ALGORITHMS FOR MANIPULATING TRIANGULATED SURFACES

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston-University Park

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Dang-Cheng Yiu

May, 1987

ACKNOWLEDGEMENT

The author wishes to express his sincere gratitude to Dr. Anne L. Simpson, the chairman of thesis committee, for her guidance and valuable advices during the period of the research.

Special thanks are gratefully extended to Dr. Ramez A. Elmasri and Dr. Richard Sanders, members of thesis committee, for their suggestions and comments.

The author wants to thank the Allied Geophysical Laboratories for the equipment and software support through the entire research.

Finally, the author deeply appreciates the strong support and encouragement of his wife and his family throughout the study at the University of Houston.

ALGORITHMS FOR MANIPULATING TRIANGULATED SURFACES

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston-University Park

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Dang-Cheng Yiu

May, 1987

ABSTRACT

Triangulations are powerful tools for surface modeling. They can be used to fit any irregular boundary shapes and surface discontinuity fault patterns. However, without the help of row and column information as in rectangular grids, manipulating a triangulated surface interactively is not an easy task. Efficiency is a major concern for manipulation functions.

This article will propose some algorithms to manipulate triangulated surfaces such as: finding the path of a given route in a triangulated surface; getting the profile of a vertex function along a given route; and partitioning a surface domain by vertical plane(s). These algorithms have applications to interactive graphics where the user wishes to slice multisurface folds to obtain various views of a solid, and to dynamical problems where one wishes to introduce fracture systems into pre-existing surfaces. Some other useful operations such as merging of two domains, moving a vertex with preset rules, and combined use with rectangular grids, are also discussed.

The proposed algorithms have been implemented in a triangulation database system developed by W.M. Smith of Cullen Image Processing Laboratory at University of Houston. This process requires both expansion and modification of Smith's system. The principal modification introduced enables the system to treat multiple surfaces in main memory rather than only one surface at a time from a file database.

The algorithms can be applied to any multi-connected surface domain.

Holes and irregular boundary conditions are given careful treatment. A theoretical analysis of time complexity is not available at this stage. Empirical results for the time required by these algorithms are given instead.

TABLE OF CONTENTS

1. INTRODUCTION

Triangulations are powerful tools for surface modeling. Triangulated grids are more flexible than quadrilateral grids in many respects. (1) They can adapt easily to arbitrarily spaced data. (2) Each triangle can be assumed to represent a small flat plane. (3) As many functions as necessary can be attached to each vertex, side and triangle to describe the characteristics of the surface, and the function values of any point within surface domain can be easily calculated by interpolation. (4) They can fit in any irregular boundary shapes and surface discontinuity fault patterns. Due to the flexibility of triangulations, it is very convenient to be used in geophysical applications.

Earth is a continually changing planet. The structure of the earth's surface is quite complicated because of the motion in the crust of the earth and in the folded mountain belts. The 3-D interactive graphic representation for the structure of earth's surface is difficult because of multisurface folds, fractures, faults and irregular boundary shapes of the surfaces. Triangulations have no problem in fitting such boundary shapes. Fractures and faults of the earth can be represented by internal boundaries defined in the triangulation system. Multisurface folds can be simulated by a stack of triangulated surfaces. Each surface of the stack represents a folded surface between layers.

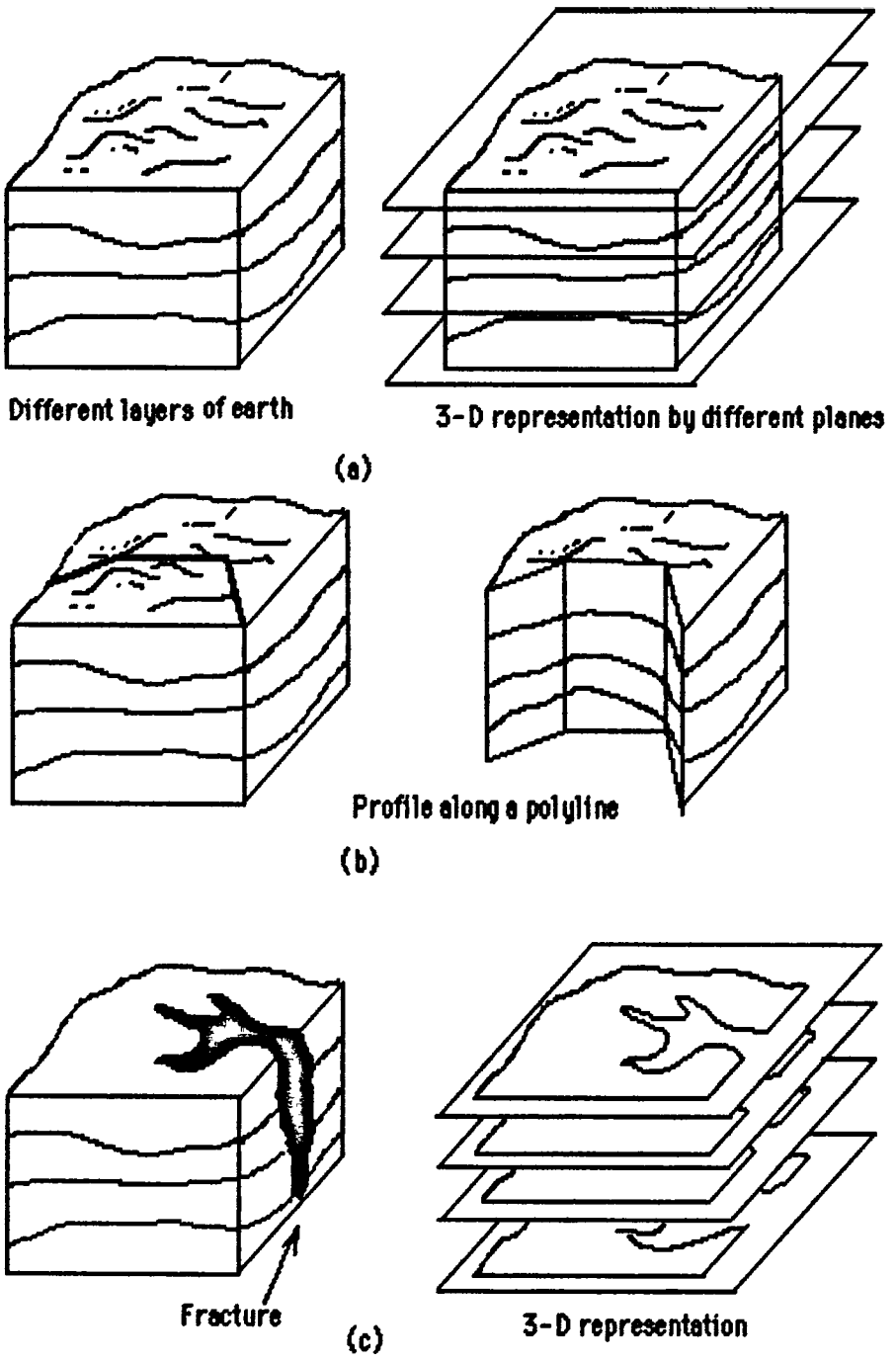


Figure 1: Triangulation system in geoscience applications (a) 3-D graphic representation (b) the profile of a solid along a polyline (c) fracture of a solid

Figure 1-a shows a solid which is a sample at the surface of the earth. To represent this solid in 3-D graphics by triangulations, various views of the solid can be obtained by slicing multisurface folds. Triangulate each view plane, then a stack of triangulated surface domains based on different planes can be used to represent the depth relation of the solid. By the stack of triangulated surface domains, display and manipulation of the solid can be done easily. Figure 1-b shows the profile of the solid along the black lines. If a fault or a fracture is on the solid as shown in Figure 1-c, the results of the fracture will cause a portion of each triangulated surface domain to be cut off.

Using triangulations to represent the complicated earth system in 3-D graphics, some manipulating functions for triangulation system are necessary. The implemented functions of interest are (1) converting a finite line in the x-y plane into a triangulated line in triangulated surface, (2) getting the tile-function values (profile) along a triangulated line, and (3) partitioning of a surface by a finite vertical plane and updating attached functions. The basic finite cut and line conversion alogrithms can then be used in a loop to introduce a very complex fault system into an initially continuous surface.

There are other applications which will be discussed in this thesis but their implementation is left for future work. Those applications include merging two isolated surface domains into a single surface domain, partitioning a surface by a finite tilt plane, varying vertices within preset constraints, and combined application with rectangular grids.

This thesis devotes itself to algorithms for manipulating surfaces stored in triangulation databases. Implementations of those algorithms were based on the IPL (The Cullen Image Processing Laboratory) T-Base, created by [Smith]. Many IPL library routines were also used and modified during implementation.

Chapter two is an introduction to the triangulation system. Included are teminologies and notations used in this thesis, assumptions regarding surface domains, some properties defined on surfaces, data structures in our triangulation database, and descriptions of new manipulating functions and different applications.

Chapter three will discuss the algorithms of the manipulation functions along with some examples. Chapter four will discuss some techniques used and problems faced during implementation.

An overview of the revised and expanded Smith System in IPL is given in APPENDIX A. APPENDIX B gives the titles and descriptions of new FORTRAN subroutines used in the new system.

2. TRIANGULATION SYSTEM AND MANIPULATING FUNCTIONS

The triangulation system used in this thesis is basically the same as the IPL T-Base [Smith] with both expansion and modification. The major restriction of IPL T-Base is that it can only load one surface domain into memory at one time. Once a surface domain is released, all data will be dropped from memory. This is not convenient for applications which need to access more than one surface domain during a single run, such as merging two surface domains into a single domain, cutting a surface domain into many separated surfaces. The new version of the [Smith] system after expansion eliminates this restriction. The memory address tables have been expanded so that the system can load more than one surface domain into main memory as long as the memory space is available. Each surface domain in memory has its own name and domain index for identification, and the file name of the database will be used as its domain name in the implementation. To save memory space, the storage spaces of each surface domain are dynamically allocated upon request. When a surface domain request a block of memory space, the database handler will acquire the space needed from the operating system and record the address and size of the block in memory. To access the data item in memory, the database handler simply goes to check the address table and calculate the exact address which is then used to perform the read/write operation on that data item.

Assumptions and limitations of the surface domain, the database structures, and a discussion of the manipulation functions will be given in the following sections.

2.1 TERMINOLOGY AND NOTATION

Triangulations consist of three basic components: vertex (V), side (S), and triangle (T). For completeness of mathematical model, it is assumed that each vertex is a closed point; each side is an open line segment without two endpoints and each triangle is an open area without three sides and three endpoints. Therefore, the area of a surface domain D is equal to the union of V, S and T.

Let V be the set of vertices of a triangulation and NV be the number of vertices. V is defined as

$$V = \{v_i = [x_i, y_i] \mid 1 \leq i \leq NV\}$$

where x_i and y_i are coordinates of v_i on base plane XY. Two or more vertices may have same coordinates.

Let S be the set of sides of a triangulation and NS be the number of sides. S is defined as

$$S = \{s_j = [v_{j1}, v_{j2}] \mid v_{j1} \text{ and } v_{j2} \text{ are in V and } 1 \leq j \leq NS\}$$

Two or more sides may have the same endpoints. The direction of side s_j is defined from point v_{j1} to point v_{j2}. Any point p on side s_j can be expressed

as $s_j(u_1)$ where

$$s_j(u_1) = (1-u_1)xv_{j1} + u_1xv_{j2}$$

u_1 = (length of p to v_{j1}) / (length of v_{j2} to v_{j1})

and $0.0 < u_1 < 1.0$

Let T be the set of triangles of a triangulation and NT be the number of triangles. T is defined as

$T = \{t_k = [s_{k1}, s_{k2}, s_{k3}] \mid |s_{k1}|, |s_{k2}|,$ and $|s_{k3}|$ are in S and $1 \leq k \leq NT\}$

$s_{k1} = [v_{j1}, v_{j2}]$, $s_{k2} = [v_{j2}, v_{j3}]$, and $s_{k3} = [v_{j3}, v_{j1}]$

If s_{ki} is negative, then $s_{ki} = [v_{j(i+1)}, v_{ji}]$

Any point p on triangle T_k can be expressed as

$$T_k(u_1, u_2) = (1-u_1-u_2)v_{j1} + u_1xv_{j2} + u_2xv_{j3} \text{ with}$$

$0.0 < u_1 < 1.0$

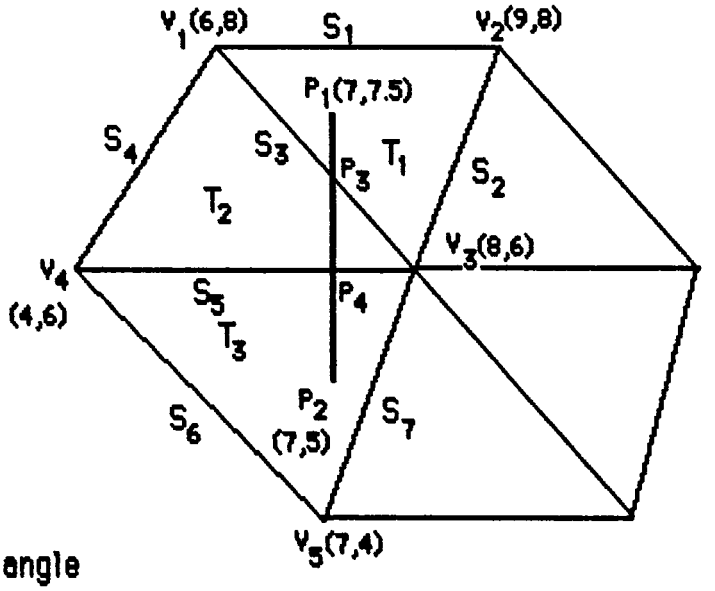
$0.0 < u_2 < 1.0$

and $(u_1 + u_2) < 1.0$

This is the representation of a traversal of a triangulation.

A line segment is usually defined by its two endpoints in x-y coordinate system. In a triangulated surface the triangulation objects traversed by the given line segment are more useful than the endpoints of the line segment. Triangulated line L_{tri} is defined as the complete path of given line segment L on a triangulated surface domain. It includes all

triangulation objects traversed by the line segment L.



Triangle

$T_1(S_1, S_2, S_3)$

$T_2(S_3, S_4, S_5)$

$T_3(S_5, S_6, S_7)$

Side

$S_1(V_1, V_2)$

$S_2(V_2, V_3)$

$S_3(V_3, V_1)$

$S_4(V_1, V_4)$

$S_5(V_4, V_3)$

$S_6(V_4, V_5)$

$S_7(V_5, V_3)$

Line segment L from P_1 to P_2

Triangulated line L_{tri}

type	Index	u_1	u_2
2	T_1	0.16667	0.25000
2	T_1	0.00000	0.50000
1	S_3	0.50000	-
2	T_2	0.50000	0.00000
2	T_2	0.00000	0.25000
1	S_5	0.75000	-
2	T_3	0.25000	0.00000
2	T_3	0.12500	0.50000

Figure 2: Triangulated line L_{tri} from point P_1 to P_2

L_{tri} is composed of a string of points. Each point P_i on L_{tri} can be

represented by four 4-byte items: type, index, u_1 and u_2. Type tells whether the current point is on a vertex, a side or inside a triangle. Type zero means P_i is on a vertex, type one means P_i is on a side and type two means P_i is inside a triangle. Index identifies the vertex, side or triangle where P_i is located. Both u_1 and u_2 are the local coordinates of P_i. If P_i is on a vertex, then both u_1 and u_2 are equal to 0.0. If P_i is on side S, then $S(u_1) = P_i$ and u_2 is equal to 0.0. If P_i is inside triangle T, then $T(u_1, u_2) = P_i$. One point in the x-y coordinates may have more than one point in the expression of triangulated line to form the completeness. For a line segment L from point P_1 to point P_2 on surface domain D, the triangulated line L_{tri} of L on D is defined as in Figure 2. L_{tri} starts from P_1 on triangle T_1, crosses side S_3 at P_3 to triangle T_2, then crosses side S_5 at P_4 to triangle T_3, and ends at point P_2. It can be seen from Figure 2 that each time when P_i exits or enters a triangle there are three different local expressions of P_i to record where P_i came from, where P_i is now and where P_i is going to.

2.2 SURFACE DOMAIN

The surface domain used in our triangulation system is a multi-connected domain under some restrictions. There are two major assumptions for the surface domain.

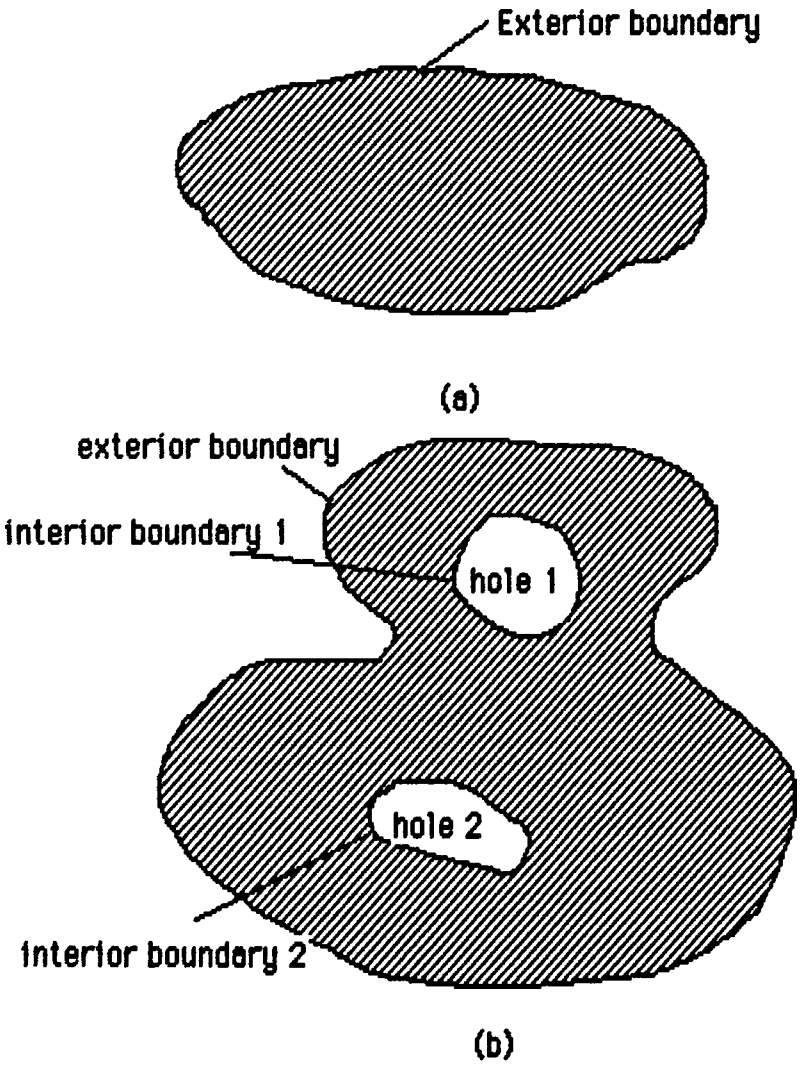


Figure 3: Surface domain in x-y plane (a) a surface domain with no hole (b) a surface domain with two holes

The first assumption is that a surface domain is the projection of a surface patch to a base plane in x-y coordinate system. Each x-y point within the surface domain has only one corresponding point to the surface patch. The second assumption is that a surface domain can be any irregular shapes bounded by a closed exterior boundary as shown in Figure 3. It is

important that there can be only one exterior boundary for each surface domain. In other words, two isolated areas (bounded by two different exterior boundaries) represent two separated domains. Within the surface domain there may be holes which are bounded by closed interior boundaries. Each surface domain may have any number of interior boundaries. Each boundary consists of a sequence of boundary sides which are connected by consecutive pairs of boundary vertices. The domain area is confined by its exterior boundary and interior boundaries. Each vertex within the surface domain is either an interior vertex or a boundary vertex located on one and only one boundary. Usually the exterior boundary will be stored in a counterclockwise direction and the interior boundary in a clockwise direction so that surface domain is always to the right of the boundary.

2.2.1 FUNCTIONS DEFINED ON SURFACE DOMAIN

A surface can be divided into many discrete areas according to the characteristics of its own. In the triangulation system each small area is a triangle. The characteristics of the surface is described by the functions attached to vertex, side or triangle. For example, we may attach the depth to each vertex, or we may attach color, density and illumination to each triangle, or attach the slope of the edge to each side. If a function value is missing, a flag should be set to indicate the function value is undefined.

When a new vertex is inserted into a surface domain, new sides and

new triangles are also created by locally triangulating the area around the new vertex. The function values of these new vertex, sides and triangles are difficult to determine without knowing the physical meaning of attached functions. For example, a triangle function TF_k is defined as the area of the associated triangle and another triangle function TF_j is defined as the density of the associated triangle. When a triangle T is split into triangle T_1 and triangle T_2, function values of TF_k of T_1 and T_2 should be recalculated by their areas and function values of TF_j of T_1 and T_2 should be the same as the function value of TF_j of T. In the implementation, system will ask the user to fill function values for new elements by default methods or set flags to indicate missing function values. The default method for vertex functions is using linear interpolation according to the location of new vertex. The default method for side functions depends on new side s_{new}. If new side s_{new} is a sub-side of an existing side S, then it has the same function values as S. If s_{new} is not a sub-side of any existing side, then null values are assigned and flags are set. The default method for triangle functions of a new triangle T_{new} is to copy the values of a existing triangle T which contains T_{new}.

2.3 STRUCTURE OF TRIANGULATION DATABASE

Memory Address Table for each domain in memory

	Domain 1	Domain 2	.....	Domain max.
Vertex Block				
SVP Block				
TSP Block				
Boundary Block				
VMAP(old) Block				
VMAP(new)Block				
VF 1 Value Block				
Missing VF 1 Flag				
⋮				
SF m Value Block				
Missing SF n Flag				
TF 1 Value Block				
Missing TF 1 Flag				
⋮				
TF m Value Block				
Missing TF n Flag				

Figure 4: Memory address table for each domain stored in memory

As mentioned before, memory blocks of each data entity are allocated upon request. After memory allocation, the system will return a relative address based on a base address for that entity. To allow more than one

surface domain in memory, larger memory address tables are necessary to record the extra addresses (See Appendix A.1 for the description of address tables which are declared in file "TDBCOMMON.FOR"). Figure 4 shows the structure of the memory address tables. The maximum number of surface domains which can be loaded into memory is controlled by parameter MAX_D. MAX_D is declared in the file "TDBCOMMON.FOR" and is currently set to ten. Each domain in memory is identified by its domain index which is in the range one to MAX_D. If the value of MAX_D is changed, all subroutines, including file "TDBCOMMON.FOR", should be recompiled. Different data blocks have their own names with MAX_D array elements. The index of the data block represents the index of domain. Access a particular domain in a data block can be done easily by giving the domain index. Variable ACTIVE_DOMAIN (declared in the file "TDBCOMMON.FOR") indicates the current working domain. The value of ACTIVE_DOMAIN can be changed to switch surface domains in memory.

The basic elements of each surface domain are vertices, sides and triangles. Each triangle has three sides and three vertices. Each side has two end vertices. Each vertex has its x and y coordinates. We therefore have the following relationships:

a. Vertex determines the values of x and y coordinate.
b. Side determines two end vertices.
c. Triangle determines both three sides and three end vertices.

