) and (d), the actual vertex is skipped by shrinking both edges.

There is an important equation that deals with this vertices problem.

\[
END\cdot POINT\cdot Y\cdot EDGE1 = START\cdot POINT\cdot Y\cdot EDGE2 - UP \ast MONO,
\]

where UP stands for edge1’s direction (whether it is up or down) and MONO indicates whether edge1 and edge2 are monotonically increasing or decreasing.

![Figure 7 Scan Line through Vertices](image)

Another special case that needs to be addressed is how to deal with horizontal edges, because horizontal edges have multiple intersections with the scan line that it passes through.
The conversion algorithm simply ignores horizontal edges. As an enclosed area, each horizontal edge must be connected with non-horizontal lines, therefore skipping one edge line in the overall area won't be a visible loss.

![Figure 8 Horizontal Edges in Polygon](image)

The final issue to be addressed is the use of antialiasing to improve display resolution. Due to computer speed and space limitations, the conversion algorithm always selects an "appropriate" resolution (i.e. rather than allowing the user to arbitrarily select a resolution). The algorithm does let the user specify the area to vectorize, so that user can zoom in the specific part of the image to get better visual effect or more accurate analysis result.

When the user specifies an area to vectorize, the algorithm still draws the scan line from very left end of the entire image, so that it doesn't lose the track of the intersection count. But the algorithm needs to draw scan lines only between the specified top and bottom limits, so that the algorithm can select higher resolution and draw the portion of the image inside the specific area. Figure 9 illustrates how to zoom in the user specified area in the algorithm.
There are two possible approaches to extend the 2-D conversion algorithm to 3-D. One way is to do the geometrical transformations that transfer each 3-D plane to an appropriate 2-D plane, then to assemble the faces by collecting all the 2-D planes. Instead I implement a 3-D version of scan-line, in the form of a 3-D scan-surface algorithm.

The scan-surface algorithm is similar to the 2-D scan-line algorithm. But instead of scanning the 2-D image with lines to find out the part of image which falls on the line, the algorithm scans the 3-D space with X-Y or Y-Z or X-Z planes to find the parts of the object that fall on this scan surface. Because a DX faces/loops/edges structure itself only represents surface
data, the actual image that falls on the scan surfaces are only intersection lines.

To deduce volume data, the algorithm follows the DX faces/loops/edges structure first to get the 3-D object which only has the surface data. Then the algorithm scans the object with scan-surface again. Each scan-surface intersects the object with an polygon frame, and then fill each polygon. In this way, the algorithm fills out the volume data for the object. For example, in Figure 11 the object is a box with a hole in the middle of it. After getting the surface data for this object, the algorithm uses the x-y plane to scan the whole object to get the volume data. Imagine using the x-y plane to cut an object with only surface data. The resulting "cut plane" has two squares (one square hole inside one square frame). Using the 2-D scan line algorithm, the cut plane is interpreted as a "scan plan", and the volume is filled (but not the hole). By filling intersection plane of each scan surface, the algorithm can fill the whole object with volume data.
4 IMPLEMENTATION AND ALGORITHM

The algorithm needs to decide which plane to chose as scan surface. To simplify the problem, it always uses the x-y, x-z or y-z plane. If the object has one face parallel to the scan surface plane, then in the first scan (to get surface data), the algorithm will change the scan surface plane to get the surface data for this special surface. In second scan, it doesn’t matter, because the intersection plane of this face and the scan surface plane is the face itself, which has already been filled in first scan.

4.3 Programming Detail in DX interface

Data Explorer provides a user interface that allows a programmer to perform various data manipulation and visualization tasks. It also provides a powerful C library with programming tools that allow a user to write his/her own special purpose modules.

To program a new module, there are three files to create: a module description file, a C file, and a makefile. In the description file, values for MODULE, CATEGORY, DESCRIPTION, INPUT and OUTPUT must be filled in. MODULE is the name of module to be
created. CATEGORY is the name of category in which the module will be included in user interface. DESCRIPTION, INPUT and OUTPUT are all explanations to be included in the "help" section to help the user to understand how to use the module.

The makefile is used to tell compiler how to compile and link the module source code with the library file. DX provides a tool called the module constructor to help the user create the makefile.

The module source file is the most important part of a new module containing the source code for the program that implements the module. A critical aspect of the source code is the use of facilities in the application interface to create the data structures appropriate for the module, and to manipulate the data structures correctly.

In DX, field objects are the fundamental objects. The information in a field is represented by several named components. Each component has a value, which is usually an array object. Table 2 below shows the standard field components. Figure 12 shows an example of a typical field.
<table>
<thead>
<tr>
<th>Component</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;data&quot;</td>
<td>arbitrary</td>
<td>user's data (dependent variable)</td>
</tr>
<tr>
<td>&quot;positions&quot;</td>
<td>float[n]</td>
<td>n-space sample points</td>
</tr>
<tr>
<td>&quot;invalid positions&quot;</td>
<td>char</td>
<td>which sample points are invalid</td>
</tr>
<tr>
<td>&quot;colors&quot;</td>
<td>float[3]</td>
<td>surface or volume colors</td>
</tr>
<tr>
<td>&quot;colors&quot;</td>
<td>char</td>
<td>color index (see &quot;color map&quot;)</td>
</tr>
<tr>
<td>&quot;color map&quot;</td>
<td>float[3]</td>
<td>color map indexed by &quot;colors&quot; component</td>
</tr>
<tr>
<td>&quot;front colors&quot;</td>
<td>float[3]</td>
<td>colors of front of surface</td>
</tr>
<tr>
<td>&quot;back colors&quot;</td>
<td>float[3]</td>
<td>colors of back of surface</td>
</tr>
<tr>
<td>&quot;opacities&quot;</td>
<td>float</td>
<td>opacity of surface or volume</td>
</tr>
<tr>
<td>&quot;opacities&quot;</td>
<td>char</td>
<td>opacity index (see &quot;opacity map&quot;)</td>
</tr>
<tr>
<td>&quot;opacity map&quot;</td>
<td>float</td>
<td>opacity map indexed by &quot;opacities&quot; component</td>
</tr>
<tr>
<td>&quot;tangents&quot;</td>
<td>float[3]</td>
<td>curve tangent</td>
</tr>
<tr>
<td>&quot;normals&quot;</td>
<td>float[3]</td>
<td>curve or surface normal</td>
</tr>
<tr>
<td>&quot;binormals&quot;</td>
<td>float[3]</td>
<td>second curve normal</td>
</tr>
<tr>
<td>&quot;connections&quot;</td>
<td>int[k]</td>
<td>interpolation elements</td>
</tr>
<tr>
<td>&quot;invalid connections&quot;</td>
<td>char</td>
<td>which interpolation elements are invalid</td>
</tr>
<tr>
<td>&quot;faces&quot;</td>
<td>int</td>
<td>faces described as a collection of loops</td>
</tr>
<tr>
<td>&quot;loops&quot;</td>
<td>int</td>
<td>loops described as a series of edges</td>
</tr>
<tr>
<td>&quot;edges&quot;</td>
<td>int</td>
<td>edges described as a series of points</td>
</tr>
<tr>
<td>&quot;neighbors&quot;</td>
<td>int[ρ]</td>
<td>pointers to connection neighbors</td>
</tr>
<tr>
<td>&quot;box&quot;</td>
<td>float[2n]</td>
<td>2n corners of a bounding box</td>
</tr>
<tr>
<td>&quot;data statistics&quot;</td>
<td></td>
<td>statistics for data component</td>
</tr>
</tbody>
</table>

Table 2 Standard DX Field Components (From [1], Table 2-1)
In the DX library there are a number of functions that define and manipulate field objects, extract component array data, extract the component information from the field, alter field component and add or delete the field component. To write a new module, the "m_nameofmodule(Object *inputfield, Object *outputfield)" routine is the module entry point that will be called by the Data Explorer executive. It will be called with two argu-
ments: a pointer to an array containing the input fields and a pointer to an array in which the module will return the output fields.

The conversion algorithm starts with the routine m_vtosc(Object *in, Object *out). This routine first extracts the position component value and the information of the position component from the input field to determine whether the conversion involves a 2-D or 3-D structure. It then applies the appropriate algorithm described above to transfer the faces/edges/loops structure to grid cell (square or cube) structure. To create the output field, I create the positions and connections (regular connection) component in the output fields. I also need to extract the data component from the input field, then apply these data values to the correspond position value in the output field according to the conversion result calculated above.

The scan-line algorithm is sketched in Figure 13. The complete source code is shown in Appendix A, B and C.
IMPLEMENTATION AND ALGORITHM

Get Data Component and Transfer to C Array Data
Get Faces/Edges/Loops Component and Transfer to C Array Data
Get User Specified Area if Any
Get Position Component info
if 2-D Position
{
  if User Specify Area
    Coordinate Transformation from Zoom Area to Full Screen
  For Each Face
  {
    For Each loop
    {
      For Each Edge
      {
        if Horizontal Edge
          Break
        Calculation Ending Point According to
        END_POINT_Y_EDGEl=START_POINT_Y_EDGEl2 UP*MONO
        Calculate Intersection with Each Scan Line
      }
    }
  }
  Sort Intersections for Each Scan Line
  Apply Odd-Parity Rule to Fill out Face and Get Rid of the Area outside
  the Screen (for User Specified Zooming)
}
else (3-D)
{
  if Scan Plane Parallel to Surface
    Exchange X Y Z Coordinate of the Vertices on this plane
  Scan Each Face with Scan Plane
  Revert to Original X Y Z Coordinate if Necessary
  if Volume Data
    Scan the Object Again
}
Make New Output Field
Make Position (Grid Cell) and Regular Connection Component
Make Data Component and Fill it According to the Result Calculated above