The following three data blocks are used to represent the

relationships for each surface domain:

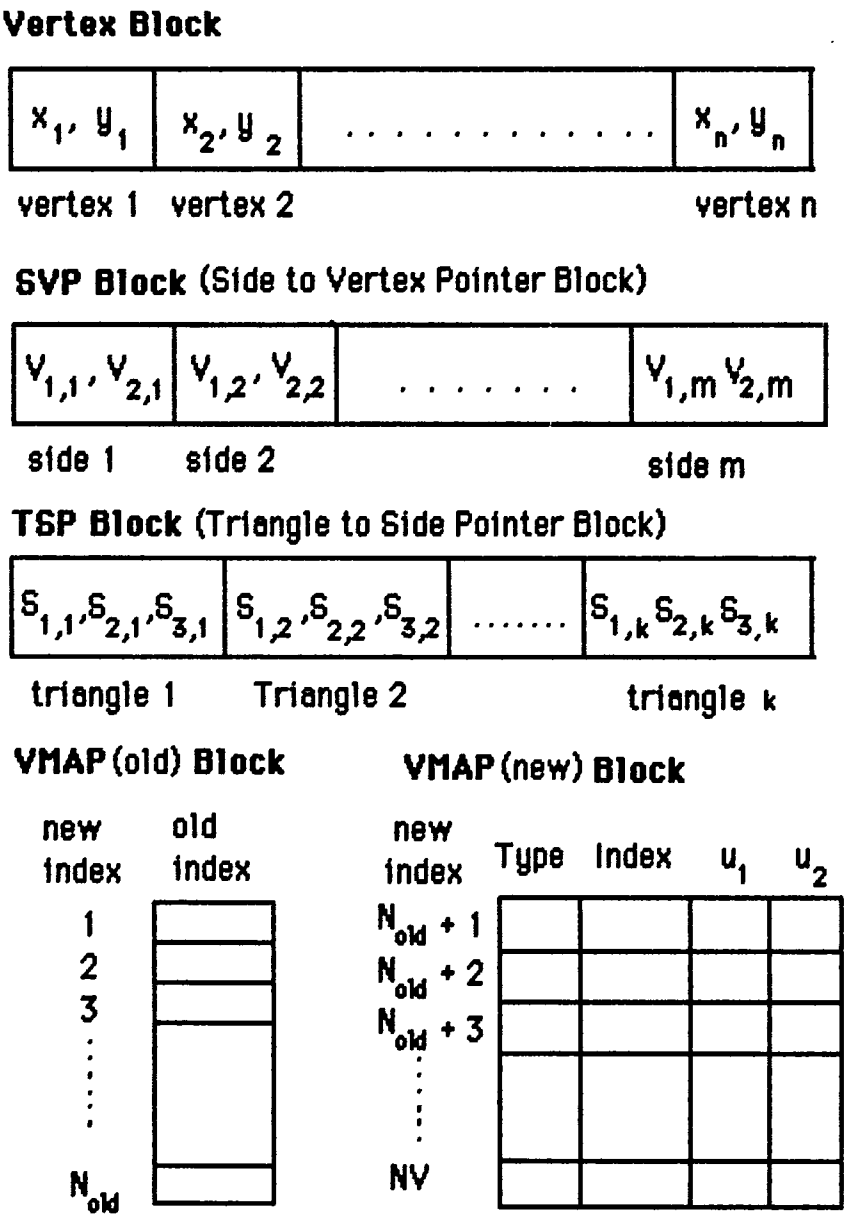


Figure 5: Data structure of vertex block, SVP block, TSP block and VMAP blocks

(1) **VERTEX BLOCK** (See Figure 5) is used to store the x and y values of each vertex. The first pair of elements are x_1 and y_1 of the first vertex; the second pair of elements are x_2 and y_2 of the second vertex, and so on. NUM_V(i) represents the number of vertices in the domain i. V_SLOTS(i) represents the number of vertices can be stored in the vertex block. Usually, a few more spaces than needed are reserved for the possible future expansion.

(2) **SVP BLOCK** (See Figure 5) is used to store the end vertices of each side. The first pair of vertex indices are end vertices of the first side; the second pair of vertex indices are end vertices of the second side, and so on. NUM_S(i) represents the number of sides in surface domain i. S_SLOTS(i) represents total number of sides can be stored by this data block.

(3) **TSP BLOCK** (See Figure 5) is used to store the sides of each triangle. The first triplet of side indices are three sides of the first triangle; the second triplet of side indices are three sides of the second triangle, and so on. NUM_T(i) represents the number of triangles in surface domain i. T_SLOTS(i) represents the number of triangles can be stored by this data block. If we want to know the three vertices of a triangle, we can find them indirectly by finding sides first, then vertices.

There are two data blocks: **VMAP_OLD** and **VMAP_NEW** (See Figure 5), which are only used in the cutting process to map the vertices in a new domain back to the original domain. VMAP_OLD is used to store the

mapping of vertices which are existed before cutting, and VMAP_NEW is used to store the mapping of vertices which are new created during cutting back to the original domain. The representation of VMAP_NEW is the same as the triangulated line discussed in section 2.1. In the line conversion process (see Section 2.4.1), VMAP_NEW is used to store the path of the triangulated line. In the tile function process (see Section 2.4.2), VMAP_OLD is also used to store the data of a profile.

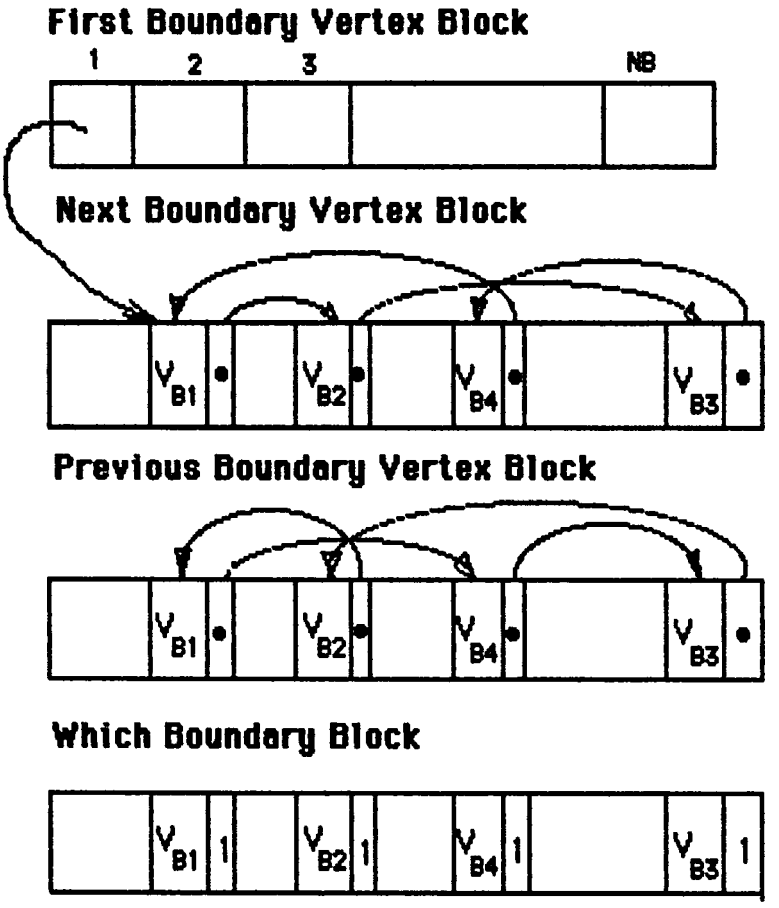


Figure 6: Data structure for boundary vertex blocks

To define the boundary, two data blocks are used to store the necessary information. One is The **First Boundary Vertex Block** and the other is **Boundary Vertex Block** as shown in Figure 6. The FIRST BOUNDAY VERTEX BLOCK is a one dimensional array which stores the first vertex of each boundary. Each boundary is given an index for identification and the exterior boundary is always the boundary one. The index of this array is the index of the boundary. Maximum number of boundaries allowed in a surface domain is controlled by parameter MAX_BOUNDARY, declared in the file "TDBCOMMON.FOR". The current maximum number of boundaries within a surface domain is set to twenty. To facilitate the boundary data retrieving, the Boundary Vertex block is divided into three data segments having the same size as V_SLOTS (See Figure 6). The index of array of each segment represents the index of vertex. The first segment is called "Next Boundary Vertex Block", the second segment is called "Previous Boundary Vertex Block" and the third segment is called "Which Boundary Block". Functions of the three segments are discussed below:

(1) The **NEXT BOUNDARY VERTEX BLOCK** is used as a pointer. The value of each element represents the index of the next boundary vertex following the direction of its boundary. For example, if the next boundary vertex of vertex 1 is vertex 7, then the first element of this block will store seven. If a vertex is not on the boundary, set the value to zero. The last vertex of a boundary will point to the first boundary vertex so that the whole boundary can be wrapped around.

(2) The **PREVIOUS BOUNDARY VERTEX BLOCK** is the same as the

next boundary vertex block except that each vertex points to the previous boundary vertex of itself.

(3) **WHICH BOUNDARY BLOCK** represents the boundary where the vertex lies on. For example, if the vertex 7 lies on boundary 2, then the 7^{th} element of this block will store 2. If a vertex is not a boundary vertex, then set the value to zero. Note that the first boundary of a surface domain always represents the exterior boundary.

As mentioned before, there are three types of functions, vertex, side and triangle functions. The number of functions can be as many as necessary. For example, N vertex functions, M side functions and K triangle functions are used to describe the different properties associated with a surface domain. Each function will need a data block to store those function values, therefore there are N vertex function blocks, M side function blocks and K triangle function blocks (See Figure 7). Function values may be missing, so a missing function value flag block will be needed for each function to indicate missing function values. Therefore, we have the following function data blocks:

(1) A **VERTEX FUNCTION VALUE BLOCK** for each vertex function. This is an array which has V_SLOTS elements. The i^{th} element of this block represents the function value of i^{th} vertex.

(2) A **MISSING VERTEX FUNCTION VALUE BLOCK** for each vertex function. This block is an array of boolean with the same length as vertex function value block. Each element is corresponding to a vertex function value. If the function value of i^{th} vertex is missing, then set i^{th} element of

this block to true. Otherwise set it to false.

(3) A **SIDE FUNCTION VALUE BLOCK** for each side function. This is an array which has S_LOTS elements. The i^{th} element of this block represents the function value of i^{th} side.

(4) A **MISSING SIDE FUNCTION VALUE BLOCK** for each side function. This block is an array of boolean with the same length as side function block. Each element is corresponding to a side function value. If the function value of i^{th} side is missing, then set i^{th} element of this block to true. Otherwise set it to false.

(5) A **TRIANGLE FUNCTION VALUE BLOCK** for each triangle function. This is an array which has T_SLOTS elements. The i^{th} element of this block represents the function value of i^{th} triangle.

(6) A **MISSING TRIANGLE FUNCTION VALUE BLOCK** for each triangle function. This block is an array of boolean with the same length as triangle function value block. Each element is corresponding to a triangle function value. If the function value of i^{th} triangle is missing, then set i^{th} element of this block to true. Otherwise set it to false.

In addition to the above data structures, there are some other control variables and scratch areas associated with each surface domain. Details will be shown in APPENDIX A.1.

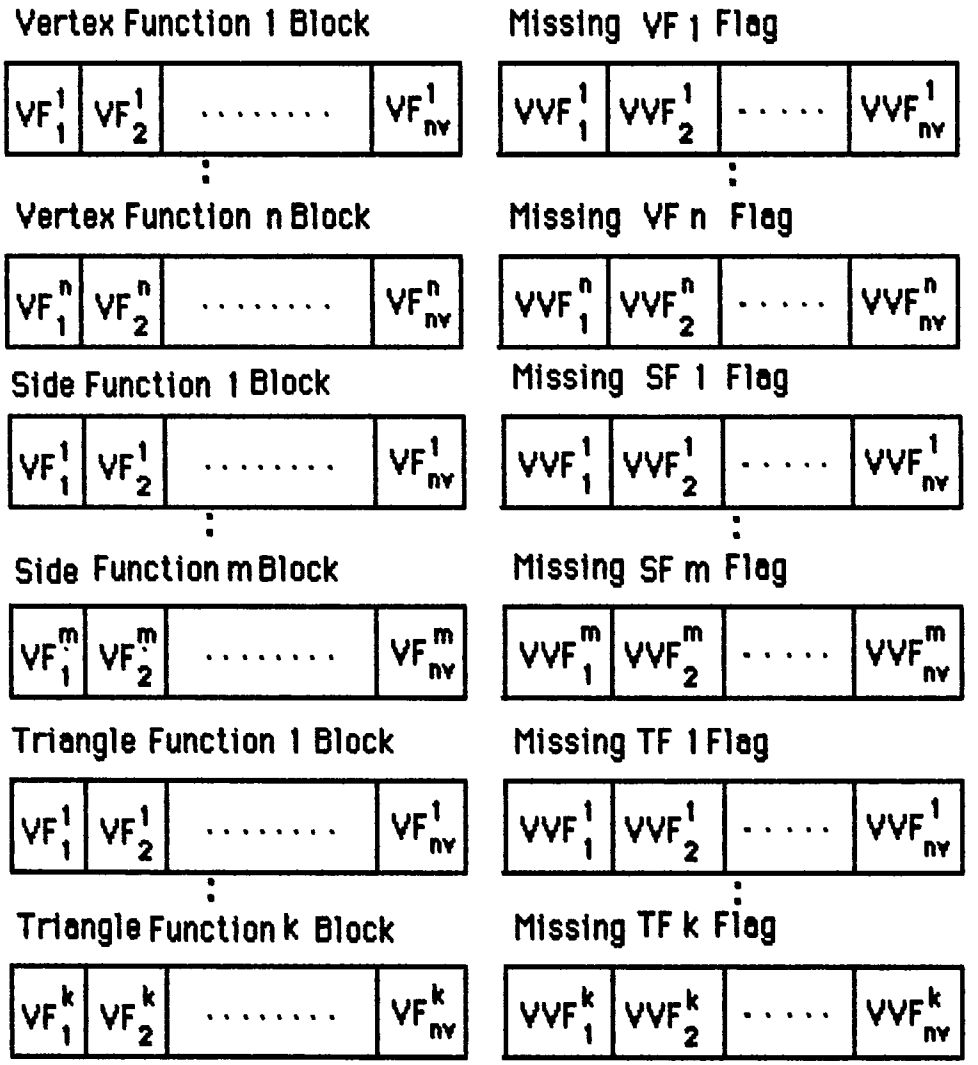


Figure 7: Vertex, side and triangle function value blocks and the missing function value flags for each function

2.4 THE MANIPULATION FUNCTIONS

Manipulating functions of interest in this thesis are more advanced

and complicated than the basic functions in the IPL database handler. The previous functions implemented by Smith are loading database, releasing database, writing database file, and functions to retrieve information without changing the triangulations. New manipulating functions are working on triangulations associated with lines, areas, or more than one surface domain. The input of new functions is a string of points to represent a polyline. Triangulations traversed by the polyline depends on length and location of the polyline and the surface itself. The path of the polyline has to be found first; then operate the function along the path. As the targets of functions are not fixed, scannings and searchings are used a lot. Efficiency of the algorithms is more important than before.

Functions discussed here include: line conversion, tile function values along a triangulated line, and partitioning of a surface. Some other applications such as merging domains, moving a vertex, and combined application with rectangular grids are also discussed in this section.

2.4.1 LINE CONVERSION

In many cases the targets of an operation are a set of triangulation objects associated with a given operating route (whether it is a line, a polyline ,or a curve). An example of this is to get the cross section of a surface domain along given route L. In this case the triangulation objects passed by L should be found first; then retrieve the values of the function along the path. The interesting thing here is not L itself but the set of

triangulation objects which are traversed by L. The function of line conversion is therefore developed to determine the set of triangulation objects passed by a route. Its purpose is to find the complete path of route L on the triangulated surface. The path of L on the surface is L_{tri} which is defined in section 2.1. (See Figure 1). The advantages of knowing the path of L are:

1. It shows not only the endpoints and turning points of route L but also a sequence of vertices, sides and triangles which were passed by L in the triangulated surface. These vertices, sides and triangles will be affected by the operation.
2. The x and y coordinates of each intersection point between route L and triangulations can be easily calculated from u_1 and u_2, the coefficients of corresponding points on L_{tri}.

If the route is a curve, we can substitute it by a polyline, and operate one line segment at a time.

Line conversion is a primary function. It can be the preliminary process of many applications to find the targets of the operation. Once the path of the operation is found, a sequence of operations can be applied to the associated triangulations within a loop. The algorithm and an example will be given in the next chapter.

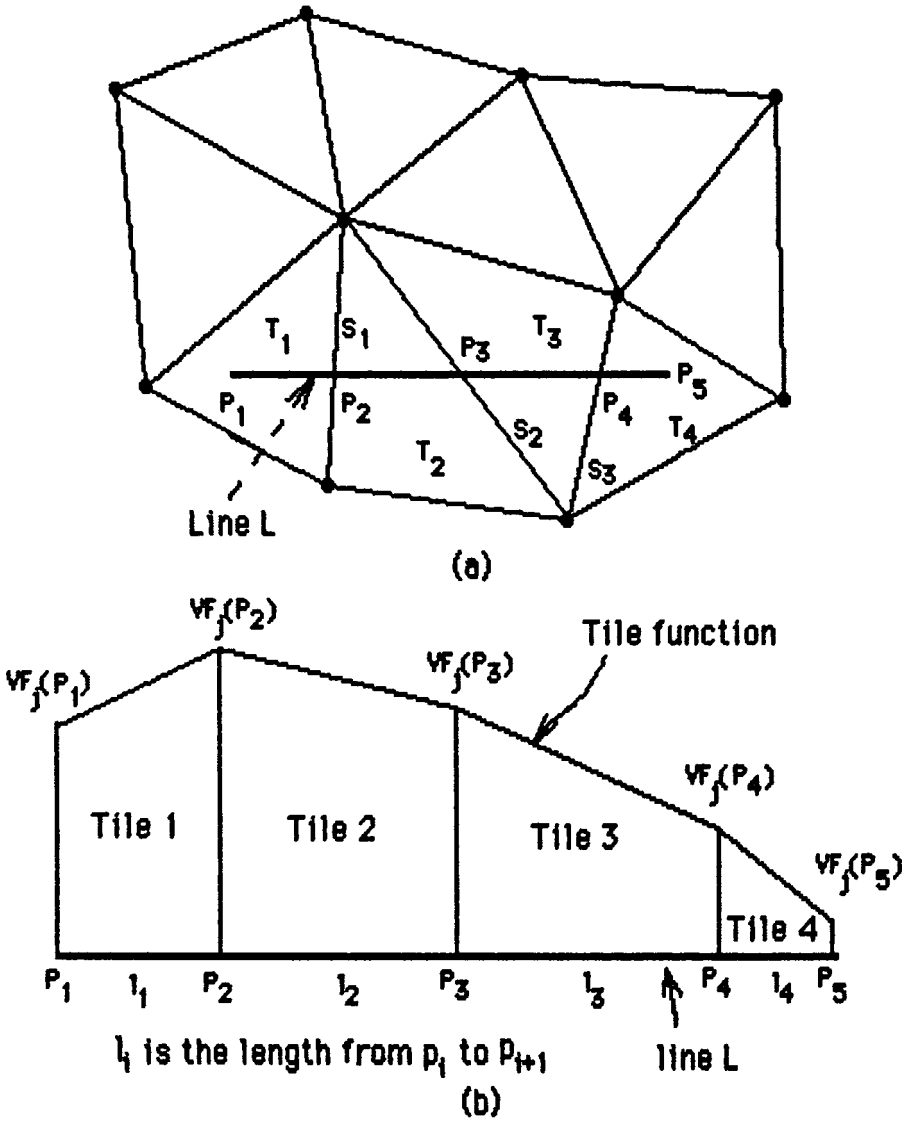


Figure 8: The tile function (a) line segment L and a triangulated surface (b) Tile function (VF_j) along line segment L

2.4.2 TILE FUNCTION

For any three dimensional object or surface, it is always interesting to see the cross section or the profile of a particular function along a

given route L. The purpose of tile function is to get a sequence of vertex function values at the intersection points between L and triangulation objects along the direction of route L. The profile along L can then be interpolated by the results of tile function.

For example, assume that the profile of j^{th} vertex function VF_j along a line L of Figure 8-a is wanted. To get the profile of triangulation objects along L the first step is to find the path of L. With the help of line conversion function, path of L which started from point P_1 then intersected sides S_1, S_2 and S_3 at points P_2, P_3 and P_4 respectively and ended at point P_5 can be found. The values of VF_j at all these five points can be calculated by interpolation according to their locations. If P_i is inside a triangle T_k, $VF_j(P_i)$ can be interpolated by the j^{th} vertex function values of three end vertices of T_k. If P_i is on a side S_m, $VF_j(P_i)$ can be interpolated by the j^{th} vertex function values of two end vertices of S_m. The profile between points P_i and P_{i+1} can then be interpolated by $VF_j(P_i)$ and $VF_j(P_{i+1})$. The profile of j^{th} vertex function along line L is shown in Figure 7(b). Each section of the profile between points P_i and P_{i+1} is called a tile, and the whole profile is called tile function. The width of a tile is the distance between points P_i and P_{i+1}.

2.4.3 SURFACE PARTITIONING

Inserting a fault into a surface domain and decomposing a surface with a complicated fault system into a set of new domains to provide better visibility and workability are two of many applications which need a partition function to achieve their purpose.

Partition of a surface domain means that a triangulated surface domain D is cut by a route and results in a set of new domains $\{D_i\}$. Each new domain D_i has its own sets of vertices, sides, triangles, boundaries and vertex mapping tables (created while cutting). Many vertices, sides, and triangles will be modified or created. By vertex mapping tables all vertices in D_i can be mapped back to original surface domain D. There are two kinds of mapping table: old vertices mapping table and new vertices mapping table. Old vertices mapping table is used by vertices which existed before cutting and new vertices mapping table is used by vertices which are created during the cutting. The value of each element in old vertices mapping table is the index of vertex before cutting. The value of each element in new vertices mapping table has the same format as triangulated line to show its location in the original domain. If a side in a new domain D_i contains a new vertex as one of its endpoints, then this side is a new side or a modified side. The same scheme can be applied to a triangle as well. If a triangle in a new domain D_i contains a new vertex as one of its endpoints, then this triangle is a new one or a modified one.

Function values of new or modified vertices, sides and triangles can

then be interpolated or readjusted according to the original surface. The completeness of a triangulation system means that the complete surface can be reconstructed without losing necessary information. The ultimate goal of partition of a surface domain is not only to perform the cutting operation to a surface domain but also to keep the completeness of all new domains as possible as we can. With the problem of filling the function values for those new vertices, sides, and triangles (see section 2.2.1), the second efforts by the user to fill or readjust the function values after partitioning is always necessary.

During the partitioning, a surface domain may be split into two. The process of splitting a domain is very expensive and it is easy to make mistakes. Vertices of both domains are renumbered so that the contents of SVP blocks have to be updated. Sides and triangles are renumbered so that the TSP blocks also have to be updated. Boundary lists, function value blocks, missing function value flag blocks, and mapping tables are all needed to update. The updating sequence of different data blocks is very important. Otherwise, some information will be lost.

The algorithm of partition function will be discussed in next chapter and some technical problems faced will be discussed in chapter four.

2.5 OTHER APPLICATIONS

There are many other useful applications in triangulation system that can be developed in the future. In the following sections, applications

of merging two surface domains, moving a vertex to a new location, and combined use with rectangular grids will be discussed.

2.5.1 Merging two domains

Decomposition of a surface with a complicated fault system into a set of smaller surfaces is a popular application. To construct a complicated system by a set of surfaces is also very important. The purpose of merging function is to merge two separated domain areas into a single and connected surface domain. Sometimes merging can be seen as the reverse function of partition of a surface but not exactly. To merge two domains D_1 and D_2 into a single domain D (See Figure 9), some assumptions have to be made:

(1) D_1 and D_2 have to be the same type of surfaces with the same number of functions attached to them.

(2) No overlapping between D_1 and D_2. If they overlapped together, inconsistency may occur at the overlapping area.

(3) **Connecting area** which is located between D_1 and D_2 and will be a part of D should be defined properly. New sides and new triangles will be created after triangulating this new area and the function values of them will be null.

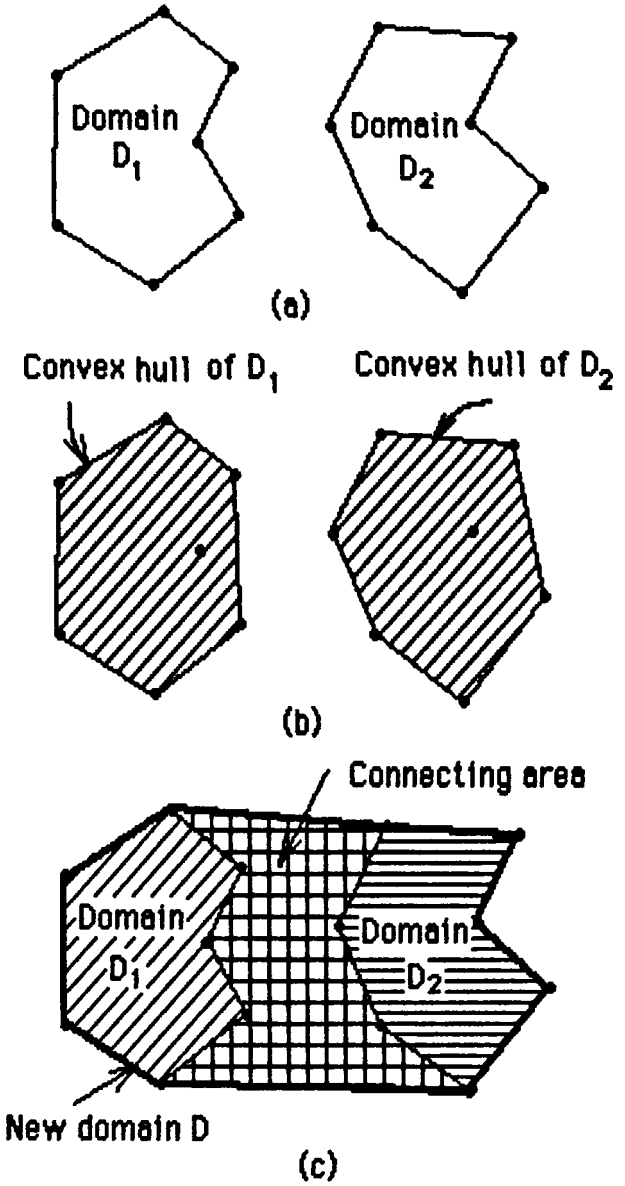


Figure 9: Merging two domains (a) Domain D_1 and domain D_2 (b) Convex hulls of D_1 and D_2 (c) New domain D after merging

The area of new domain D is the union of areas of D_1, D_2, and connecting area. If D_1 and D_2 are physically connected together before

merging, no connecting area is needed in this case. The connecting area covered by D is determined by the convex hulls of D_1 and D_2. Let the area of convex hulls of D_1 and D_2 be A_1 and A_2 respectively. If A_1 and A_2 are separated without overlapping, the connecting area is confined by two tangent lines and portions of boundaries of D_1 and D_2 (See Figure 9c). If A_1 and A_2 are overlapping together, the connecting area is the union of A_1 and A_2 minus areas of D_1 and D_2 with some adjustments to fit existing vertices. (See Figure 10c). The connecting area is a polygon region. Triangulating in the connecting area can be done more efficiently by taking advantage of the polygon region.

If a surface domain D was cut into a set of new domain $\{D_i\}$, then $\{D_i\}$ were merged back into a single surface domain D_{new} at some time later. Domains D and D_{new} are not the same triangulations any more, because some vertices and sides had been added into $\{D_i\}$ and some sides and triangles were broken while the cutting was performed. Those new elements won't be removed or reconnected after merging.

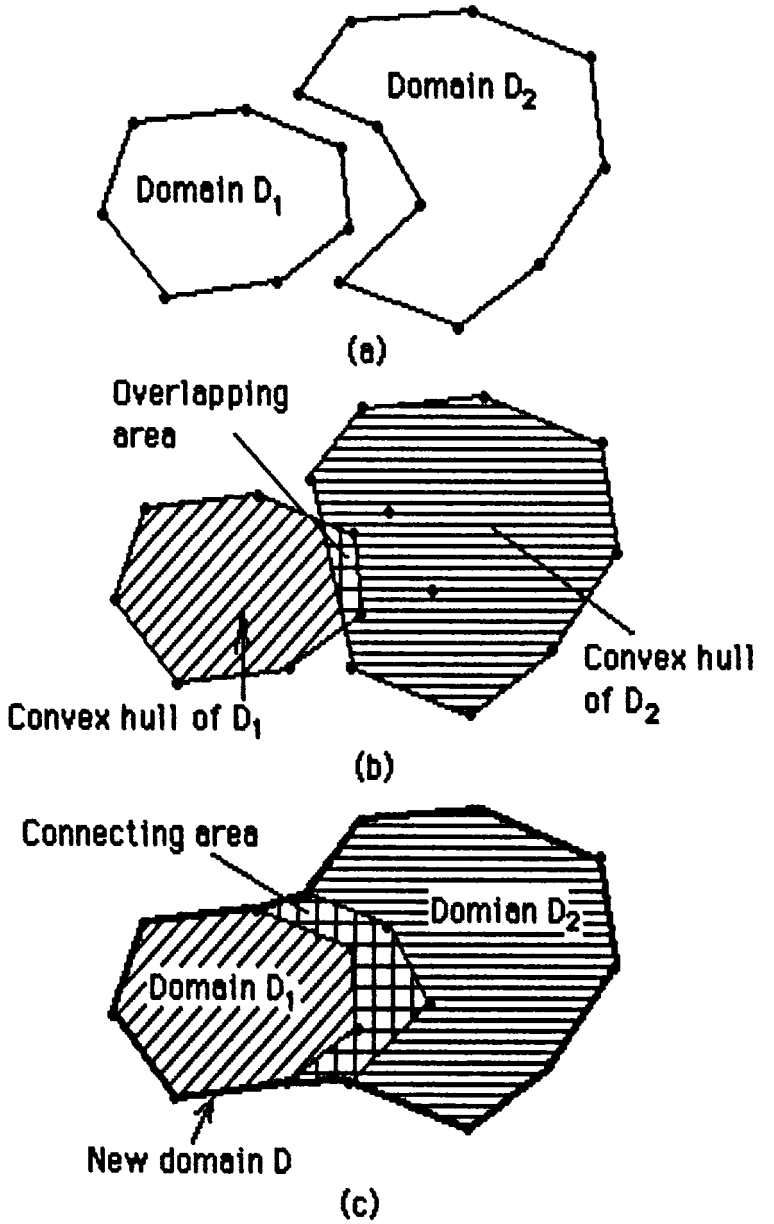


Figure 10: Merging two domains (a) D_1 and D_2 overlapped (b) Convex hulls of D_1 and D_2 (c) New domain D after merging with adjustment at connecting area

2.5.2 Moving a vertex

To observe the deflections of a surface being pulled or pushed at a point V is common and useful in engineering designs and science applications. The problems accompanied with the movement of V are: the area of the surface affected by the movement of V, the relative movement by the individual vertex in affected area and the resulting triangulations after this movement. In mechanics the deflections of a shell structure are functions of forces, material properties, geometrical shapes of the surface and some other conditions. A complete analysis of the surface is expensive and time consuming, and the long and complicated calculations are not affordable. Some assumptions to simplify the problem should be made:

1. The pulling or pushing of the surface can only be given at an existing vertex V. To ensure the smoothness of the surface, any vertex whose radius of vertex vicinity of V is less than or equal to a given radius r will also be pushed or pulled by this movement to some different extents. Radius r is determined by the user.
2. The affected area by the movement of V is bounded by the vertices of vertex vicinity of V with radius (r+1). The boundary of the affected area is called the affecting boundary. No vertex can be moved on or outside the affecting boundary in x-y plane.
3. After the movement, triangulations of the surface domain remain

unchanged. No turnover of any triangle is allowed. Only the shapes of triangle and values of attached functions can be changed.

4. Let new location of V after the movement be V'. The direction from V to V' is called moving direction. All affected vertices will be moved to a new location in moving direction by the movement of V. The distance of each vertex V_i moved may be defined as

$$m_i = M / (r_i + 1)^2 \qquad \text{where}$$

M is the total distance moved by V

r_i is the radius of V_i within the vertex vicinity of V.

m_i is the distance moved by V_i.

The advantages of above assumptions to handle the surface movement are simplicity and efficiency. We can start the moving from the vertices with radius r, r-1, r-2, to radius 0 (V itself). Once a triangle is turned over, the process stops and rejects the movement of V. The disadvantages of these assumptions are:

1. No quick way to determine the movement is acceptable or not.
2. The idea of using radius of vertex vicinity instead of distance in the process may be good and simple but it doesn't reflect the reality. Distance is a better candidate.

2.5.3 COMBINED APPLICATIONS WITH RECTANGULAR GRIDS

No matter what kind of grid used in a single grid system, some disadvantages are inevitable. Rectangular grids used in a rectangulated system can preserve the integrity of the surface by row and column information and rectangle based interpolations are easy. But rectangular grids have a problem fitting in irregular boundary shapes and complicated fault system. Also, the errors generated by converting an arbitrarily scattered data set to values at rectangular grid nodes may be propagated to the final results. Triangulated grids used in a triangulation system are more flexible than rectangular grids fitting in a surface domain with irregular boundary shapes and it will preserve the better results of the surface. But it lacks integrity among triangles and more calculations are needed for triangle based interpolations. A series of rectangles traversed by the path of an operation in a rectangulated surface can be easily constructed through row by row. The same job to be done in a triangulated surface is not easy and needs a lot of effort. To take advantage of integrity of rectangular grids, combined applications of rectangular and triangular grids for a surface can improve the efficiency of surface manipulations while the accuracy of the surface is still preserved.

In combined applications of both rectangular and triangular grids, a two-level hierarchy scheme is proposed. The top level of hierarchy is based on rectangular grids with some triangular grids on the boundary to fit irregular boundary shapes. The second level is based on triangular grids as the children of top level grids to preserve the accuracy of rough areas. The scattered data set of a surface domain is converted into values

at rectangular grid nodes (or vertices) in rows and columns except for the areas along the boundaries (See Figure 11). Along the boundaries, triangular grids are used to fit irregular boundary shapes. The boundary vertices and rectangular grid nodes are vertices of top level. Boundary triangles, and rectangles are top level triangles and rectangles.

If the patch of a rectangular grid folds, the values of top level vertices may fail to describe the folding of the surfaces. Therefore, extra vertices (lower level vertices) will be used to assist top level grids in those areas to describe the folding. If a top level rectangle or a top level triangle is assisted by some lower level vertices to reflect the folding surface, it will be further triangulated by its endpoints and lower level vertices. Newly created triangles after triangulation are lower level triangles. The number of lower level vertices to assist a top level grid depends on the complexity of the folding of that grid.

A top level grid (a rectangle or a triangle) may or may not have children (the attached triangulations) but a lower level grid (a triangle) does have a parent (a top level grid). Vertices are in lines (top level vertices) or inside a grid (lower level vertices). A row map consists of rows from top to bottom. Each row is a sequence of top level grids (triangles and rectangles) from left to right according to their locations. Besides row map, a line map to connect top level vertices is also necessary. With the row map and line map, the relative position of a given point to the surface can be easily decided.

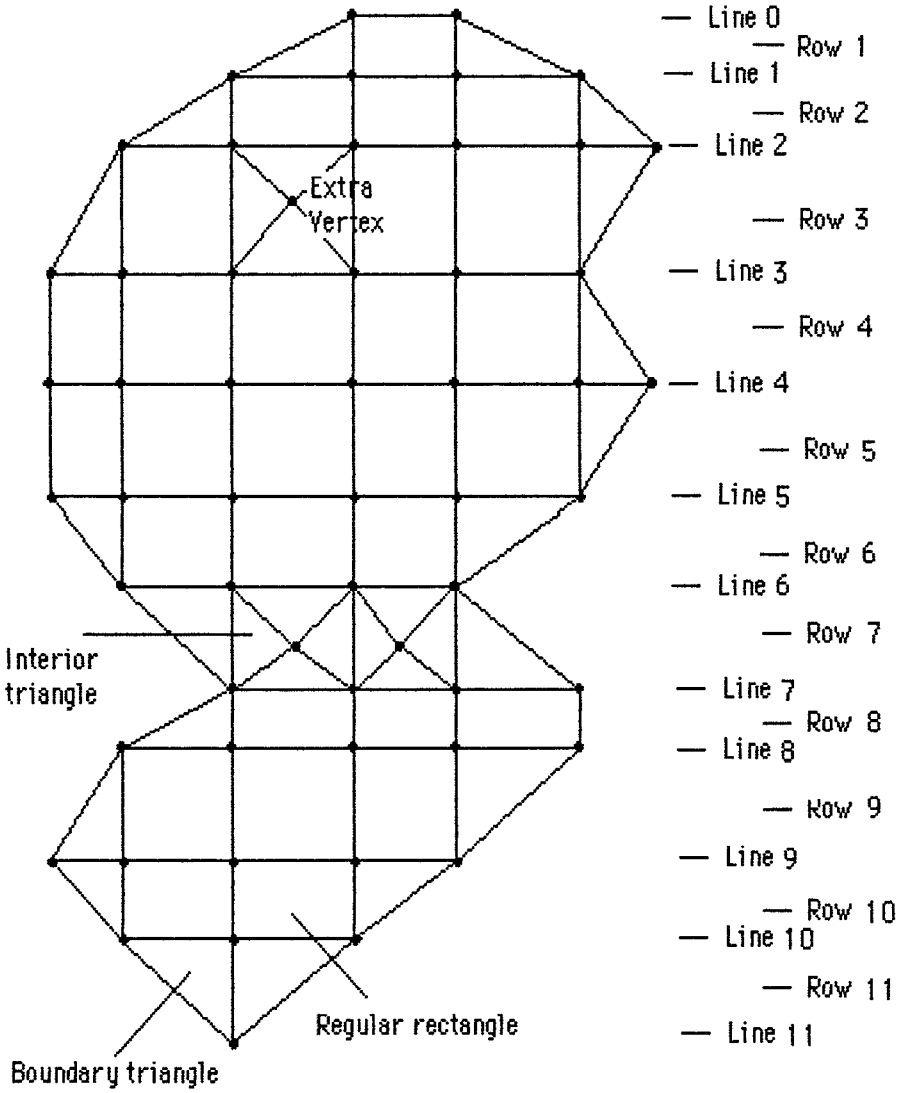


Figure 11: Surface representation by both rectangular grids and triangular grids

A pointer is attached with top level grids to point to the attached lower level triangulations or to nil (if no attached triangulations). Each lower level triangulation is given a pointer to point back to its parent.

Each lower level triangulation can be treated as an independent sub-domain under the main domain. Scanning and searching of this database scheme will start from top level line map or row map, then down to lower level triangulations if necessary.

The data structures of top level grids are discussed below. Vertex block to store x and y coordinate of each top level vertex is needed. Side to top level triangle or top level rectangle block, triangle to top level vertex block and rectangle to top level vertex block are also needed. Boundary lists are still important to the scheme but the attached function values will focus on vertices to avoid the potential inconsistency between rectangle and triangle. In addition to the above structures, a row map of top level grids and a line map of top level vertices are needed to support the integrity of the network. The data structures of low level triangulations remains the same as before with an extra pointer to point its parent.

The new scheme needs more memory space to store the extra data structures. The overhead and updating the new database needs more work with extra caution. Also because of the extra data structures, the new scheme will run more efficiently after the system is initialized and will preserve the accuracy of the surface.

3.0 ALGORITHMS FOR SURFACE MANIPULATING

Algorithms of the new manipulating functions are more complicated than the basic ones because more triangulations are involved in the operations of the new functions. As mentioned before, the triangulation system lacks row and column information. Connections among triangles have no fixed pattern and are not predictable. Searching and scanning are necessary to find out the relative locations among triangulations. To ensure the correctness of the algorithms and implementations, many tests have been made in the past six months. Simplified pseudo-codes of these algorithms and some examples will be given in the following sections. The theoretical analysis of time complexity of functions is quite difficult. Empirical results for time required by these algorithms will be given instead. Some problems that had been confronted during implementations will be discussed in chapter four. A triangulated surface domain D shown in Figure 12 will be used for illustration throughout the entire chapter. Figure 13 shows the database of D.

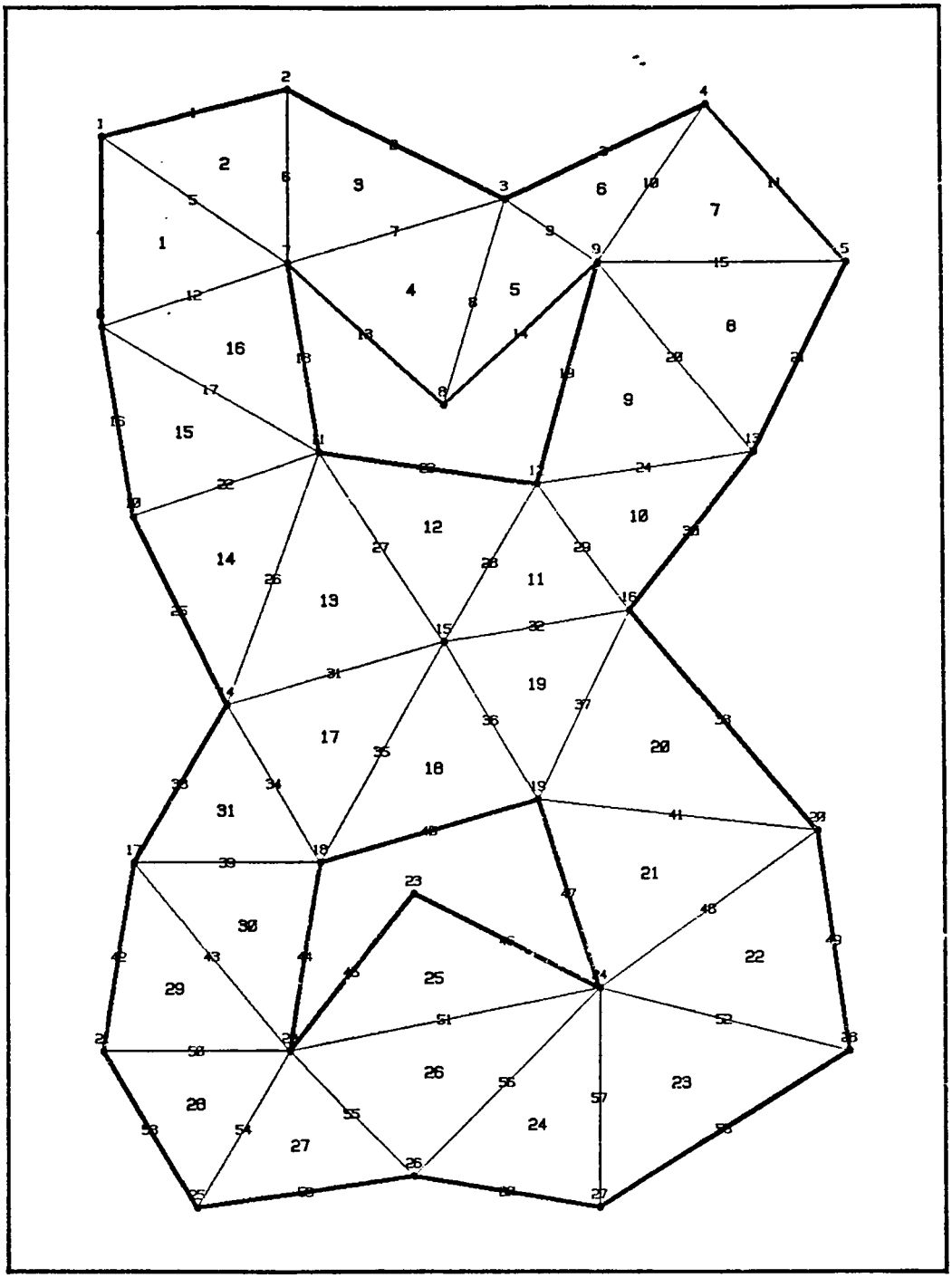


Figure 12: The triangulated surface domain D in x-y plane. Vertex, side and triangle indices are also shown. Black thick sides represent boundary sides.

Vertex Block

V#	x and y	V#	x and y	V#	x and y
1	(3.00, 36.00)	11	(10.00, 26.00)	21	(3.00, 7.00)
2	(9.00, 37.50)	12	(17.00, 25.00)	22	(9.00, 7.00)
3	(16.00, 34.00)	13	(24.00, 26.00)	23	(13.00, 12.00)
4	(22.50, 37.00)	14	(7.00, 18.00)	24	(19.00, 9.00)
5	(27.00, 32.00)	15	(14.00, 20.00)	25	(6.00, 2.00)
6	(3.00, 30.00)	16	(20.00, 21.00)	26	(13.00, 3.00)
7	(9.00, 32.00)	17	(4.00, 13.00)	27	(19.00, 2.00)
8	(14.00, 27.50)	18	(10.00, 13.00)	28	(27.00, 7.00)
9	(19.00, 32.00)	19	(17.00, 15.00)		
10	(4.00, 24.00)	20	(26.00, 14.00)		

Vertex Function Value Block

V#	VF value	V#	VF value	V#	VF value	V#	VF value
1	3.000	8	8.700	15	10.500	22	10.600
2	5.800	9	7.200	16	9.000	23	10.900
3	7.800	10	5.700	17	7.500	24	9.300
4	5.000	11	7.900	18	9.900	25	6.700
5	3.500	12	9.400	19	11.000	26	9.000
6	3.000	13	6.300	20	7.800	27	7.200
7	6.700	14	8.300	21	8.300	28	4.800

Figure 13(a): The data of vertex block and vertex function value block of surface domain D. There are total of 28 vertices and one vertex function.