Figure 13 Complete Algorithm
5 Summary

That is about how my project is created and what it is used for. I create two new modules in DX. One module is used to convert 3-D faces/edges/loops structures to grid cell structure where only surface data is present. The other module will do the conversion applying volume data. Both modules will process 2-D conversions in same way. These modules add a useful tool for DX user interface.

References


Appendix A

/* Module Definition File*/

MODULE vtoscface
CATEGORY new
DESCRIPTION transform ELF to grid cell data for face data
OUTBOARD midface
INPUT inELFface; field; (none); input ELF structure
INPUT inPOSface; field; (none); input limit
OUTPUT outface; field; out gridcell structure for face data

MODULE vtosc
CATEGORY new
DESCRIPTION transform ELF to grid cell data
OUTBOARD mid
INPUT inELF; field; (none); input ELF structure
INPUT inPOS; field; (none); input limit
OUTPUT out; field; out gridcell structure
/*Makefile*/

FILES_vtoscface = vtoscface.o
BASE = /usr/lpp/dx
CFLAGS = -Dibm6000 -O -I$(BASE)/include
LDFLAGS = -bE:$(BASE)/lib_ibm6000/dxexec.exp -L$(BASE)/lib_ibm6000
LIBS = -1DX -ly -ll -1X11 -lm
OLIBS = -1DXlite -lm

midface: $(FILES_vtoscface) outboard.o
    $(CC) $(CFLAGS) $(LDFLAGS) $(FILES_vtoscface) outboard.o $(OLIBS) -o midface

outboard.o: $(BASE)/lib/outboard.c
    $(CC) $(CFLAGS) outboard.c

uservtoscface.c: vtoscface.mdf
    mdf2c vtoscface.mdf > uservtoscface.c
APPENDIX C
/*C File*/
#include <stdio.h>
#include "dx/dx.h"

int final[500][500];
int mark[500][500];
int f3[50][50][50];
int m3[50][50][50];
float minx,maxx,miny,maxy,minz,maxz;
Array apos, aedge, aloop, aface, acolo;
float *pos, *colo;
int *edge,*loop,*face;
int pn,ln,fn,en;
int shape[1];
static Error dovtoasc(Object *, Object *);
static Error d3s(Object *);
void fill(float pos[],int edge[],int loop[],int face[],int fn,int ln,
int en);
void dff(float pos[],int edge[],int loop[],int face[],int fn,int ln,
int en);

Error
m_vtoscface(Object *, Object *)
{
    int i,n,rank;
    Type type;
    Category category;
    Array alimit;
    float *limit;

    out[0] = DXNewField();
    if (! out[0])
        goto error;

    if (in[0] == NULL)
    {
        DXSetError(ERROR_MISSING_DATA, "\"inpu\" must be specified");
        return ERROR;
    }

    if(in[1] == NULL) (maxx=0;minx=0;maxy=0;miny=0;minz=0;maxz=0;)
    else {
        alimit=(Array)DXGetComponentValue((Field)in[1], "positions");
        DXGetArrayInfo(alimit,&n,&type,&category,&rank,shape);
        if((rank!=1)||(type!=TYPE_FLOAT))
        {  
            DXSetError(ERROR_BAD_TYPE,"data is not valid");
            return ERROR;
        }
        limit=(float *)DXGetArrayData(alimit);
        if(shape[0]==2)
            minx=limit[0];miny=limit[1];maxx=limit[2];maxy=limit[3];
        if(shape[0]==3)
            minx=limit[0];miny=limit[1];minz=limit[2];
            maxx=limit[3];maxy=limit[4];maxz=limit[5];}

    if (!dovtoasc(in, out))
        goto error;

    return OK;
}
if (in[i] != out[i])
    DXDelete(out[i]);
out[i] = NULL;
}
return ERROR;