SVP Block

S#	v_1	v_2	S#	v_1	v_2	S#	v_1	v_2	S#	v_1	v_2	S#	v_1	v_2
1	1	2	13	7	8	25	10	14	37	16	19	49	20	28
2	2	3	14	8	9	26	11	14	38	16	20	50	21	22
3	3	4	15	9	5	27	11	15	39	17	18	51	22	24
4	1	6	16	6	10	28	12	15	40	18	19	52	24	28
5	1	7	17	6	11	29	12	16	41	19	20	53	21	25
6	2	7	18	7	11	30	13	16	42	17	21	54	22	25
7	3	7	19	9	12	31	14	15	43	17	22	55	22	26
8	3	8	20	9	13	32	15	16	44	18	22	56	24	26
9	3	9	21	5	13	33	14	17	45	22	23	57	24	27
10	4	9	22	10	11	34	14	18	46	23	24	58	27	28
11	4	5	23	11	12	35	15	18	47	19	24	59	25	26
12	6	7	24	12	13	36	15	19	48	20	24	60	26	27

First Boundary Vertex

	Boundary 1	Boundary 2	Boundary 3
First Vertex	1	7	18

Boundary Lists

Boundary 1: 1 -> 2 -> 3 -> 4 -> 5 -> 13 -> 16 -> 20 -> 28 -> 27 -> 26 -> 25 -> 21 -> 17 -> 14 -> 10 -> 6 -> 1 (First vertex)

Boundary 2: 7 -> 11 -> 12 -> 9 -> 8 -> 7 (First vertex)

Boundary 3: 18 -> 22 -> 23 -> 24 -> 19 -> 18 (First vertex)

Figure 13(b): The data of SVP block, First Boundary Vertex block and boundary lists. Surface domain D has 60 sides, three boundaries and no side function.

TSP Block

T#	s_1 s_2 s_3	T#	s_1 s_2 s_3	T#	s_1 s_2 s_3	T#	s_1 s_2 s_3
1	(4,12, 5)	9	(19,24,20)	17	(31,34,35)	25	(45,51,46)
2	(1, 5, 6)	10	(24,29,30)	18	(35,40,36)	26	(51,55,56)
3	(2, 6, 7)	11	(28,32,29)	19	(32,36,37)	27	(54,59,55)
4	(7,13, 8)	12	(23,27,28)	20	(37,41,38)	28	(50,53,54)
5	(8,14, 9)	13	(26,31,27)	21	(41,47,48)	29	(42,50,43)
6	(3, 9,10)	14	(22,25,26)	22	(48,52,49)	30	(39,43,44)
7	(10,15,11)	15	(16,22,17)	23	(52,57,58)	31	(33,39,34)
8	(15,20,21)	16	(12,17,18)	24	(56,60,57)		

Triangle Function Value Block

T#	TF value	T#	TF value	T#	TF value	T#	TF value
1	1.000	9	9.000	17	17.000	25	25.000
2	2.000	10	10.000	18	18.000	26	26.000
3	3.000	11	11.000	19	19.000	27	27.000
4	4.000	12	12.000	20	20.000	28	28.000
5	5.000	13	13.000	21	21.000	29	29.000
6	6.000	14	14.000	22	22.000	30	30.000
7	7.000	15	15.000	23	23.000	31	31.000
8	8.000	16	16.000	24	24.000		

Figure 13(c): The data of TSP block and triangle function value block of surface domain D. There are total 31 triangles and one triangle function.

3.1 LINE CONVERSION

The line conversion function will convert a polyline (or a line) in the x-y coordinate system into a triangulated line in the triangulation system. To convert a polyline, the function will only take care of one line segment at a time. The conversion direction for each line segment L_i is from the endpoint with larger x coordinate to the other end. If L_i is vertical, the direction is from the endpoint with larger y coordinate to the other end.

Three major subroutines of line conversion function are: finding the location of a point in triangulations (see subroutine Pt_in_2DTDB), finding intersection points between the exterior boundary and line segment L_i (see subroutine Get_Intersections) and searching for the next crossing point by L_i in triangulations (see subroutine Get_Next_Point). The main flow of the algorithm (see subroutine XYCRV_TRICRV) is guiding the conversion line to the correct direction during entering or exiting the boundary. Due to the irregularity of the boundaries, L_i may enter and exit the domain quite often. Therefore, the sequence of intersection points between L_i and the working boundary is very important. It shows where the line segment enters and exits the surface domain.

Input:

1. ACTIVE: The index of active domain where polyline PL is on.
2. NPT: number of endpoints on polyline PL.
3. XY: x and y coordinates of each endpoint.
4. TFS: true if first line segment of PL extends to infinite.
5. TFE: true if the last line segment of PL extends to infinite.

Output:

The outputs of this function are PL_{tri} which will be stored in the VMAP_NEW block with starting address NV_ORG(active). Variable NV_LEN(active) is used to indicate number of points in PL_{tri}. Each points consists of four 4-byte data. (See section 2.1.1) Before the termination of the process, system will ask you to save PL_{tri} to a file. If answer is "yes", a subroutine called "Write_Profile" will create a ".PRF" file with the same file name and version number as the surface domain to save the triangulated line. The database of the surface domain remains unchanged.

Method: (a brief pseudo code, see subroutine XYCRV_TRICRV)

```
NV_LEN(active) = 0
Allocate space to store PL_tri.
Do i = 1, N          /* N line segments */
   Done = .false.
```

B_{new} = 0 /* new boundary met by line segment P_1P_2 */

Find corresponding locations of P_1 and P_2 in triangulations

/* see subroutine "PT_IN_2DTDB" to locate a point in triangulations */

If (P_1 is inside of boundary) then

B_{work} = 0 /* B_{work} is the current working boundary */

Else If (P_1 is outside of exterior boundary) then

B_{work} = 1

Else If (P_1 is inside of a hole surrounded by interior boundary b) then

B_{work} = b

Endif

If (B_{work} .ne. 0) then

Find all intersection points between L_i and boundary B_{work}

/* See subroutine Get_Intersections */

P_1 = intersection point with largest x value

P_c = P_1 (the current intersection point)

P_n = the next intersection point of P_1 along the direction of B_{work}

P_p = the previous intersection point of P_1 along the direction of B_{work}

Endif

Calculate local coordinates of P_1 and save data of P_1

Increase NV_LEN(active) by 1

Do while (.not. Done)

If (P_1 and P_2 are the same coordinates) then

Done = .true.

Else if (B_{new} .gt. 0) and (B_{new} .ne. B_{work}) then

$B_{work} = B_{new}$

$B_{new} = 0$

Find all intersection points between P_1P_2 and B_{work}

/* See subroutine Get_Intersections */

If no intersection point found then

Done = .true.

Else

P_1 = intersection point with largest x value

$P_c = P_1$ (the current intersection point)

P_n = the next intersection point of P_1 along B_{work}

P_p = the previous intersection point of P_1 along B_{work}

Calculate local coordinates of P_1 and save data of P_1

Increase NV_LEN(active) by 1

Endif

Else if (B_{new} .gt. 0) and (B_{new} .eq. B_{work}) then

If (P_n .eq. P_p) then

Done = .true.

Else If (P_1 .eq. P_n) then

P_1 = the next intersection point of P_n in B_{work}

$P_c = P_1$

P_n = the next intersection point of P_1 in B_{work}

Else

P_1 = the previous intersection point of P_p in B_{work}

$P_c = P_1$

P_p = the previous intersection point of P_1 in B_{work}

Endif

Calculate local coordinates of P_1 and save data of P_1

Increase NV_LEN(active) by 1

Endif

If (.not. Done) then

Find the next crossing point P_{next} of P_1P_2

/* See subroutine Get_Next_Point */

Set $P_1 = P_{next}$ just found

Calculate local coordinates of P_1 and save the data of P_1

Increase NV_LEN(active) by 1

If (P_1 and P_2 are the same location) then

Done = .true.

Else

If (P_1 is on the boundary) then

B_{new} = Boundary where P_1 is on

Else

B_{new} = 0

Endif

Endif

Endif

Enddo

Enddo

3.1.1 Example (Function of line conversion)

Given a triangulated surface domain D (see Figure 12) and a polyline PL (see Figure 14) find the triangulated line PL_{tri}. PL has five endpoints and both end line segments are finite. Coordinates of the five points are (15,39), (12,35), (25,22), (17,12) and (17,1). The line conversion will start from the first line segment L_1, P_1(15,39) to P_2(12,35).

P_1 is outside the exterior boundary and P_2 is inside triangle #3. Working boundary B_{work} is pointing to the exterior boundary. There is only one intersection point between L_1 and B_{work} at side #2. Set P_1 to the intersection point where L_1 enters the surface domain, then express P_1 in

terms of side #2, $S_2(u_1)=P_1$, as the first point of triangulated line. Along L_1, P_1 leaves side #2 and enters triangle #3. To show L_1 enters triangle #3 at point P_1, P_1 is also expressed in the terms of triangle #3, $T_3(u_1,u_2)=P_1$, as the next point in triangulated line. Finally, L_1 will end in triangle #3 at point P_2. Express P_2 in terms of triangle #3, then go back for the next line segment.

The next line segment L_2 is from $P_1(25,22)$ to $P_2(12,35)$ because P_1 has a larger x coordinate. P_1 is outside the exterior boundary and P_2 is inside triangle #3. Let B_{work} point to the exterior boundary. Again, we only have one intersection point between line segment L_2 and B_{work} at side #30. Set P_1 to the intersection point. Along line segment L_2, P_1 enters and leaves triangle #18 and crosses side #24. Record the track of P_1 and set P_1 to the intersection point on side #24. After passing through the triangle #9, an interior boundary pointed by B_{new} is met at side #19. Now, set B_{work} to point the same boundary as the boundary pointed by B_{new}. One intersection point (excluding P_1 itself) at side #14 has been found between line L and B_{work}. Set P_1 to this new point and keep on going until it meets P_2 in triangle #3. Reverse the the path of L_2 to fit the original direction of the polyline.

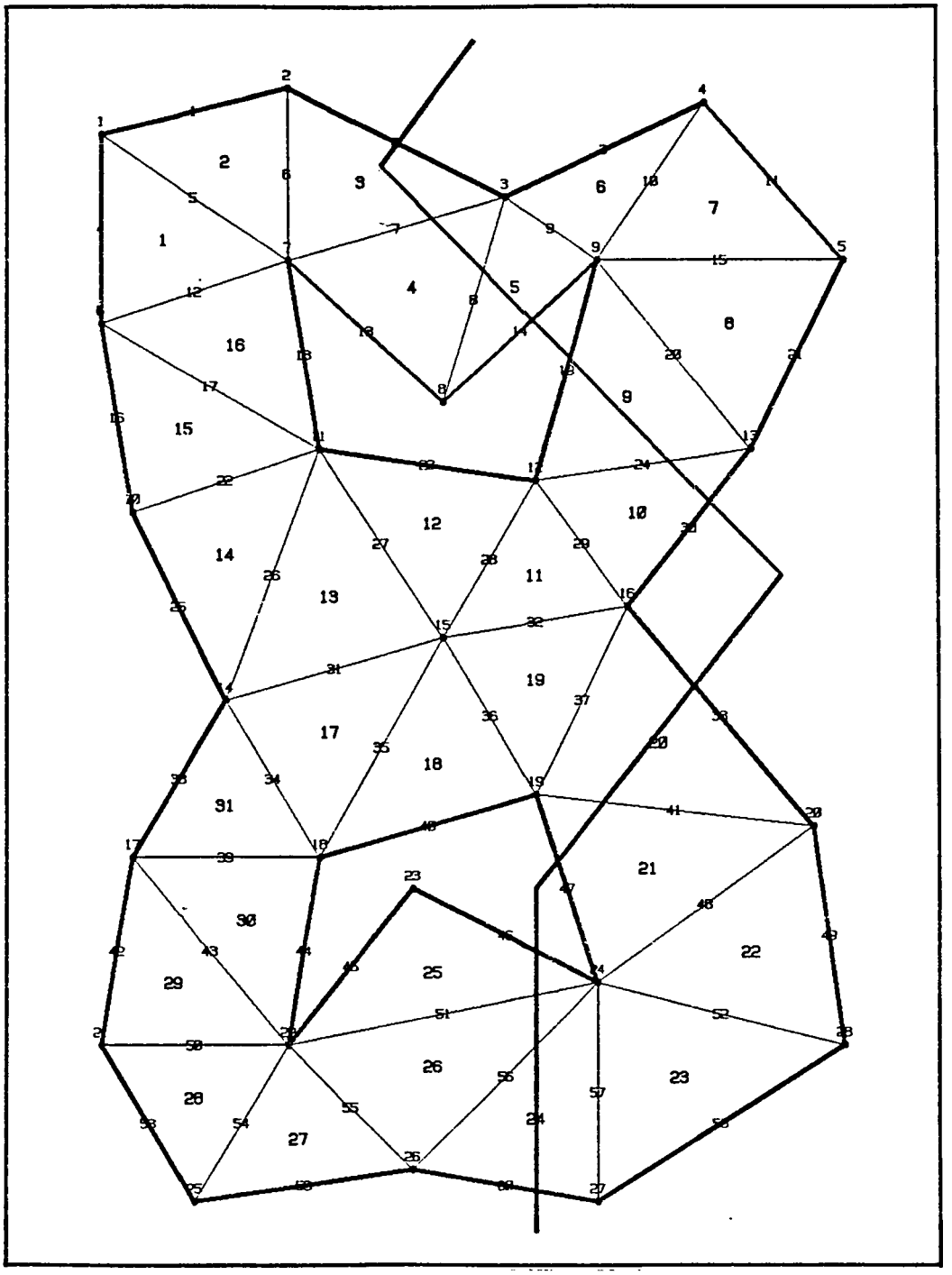


Figure 14: A polyline starts from top down to the bottom was shown on surface domain D.

Result of triangulated line PL_{tri}

Point list	Type	Index	U_1	U_2
1	1	2	0.506493	0.000000
2	2	3	0.493506	0.000000
3	2	3	0.389610	0.181818
4	2	3	0.000000	0.333334
5	1	7	0.333333	0.000000
6	2	4	0.333333	0.000000
7	2	4	0.000000	0.352941
8	1	8	0.352941	0.000000
9	2	5	0.352941	0.000000
10	2	5	0.421052	0.578948
11	1	14	0.578947	0.000000
12	1	19	0.444444	0.000000
13	2	9	0.444444	0.000000
14	2	9	0.375000	0.625000
15	1	24	0.625000	0.000000
16	2	10	0.375000	0.000000
17	2	10	0.000000	0.333333
18	1	30	0.333333	0.000000
19	1	38	0.362069	0.000000
20	2	20	0.000000	0.362069
21	2	20	0.755102	0.244898
22	1	41	0.244898	0.000000
23	2	21	0.755102	0.000000
24	2	21	0.647059	0.352941
25	1	47	0.352941	0.000000
26	1	46	0.666667	0.000000
27	2	25	0.000000	0.666667
28	2	25	0.200000	0.800000
29	1	51	0.800000	0.000000
30	2	26	0.200000	0.000000
31	2	26	0.000000	0.333333
32	1	56	0.333333	0.000000
33	2	24	0.333333	0.000000
34	2	24	0.333333	0.666667
35	1	60	0.666667	0.000000

Figure 15: Result of a polyline PL being converted in D. The original polyline have 5 points, (15,39), (12,35), (25,22), (17,12) and (17,1), and both ends are finite line segment. There are total of 35 points in PL_{tri}.

The same procedure is applied to the third and fourth line segments. After that, the process of line conversion is then completed. Figure 15 shows the complete listing of triangulated line PL_{tri}. If we convert the triangulated line PL_{tri} back to the line in the x-y coordinate system PL_{xy}, we found that there were nine points in the new polyline PL_{xy} which is not the original polyline PL. The reason for this difference is that PL_{tri} only shows portions of the polyline PL which are inside surface domain D and clips out the rest. These nine points of PL_{xy} tell the discontinuities of line PL in surface domain D.

3.1.2 RELATED SUBROUTINES

The related subroutines in implementation are:

1. Subroutine "XYXRV_TRICRV": This is the main routine of the function to find the path of the given route.
2. Subroutine "Get_Profile": This routine will find the path of a single line segment.
3. Subroutine "PT_IN_2DTDB": This routine is used to find the location of a given point P in a triangulated surface. Two variables, Position1 and Position2, are returned to indicate the location of P.

Position1	Description
0	P is outside exterior boundary

1	P is inside a hole
2	P is on a vertex
3	P is on a boundary side
4	P is on a non-boundary side
5	P is inside a triangle

Position1 =>	Position2
0	index of the exterior boundary (it is 1)
1	index of a interior boundary
2	index of a vertex
3, 4	index of a side
5	index of a triangle

4. Subroutine "GET_INTERSECTIONS": This routine will find all intersection points between a given line segment and a given boundary.
5. Subroutine "GET_NEXT_POINT": this routine will find the next crossing point between a given line segment and triangulations from the current point.
6. Subroutine "XYPT_SIDEPT": this routine will convert a point P(x,y), locating on a given side S, to the side coordinates $S(u_1)$= P.
7. Subroutine "XYPT_TRIPT": this routine will convert a point P(x,y), locating inside of a given triangle T, to the triangle coordinates $T(u_1, u_2)$= P.

Some other basic routines are also called by above subroutines. They are either in IPL Library or in Appendix B.

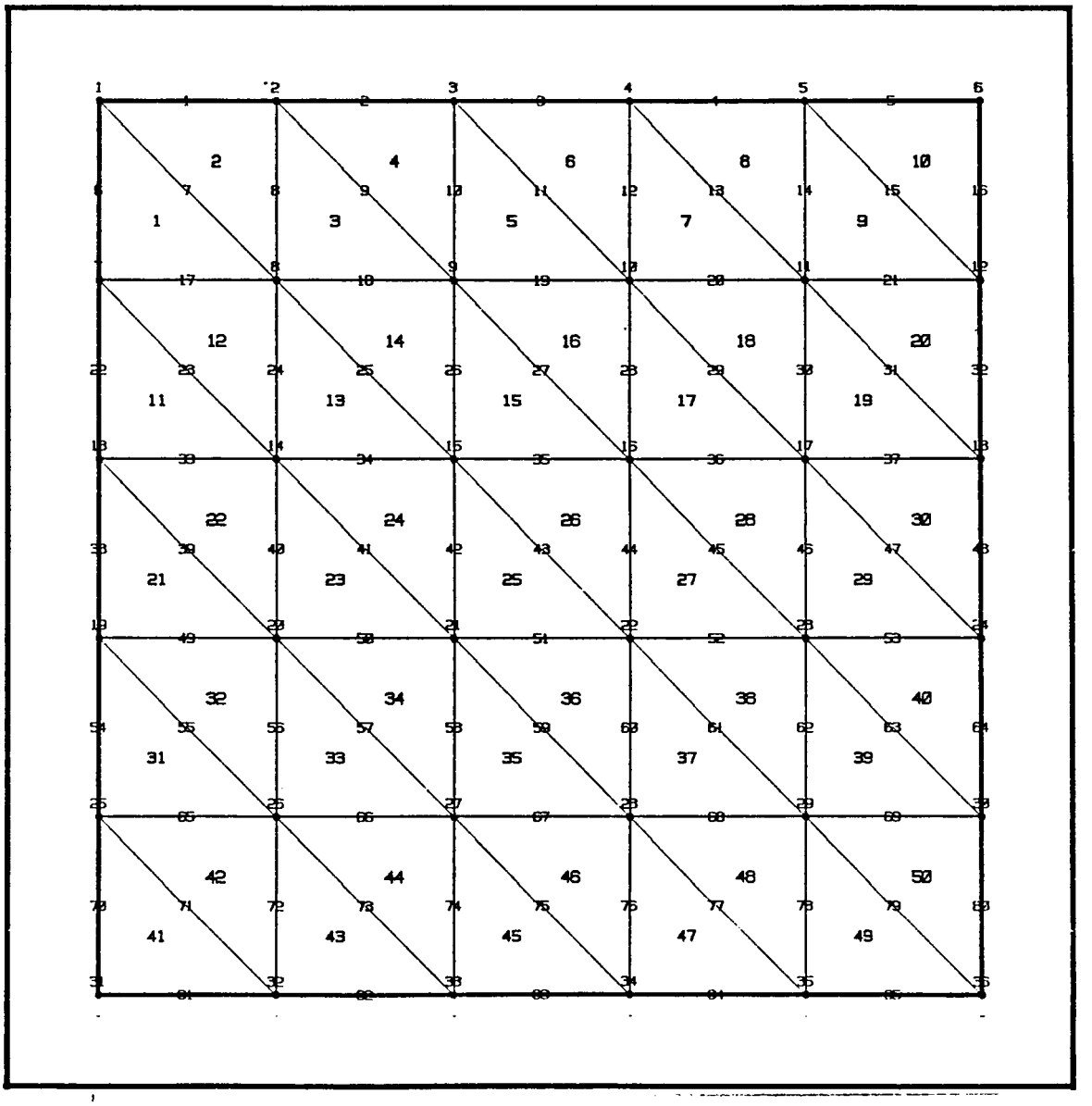


Figure 16: The surface domain used in experiment of time requirements analysis

3.1.3 EMPIRICAL ANALYSIS OF TIME COMLEXITY

The time complexity of this algorithm depends on many factors such as the size of the database, number of sides passed by the given line. In order to estimate the required time for the algorithm, an empirical analysis was made. The surface domain used for experiment is a square surface with 36 vertices, 85 sides and 50 triangles (see Figure 16). A single line segment, from P_1 to P_2, is going to be converted by the subroutine "XYCRV_TRICRV". P_1 is fixed at a point outside the boundary and P_2 is moveable along the same line many times. The number of sides crossed by line segment P_1P_2 each time will be increased by one until P_2 is out of the boundary. Required time is dominated by two major parts: time to locate points P_1 and P_2 in the triangulated surface and time to trace the path of line segment P_1P_2. Time to locate the points P_1 and P_2 in the triangulated surface is propotional to the number of sides in the surface domain. Time to trace the path is propotional to the number of sides crossed by the line segment. Let T be the time required, N_s be the number of sides in the surface domain, and S_{cross} be the number of sides crossed by the line segment. T is assumed of the form

$$T = t_1 + t_2$$

$t_1 = aN_s$ (time to locate point)

$t_2 = bS_{cross}$ (time to convert line)

Do the same experiment a few times with different location of P_1. The resulting time requirements are listed below:

S_{cross}	t_1 (10^{-2}sec.)	t_2 (10^{-2}sec.)
1	3	1
2	4	1
3	4	2
4	4	3
5	5	3
6	4	3
7	5	4
8	5	4
9	4	5
10	4	5
11	6	6

The average time to locate points P_1 and P_2 in the triangulated surface is about 0.045 second. The total number of sides in the domain is 85. Coefficient a is equal to 0.053 ($a = t_1/N_s$). The average time for line conversion for each crossed side is 0.005 second. Coefficient b is equal to

0.5. The empirical time requirements T is

$$T\ (10^{-2}\ \text{sec}) = 0.053N_s + 0.5S_{cross}$$

3.2 TILE FUNCTION

Tile function will calculate values of a particular vertex function along a given polyline (or a line) in a surface domain (see subroutine "TileF_Profile"). This function has two major steps. The first step is line conversion and the second step is using the triangulated line, output from the first step, to calculate the function values along the line. The number of points after the process of tile function will be less than the number of points in PL_{tri} because three points are used to describe the intersection in PL_{tri}: leaving a triangle, crossing a side, and entering a new triangle. While in tile function, only one point is used. The algorithm of the second step is giving below.