static Error
dovtosc(Object *in, Object *out)
{
    Array anpos=NULL;
    Array ancolo=NULL;
    Array anopa=NULL;
    float *npos,*ncolo,*nopa;
    int i,j,k,n,m,1,ko;
    float a,b,c;
    int d;
    static Error re;
    float sc;
    apos=(Array)DXGetComponentValue((Field)(*in),"positions");
    if(!apos){
        DXSetError(ERROR_MISSING_DATA,"input has no positions1");
        return ERROR;
    }
    if(!DXGetArraylnfo(apos,&pn,NULL,NULL,&d,shape))
        return ERROR;
    pos=(float *)DXGetArrayData(apos);
    if(!pos){
        DXSetError(ERROR_MISSING_DATA,"input has no positions2");
        return ERROR;
    }
    aedge=(Array)DXGetComponentValue((Field)(*in),"edges");
    if(!aedge){
        DXSetError(ERROR_MISSING_DATA,"input has no edges1");
        return ERROR;
    }
    if(!DXGetArraylnfo(aedge,&en,NULL,NULL,NULL,NULL))
        return ERROR;
    edge=(int *)DXGetArrayData(aedge);
    if(!edge){
        DXSetError(ERROR_MISSING_DATA,"input has no edges2");
        return ERROR;
    }
    aloop=(Array)DXGetComponentValue((Field)(*in),"loops");
    if(!aloop){
        DXSetError(ERROR_MISSING_DATA,"input has no loops1");
        return ERROR;
    }
    if(!DXGetArraylnfo(aloop,&ln,NULL,NULL,NULL,NULL))
        return ERROR;
    loop=(int *)DXGetArrayData(aloop);
    if(!loop){
        DXSetError(ERROR_MISSING_DATA,"input has no loops2");
        return ERROR;
    }
    aface=(Array)DXGetComponentValue((Field)(*in),"faces");
    if(!aface){
        DXSetError(ERROR_MISSING_DATA,"input has no faces1");
        return ERROR;
    }
if(!DXGetArrayInfo(aface,&fn,NULL,NULL,NULL,NULL))
    return ERROR;
face=(int *)DXGetArrayData(aface);
if(!face){
    DXSetError(ERROR_MISSING_DATA,"input has no faces2");
    return ERROR;
}
acolo=(Array)DXGetComponentValue((Field)(*in),"colors");
if(!acolo){
    DXSetError(ERROR_MISSING_DATA,"input has no color1");
    return ERROR;
}
colo=(float *)DXGetArrayData(acolo);
if(!colo){
    DXSetError(ERROR_MISSING_DATA,"input has no color2");
    return ERROR;
}
if(shape[0]==2){
    if((minx==0)&&(miny==0)&&(maxx==0)&&(maxy==0)){
        minx=pos[0];
        maxx=pos[0];
        miny=pos[13];
        maxy=pos[ 1  j  ;
        for(i=0;i<pn;i++)
        {
            if(pos[i*2]<minx) minx=pos[i*2];
            if(pos[i*2]>maxx) maxx=pos[i*2];
            if(pos[i*2+1]<miny) miny=pos[i*2+1];
            if(pos[i*2+1]>maxy) maxy=pos[i*2+1];
        }
    }
    else sc=maxy-miny;
    for(i=0;i<pn;i++)
    { 
        pos[2*i]=(pos[2*i]-minx)/sc*399;
        pos[2*i+1]=(pos[2*i+1]-miny)/sc*399;
    }
    for(i=0;i<400;i++)
    { 
        for(j=0;j<400;j++)
        { 
            mark[i][j]=0;
        }
    }
    fill(pos,edge,loop,face,fn,ln,en);
}
}

anpos=(Array)DXNewArray(TYPE_FLOAT,CATEGORY_REAL,1,2);
if(!anpos){
    DXSetError(ERROR_MISSING_DATA,"input has no color3");
    return ERROR;
}
if(!DXAddArrayData(anpos,0,160000,NULL)){
    DXSetError(ERROR_MISSING_DATA,"input has no color4");
    return ERROR;
}
npos=(float *)DXGetArrayData(anpos);
if(!npos){
    DXSetError(ERROR_MISSING_DATA,"input has no colorR");
    return ERROR;
}
if(!DXSetComponentValue((Field)(*out),"positions",(Object)anpos)){
    DXSetError(ERROR_MISSING_DATA,"input has no colorQ");
}
return ERROR;
}

ancolo=(Array)DXNewArray(TYPE_FLOAT,CATEGORY_REAL,1,3);
if(!ancolo){
    DXSetError(ERROR_MISSING_DATA,"input has no colorP");
    return ERROR;
}
if(!DXAddArrayData(ancolo,0,160000,NULL)){
    DXSetError(ERROR_MISSING_DATA,"input has no colorO");
    return ERROR;
}
ncolo=(float *)DXGetArrayData(ancolo);
if(!ncolo){
    DXSetError(ERROR_MISSING_DATA,"input has no colorN");
    return ERROR;
}
if(!DXSetComponentValue((Field)(*out),"colors",(Object)ancolo)){
    DXSetError(ERROR_MISSING_DATA,"input has no colorM");
    return ERROR;
}
DXSetComponentAttribute((Field)(*out),"colors","dep",
(Object)DXNewString("positions"));