Input:

The index of vertex function IVF to be projected should be known by the tile function and the triangulated line PL_{tri} is already stored in VMAP_NEW block with starting address NV_ORG(active_domain). The number of points in PL_{tri} is indicated by variable NV_LEN(active_domain).

Where active_domain, a global variable known by the system, indicates the surface domain to work with.

Output:

The output of this function will be stored in the VMAP_OLD block with the starting address OV_ORG(active_domain). Number of points after this operation will be indicated by OV_LEN(active_domain). Each point has four 4-byte data. The first two are x and y coordinates of the point. The third one is the desired vertex function value. The fourth one is the distance between this point and the previous point along the line. If a point is a boundary entry point, the distance is equal to zero.

Method: (See subroutine TILEF_PROFILE)

```
OV_LEN(active_domain) = 0
addr2 = V_FVAL_ORG(IVF, active_domain)    /* Address of VF block */
i = 1
enter = .true.  /* indicate PL_tri just entered surface domain */
Do While (i .le. NV_LEN(active_domain))
   addr = OV_ORG(active_domain) + 4 * OV_ORG(active_domain)
   /* Output address */
   addr1 = NV_ORG(active_domain) + 4 * (i-1)  /* address  of PL_tri */
```

Increase OV_ORG(active_domain) by 1

Get the current point from address "addr1"

If (type of current .eq. 0) then

Get x and y coordinates of current point

Get the function value of the current point

Else if (type .eq. 1) then

Get information on side s where the current point is located

Convert x and y of the current point by u_1 and side s.

Get the function value of the current point by linear interpolation

Else if (type .eq. 2) then

Get information on triangle t where the current point is located

Convert x and y of the current point by u_1 and u_2 and triangle t

Get the function value of the current point by linear interpolation

Endif

If (enter) then

enter = .false.

distance of current = 0.0

Else

Calculate the distance between the current point and previous point

Endif

Find the next point in PL_{tri} with different x and y coordinates

Point i to the next point.

Enddo

Profile of Vertex Function 1 Along Triangulated line PL_{tri}

Point List	x	y	VF value	Distance
1	12.54545	35.72727	6.812987	0.000000
2	12.00000	35.00000	6.820779	0.909090
3	13.66667	33.33333	7.433333	2.357024
4	15.29412	31.70588	8.117647	2.301562
5	16.89474	30.10526	7.831579	2.263618
6	18.11111	28.88889	8.177777	0.000000
7	21.37500	25.62500	7.462500	4.615837
8	22.66667	24.33333	7.200000	1.826692
9	22.17241	18.46552	8.565517	0.000000
10	19.20408	14.75510	10.216327	4.751650
11	17.70588	12.88236	10.400001	2.398288
12	17.00000	10.00000	9.833333	0.000000
13	17.00000	8.60000	9.560000	1.400000
14	17.00000	7.00000	9.200000	1.600000
15	17.00000	2.33333	7.800000	4.666667

Figure 17: The profile of vertex function 1 along the triangulated line PL_{tri}. PL entered and exited the surface D four times. The underlined data indicates the entry points.

3.2.1 **Example** (Tile Function)

The data of profile along the triangulated line PL_{tri} (in Figure 15) is shown in Figure 17. The PL_{tri} has entered and exited the boundary of surface domain D four times. The distance of point i is the distance between points (i-1) and point i. The distance of each boundary entry point is equal to zero. So the first section of the profile is from point 1 to point 5 (see Figure 16), the second section of the profile is from point 6 to point 8, the third section of the profile is from point 9 to point 11, and the fourth section of profile is from point 12 to point 15.

3.2.2 RELATED SUBROUTINES

The related subroutines in implementation includes:

1. Subroutine "TILEF_PROFILE": This routine will use the results of subroutine "XYCRV_TRICRV" to find the tile function values along the given route.
2. Subroutine "WRITE_TILE": this routine will save the result of tile function values to a file. The file type will be ".TLF".
3. Subroutine "SIDEPT_XYPT": this routine will convert a point P, locating on a given side S with coordinates u_1, back to the x and y coordinates.
4. Subroutine "TRIPT_XYPT": this routine will convert a point P,

locating on a given triangle T with coordinates u_1 and u_2, back to the x and y coordinates.

Note that, before the subroutine "TIFEF_PROFILE" is called, subroutine "XYCRV_TRICRV" has to be called first to generate the path of the given route.

3.3 PARTITION A SURFACE DOMAIN (CUTTING)

This function will partition a surface domain into a set of new domain(s). The process of cutting is quite expensive in searching points, inserting new points, updating the database, and splitting the database of the surface domain. After database of the surface domain split, indices of vertex, side and triangle are all renumbered. Vertex mapping tables should be created to keep track of the relationships between the new domains and the original domain.

There are two approaches to perform this function. The first approach divides the process in two steps: finding the path of cutting route then cutting the surface along the path. The second approach is cutting the surface immediately once a cut point is found. The advantages and disadvantages of two approaches are compared below:

1. In the first approach, the path of cutting route can be obtained while cutting. But if the path of cutting route is not wanted, the extra cost for this unnecessary information is a waste.

2. Path of cutting route is based on the original database. If no surface splitting occurs, the first approach is good and clear. Once the surface is split, indices of vertex, side and triangle are all renumbered in the new domains. Extra cost is needed to map the points on the path to the new domains in the first approach.
3. The scanning and searching of the second approach is always based on the current working domain. After the surface is split, the new working domain is smaller than the original one. Scanning and searching for the next cutting point in the new working domain for the second approach will be faster than the first approach.

The implementation of this function in IPL system is based on the second approach because of its efficiency. The cutting procedure can be imagined as zipping down a zipper. All vertices, sides and triangles along the cut line are split into two, until the cutting is finished. The algorithm of second approach (see subroutine Lines_Cut) will be showed below. Some major components such as inserting a vertex (see subroutine Insert_Vertex), the zipping procedure (see subroutine Zip) and splitting of the surface domain (see subroutine Create_New_Domain), will be discussed in the next chapter.

Input:

1. ORGD: Index of surface domain to be cut.
2. NPT: number of points in the given cutting line. The cutting line is supposed to be given as a polyline (PL).

3. XY: x and y coordinates of each point of the polyline.
4. TFSINF: true if first line segment of PL which extends to infinite
5. TFEINF: true if last line segment of PL which extends to infinite
6. SWAP: Perform side-swapping when the value is true.
7. FILL: true if all function values of new vertices, new sides and new triangles are filled by default methods. The default method for vertex function values is linear interpolation; for side function values and triangle function values, default methods are copying.

Output:

1. N_D: number of new domain(s) created after cutting.
2. LIST: a list of new domain indices.

All new domains have already been assigned a domain index and a domain name in memory. The default domain name of each new domain is the same as the name of original domain except the version number. If the version number of the original domain is n and there are m new domains, the version number of new domains will start from n+1 to n+m. If no version number shows in the name of original domain, version numbers of new domains will start from 101 to 100+m.

Method: (subroutine Lines_Cut)

Increase number of domain in memory, N_DB, by one

Duplicate domain, ORGD, in memory

N_D = 1

LIST(N_D) = N_DB

active_domain = LIST(N_D)

AD = active_domain

Do i = 1, N /* N line segments. L_i means current line segment */

Done = .false.

B_{new} = 0 /* new boundary met by line segment P_1P_2 */

Find corresponding locations of P_1 and P_2 in triangulations

/* See subroutine Pt_In_2DTDB */

If (P_1 is inside of boundary) then

B_{work} = 0 /* B_{work} is the current working boundary */

Else If (P_1 is outside of exterior boundary) then

B_{work} = 1 /* index of exterior boundary is one */

Else If (P_1 is inside of a hole surrounded by interior boundary b) then

B_{work} = b

Endif

If (B_{work} .ne. 0) then

Find and **insert** intersection points between L_i and boundary B_{work}

/* See subroutine Find_Intersections */

v_{next} = intersection point with largest x value

$P_c = v_{next}$ (the current intersection point)

P_n = the next intersection point of P_1 on the boundary B_{work}

P_p = the previous intersection point of P_1 on the boundary B_{work}

Else

Insert point P_1 into database

Find the next cutting points v_{next} of P_1, and **insert** it to database

/* See subroutines "Find_Next_Cut_Point" and "Insert_Vertex" */

If (v_2 is not on boundary) then

Form a new interior boundary b by P1 and v_{next}

$B_{work} = b$

Else if (v_2 is on boundary b_{next}) then

Zip the side $s(v_{next}, P_1)$ /* see subroutine Zip */

$B_{new} = b_{next}$

Endif

Endif

cutit = .false. /* split domain when it is true */

Do while (.not. Done)

If (v_{next} and P_2 are the same coordinates) then

Done = .true.

Create the new domain and increase N_DB and N_D by 1

/* See subroutine "Create_New_Domain" */

LIST(N_D) = N_DB

Else if ((B_{new} .gt. 0) and (.not. cutit) then

$B_{work} = B_{new}$

$P_1 = v_{next}$

Find and **insert** intersection points between P_1P_2 and B_{work}

/* See subroutines "Find_Intersections" */

If no intersection point found then

Done = .true.

Else

$B_{new} = 0$

v_{next} = intersection point with largest x value

$P_c = v_{next}$ (the current intersection point)

P_n = the next intersection point of P_1 along B_{work}

P_p = the previous intersection point of P_1 along B_{work}

Endif

$P_1 = v_{next}$

Else if (B_{new} .gt. 0) and (cutit) then

Create the new domain and increase N_DB and N_D by 1

/* See subroutines "Create_New_Domain" */

LIST(N_D) = N_DB

If (P_n .eq. P_p) then

Done = .true.

Else If (P_1 .eq. P_n) then

v_{next} = the next intersection point of P_n in B_{work}

$P_c = v_{next}$

P_n = the next intersection point of P_1 in B_{work}

Else

v_{next} = the previous intersection point of P_p in B_{work}

$P_c = v_{next}$

P_p = the previous intersection point of P_1 in B_{work}

Endif

If (.not. Done) then

$P_1 = v_{next}$

$B_{new} = 0$

Endif

Endif

If (.not. Done) then

Find and **insert** the next cutting point v_{next} of line segment P_1P_2

/* See subroutines "Find_Next_Cut_Point" and "Insert_Vertex" */

Zip the surface domain from P_1 to v_{next}

/* See subroutines "Zip" */

If (v_{next} is on boundary b_{next}) then

$B_{new} = b_{next}$

Endif

Endif

Enddo

Enddo

3.3.1 EXAMPLE OF PARTITIONING

Figure 21 shows the results of surface domain D which is cut by the same polyline in example 3.1.1. There are three new domains created after partitioning. Figure 18 shows the database of new domain D_1; Figure 19 shows the database of new domain D_2, and Figure 20 shows the database of new domain D_3. All function values were filled in by the default methods. In Figure 21, domains D_1 and D_3 were translated to the right for a better view on the new boundaries.

No side-swapping is performed in this example. The reason for not side-swapping is because function values were defined according to their geometric shapes or local properties. If geometric shapes have been changed, those function values may be not valid any longer. For the same reason, the system cannot perceive the physical meaning of those functions so that filling function values of new elements is an option left for the user to decide.

Vertex Block (13 vertices)

V#	x and y	V#	x and y	V#	x and y
1	(16.000, 34.000)	6	(12.000, 35.000)	11	(16.895, 30.105)
2	(22.500, 37.000)	7	(12.545, 35.727)	12	(15.294, 31.706)
3	(27.000, 32.000)	8	(22.667, 24.333)	13	(13.667, 33.333)
4	(19.000, 32.000)	9	(21.375, 25.625)		
5	(24.000, 26.000)	10	(18.111, 13.667)		

Vertex Function Value Block

V#	VF value	V#	VF value	V#	VF value	V#	VF value
1	7.800	5	6.300	9	7.463	13	7.433
2	5.000	6	6.821	10	8.178		
3	3.500	7	6.813	11	7.831		
4	7.200	8	7.200	12	8.118		

SVP Block (23 sides)

S#	v_1 v_2	S#	v_1 v_2	S#	v_1 v_2	S#	v_1 v_2	S#	v_1 v_2
1	(1, 2)	6	(2, 3)	11	(7, 1)	16	(8, 9)	21	(11,12)
2	(1,13)	7	(4, 3)	12	(7, 6)	17	(4,10)	22	(6,13)
3	(1,12)	8	(4, 5)	13	(5, 8)	18	(9,10)	23	(12,13)
4	(1, 4)	9	(3, 5)	14	(5, 9)	19	(4,11)		
5	(2, 4)	10	(1, 6)	15	(4,10)	20	(1,11)		

Figure 18(a): Data of vertex block, vertex function value block and SVP block of new surface domain D_1. Total of 13 vertices and 23 sides and one vertex function. No side function exists.

TSP Block (11 triangles)

T#	S_1 S_2 S_3	T#	S_1 S_2 S_3	T#	S_1 S_2 S_3
1	(3, 2,23)	5	(7, 8, 9)	9	(13,16,14)
2	(20,21, 3)	6	(10,22, 2)	10	(15,17,18)
3	(1, 4, 5)	7	(12,10,11)	11	(4,19,20)
4	(5, 7, 6)	8	(8,14,15)		

Triangle Function Value Block

T#	TF value	T#	TF value	T#	TF value
1	4.000	5	8.000	9	10.000
2	5.000	6	3.000	10	9.000
3	6.000	7	3.000	11	5.000
4	7.000	8	9.000		

Boundary List

Boundary 1: 13 -> 6 -> 7 -> 1 -> 2 -> 3 -> 5 -> 8 -> 9 -> 10 -> 4 -> 11 -> 12 -> 13 (first boundary vertex)

Figure 18(b): Data of TSP block and triangle function value block and boundary list of new domain D_1. There are total of 11 triangles and one triangle function. Only one boundary exists in D_1.

VMAP_OLD Block

Vertex Index	Original Vertex Index
1	3
2	4
3	5
4	9
5	13

VMAP_NEW Block

Vertex Index	Type	Index	U_1	U_2
6	2	3	0.3896104	0.1818182
7	1	2	0.5064934	0.0000000
8	1	30	0.3333333	0.0000000
9	1	24	0.6250000	0.0000000
10	1	19	0.4444449	0.0000000
11	1	14	0.5789474	0.0000000
12	1	8	0.3529411	0.0000000
13	1	7	0.3333332	0.0000000

Figure 18(c): The mapping tables of new domain D_1 . VMAP_OLD block stores all old existing vertices. VMAP_NEW block stores all new created vertices after cutting.

Vertex Block (34 vertices)

V#	x and y	V#	x and y	V#	x and y
1	(3.000, 36.000)	13	(10.000, 13.000)	25	(15.294, 31.706)
2	(9.000, 37.500)	14	(17.000, 15.000)	26	(13.667, 33.333)
3	(3.000, 30.000)	15	(3.000, 7.000)	27	(12.000, 35.000)
4	(9.000, 32.000)	16	(9.000, 7.000)	28	(22.172, 18.466)
5	(14.000, 27.500)	17	(13.000, 12.000)	29	(19.204, 14.755)
6	(4.000, 24.000)	18	(6.000, 2.000)	30	(17.706, 12.882)
7	(10.000, 26.000)	19	(13.000, 3.000)	31	(17.000, 10.000)
8	(17.000, 25.000)	20	(12.545, 35.727)	32	(17.000, 2.333)
9	(7.000, 18.000)	21	(22.667, 24.333)	33	(17.000, 8.600)
10	(14.000, 20.000)	22	(21.375, 25.625)	34	(17.000, 7.000)
11	(20.000, 21.000)	23	(18.111, 28.889)		
12	(4.000, 13.000)	24	(16.895, 30.105)		

Vertex Function Value Block

V#	VF value	V#	VF value	V#	VF value	V#	VF value
1	3.000	10	10.500	19	9.000	28	8.566
2	5.800	11	9.000	20	6.813	29	10.216
3	3.000	12	7.500	21	7.200	30	10.400
4	6.700	13	9.900	22	7.463	31	9.833
5	8.700	14	11.000	23	8.178	32	7.800
6	5.700	15	8.300	24	7.832	33	9.560
7	7.900	16	10.600	25	8.118	34	9.200
8	9.400	17	10.900	26	7.433		
9	8.300	18	6.700	27	6.821		

Figure 19(a): Data of vertex block and vertex function value block of new surface domain D_2. There are total of 34 vertices and one vertex function.

SVP Block (66 sides)

S#	v_1 v_2	S#	v_1 v_2	S#	v_1 v_2	S#	v_1 v_2	S#	v_1 v_2
1	(1, 2)	15	(22, 8)	29	(12,13)	43	(27, 2)	57	(28,29)
2	(2,20)	16	(6, 9)	30	(13,14)	44	(27, 4)	58	(14,30)
3	(1, 3)	17	(7, 9)	31	(14,29)	45	(27,20)	59	(29,30)
4	(1, 4)	18	(7,10)	32	(12,15)	46	(21, 8)	60	(17,31)
5	(2, 4)	19	(8,10)	33	(12,16)	47	(22,21)	61	(16,31)
6	(3, 4)	20	(8,11)	34	(13,16)	48	(22,23)	62	(31,33)
7	(4, 5)	21	(21,11)	35	(16,17)	49	(25, 5)	63	(19,33)
8	(24, 5)	22	(9,10)	36	(15,16)	50	(25, 4)	64	(19,34)
9	(3, 6)	23	(10,11)	37	(16,33)	51	(25,24)	65	(32,34)
10	(3, 7)	24	(9,12)	38	(15,18)	52	(26, 4)	66	(33,34)
11	(4, 7)	25	(9,13)	39	(16,18)	53	(26,25)		
12	(23, 8)	26	(10,13)	40	(16,19)	54	(26,27)		
13	(6, 7)	27	(10,14)	41	(18,19)	55	(11,28)		
14	(7, 8)	28	(11,14)	42	(19,32)	56	(14,28)		

Boundary List

Boundary 1: 34 -> 32 -> 19 -> 18 -> 15 -> 12 -> 9 -> 6 -> 3 -> 1 -> 2 -> 20 -> 27 -> 26 -> 25 -> 24 -> 5 -> 4 -> 7 -> 8 -> 23 -> 22 -> 21 -> 11 -> 28 -> 29 -> 30 -> 14 -> 13 -> 16 -> 17 -> 31 -> 33 -> 34 (first boundary vertex)

Figure 19(b): Data of SVP block and boundary list of new domain D_2. There are total of 66 sides and no side function. Only one boundary exists in D_2.

TSP Block (33 triangles)

T#	S_1 S_2 S_3	T#	S_1 S_2 S_3	T#	S_1 S_2 S_3
1	(3, 6, 4)	12	(22,25,26)	23	(5,43,44)
2	(1, 4, 5)	13	(26,30,27)	24	(47,15,46)
3	(43, 2,45)	14	(23,27,28)	25	(51,49, 8)
4	(48,12,15)	15	(56,57,31)	26	(7,50,49)
5	(20,21,46)	16	(61,62,37)	27	(54,44,52)
6	(19,23,20)	17	(40,37,63)	28	(53,52,50)
7	(14,18,19)	18	(39,41,40)	29	(28,55,56)
8	(17,22,18)	19	(36,38,39)	30	(31,58,59)
9	(13,16,17)	20	(32,36,33)	31	(35,60,61)
10	(9,13,10)	21	(29,33,34)	32	(42,64,65)
11	(6,10,11)	22	(24,29,25)	33	(63,66,64)

Triangle Function Value Block

T#	TF value	T#	TF value	T#	TF value
1	1.000	12	17.000	23	3.000
2	2.000	13	18.000	24	10.000
3	3.000	14	19.000	25	5.000
4	9.000	15	20.000	26	4.000
5	10.000	16	25.000	27	3.000
6	11.000	17	26.000	28	4.000
7	12.000	18	27.000	29	20.000
8	13.000	19	28.000	30	21.000
9	14.000	20	29.000	31	25.000
10	15.000	21	30.000	32	24.000
11	16.000	22	31.000	33	26.000

Figure 19(c): Data of TSP block and triangle function value block of new domain D_2. There are total of 33 triangles and one triangle function.

VMAP_OLD Block

Vertex Index	Original Vertex Index
1	1
2	2
3	6
4	7
5	8
6	10
7	11
8	12
9	14
10	15
11	16
12	17
13	18
14	19
15	21
16	22
17	23
18	25
19	26

VMAP_NEW Block

Vertex Index	Type	Index	U_1	U_2
20	1	2	0.5064934	0.0000000
21	1	30	0.3333333	0.0000000
22	1	24	0.6250000	0.0000000
23	1	19	0.4444449	0.0000000
24	1	14	0.5789474	0.0000000
25	1	8	0.3529411	0.0000000
26	1	7	0.3333332	0.0000000
27	2	3	0.3896104	0.1818182
28	1	38	0.3620689	0.0000000
29	1	41	0.2448978	0.0000000
30	1	47	0.3529409	0.0000000
31	1	46	0.6666667	0.0000000
32	1	60	0.6666666	0.0000000
33	1	51	0.8000000	0.0000000
34	1	56	0.3333333	0.0000000

Figure 19(d): The mapping tables of new domain D_2. VMAP_OLD block stores all old existing vertices. VMAP_NEW block stores all new created vertices after cutting.

Vertex Block (11 vertices)

V#	x and y	V#	x and y	V#	x and y
1	(26.000, 14.000)	6	(19.204, 14.755)	11	(17.000, 2.333)
2	(19.000, 9.000)	7	(17.706, 12.882)		
3	(19.000, 2.000)	8	(17.000, 10.000)		
4	(27.000, 7.000)	9	(17.000, 8.600)		
5	(22.172, 18.466)	10	(17.000, 7.000)		

Vertex Function Value Block

V#	VF value	V#	VF value	V#	VF value	V#	VF value
1	7.800	4	4.800	7	10.400	10	9.200
2	9.300	5	8.566	8	9.833	11	7.800
3	7.200	6	10.216	9	9.560		

SVP Block (19 sides)

S#	v_1 v_2	S#	v_1 v_2	S#	v_1 v_2	S#	v_1 v_2	S#	v_1 v_2
1	(5, 1)	5	(1, 4)	9	(3, 4)	13	(6, 7)	17	(9, 8)
2	(8, 2)	6	(2, 4)	10	(6, 1)	14	(11, 3)	18	(10, 9)
3	(7, 2)	7	(10, 2)	11	(6, 2)	15	(11, 2)	19	(10,11)
4	(1, 2)	8	(2, 3)	12	(6, 5)	16	(9, 2)		

Figure 20(a): Data of vertex block, vertex function value block and SVP block of new surface domain D_3. Total of 11 vertices and 19 sides and one vertex function. No side function exists.

TSP Block (9 triangles)

T#	S_1 S_2 S_3	T#	S_1 S_2 S_3	T#	S_1 S_2 S_3
1	(13, 3,11)	4	(19,15, 7)	7	(8,14,15)
2	(4, 6, 5)	5	(12,10, 1)	8	(17,16, 2)
3	(6, 8, 9)	6	(4,11,10)	9	(18, 7,16)

Triangle Function Value Block

T#	TF value	T#	TF value	T#	TF value
1	21.000	4	24.000	7	24.000
2	22.000	5	20.000	8	25.000
3	23.000	6	21.000	9	26.000

Boundary List

Boundary 1: 11 -> 10 -> 9 -> 8 -> 2 -> 7 -> 6 -> 5 -> 1 -> 4 -> 3 -> 11 (first boundary vertex)

Figure 20(b): Data of TSP block and triangle function value block and boundary list of new domain D_3. There are total of 9 triangles and one triangle function. Only one boundary exists in D_3.

VMAP_OLD Block

Vertex Index	Original Vertex Index
1	20
2	24
3	27
4	28

VMAP_NEW Block

Vertex Index	Type	Index	U_1	U_2
5	1	38	0.3620689	0.0000000
6	1	41	0.2448978	0.0000000
7	1	47	0.3529409	0.0000000
8	1	46	0.6666667	0.0000000
9	1	51	0.8000000	0.0000000
10	1	56	0.3333333	0.0000000
11	1	60	0.6666666	0.0000000

Figure 20(c): The mapping tables of new domain D_3. VMAP_OLD block stores all old existing vertices. VMAP_NEW block stores all new created vertices after cutting.

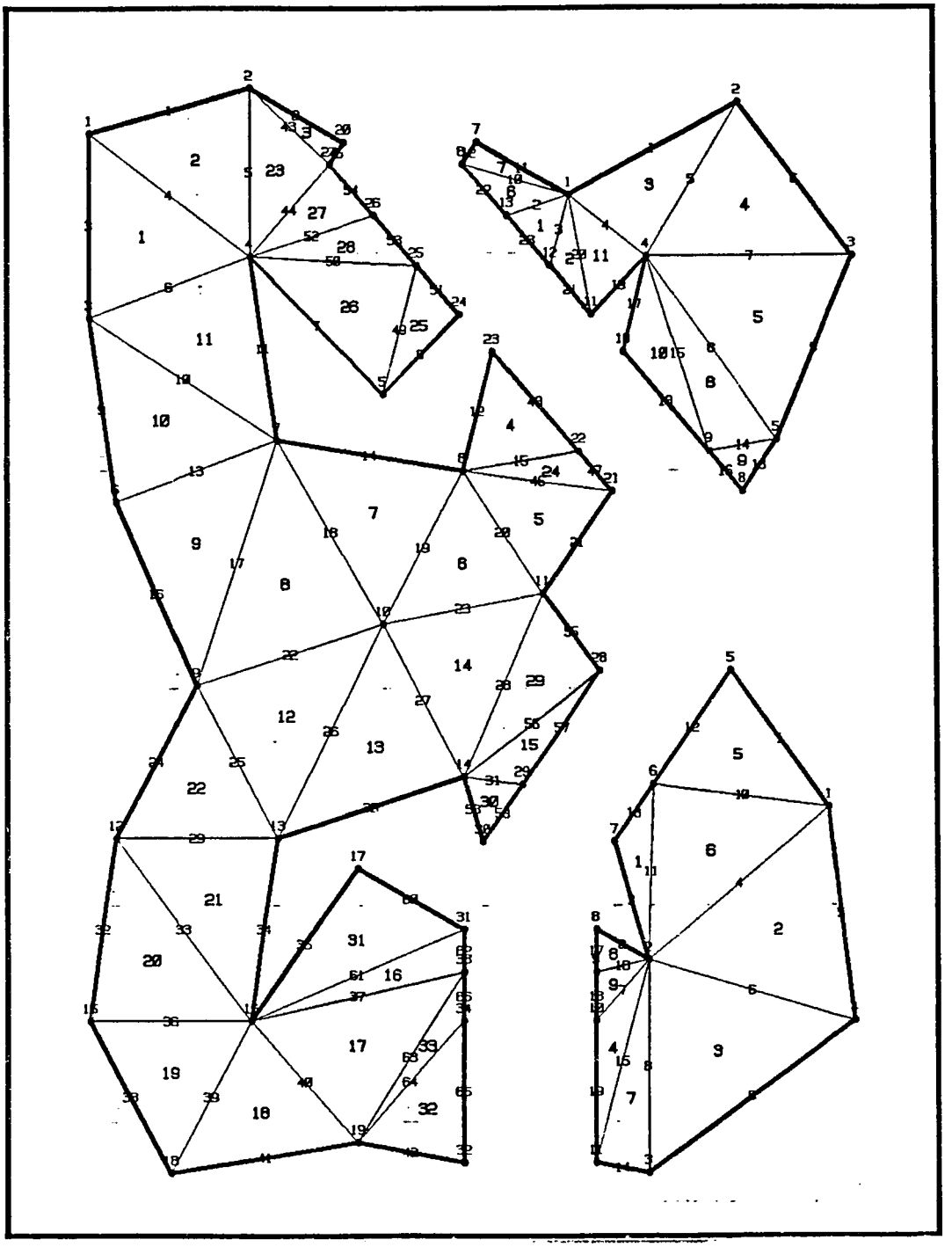


Figure 21: Three new domains D_1, D_2 and D_3 were created after the surface domain D was cut by the polyline shown in Figure 13.

In Figure 21, two interior boundaries were broken and they turned out to be part of exterior boundaries. Because no side-swapping was done, shapes of some triangles are long and narrow. Upgrading the quality of the triangle will be left to the user. Along the cut line, all boundary vertices and sides are paired. Two components in each pair have the same physical location but are located in different domains.

3.3.2 RELATED SUBROUTINES

The related subroutines in implementation includes:

1. Subroutine "LINES_CUT": This is the main routine of the function to partition a surface domain along the given route.
2. Subroutine "CUT_DOMAIN": This routine will cut the surface domain along a single line segment.
3. Subroutine "PT_IN_2DTDB": This routine is used to find the location of a given point P in a triangulated surface. Two variables, Position1 and Position2, are returned to indicate the location of P.

Position1	Description
0	P is outside exterior boundary
1	P is inside a hole
2	P is on a vertex
3	P is on a boundary side
4	P is on a non-boundary side

5	P is inside a triangle

Position1 =>	Position2
0	index of the exterior boundary (it is 1)
1	index of an interior boundary
2	index of a vertex
3, 4	index of a side
5	index of a triangle

4. Subroutine "FIND_INTERSECTIONS": This routine will find and insert all intersection points between a given line segment and a given boundary.
5. Subroutine "GET_NEXT_CUT_POINT": this routine will find and insert the next cutting point between a given line segment and triangulations from the current point.
6. Subroutine :INSERT_VERTEX": This routine will insert a new vertex into the surface domain.
7. Subroutine "ZIP": This routine will split the surface from the current point to the next point.
8. Subroutine "CREATE_NEW_DOMAIN": This routine will split the surface into two new surface domains.

Some other basic routines are also called by the above subroutines. They are either in IPL Library or in Appendix B.

3.3.3 ANALYSIS OF TIME COMPLEXITY

The time complexity of this algorithm depends on different factors at different stages such as initial time is proportional to the size of the database, zipping time is proportional to number of sides passed by the cutting line. In order to estimate the time required for the algorithm, an empirical analysis was made. The surface domain and the method used for this experiment is the same surface (see Figure 16) and the same method as used in the experiment for the function of line conversion. The only difference is that the surface is going to be split by the subroutine "LINES_CUT". The total time required for this algorithm has four major parts: time to initialize the process, time to locate points P_1 and P_2 in the surface, time to find the cutting points and zip the surface, and time to split the database of the working domain. Time to initialize process is proportional to the size of the database; time to locate points P_1 and P_2 is proportional to the total number of sides in the domain; time to zip the surface is proportional to the number of sides crossed by cutting line, and time to split the database is also proportional to the size of the database. Let T be the time required, N_s be the number of sides in the domain, S_{cross} be the number of sides crossed by the line segment P_1P_2, and N_{size} be the size of the database. We use the sum of number of vertices, number of sides, and number of triangles in the surface domain to replace the real size of the database. Assume that T is of the form

$$T = t_1 + t_2 + t_3 + t_4$$

$t_1 = aN_{size}$ (time to initialize the process)

$t_2 = bN_s$ (time to locate point)

$t_3 = cS_{cross}$ (time to zip the surface)

$t_4 = dN_{size}$ (time to split the database)

If no surface splitting occurs, t_4 can be omitted. Do the same experiment a few times with different location of P_1. The resulting time requirements are listed below:

S_{cross}	aN_{size}	bN_s	S_{cross}	dN_{size}
1	5	3	4	-
2	5	4	6	-
3	4	4	9	-
4	4	3	11	-
5	4	3	13	-
6	4	4	16	-
7	5	3	18	-
8	5	4	20	-
9	5	4	22	-
10	4	4	24	-
11	6	6	26	62

All time listed above is based on 10^{-2} second.

The total number of vertices, sides, and triangles in the domain are 36, 85 and 50 respectively. The average time to initialize the process is

0.05 second so we get a = 0.03. The average time to locate points P_1 and P_2 in the triangulated surface is about 0.045 second. The total number of sides in the domain is 85. We get b = 0.053. The time for zipping the surface is proportional to the number of crossed side. We get c is equal to 0.022 second for zipping each crossed side. The time for surface splitting is proportional to the size of the domain and we get d = 0.36.

The empirical time requirements T for function of partitioning is

$$T\ (10^{-2}\ \text{sec}) = 0.03N_{size} + 0.053N_s + 2.2S_{cross} + 0.36N_{size}$$

The main flow of a partition algorithm is very similar to the flow of line conversion algorithm. Both of them are guiding the process to follow the given route across the triangulations with different purposes. Algorithms described in this chapter were simplified to brief pseudo codes. The real implementations are much more complicated than the pseudo codes. Inserting a vertex, finding the next points, finding intersection points along boundary, and zipping procedure, splitting the domain (creating a new domain) and some numerical problems faced in implementation will be discussed in the next chapter.

4.0 DISCUSSION

The pseudo codes given in the previous chapter include some pseudo instructions such as inserting a vertex, finding next cutting point and splitting the domain. They are key roles to execute the functions. If they can be handled properly, the manipulation functions will run more efficiently. In the following sections, details of those key roles will be discussed.

4.1 FINDING INTERSECTION POINTS ON BOUNDARY

Given line segment L, it may enter and exit boundary B many times. To find a sequence of intersection points between L and B, all boundary sides on B will be checked one by one following the boundary direction. If boundary vertex v is on L and the pair of consecutive boundary sides on B containing v as an endpoint are in the same side of L, then vertex v will not be included in the sequence. Let endpoints of L be P_1 and P_2. Suppose the direction of L is from P_1 to P_2 and P_1 is outside of the boundary, then the closest intersection point to P_1 is the first domain entry point of L. Pairs of domain entry and exit points can then be found by tracing through the sequence from the first entry point.

Given two arbitrary line segments, they may intersect each other or be completely separate. To determine the relative position between them,

it is necessary to calculate four cross-products. If there are N boundary sides on a boundary, 4N cross products will be calculated to check all intersection points between line segment L and the boundary. To reduce unnecessary calculations, a rectangular window will be set to skip boundary sides which obviously have no intersection with L. The rectangular window, which is parallel to both x and y axes, can be defined by endpoints of L. If both endpoints of a boundary side are above, below, right or left to the window, no intersection will occur between this side and line segment L. If a boundary side is known to be intersected by line segment L, then the intersection point will be calculated.

4.2 LOCATION OF A POINT IN TRIANGULATIONS

For any given point P, it may be outside the exterior boundary, in a hole or inside the boundary. If P is inside the boundary, then whether P is on a vertex, on a side or inside a triangle is interesting to know. To determine that point P is inside or outside a boundary, an arbitrary point P_0 outside the boundary will be used. If line segment P_0P intersects the boundary even times, then point P is outside the boundary. Otherwise, P is inside the boundary. If boundary vertex v is touched by line segment P_0P and the pair of consecutive boundary sides containing v as an endpoint are in the same side of line segment P_0P, then boundary vertex v is not counted as an intersection point.

To determine whether point P is inside triangle T_i, three cross products will be calculated between P and each of the three sides of T_i. P may be on an end vertex of T_i. If one of the three cross products is equal to zero, then P is on that side. If all three cross products are non-zero and have the same direction, then P is inside of T_i. Otherwise, P is outside of T_i.

Determining the location of point P in triangulations is very expensive. The complete search and comparison through the entire database will be executed until the location of P is found. The search can be done by either checking whether P is outside the boundaries first or checking whether P is inside a triangle first. If point P is inside the domain, checking the boundaries is a waste for point P. But if point P is outside the domain, checking triangles will also be a waste. Without any further information about P except x and y coordinates, the best way to minimize the waste is to minimize the number of calculatiuon. If a surface domain D has a total of N_s sides and N_{bs} boundary sides, $4N_{bs}$ cross products will be needed for checking the boundaries, while N_s cross products will be needed for checking the triangles. If $4N_{bs}$ is less than N_s, boundaries will be checked first. Otherwise, triangles will be checked first. Once the location of P has been found, two arguments will be returned to tell the type and the index of location P.

Type	Type Description	Index

0	outside the exterior boundary	1 (exterior boundary)
1	in a hole	index of the boundary
2	on a vertex	index of the vertex
3	on a boundary side	index of the side
4	on a non-boundary side	index of the side
5	inside a triangle	index of the triangle

4.3 INSERTION OF A VERTEX

To insert a new vertex into a surface domain, the location of the new vertex in triangulations has to be found first. After that triangulations in the neighborhood of the new vertex will be retriangulated. Some sides and triangles will be modified and created. The number of new sides and new triangles created depends on the location of new vertex. Figure 22 shows the insertion of new vertex P in different locations. If the new vertex is on the boundary, it also needs to be inserted into the boundary list.

The function values associated with new vertices, sides and triangles calculated by predefined methods is an option up to the user. If user does not want the function values of new elements to be calculated automatically, flags will be set to indicate missing function values. Side-swapping is another option. If user requests it, local side-swapping will be performed around the neighborhood of the new triangles. After side-swapping, some function values may be not valid any more. Due to the fact that the function values may be missing, the mapping between the

new vertex and the original surface domain is recorded so that user can refer the location of the new vertex to calculate or readjust the function values of those new elements at some time later.

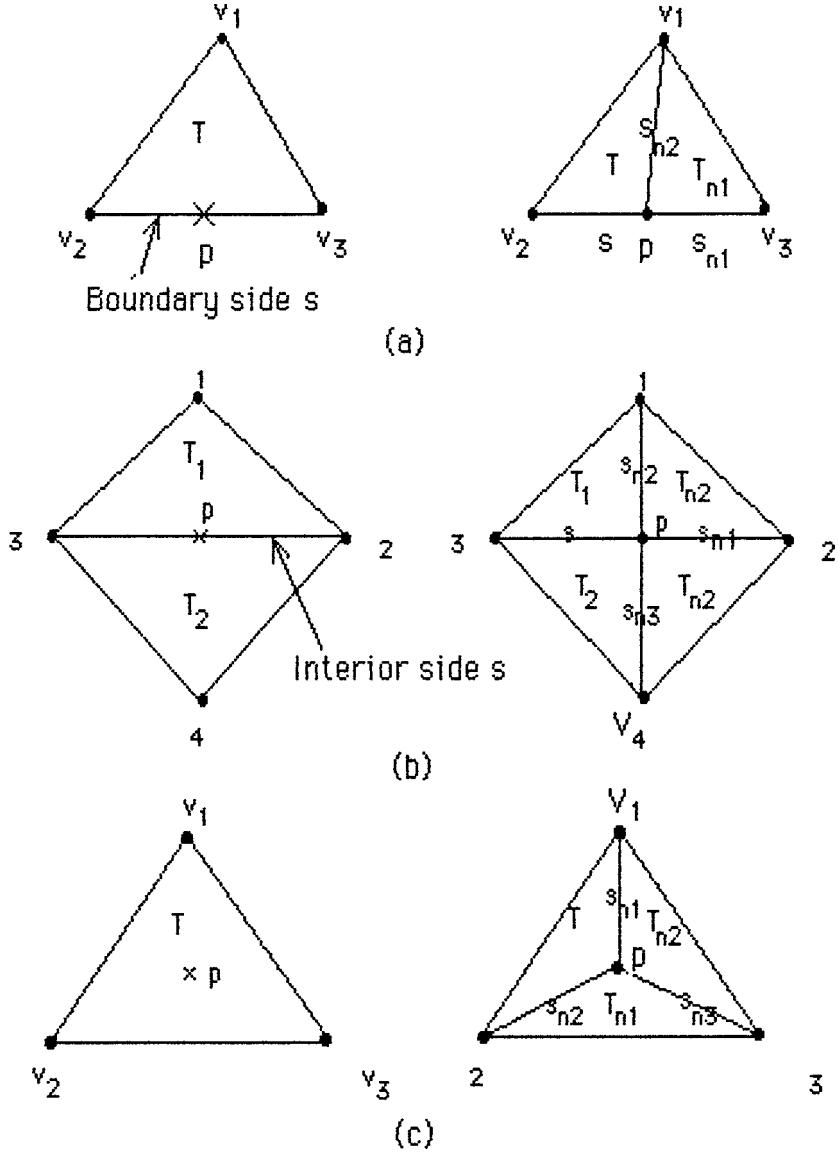


Figure 23: Insertion of point p into a triangulated surface (a) p is on a boundary side (b) p is on an interior side (c) p is inside a triangle. S_{ni} and T_{ni} represent new sides and new triangles created by the insertion

4.4 FINDING THE NEXT CUTTING (CROSSING) POINT

To find the next point on the path of line segment L, the vicinity of the current point on the path should be searched. Let P_1 and P_2 be the endpoints of L and P_{cur} be the current point on the path of L. The direction of path is from P_1 to P_2. P_{cur} may be on a vertex, a side, or inside a triangle. To find the next point P_{next} on the path in the direction of L, different location of P_{cur} will be treated separately.

If P_{cur} is on vertex V, all three vertices, side and triangle vicinity of V with radius one will be retrieved. Line segment L may be ended inside the boundary of vicinity one. If this is the case, then P_{next} has the same location as P_2. Otherwise, line segment L will exit the boundary of vicinity one from a vertex, or a side. Then P_{next} is the the intersection point between L and the exit side.

If P_{cur} is on boundary side S_b, there is only one triangle containing S_b. This is the triangle where L is going. If P_{cur} is on an interior side S_i, there are two triangles containing side S_i. One is the triangle where line L came from, and the other one is the triangle where line L is going. Let T be the triangle where line L is going . P_{next} is on one of the other two sides of T unless line segment L is end inside the triangle T.

If P_{cur} is inside of triangle T, this could only happen when P_{cur} is one of the endpoints of L. P_{next} is on one of the three sides of T unless the other endpoint of L is also in T.

In the cutting process, once P_{next} is found it will be inserted into the current working domain immediately if it is not an existing vertex. At the next turn, new P_{cur} has been set to P_{next} so that P_{cur} is always on a vertex. But in the line conversion process, no vertex will be inserted into the surface so that P_{cur} can be on a vertex, on a side or inside a triangle.

4.5 ZIPPING PROCEDURE

The surface splitting is done by the zipping process. When zipping down the zipper, we have to know where to start and where to end. Assume that the zipping starts from vertex V_1 down to vertex V_2, i.e., a slit will be cut from V_1 to V_2. Note that zipping always starts from a boundary vertex. Even if both V_1 and V_2 are not on the boundary, an interior boundary, which is from V_1 to V_2 and then back to V_1, will be created. Therefore, V_1 is always on a boundary.

Assume V_1 is on the boundary B_1. If V_2 is an interior vertex, vertex V_{1c} will be added to the same location as V_1. V_{1c} is created as the counter vertex of V_1 on the zipper. A new side connecting V_{1c} and V_2 will be added.

Boundary B_1 will be extended from V_1 via V_2 and V_{1c} then back to boundary B_1. Sides and triangles containing V_1 as one of the endpoints should be reconnected to V_{1c} instead of V_1 if they are in the same side of V_{1c}. See Figure 23(a).

If V_2 is on boundary B_2, vertices V_{1c} and V_{2c} will be added as the counter vertex of V_1 and V_2 on the zipper respectively and a new side connecting V_{1c} and V_{2c} will be added. V_{1c} and V_{2c} will also be inserted into the boundary. If B_1 and B_2 are the same boundary, boundary B_1 is actually separated into two parts except that V_2 and V_{2c} have the same location to connect the boundary as a whole. Boundary B_1 will be split from V_{1c} to V_{2c} to form two separated boundaries. See Figure 23(b). If B_1 and B_2 are different boundaries, V_{1c}, V_2 and V_{2c} will be inserted into the boundaries. At this point, B_1 and B_2 are actually connected together at V_2 and V_{2c}. Boundary B_1 and B_2 will be merged from V_1, via V_2, boundary B_2, V_{2c}, V_{1c}, then back to boundary B_1. See Figure 23(c). Sides and triangles that contain V_1 as one of the endpoints should be reconnected to V_{1c} instead of V_1, if they are in the same side of V_{1c}. Also, Sides and triangles contain V_2 as one of the endpoints should be reconnected to V_{2c} instead of V_2, if they are in the same side of V_{2c}.

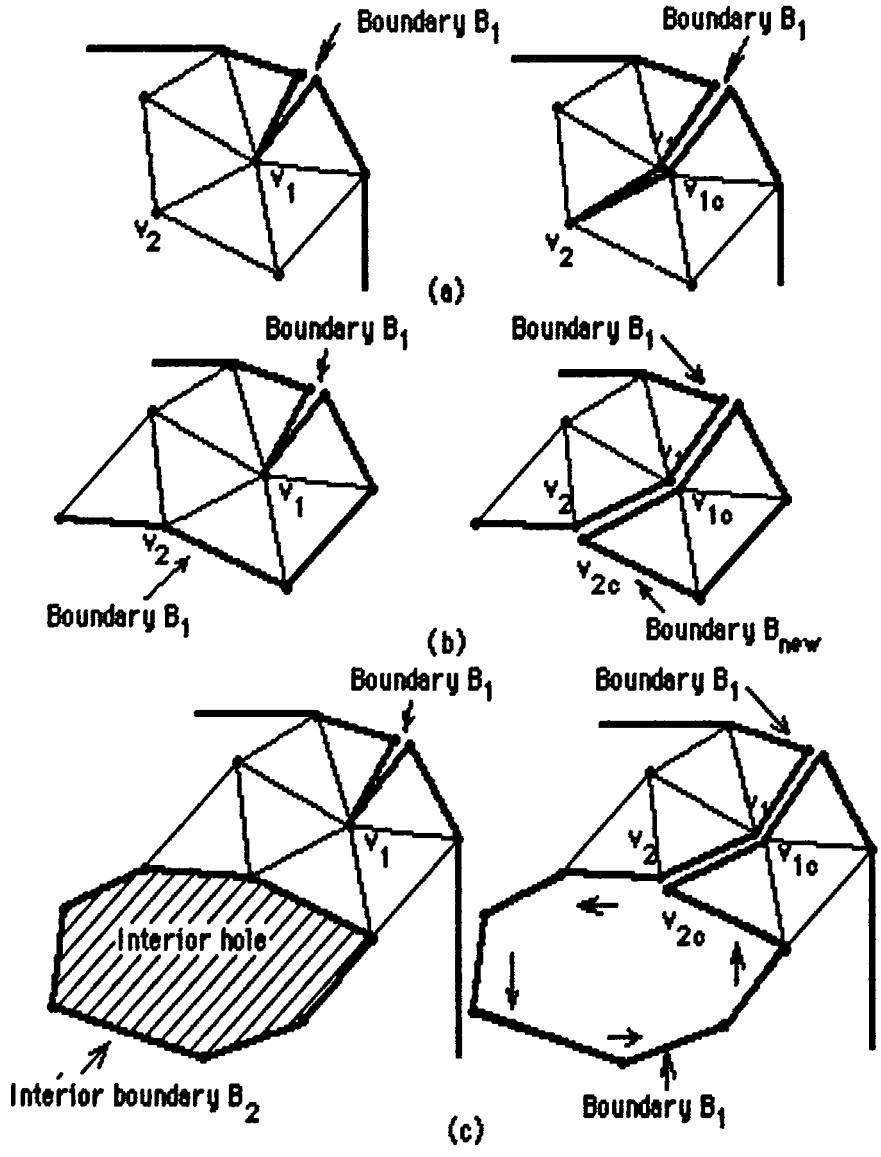


Figure 23: Zip down the surface from vertex v_1 to v_2. Vertex v_1 is on the boundary B_1 and v_2 is (a) an interior vertex (b) also on the boundary B_1 (c) on another boundary B_2

If the boundary is split, a cutting flag should be set to inform the process to create a new domain. Index of the new boundary, the exterior

boundary of new domain, should also be returned. Caution must be taken during reconnecting the boundary.