anopa=(Array)DXNewArray(TYPE_FLOAT,CATEGORY_REAL,0);
if(!anopa){
    DXSetError(ERROR_MISSING_DATA,"input has no colorP");
    return ERROR;
}
if(!DXAddArrayData(anopa,0,160000,NULL)){
    DXSetError(ERROR_MISSING_DATA,"input has no colorO");
    return ERROR;
}
nopa=(float *)DXGetArrayData(anopa);
if(!nopa){
    DXSetError(ERROR_MISSING_DATA,"input has no colorN");
    return ERROR;
}
if(!DXSetComponentValue((Field)(*out),"opacities",(Object)anopa)){
    DXSetError(ERROR_MISSING_DATA,"input has no colorM");
    return ERROR;
}
DXSetComponentAttribute((Field)(*out),"opacities","dep",
(Object)DXNewString("positions"));

anpos=NULL;
a=0;
b=0;
i=0;
for(k=0;k<400;k++)
{
    for(j=0;j<400;j++)
    {
        npos[i]=a;
        i++;
        npos[i]=b;
        i++;
        b=b+0.025;
    }
    a=a+0.025;
b=0;
}
ancolo=NULL;
/* for(i=0;i<(480*480*3);i++)
ncolo[i]=0.5;
}*/
k=0;
k=0;
for(i=0;i<400;i++)
{
  for(j=0;j<400;j++)
  {
    if (mark[i][j]==0){ncolo[k]=0;
      ncolo[k+1]=0;
      ncolo[k+2]=0;
      nopa[ko]=0;}
    else {ncolo[k]=colo[final[i][j]]*3];
      ncolo[k+1]=colo[(final[i][j]]*3+1];
      ncolo[k+2]=colo[(final[i][j])*3+2];
      nopa[ko]=1;}
    k=k+3;
    ko=ko+1;
  }
}

/* if(!DXEndField((Field)(*out))){
    DXSetError(ERROR_MISSING_DATA,"input has no colorL"
    return ERROR;
}*/

return OK;

if(shape[0]==3)
{
  re=d3s(out);
  return re;}
return ERROR;

static Error d3s(Object *out)
{
  Array ancon=NULL;
  Array anpos=NULL;
  Array ancolo=NULL;
  Array anopa=NULL;
  float *npos,*ncolo,*nopa;
  int i,j,k,n,m,l,ko;
  float a,b,c;
  int d;
  float sc;

  int counts[3];
  float origin[3],deltas[9];
  counts[0]=20;
  counts[1]=20;
  counts[2]=20;
  origin[0]=0;
  origin[1]=0;
  origin[2]=0;
  deltas[0]=0.5;
  deltas[1]=0;
  deltas[2]=0;
  deltas[3]=0;
  deltas[4]=0.5;
  deltas[5]=0;
deltas[6]=0;
deltas[7]=0;
deltas[8]=0.5;

if((minx==0)&&(miny==0)&&(maxx==0)&&(maxy==0)&&(minz==0)&&(maxz==0)){
  minx=pos[0];
  maxx=pos[0];
  miny=pos[1];
  maxy=pos[1];
  minz=pos[2];
  maxz=pos[2];
  for(i=0;i<pn;i++)
    if(pos[i*3]<minx) minx=pos[i*3];
    if(pos[i*3]>maxx) maxx=pos[i*3];
    if(pos[i*3+1]<miny) miny=pos[i*3+1];
    if(pos[i*3+1]>maxy) maxy=pos[i*3+1];
    if(pos[i*3+2]<minz) minz=pos[i*3+2];
    if(pos[i*3+2]>maxz) maxz=pos[i*3+2];
}

if((maxy-miny)>(maxx-minx)) sc=maxy-miny;
else sc=maxx-minx;
if((maxz-minz)>sc) sc=maxz-minz;
for(i=0;i<pn;i++)
  pos[3*i] = (pos[3*i]-minx)/sc*19;
  pos[3*i+1]= (pos[3*i+1]-miny)/sc*19;
  pos[3*i+2]= (pos[3*i+2]-minz)/sc*19;
}

dfor(pos,edge,loop,face,fn,In,en);
anpos=(Array)DXMakeGridPositionsV(3,counts,origin,deltas);
if(!anpos)
  return ERROR;
if(!DXSetComponentValue((Field)("out","positions",(Object)anpos))
  return ERROR;

/*
anpos=(Array)DXNewArray(TYPE_FLOAT,CATEGORY_REAL,1,3);
if(!anpos)
  DXSetError(ERROR_MISSING_DATA,"input has no color3");
return ERROR;
if(!DXAddArrayData(anpos,0,8000,NULL)){
  DXSetError(ERROR_MISSING_DATA,"input has no color4");
  return ERROR;
}
npos=(float *)DXGetArrayData(anpos);
if(!npos)
  DXSetError(ERROR_MISSING_DATA,"input has no colorR");
  return ERROR;
if(!DXSetComponentValue((Field)("out","positions",(Object)anpos)){
DXSetError(ERROR_MISSING_DATA, "input has no colorQ");
return ERROR;
}

a=0;
b=0;
c=0;
i=0;
for(k=0;k<20;k++)
{
    for(j=0;j<20;j++)
    {
        for(l=0;l<20;l++)
        {
            npos[i]=a;
i++;
            npos[i]=b;
i++;
            npos[i]=c;
i++;

            c=c+0.5;
        }
c=0;
b=b+0.5;
    }
a=a+0.5;
b=0;
c=0;
}

ancon=(Array)DXMakeGridConnectionsV(3,counts);
if(!DXSetComponentValue((Field)(*out), "connections", (Object)ancon)) {
    DXSetError(ERROR_MISSING_DATA, "input has no connections");
    return ERROR;
}

DXSetComponentAttribute((Field)(*out), "connections", "ref", (Object)DXNewString("positions"));

ancolo=(Array)DXNewArray(TYPE_FLOAT, CATEGORY_REAL, 1, 3);
if(!ancolo) {
    DXSetError(ERROR_MISSING_DATA, "input has no colorP");
    return ERROR;
}

if(!DXAddArrayData(ancolo, 0, 8000, NULL)) {
    DXSetError(ERROR_MISSING_DATA, "input has no colorQ");
    return ERROR;
}

ncolo=(float *)DXGetArrayData(ancolo);
if(!ncolo) {
    DXSetError(ERROR_MISSING_DATA, "input has no colorN");
    return ERROR;
}

if(!DXSetComponentValue((Field)(*out), "colors", (Object)ancolo)) {
    DXSetError(ERROR_MISSING_DATA, "input has no colorM");
    return ERROR;
}

DXSetComponentAttribute((Field)(*out), "colors", "dep", (Object)DXNewString("positions"));

anopa=(Array)DXNewArray(TYPE_FLOAT, CATEGORY_REAL, 0);
if(!anopa) {
DXSetError(DXERROR_MISSING_DATA, "input has no colorP");
return ERROR;
}
if(!DXAddArrayData(anopa,0,5000,NULL)){
    DXSetError(DXERROR_MISSING_DATA, "input has no colorO");
    return ERROR;
}
nopa=(float *)DXGetArrayData(anopa);
if(!nopa){
    DXSetError(DXERROR_MISSING_DATA, "input has no colorN");
    return ERROR;
}
if(!DXSetComponentValue((Field)(*out), "opacities", (Object)anopa)){
    DXSetError(ERROR_MISSING_DATA, "input has no colorM");
    return ERROR;
}
DXSetComponentAttribute((Field)(*out), "opacities", "dep",
                        (Object)DXNewString("positions"));

l=0;
k=0;
for(i=0;i<20;i++)
{
    for(j=0;j<20;j++)
    {
        for(k=0;k<20;k++)
        {
            if (m3[i][j][k]==0){
                ncolo[1]=0;
                ncolo[1+1]=0;
                ncolo[1+2]=0;
                nopa[ko]=0;
            }
            else {
                ncolo[1]=colo[(f3[i][j][k])*3];
                ncolo[1+1]=colo[(f3[i][j][k])*3+1];
                ncolo[1+2]=colo[(f3[i][j][k])*3+2];
                nopa[ko]=1;
            }
            l=l+3;
            ko=ko+1;
        }
    }
}
return OK;

void fill(float pos[],int edge[],int loop[], int face[],int fn, int ln, int en)
{
    int x[20],y[20],sy;
    int i,j,ji,t,k,l;
    int ma,mi,up,nextpt,endpt_y,mono,adj_y,flag,temp;
    int nextli,nextei,count;
    int int_count[400];
    int int_x[400][20];
    for(i=0;i<fn;i++)
    {
        for (j=0;j<400;j++)
        {
            int_count[j] =0;
            ma=0;
            mi=400;
            if(i==(fn-1)) nextli=ln;
            else nextli=face[i+1];
            for(j=face[i];j<nextli;j++)
            {
count=0;
if(j==ln-1) nextei=en;
else nextei=loop[j+1];
for(k=loop[j];k<nextei;k++)
{
    x[count]=(int)(pos[edge[k]*2]);
y[count]=(int)(pos[edge[k]*2+1]);
count++;
}
x[count]=x[0];
y[count]=y[0];