4.6 Split domain

Since the cutting flag was set and the index of the exterior boundary of the new domain has been returned. A new domain will be split from the current working domain. First, we have to separate vertices, sides and triangles by their own boundaries, then renumber the indices and update the entire databases.

Assume that part of surface domain D_1 will be split out to form a new domain D_2 which is surrounded by boundary B_{e2}. The remaining part of D_1 is surrounded by boundary B_{e1}. All vertices of D_1 are either inside B_{e1} or inside B_{e2}. To check that, we will use B_{e1} or B_{e2} whichever has fewer vertices. Vertices, on boundary B_{e1} and B_{e2}, are in the domain D_1 and D_2 respectively. For vertices on interior boundaries, we only need to check one vertex for each interior boundary. If the vertex on an interior boundary is inside of B_{e1}, then all vertices on that boundary are in domain D_1. Otherwise they are in domain D_2. For interior vertices, we have to check them one by one.

Suppose that boundary B_{e2} has fewer vertices. Usually, there are fewer interior vertices, sides and triangles are inside shorter boundary.

Find all vertices either inside or on the boundary B_{e2}, i.e. they belong to domain D_2. Sort the vertex list of D_2 by vertex index in ascending order so that the first part of the vertex list is all old vertices existing in the original domain and the second part of list is all new vertices created during cutting. New vertex index of D_2 is given by the order of this list. Allocate memory space and copy x-y coordinates for the vertices of D_2. Boundary lists of D_2 will be reconstructed according to new vertex index. Mapping tables and vertex function values of D_2 are also copied. All information related to those vertices should be removed from D_1, then re-number the vertex index of D_1. Note that a mapping between new and old vertex index of D_1 should be recorded for later use.

Next, pick out all sides which belong to D_2 to form SVP block of D_2. All vertex indices of SVP of D_2 should be updated to new vertex index of D_2 and side function values of D_2 copied. Remove sides of D_2 from SVP of D_1 and re-number the side vertex. Update the vertex indices of SVP of D_1 to new vertex indices. A mapping table between new and old side indices should be recorded.

Do the same process for triangles. Finally, we will update the boundary list and first boundary vertex of D_1 according to the new vertex index of D_1. Now, domain D_1 and D_2 are seperated completely.

This is the most expensive part of the entire cutting process. In IPL T-Base, arrays are basic structures. Array structure is convenient in direct access but weak in insertion or deletion of an element from a list. In conventional database systems, they never load everything into the memory because records in the same entity are independent. In triangulation database, the entire surface is connected as a whole. The entire database has to be loaded into memory so that the integrity of the surface can then be perceived. Dynamic array structures allow the database to be stored in less memory space with fast data retrieving, but difficult for updating.

5.0 CONCLUSION

The idea of surface representation by triangulation is simple and easy to understand, but to manipulate a digital triangulation system is quite difficult and time consuming. Throughout the applications, we must battle for the current location, affected triangulations, updating the database. Regardless of the problems faced during manipulations, digital representation of a surface domain by triangulations is a systematic way to save, reconstruct and manipulate a surface.

The algorithm to convert a line segment in the x-y coordinate system to its path in the triangulation system was discussed. With the help of this algorithm, the path of an operation along a given route can be easily found. Getting the profile of a particular vertex function along a given route is a very useful tool in taking a look into the surface. Partitioning of a surface by vertical plane(s) is also a very useful tool in classifying or trimming the surface to fit our needs. The current partition algorithm can only be used to cut a surface domain by vertical planes. This is a major limitation of the function. Partitioning a surface by a tilt plane is the more general and powerful case in reality.

In the future development, merging two surface domains, and setting the rules for moving a vertex are also very useful functions in manipulating the surface. The merging of two domains can be used to construct a more complicated surface. Rules of moving a vertex can free the contradictions between modifying the location of a vertex and the

original structure of triangulations.

The combined use of triangular grids and rectangular grids may be a very good idea in the future. Rectangular grids are easy to work with because of the rows and columns but are difficult to fit into irregular boundary shapes and preserve the accuracy of the surface. Triangular grids has no problem to fit into irregular boundary shapes and preserve the accuracy but are harder to work with (compared with rectangular grids). By combined use of both grids as proposed in this thesis, the surface domain can be handled more efficiently without losing its accuracy.

APPENDIX A
INTRODUCTION OF NEW SMITH/YIU SYSTEM

This is an introduction of SMITH/YIU triangulation database system which is revised and expanded from the SMITH triangulation system. Many new subroutines have been written and some subroutines of IPL Library have been modified. We will discuss and list all subroutines related to this work in the following paragraphs.

A.1 INTRODUCTION

To have a better idea of how to access and use this new system, it is worth reading "Triangulation Databases and Algorithms", Master thesis of W.M. Smith, first. This revised system still holds the same philosophy as the Smith's system except that

1. It allows more than one surface domain loaded in memory
2. Some new functions were added into the system.

To allow more than one surface domain loaded in memory, a new included file named "TDBCOMMON.FOR" has been created. In this file, there are five common areas declared. They are used to store the status and the starting addresses of data blocks of all domains. The five common blocks include:

1. Blank Common Area: Integer*4 IMEM(1)
Real*4 RMEM(1)

```
Logical      LMEM(1)
Byte         BMEM(1)
COMMON       IMEM(1)
Equivalence  (IMEM, RMEM)
Equivalence  (IMEM, LMEM)
Equivalence  (IMEM, BMEM)
```

We use dynamic array IMEM(1) as the base address of database system. With those equivalence statements, elements of any kind of data type can be stored and retrieved from this giant array. The starting address of each data block is a memory offset relative to IMEM. For example, to retrieve n^{th} element of an integer data block with starting address IB, we can write a statement

```
IDATA = IMEM(IB + n -1)
```

to assign the value to IDATA. Note that IB points to the first element of that data block.

2. TDB_Status common area: Many system status variables are stored in this common block.

1. Parameter MAX_DB =10

   MAX_DB is the maximum number of domains allowed in memory. It can be changed to allow more domains.

2. Integer*4 N_DB

   N_DB is number of domains currently in memory.

3. Integer*4 Active_domain

Active_Domain points to the current working domain.

4. Character Dbfname(MAX_DB)*120

   Dbfname is name of each domain.

5. Integer*4 Num_V(MAX_DB), Num_S(MAX_DB), Num_T(MAX_DB)

   They are number of vertices, sides and triangles of each domain respectively.

6. Integer*4 V_lots(MAX_DB), S_lots(MAX_DB), T_lots(MAX_DB)

   They are a number of 4-byte-word memory that have been allocated to vertex, SVP and TSP block of each domain respectively.

7. Integer*4 Ov_len(Max_DB), Ov_slots(MAX_DB)

   They are a number of data stored in VMAP_OLD block and a number of 4-byte words that have been allocated to VMAP_OLD block of each domain respectively.

8. Integer*4 Nv_len(Max_DB), Nv_slots(MAX_DB)

   They are a number of data stored in VMAP_NEW block and a number of four 4-byte words that have been allocated to VMAP_NEW block of each domain respectively.

9. Integer*4 Num_V_Func(MAX_DB), Num_T_Func(MAX_DB),
   Num_V_Func(MAX_DB)

   They are number of vertex functions, side functions and triangle functions of each domain respectively.

10. Integer*4 N_Boundaries(MAX_DB)

    N_Boundaries is number of boundaries of each domain.

3. **ADDR_INFO Common Area:** stores the starting addresses of all data blocks in dynamic memory.

   1. Integer*4 V_org(MAX_DB), SVP_org(MAX_DB),
      TSP_org(MAX_DB), BV_org(MAX_DB)

      They are the starting addresses of Vertex blocks, SVP blocks, TSP blocks and Boundary blocks of each domain respectively.

   2. Integer*4 Ov_org (MAX_DB), Nv_org (MAX_DB)

      They are the starting addresses of VAMP_OLD blocks and VMAP_NEW blocks of each domain respectively.

   3. PARAMETER Max_N_Vfvals = 24
      PARAMETER Max_N_Sfvals = 4
      PARAMETER Max_N_Tfvals = 30

      They are the maximum number of vertex, side and triangle functions allowed in each domain. Each of them is changeable.

   4. Integer*4 V_Fval_org(Max_N_Vfvals,Max_DB),
      MVFV_Flag_org (Max_N_Vfvals,Max_DB)

      They are the starting addresses of Vertex Function Value blocks and Missing Vertex Function Value Flag blocks of each domain respectively.

   5. Integer*4 S_Fval_org(Max_N_Sfvals,Max_DB),
      MSFV_Flag_org (Max_N_Sfvals,Max_DB)

      They are the starting addresses of Side Function Value blocks and Missing Side Function Value Flag blocks of each domain respectively.

6. Integer*4 T_Fval_org(Max_N_Tfvals,Max_DB),
   MTFV_Flag_org (Max_N_Tfvals,Max_DB)
   They are the starting addresses of Triangle Function Value blocks and Missing Triangle Function Value Flag blocks of each domain respectively.

4. **Boundary_Info Common area**: stores boundary information.
   1. PARAMETER Max_N_Boundaries = 20
      Max_N_Boundaries is the maximum number of boundaries allowed in each domain. It is changeable.
   2. Integer*4 First_Point_in_Boundary(Max_N_Boundaries,MAX_DB)
      It stores the first vertex of each boundary of each domain.

5. **MGR_STATUS Common area**: Information associated with VVP, VTP and STP managers.
   1. Logical*4 VVP_Present(MAX_DB)
      It is the flag to indicate VVP list of each domain has been created or not.
   2. Integer*4 VVPorg_org(MAX_DB)
      It is the starting address of VVP list of each domain
   3. Logical*4 VTP_Present(MAX_DB)
      It is the flag to indicate VTP list of each domain has been created or not.
   4. Integer*4 VTPorg_org(MAX_DB)

It is the starting address of VTP list of each domain.

5. Logical*4 STP_Present(MAX_DB)

   It is the flag to indicate STP list of each domain have been created or not.

6. Integer*4 STPorg_org(MAX_DB)

   It is the starting address of STP list of each domain.

An local VAX/VMS logical name "**TDBCOM**" under [IPLDCY] has been assigned to have the same meaning as the sub-directory **[IPLDCY.DBASE]**. To include this file by a new subroutine, a statement

INCLUDE 'TDBCOM:TDBCOMMON.FOR'

should be included in that subroutine.

Another included file is included by VVP, VTP and STP managers named "TDBPARAMS.FOR" have added a new parameter to indicate deletion of an existing vertex. To include this file, a statement

INCLUDE 'TDBCOM:TDBPARAMS.FOR'

should be included in subroutine.

The new functions that have been added to the systems are: Insertion of a vertex, deletion of vertex, line conversion, tile function values, cutting function, and plotting the surface domain(s).

Function	File-name	Subroutine name
Insertion of a vertex	INSERTPT.FOR	INSERT_VERTEX
Deletion of a vertex	DELETEPT.FOR	DELETE_VERTEX

Line Conversion	PROFILE.FOR	XYCRV_TRICRV
Tile Function Value	TILEF.FOR	TILEF_PROFILE
Cutting Function	CUT.FOR	LINES_CUT
Plotting Surface Domain(s)	PLTTRI.FOR	PLTTRI

See APPENDIX B for the details of these routines.

A.2. Routines have been modified

The reasons to modify a routine are:

a. Include file "TDBPARAMS.FOR" have been changed, so all those routines which include this file have to be recompiled. The parameters declared in this file are used by those VVP, SVP and TVP MANAGER and related routines. Now, one more parameter has been added into the file to allow deletion of a vertex. Those routines are listed below. The default subdirectory is [IPLDCY.DBASE]

	File-name	Subroutine name
1.	ATSTP.FOR	ADD_TRI_TO_STP
2.	ATVTP.FOR	ADD_TRI_TO_VTP
3.	CRESTP.FOR	CREATE_STP
4.	CREVTP.FOR	CREATE_VTP
5.	CREVVP.FOR	CREVVP

6.	DELBS.FOR	DEL_BSIDES
7.	GVVFDS.FOR	GVVFDS
8.	GVVFDV.FOR	GVVFDV
9.	RPVVP.FOR	REMOVE_PAIR_FROM_VVP
10.	RTSTP.FOR	REMOVE_TRI_FROM_STP
11.	RTVTP.FOR	REMOVE_TRI_FROM_VTP

b. Inculde file "TDBCOMMON.FOR" has been changed. The old program only allows one database to be loaded into memory at a single run. The modified file has changed all necessary variables of a database into arrays so that more than one database can be stored in memory at any time. The maximum number of surface domains stored in memory is controlled by a parameter MAX_DB which is currently set to ten. Those routines include both "TDBCOMMON.FOR" AND "TDBPARAMS.FOR" are listed below. The default sub-directory is [IPLDCY.DBASE]

	File-name	Subroutine name
1.	STPMAN.FOR	STP_MANAGER
2.	VTPMAN.FOR	VTP_MANAGER
3.	VVPMAN.FOR	VVP_MANAGER

c. Two routines are modified from Dr. Simpson's routines for adding epsilon to eliminate the round-off errors.

	File-name	Subroutine name
1.	PTLSEG.FOR	LSEG_VS_PT
	Same as Simpson's LPTRDL	
2.	PTLSEG.FOR	RLSEGLSEG
	Same as Simpson's	LSEGS_INTERSECT

d. Two main routines have been modified to fit into the new frame of the system.

	File-name	Subroutine name
1.	T1MANIP.FOR	This is the main program
2.	MANIP.FOR	MANIPULATE_2DTDB

The main program now includes all new functions in main menu.

A.3 LINKING THE SYSTEM

To link the system, a library, MANIP.OLB, with 42 files should be created first. By a VAX/VMS command file "TDBCOM:CRLIB.COM", MANIP.OLB will be generated automatically. The command is

```
$Libaray/Create  MANIP  ATSTP, ATVTP, CRESTP, CREVTP, CREVVP, -
                        DBMS, DELBS, DELETEPT, GETTV, GVVFDS, GVVFDV, -
                        INSERTPT, MANIP, PINTDB, REMXS, RPVVP, RTSTP, -
                        SRCHLS, RTVTP, STPMAN, VTPMAN, VVPMAN, -
                        STRSORT, LOCALTRI, 2DCROSS, LSWAP, CUT, -
```

```
                    CUTLIB, CUTNEWD, REMS, PTLSEG, PLTTRI, THKDOT, -
                    VMAP, PROFILE, PROFLIB, PMAP, TRIPT, XYCRV, -
                    PLANE_3PT, TILEF
```

A link command file "TDBCOM:T1TDB" will link the testing program "T1MANIP.OBJ" with all necessary libraries. The command is

```
$Link/Debug  TDBCOM:T1MANIP,-
             TDBCOM:MANIP/LIB, -
             IPL_LIB:IPLSUBS1/LIB,ALS/LIB, -
             DCI_LIB:PLTLIB/LIB,-
             TEKLIB/LIB,-
             HSRLIB/LIB,-
             UTLLIB/LIB
```

After linking, a testing program named T1MANIP.EXE is ready to run. There is a program named MAKEFILE.EXE in TDBCOM. It will allow the user to generate a testing surface domain datafile interactively.

APPENDIX B

TITLES AND DESCRIPTIONS OF FORTRAN PROGRAM

The titles and descriptions of new FORTRAN programs will give in the following paragraphs.

B.1 TITLE OF SUBROUTINES

Source	File-name	Entry	Title
BDY	2DCROSS.FOR	Cross_Product2	Perform 2-D cross product
BDY	2DCROSS.FOR	CompReal	Compare the values of two real numbers
WMS/BDY	ATSTP.FOR	Add_Tri_To_STP	Add a triangle to STP list
WMS/BDY	ATVTP.FOR	Add_Tri_To_VTP	Add a triangle to VTP list
WMS/BDY	CRESTP.FOR	Create_STP	Create STP list for T-Base
WMS/BDY	CREVTP.FOR	Create_VTP	Create VTP list for T-Base
WMS/BDY	CREVVP.FOR	Create_VVP	Create vertex to vertex pointers for T-Base
BDY	CUT.FOR	Lines_Cut	Main routine to cut a surface domain by a polyline
BDY	CUT.FOR	Cut_Domain	Cut a surface domain by a single line segment
BDY	CUTLIB.FOR	Find_Intersections	Find and insert the intersection points between a boundary and a line segment
BDY	CUTLIB.FOR	Find_Next_Cut_Point	Find and insert the next cutting point along the cutting line
BDY	CUTLIB.FOR	Zip	Split the surface domain from vertex 1 to vertex 2
BDY	CUTNEWD.FOR	Create_New_Domain	Split the working domain to

			form two new doamins
BDY	DBMS.FOR	Add_Triangle	Add a new triangle to a surface domain
BDY	-	Add_Function	Add a new function to a surface domain
BDY	-	Find_Side_Given_Vertex	Find a side by given one of its endpoints
BDY	-	Find_Triangle_Given_2sides	Find a triangle by given two of its three sides
BDY	-	Find_Vertex_Given_XY	Find a vertex by given x and y coordinates
BDY	-	Remove_Boundary_Vertex	Remove a list of consecutive vertices from boundary list
BDY	-	Remove_Triangle_Entry	Remove a triangle entry from TSP list
BDY	-	Remove_Vertex_Entry	Remove a vertex entry from vertex list
BDY	-	Swap_Triangle	Swap diagonal of a convex quadrilateral
WMS/BDY	DELBS.FOR	Del_Bsides	Delete boundary sides
BDY	DELETEPT.FOR	Delete_Vertex	Delete a vertex from a surface domain
BDY	DELETEPT.FOR	VST_Vcn1_V	Find vertex, side and triangle vicinity of radius 1
WMS/BDY	GVVFDS.FOR	GVVFDS	Get vertex vicinity in function domain of a side
WMS/BDY	GVVFDV.FOR	GVVFDV	Get vertex vicinity in function domain of a vertex
BDY	INSERTPT.FOR	Insert_Vertex	Insert a vertex into a surface domain

BDY	INSERTPT.FOR	Add_V_Outside_BD	Connect an out-boundary point to a surface domain
BDY	LOCALTRI.FOR	Local_Tri	Triangulate a polygon in a surface domain
BDY	LSWAP.FOR	Local_Side_Swap	Perform local side swapping for triangles
BDY	LSWAP.FOR	Distance	Compute distance between two points in x-y space
BDY	MANIP.FOR	Set_Working_Domain	Select a surface domain to work with
BDY	-	Rename_Domain	Rename the name of a surface domain
WMS/BDY	-	Read_Database	Transfer database to virtual memory
WMS/BDY	-	How_Many	How many vertices, sides, triangles, etc..
WMS/BDY	-	Xfer_XYF_Array	Transafer all x-y plus the specified function values
WMS/BDY	-	Release_Database	Release the database from virtual memory
WMS/BDY	-	Write_2DTDB	Write database to a file
WMS/BDY	-	Add_Vertex	Add a new vertex
WMS/BDY	-	Get_Vertex	Retrieve x and y values of a given vertex
WMS/BDY	-	Put_Vertex	Assign x and y values of a given vertex
WMS/BDY	-	Get_First_Bounday_Vertex	Get the first vertex of a given boundary
WMS/BDY	-	Get_Next_Boundary_Vertex	Get the next vertex on the boundary
WMS/BDY	-	Get_Prev_Boundary_Vertex	Get the previous vertex on the boundary

WMS/BDY -	Insert_boundary_Vertex	Make the given vertex be part of a boundary
WMS/BDY -	On_Boundary	Determine if a given vertex is on a boundary
WMS/BDY -	Which_Boundary	Determine which boundary a vertex is part of
WMS/BDY -	Orient_Boundaries	Force orientation of all boundaries
WMS/BDY -	Join_Boundaries	Join two boundaries
WMS/BDY -	Split_Boundary	Split a boundary into two
WMS/BDY -	Add_Boundary	Define a new boundary
WMS/BDY -	Get_V_VCN_V	Get vertex vicinity of a vertex
WMS/BDY -	Get_V_VCNFD_V	Get vertex vicinity in a function's domain of a vertex
WMS/BDY -	Get_S_VCN_V	Get side vicinity of a vertex
WMS/BDY -	Get_T_VCN_V	Get triangle vicinity of a vertex
WMS/BDY -	Get_Side	Return coordinates of a given side
WMS/BDY -	Find_Side_Given_Endpoints	Find a side with the given endpoints
WMS/BDY -	Find_Sides_Given_Endpoints	Find two sides with the given two endpoints
WMS/BDY -	Get_V_VCNFD_S	Get vertex vicinity in a function's domain of a side
WMS/BDY -	Put_Side	Replace vertices of a given side
WMS/BDY -	Add_Side	Add a side
WMS/BDY -	Delete_Side	Delete a side
WMS/BDY -	Find_Triangle_Given_Side	Find a triangle with the

			given side
WMS/BDY	-	Find_Triangles_Given_Side	Find two triangle with the given side
WMS/BDY	-	Put_Triangle	Replace sides of a triangle
WMS/BDY	-	Get_V_Fval	Retrieve a function value of a given vertex
WMS/BDY	-	Get_Fval	Earlier version of Get_V_Fval
WMS/BDY	-	Put_V_Fval	Assign a function value of a given vertex
WMS/BDY	-	Put_Fval	Earlier version of Put_V_Fval
WMS/BDY	-	Add_V_Func	Add a vertex function
WMS/BDY	-	Set_MVFV_Flag	Set missing vertex function value flag
WMS/BDY	-	Clr_MVFV_Flag	Clear missing vertex function value flag
WMS/BDY	-	Missing_VFV	Get missing vertex function value flag
WMS/BDY	-	Get_S_Fval	Retrieve a function value of a given side
WMS/BDY	-	Put_S_Fval	Assign a function value to a given side
WMS/BDY	-	Add_S_Func	Add a side function
WMS/BDY	-	Set_MSFV_Flag	Set missing side function value flag
WMS/BDY	-	Clr_MSFV_Flag	Clear missing side function value flag
WMS/BDY	-	Missing_SFV	Get missing side function value flag
WMS/BDY	-	Get_T_Fval	Retrieve a function value of a given triangle
WMS/BDY	-	Put_T_Fval	Assign a function value to a given triangle
WMS/BDY	-	Add_T_Func	Add a triangle function