for(l=0;l<count;l++) {/*process each edge in turn*/
    if(y[l] > ma) ma=y[l];
    if(y[l] < mi) mi=y[l]; {/*get max and min vertex y coord*/
    if(y[l] != y[l+1]) {/*only need consider non-horizontal line*/
        up=1;
        else up=-1;
        nextpt = l+1;
        endpt_y = y[nextpt];
        do { {/*skip possible horizontal line(s)*/
            nextpt = nextpt + 1;
        if(nextpt > count) nextpt = 1; /*span to the beginning*/
        }
        while((y[nextpt]==endpt_y);
        if(((up==1) &&(y[nextpt]<endpt_y)) ||(up==-1) &&(y[nextpt]>endpt_y))
            mono=0;
        else
            mono=1; /*check if monotonically increase or decrease*/
            adj_y = endpt_y - up * mono; /*adjust end point y values for edges*/
            switch(up){
                case 1:
                    if(y[l]<399)
                        { if (y[l]<0) sy=0; else sy=y[l];
                            if (adj_y<399) adj_y=399;
                                for(k=sy;k<=adj_y;k++)
                                    { int_x[k][int_count[k]]=
                                        (x[l]+(k-y[l])*(x[l+1]-x[l])/(endpt_y-y[l]));
                                        int_count[k]=int_count[k]+1;
                                    }
                                break;
                                case -1:
                                    if(y[l]>0)
                                        { if(y[l]>399) sy=399; else sy=y[l];
                                            if(adj_y>0) adj_y=0;
                                                for(k=sy;k>=adj_y;k--)
                                                    { int_x[k][int_count[k]]=
                                                        (x[l]+(k-y[l])*(x[l+1]-x[l])/(endpt_y-y[l]));
                                                        int_count[k]=int_count[k]+1;
                                                    }
                                                break;
                                        }
                                    }
                                }
                            }
                        }
                    else
                        { mi=0;
                        if(ma>399) ma=399;
                        for(l=mi;l<=ma;l++)
                            if(int_count[l]>0) /*skip degenerated polygon*/
                                for (jj= int_count[l]-1;jj>=l;jj--)
                                    for(k=0;k<jj;k++)
                                        if(int_x[l][k] > int_x[l][k+1])
                                            temp=int_x[l][k]
```c
int_x[l][k]=int_x[l][k+1];
int_x[l][k+1]=temp;
}/*if*/
for(j=mi;j<=ma;j++)
    if(int_count[j]>0)
    {
        jj=(int)(int_count[j]/2);
        for(k=0;k<=j-1;k++)
        {
            if(int_x[j][k*2]<399)
                if ((int_x[j][k*2+1]>399) int_x[j][k*2+1]=399;
                for(t=int_x[j][k*2];t<=int_x[j][k*2+1];t++)
                {
                    final[t][j]=i;
                    mark[t][j]=1;
                }
        }
    }
    return;
}

void dff(float pos[],int edge[],int loop[], int face[],int fn,int ln,
    int en)
{
    int x[20], y[20], z[20], scal, sy;
    int i,j,k,l;
    int ma,mi,up,nextpt,endpt_y, mono, adj_y, flag, temp, tempz;
    int nextli, nextei, count;
    int keep[5];
    int int_count[100];
    int int_x[100][20];
    int int_z[100][20];
    int sm,nn,num;
    for(i=0;i<fn;i++)
    {
        sm=0;
        for (j=0; j<20; j++)
            int_count[j] =0;
        ma=0;
        mi=20;
        if(i==((fn-1)) nextli=ln;
        else nextli=face[i+1];
        for(j=face[i];j<nextli;j++)
        {
            count=0;
            if(j==((ln-1)) nextei=en;
            else nextei=loop[j+1];
            for(k=loop[j];k<nextei;k++)
            {
                x[count]=(int)(pos[edge[k]*3]);
                y[count]=(int)(pos[edge[k]*3+1]);
                z[count]=(int)(pos[edge[k]*3+2]);
                /*printf("%d %d %d\n",x[count],y[count],z[count]);*/
                count++;
            }
            x[count]=x[0];
            y[count]=y[0];
            z[count]=z[0];
            if((x[0]==x[1])&&x[1]==x[2])
            {
```
sm=1;
for(num=0; num<=count; num++)
{
    temp=x[num];
x[num]=z[num];
z[num]=temp;
}
}
else{
    if ((y[0]==y[1])& (y[1]==y[2]))
    {
        sm=2;
        for(num=0; num<=count; num++)
        {
            temp=y[num];
y[num]=z[num];
z[num]=temp;
        }
    }

/*printf("sm \d\n", sm);
printf("count: \d\n", count);*/
for(l=0; l<count; l++) {
/*printf("xyz :\d \d \d \n", z[l], y[l], x[l]);*/
    if(y[l]!=y[l+1])
    {
        if(y[l] > ma) ma=y[l];
        if(y[l] < mi) mi=y[l];
        if(y[l] < y[l+1]) up=1;
        else up=-1;
        nextpt = l+1;
        endpt_y = y[nextpt];
do {
            nextpt = nextpt + 1;
            if(nextpt >count) nextpt = 1;
        } while(y[nextpt]==endpt_y);
        if((up==1) & (y[nextpt]<endpt_y)) || ((up==-1) & (y[nextpt]>endpt_y))
            mono=0;
        else
            mono=1;
        adj_y = endpt_y - up * mono;
switch (up){
    case 1:
        if(y[l]<19){
            if (y[l]<0) sy=0; else sy=y[l];
            if (adj_y>19) adj_y=19;
            for(k=sy; k<adj_y; k++){
                int_x[k][int_count[k]]=
                    (x[l]+(k-y[l])*(x[l+1]-x[l])/(endpt_y-y[l]));
                int_z[k][int_count[k]]=
                    (z[l]+(k-y[l])*(z[l+1]-z[l])/(endpt_y-y[l]));
                int_count[k]=int_count[k]+1;
            }
            break;
    case -1:
        if(y[l]>19) sy=19; else sy=y[l];
        if(adj_y<0) adj_y=0;
        for(k=sy; k>adj_y; k--){
            int_x[k][int_count[k]]=
                (x[l]+(k-y[l])*(x[l+1]-x[l])/(endpt_y-y[l]));
            int_z[k][int_count[k]]=
                (z[l]+(k-y[l])*(z[l+1]-z[l])/(endpt_y-y[l]));
            int_count[k]=int_count[k]+1;
        }
    }
if(mi<0)mi=0;
if(ma>19)ma=19;
for(l=mi;l<=ma;1++)
  if(int_count[l]>0)
    for jj= int_count[l]-1; jj>=1; jj--)
      for(k=0;k<jj;k++)
        if(int_x[l][k] > int_x[l][k+1]){
          temp=int_x[l][k];
          tempz=int_z[l][k];
          int_x[l][k]=int_x[l][k+1];
          int_z[l][k]=int_z[l][k+1];
          int_x[l][k+1]=temp;
          int_z[l][k+1]=tempz;
        }
  for(j=mi;j<=ma;j++)
    if(int_count[j]>0)
      {jj=(int) (int_count[j]/2);
      for(k=0;k<jj-1;k++)
        if((int_x[j][k*2]<19)&&(int_x[j][k*2+1]>=0)){
          if(int_x[j][k*2]<0)
          {int_z[j][k*2]=(int_z[j][k*2+1]-int_z[j][k*2])*2
            / (int_x[j][k*2+1]-int_x[j][k*2]);
            int_x[j][k*2]=0;}
          else {if((int_x[j][k*2+1])>19)
             {int_z[j][k*2+1]=(int_z[j][k*2+1]-int_z[j][k*2])*2
              / (int_x[j][k*2+1]-int_x[j][k*2]);
              int_x[j][k*2+1]=19;}}
        }
    }for(t=int_x[j][k*2];t<=int_x[j][k*2+1];t++)
      {zcal=(int_z[j][k*2+1]-int_z[j][k*2])*2
        / (int_x[j][k*2+1]-int_x[j][k*2]);
        if((zcal>=0)&&(zcal<=19))
          {if(s==0){
              f3[t][j][zcal]=i;
              m3[t][j][zcal]=1;}
            else{if(s==1){
              f3[zcal][j][t]=i;
              m3[zcal][j][t]=1;}
              else {
              f3[t][zcal][j]=i;
              m3[t][zcal][j]=1;}}
        } /*printf("coordinate :%d %d %d %d
",t,j,zcal,i);*/
    }
/*for(l=0;l<20;l++)
  {for(j=0;j<20;j++)

{ jj=0; 
  for(k=0;k<20;k++) 
  { 
    if(m3[1][j][k]==1) {keep[jj]=k;jj++;} 
  } 
  if(jj!=0){
    t=0;
    while(t<=(jj-2)) 
    { 
      for(k=keep[t];k<=keep[t+1];k++) 
      { 
        f3[1][j][k]=0;
        m3[1][j][k]=1;
      }
      t=t+2;
    }
  }
  */
  return;
}