WMS/BDY	-	Set_MTFV_Flag	Set missing triangle function value flag
WMS/BDY	-	Clr_MTFV_Flag	Clear missing triangle function value flag
WMS/BDY	-	Missing_TFV	Get missing triangle function value flag
BDY	PINTDB.FOR	Find_PT_In_2DTDB	Find point related to a 2-D triangulated surface
BDY	PLTTRI.FOR	PLTTRI	Create a HSR file to plot a triangulation (Modified from IPL's "PLOTTRI")
BDY	PMAP.FOR	Write_Profile	Create a "PRF" file to write a triangulated curve
BDY	PROFILE.FOR	XYCRV_TRICRV	Convert a x-y curve to triangulated curve in a given surface domain
BDY	PROFILE.FOR	Get_Profile	Convert a single line segment in x-y space to triangulated space
BDY	PROFLIB.FOR	Get_Intersections	Find intersections between boundary and a line
BDY	PROFLIB.FOR	Get_Next_Point	Find the next crossing point along a line in triangulated space
BDY	PTLSEG.FOR	Lseg_VS_PT	Location of point related to a directed line segment (modified from IPL's "LPTRDL")
BDY	PTLSEG.FOR	RLsegLseg	Relationship of a line segment to a line segment (This one is modified from IPL's "LSEGS_INTERSECT")
BDY	REMS.FOR	Boundary_Side	Check a given side is on a boundary or not
BDY	REMS.FOR	Remove_Side_Entry	Remove an entry from SVP

			list of a surface domain
BDY	REMS.FOR	Copy_Domain	Duplicate a given domain in memory
WMS/BDY	RPVVP.FOR	Remove_Pair_From_VVP	Record in VVP lists that a pair are disconnected
WMS/BDY	RTSTP.FOR	Remove_Tri_From_STP	Remove a triangle from STP list
WMS/BDY	RTSTP.FOR	Remove_Tri_From_VTP	Remove a triangle from VTP list
WMS/BDY	STPMAN.FOR	STP_Manager	Handle STP lists for triangulation database
BDY	STRSORT.FOR	Str_Sort_Int	Sort a list of integers by straight sort algorithm
BDY	STRSORT.FOR	Bin_Search	A binary search routine for a list of integers
BDY	STRSORT.FOR	Set_Domain_Name	Set surface domain name
BDY	TILEF.FOR	TileF_Profile	Get the profile along a triangulated curve
BDY	TILEF.FOR	Write_Tile	Create a "TLP" file to write the profile of a curve
BDY	TRIPT.FOR	XYPT_SidePT	Convert a x-y point to side coordinate $S(u_1)$
BDY	TRIPT.FOR	SidepT_XYPT	Given side coordinate $S(u_1)$ of a point to find its x, y values
BDY	TRIPT.FOR	XYPT_TriPT	Convert a x-y point to triangle coordinate $T(u_1,u_2)$
BDY	TRIPT.FOR	TriPT_XYPT	Given triangle coordinate $T(u_1,u_2)$ of a point to find its x, y values
WMS/BDY	VTPMAN.FOR	VTP_Manager	Handles VTP lists in triangulation database

BDY	VMAP.FOR	Write_Vertex_Map	Create a "VMP" file to write the vertex mapping tables
WMS/BDY	VVPMAN.FOR	VVP_Manager	Handle VVP list in triangulation database
BDY	XYCRV.FOR	TriCrv_XYCrv	Convert a triangulated curve to x-y curve
BDY	XYCRV.FOR	Convert_Point	Convert a triangulated point to its x-y coordinate

B.3 Descriptions of New Routines

1. File-name : 2DCROSS.FOR

```
C-----------------------------------------------------------------
C  CROSS_PRODUCT2: Perform 2-D cross product.
C
C       Given three vertex V0, V1 and V2, cross product is equal to
C       Product = (V1 - V0) x (V2-V1)
C-----------------------------------------------------------------
C                                Usage
C
C  Call CROSS_PRODUCT2(X0, Y0, X1, Y1, X2, Y2, PRODUCT)
C
C  INPUT
C  R*4 X0, Y0 are x and y values of vertex V0.
C  R*4 X1, Y1 are x and y values of vertex V1.
C  R*4 X2, Y2 are x and y values of vertex V2.
C
C  OUTPUT
C  R*4 PRODUCT is the result of cross product
C-----------------------------------------------------------------
```

2. File-name: 2DCROSS.FOR

```
C-----------------------------------------------------------------
C  COMPREAL: Compare two real numbers
```

```
C
C      Compare two real numbers and return the difference
C between the first number and the second number. Epsilon is used to
C eliminate round off error.
C-------------------------------------------------------------
C                          Usage
C
C Call COMPREAL(R1, R2, EPS, DIFF)
C
C INPUT
C R*4 R1, R2 are the real numbers to be compared
C R*4 EPS is the epsilon to be used during comparison.
C
C OUTPUT
C R*4 DIFF is the result of (R1 - R2)
C-------------------------------------------------------------
```

3. File-name: CUT.FOR

```
C-------------------------------------------------------------
C LINES_CUT: To cut a surface domain by a polyline
C
C      This routine allow user to cut the surface domain by a polyline.
C Either or both ends of line may be extended to infinite. It will call
C subroutine "CUT_DOMAIN" to cut the polyline one line segment by one
C line segment. Before cutting, we will duplicate the original domain
C in memory under a new domain name, and the real operation will be
C executed on the duplicated one so that we can always refer and keep
C track of the mapping between the new domain(s) and original domain.
C-------------------------------------------------------------
C                          Usage
C
C Call LINES_CUT([*SNER,] ORGD, NPT, PT, TFSINF, TFEINF, SWAP,
C                  FILL, N_D, LIST)
C
C Where SNER is statement number for error return
```

```
C
C INPUT
C I*4 ORGD is the index of the original domain to be cut
C I*4 NPT is the number of points in polyline.  It should be equal or
C       greater than 2
C R*4 PT is a list of x and y values of each point of the polyline.
C L   TFSINF indicates the first line segment extended to infinite if
C       the value is true
C L   TFEINF indicates the last line segment extended to infinite if
C       the value is true
C L   SWAP indicates the local side-swapping is performed during
C       the process if the value is true
C L   FILL fill the function values of new vertices, sides and triangles
C            if value is true.
C
C OUTPUT
C I*4 N_D is number of new domains created after operation
C I*4 LIST(i) (i= 1, N_D) is a list of domain indices of all new
C       domains
C-----------------------------------------------------------------
```

4. File-name: CUT.FOR

```
C-----------------------------------------------------------------
C CUT_DOMAIN: Cut the surface domain by a single line segment.
C
C       In this routine, the surface domain will be cut by one line
C segment from the endpoint with larger x-value to the point with less
C x-value.  The cutting algorithm is quite complicated and we will skip
C here.  All new domains have their own domain names, domain indices
C and memory locations.
C-----------------------------------------------------------------
C                                 Usage
C
C Call CUT_DOMAIN([*SNER,] ORIGIN_D, X1, Y1, X2, Y2, SWAP, Fill, EPS,
C                          N_D, LIST)
C
```

```
C Where SNER is the statement number for error return
C
C INPUT
C I*4  ORIGIN_D is the index of the original domain to be cut
C R*4 X1, Y1 are the x and y value of one end point
C R*4 X2, Y2 are the x and y value of the other end point
C L    SWAP indicates the local side-swapping is performed during
C      the process if the value is true
C L    FILL fill the function values of new vertices, sides and triangles
C           if value is true.
C R*4 EPS is the epsilon used in this cutting
C
C OUTPUT
C I*4  N_D is total number of new surface domains created
C I*4  LIST(I) (I=1, N_D) is a list of domain indices of new domains
C-------------------------------------------------------------
```

5. CUTLIB.FOR

```
C-------------------------------------------------------------
C FIND_INTERSECTIONS: Find and insert all intersection points between
C      boundary and a line segment.
C
C      This is a special routine called by subroutine "CUT_DOMAIN" to
C find and insert all intersection points between a boundary of a
C surface domain and a line segment.
C-------------------------------------------------------------
C                              Usage
C Call FIND_INTERSECTIONS([*SNER,] POINT, SWAP, FILL, WB, ORGD,
C              NUM_SEC, B_SEC_XY, START, CLOCKWISE, COUNT_CLOCK)
C
C Where SNER is statement number of error return
C
C INPUT
C R*4 POINT(2,2) are x and y values of end points of given line segment
C L    SWAP indicates local side-swapping is performed during the
C      process if value is true
```

```
C L     FILL fill the function values of new vertices, sides and triangles
C             if value is true.
C I*4  WB is the index of working boundary
C I*4  ORGD is index of the original domain to be cut
C
C OUTPUT
C I*4  NUM_SEC is total number of intersection points found
C R*4 B_SEC_XY(2,j) (j=1,NUM_SEC) are x and y values of each
C       intersection point
C I*4  START is an index of intersection point which has the largest
C       x-value
C I*4  CLOCKWISE is an index of the next intersection point of START
C I*4  COUNT_CLOCK is an index of the previous intersection point of
C       START
C-------------------------------------------------------------
```

6. File-name: CUTLIB.FOR

```
C-------------------------------------------------------------
C FIND_NEXT_CUT_POINT: To find and insert the next cut point along
C       cutting line
C
C       This is a special routine called by "CUT_DOMAIN". Given the
C last cutting point and the cutting line, it will use the vicinity of
C last cutting point to find the next cutting point
C-------------------------------------------------------------
C                              Usage
C
C Call  FIND_NEXT_CUT_POINT([*SNER,] CLINE, LASTV, ORGD, MAXV,
C                       SWAP, FILL, VLIST, SLIST, TLIST, NEXTV, X, Y)
C
C Where SNER is statement number for error return
C
C INPUT
C R*4 CLINE(2,2) are x and y values of end points of current cutting
C       line segment
C I*4  LASTV is the vertex index of last cutting point
```

```
C I*4  ORGD is the index of original domain to be cut
C I*4  MAXV is the total number of vertices of working domain
C L    SWAP indicates local side-swapping is performed during the
C      process if the value is true
C L    FILL fill the function values of new vertices, sides and triangles
C           if value is true.
C
C OUTPUT
C I*4  NEXTV is the vertex index of next cutting point.  If no new point
C      is found return zero
C R*4 X, Y are x and y value of next cutting point
C-------------------------------------------------------------
```

7. File-name: CUTLIB.FOR

```
C-------------------------------------------------------------
C ZIP: Split the surface domain from vertex 1 to vertex 2
C
C      This is a special routine called by "CUT_DOMAIN".  The idea of
C cutting is just like we zip down a zipper.  Given two vertices V1 and
C V2, where V1 is a boundary vertex and V2 is the point to be cut in.
C To perform as a zipper, we will create a new vertex V1' with the
C same location as V1 and a new side (V1', V2) as the counter part of
C vertex V1 and side (V1, V2), then add vertices V2 and V1' into
C boundary list.  Then reconnect all those sides and triangles,
C containing V1 as one of the endpoints and locating at the same side
C of V1, to V1' instead of V1.  Finally, if V2 is on the same boundary
C then a new isolated domain created, or if V2 is on a different
C boundary then merge these two boundaries
C-------------------------------------------------------------
C                              Usage
C
C Call ZIP([*SNER,] POINT, V1, XY1, V2, XY2, RIGHT,FILL,  ORGD, W_B,
C          NEWB, NEXTS, CUTIT)
C
C Where SNER is statement number for error return
C
```

```
C INPUT
C R*4 POINT(2,2) are x and y values of endpoints of current cutting
C       line segment
C I*4  V1, V2 are vertex indice of last and next cut points
C R*4 XY1(2), XY2(2) are x and y value of point V1 and V2
C L    RIGHT indicates the right part or the left part of domain will
C       be cut out of the working domain, if necessary
C               If RIGHT = True, then cut off right part of domain
C               If False, then cut off left part of domain
C L    FILL fill the function values of new vertices, sides and triangles
C            if value is true.
C I*4 ORGD is the index of the original domain to be cut
C I*4 W_B is the boundary index where V2 is on. If V2 is an interior
C       vertex then W_B is equal to zero
C
C OUTPUT
C I*4 NEWB is the boundary index where V2 is on after cutting
C I*4 NEXTS is set to V2 or V2' or zero.  When two boundary joined
C       together, it is difficult to tell the next cutting will continue
C       from V2 or V2'.  With the help of NEXTS, We can know where
C       to go next. If no joining happened, NEXTS is set to zero
C L    CUTIT indicates a new isolated domain is formed if the value
C       is true.
C-------------------------------------------------------------
```

8. File-name: CUTNEWD.FOR

```
C-------------------------------------------------------------
C CREATE_NEW_DOMAIN: Split a surface domain into two new domains
C
C       This is a special routine called by CUT_DOMAIN.  When an
C isolated domain has been formed, we use this routine to cut it out
C from the working domain and store it to a new  memory location.  All
C the indices in original working domain and new created domain must
C be packed and all associated information must be updated.
C-------------------------------------------------------------
C                              Usage
```

```
C
C Call CREATE_NEW_DOMAIN([*SNER,] EXTB)
C
C Where SNER is the statement number for error return
C
C INPUT
C I*4 EXTB is a boundary index of working domain.  It will be the
C       exterior boundary of the new domain
C
C Information of both old and new surface domains has been updated.
C-------------------------------------------------------------
```

9. File-name: DBMS.FOR

```
C-------------------------------------------------------------
C This is a multiple entry subroutine contains some basic functions
C needed by database handler.  All functions here should be
C transparent to users and only be called by other subroutines.
C-------------------------------------------------------------
C
C Entry points in this subroutine:
C
C 1.   Add_Triangle: To add a triangle into surface
C 2.   Add_Function : To add a function to a surface domain
C 3.   Find_Side_Given_Vertex: Find a side by given one of its endpoint
C 4.   Find_Triangle_Given_2Sides: Find a triangle by given two sides
C 5.   Find_Vertex_Given_XY: Find a vertex by given x and y values
C 6.   Remove_Boundary_Vertex: Remove vertices from boundary list
C 7.   Remove_Triangle_Entry: Remove triangle entries from TSP list
C 8.   Remove_Vertex_Entry: Remove vertex entries from vertex list
C 9.   Swap_Triangle: Swap the diagonal of a convex quadrilateral
C-------------------------------------------------------------
C ENTRY: Call ADD_TRIANGLE([*SNER,], IS1, IS2, IS3, IT)
C
C Where SNER is statement number for error return
C
C INPUT
```

```
C I*4  IS1, IS2, IS3 are three sides of a triangle
C
C OUTPUT
C I*4  IT is the index of the new added triangle
C-------------------------------------------------------------
C ENTRY Call ADD_FUNCTION([*SNER,], ITYPE, INDEX)
C
C Where SNER is statement number for error return
C
C INPUT
C I*4  ITYPE indicates the type of function.
C              If ITYPE = 0, then add a vertex function
C              If ITYPE = 1, then add a side function
C              If ITYPE = 2, then add a triangle function
C
C OUTPUT
C I*4  INDEX is the index of new function
C-------------------------------------------------------------
C ENTRY Call FIND_SIDE_GIVEN_VERTEX([*SNER,] IV, IS)
C
C Where SNER is statement number for error return
C
C INPUT
C I*4  IV is the index of given vertex
C
C OUTPUT
C I*4  IS is the first side in SVP list which contains IV as one of its
C      endpoint
C-------------------------------------------------------------
C ENTRY Call FIND_TRIANGLE_GIVEN_2SIDES([*SNER,] IS1, IS2, IT)
C
C Where SNER is statement number for error return
C
C INPUT
C I*4  IS1, IS2 are two given sides
C
```

```
C OUTPUT
C I*4  IT is the triangle which contains IS1 and IS2 as two of its
C      three sides
C-------------------------------------------------------------
C ENTRY Call FIND_VERTEX_GIVEN_XY([*SNER,] X1, Y1, IV)
C
C Where SNER is statement number for error return
C
C INPUT
C R*4 X1, Y1 are x and y values of the vertex to be found
C
C OUTPUT
C I*4  IV is the index of a vertex which locates in (X1, Y1)
C-------------------------------------------------------------
C ENTRY Call REMOVE_BOUNDARY_VERTEX([*SNER], LEN, VLIST)
C
C Where SNER is statement number for error return
C
C INPUT
C I*4  LEN is number of boundary vertices to be removed
C I*4  VLIST(i) (i=1, LEN) is a list of consecutive boundary vertices to
C      be removed.
C
C The boundary list of database will be updated
C-------------------------------------------------------------
C ENTRY REMOVE_TRIANGLE_ENTRY([*SNER,] LEN, LIST)
C
C Where SNER is statement number for error return
C
C INPUT
C I*4  LEN is number of triangles to be removed
C I*4  LIST(i) (i=1, LEN) is a list of triangles to be removed
C
C All information in database related to these triangles are
C updated, and the indices of triangle are packed
C-------------------------------------------------------------
```

```
C ENTRY REMOVE_VERTEX_ENTRY([*SNER,] LEN, LIST)
C
C Where SNER is statement number for error return
C
C INPUT
C I*4  LEN is number of vertices to be removed
C I*4  LIST(i) (i=1, LEN) is a list of vertices to be removed
C
C All information in database related to these vertices are updated,
C and the indices of vertex are packed
C-------------------------------------------------------------
C ENTRY Call SWAP_TRIANGLE([*SNER,] IT1, T1S1, T1S2, T1S3, IT2,
C                                  T2S1, T2S2, T2S3)
C
C Where SNER is statement number for error return
C
C INPUT
C I*4 IT1, IT2 are indices of two neighbored triangles which form a
C       convex quadrilateral
C I*4  T1S1, T1S2, T1S3 are three sides of triangle IT1
C I*4  T2S1, T2S2, T2S3 are three sides of triangle IT2
C
C Triangles IT1 and IT2 are side-swapped to another diagonal
C-------------------------------------------------------------
```

10. File-name: DELETEPT.FOR

```
C-------------------------------------------------------------
C DELETE_VERTEX: Delete a vertex from the surface domain
C
C       This routine delete an existing vertex from surface domain. The
C sides and triangles which contain this vertex should be removed. If
C the vertex is on a boundary, vertex is also removed from boundary
C list.  If the vertex is not on the boundary, then we have to
C re-triangulate the local area after the vertex has been deleted.
C-------------------------------------------------------------
C                              Usage
```

```
C
C Call DALETE_VERTEX([*SNER,] IV, MAXV, NV, VLIST, NS, NEWS,
C                       SLIST, NT, NEWT, TLIST, SCR1, SCR2)
C
C Where SNER is statement number of error return
C
C INPUT
C I*4 IV is the index of the vertex to be deleted
C I*4 MAXV is the total number of vertices of surface domain
C
C OUTPUT
C I*4 NV is number of vertices which are the first vertex vicinity of
C     IV
C I*4 VLIST(i) (i=1, NV) is a list of vertices which are the first
C     vertex vicinity of IV
C I*4 NWS is number of new sides created after re-triangulate
C     local area
C I*4 NS is number of sides to be deleted due to deletion of IV
C I*4 SLIST(i) (i=1, NS+NEWS) is a list side indices. The first NS
C     sides are sides to be deleted and the last NEWS sides are new
C     sides created after re-triangulation
C I*4 NT is number of triangles to be deleted due to deletion of IV
C I*4 NEWT is number of new triangles created after re-triangulate
C     the local area
C I*4 TLIST(i) (i=1,NT+NEWT) is a list of triangle indices. The first
C     NT triangles are triangles to be deleted and the last NEWT
C     triangles are new triangles created after re-triangulation
C-----------------------------------------------------------------
```

11. File-name: DELETEPT.FOR

```
C-----------------------------------------------------------------
C VST_VCN1_V: Find vertex, side and triangle vicinity of radius 1
C
C     This routine can find all vertex, side and triangle vicinity of a
C vertex with radius 1. All three lists of vertex, side and triangle are
C ordered in clockwise or counterclockwise direction
```

```
C-------------------------------------------------------------
C                              Usage
C Call CST_VCN1_V([*SNER,] IV, MAXV, NV, VVCN, NS, SVCN, NT,
C                      TVCN, CLOZ)
C
C Where SNER is statement number for error return
C
C INPUT
C I*4  IV is the center vertex
C I*4  MAXV is the total number of vertices in database
C
C OUTPUT
C I*4  NV is total number of vertices found in vicinity 1 of IV
C I*4  VVCN(i) (i=1, NV) is a list of vertex indices in vicinity 1 of IV
C I*4  NS is total number of sides found in vicinity 1 of IV
C I*4  SVCN(i) (i=1, NS) is a list of side indices in vicinity 1 of IV
C I*4  NT is total number of triangles found in vicinity 1 of IV
C I*4  TVCN(i) (i=1, NT) is a list of triangles found in vicinity 1 of
C      IV
C L    CLOZ indicates the close vicinity if value is true.
C-------------------------------------------------------------
```

12. File-name: INSERTPT.FOR

```
C-------------------------------------------------------------
C INSERT_VERTEX: Insert a new vertex into surface domain
C
C      This routine insert a new vertex, V, into surface domain. The
C new vertex may be inside or outside of domain. The area neighboring
C the new vertex may or may not perform side-swapping. If the
C boundary is out side of the boundary, we will find a boundary vertex
C which has the shortest distance to the new vertex, then connect the
C vertex V and boundary vertices along both clockwise and
C counterclockwise directions, until the next boundary vertices in both
C directions can not be seen directly from V.
C-------------------------------------------------------------
C Call INSERT_VERTEX([*SNER,] ORGD, X, Y, IV, SWAP, FILL, WV, NWS,
```

```
C                          NWT)
C
C Where SNER is statement number for error return
C
C INPUT
C I*4  ORGD is the index or the original database domain
C R*4 X, Y are x and y values of the new vertex
C L    SWAP indicates the local side-swapping is performed during the
C      process if value is true
C L    FILL fill the function values of new vertices, sides and triangles
C           if value is true.
C
C OUTPUT
C I*4  IV is the index of the new vertex
C I*4  WV is the index of a vertex which has the same location as
C      the new vertex, if any.
C I*4  NWS is number of sides have been created or changed.
C I*4  NWT is number of triangles have been created or changed
C
C The list of sides which have been created or changed are stored in
C address of SCR_ORG(1,ACTIVE_DOMAIN)
C The list of triangles which have been created or changed are
C stored in address of SCR_ORG(2,ACTIVE_DOMAIN)
C-------------------------------------------------------------
```

13.File-name: INSERTPT.FOR

```
C-------------------------------------------------------------
C ADD_V_OUTSIDE_BD: Connect an outside point to a surface domain
C
C      This is a special routine called by "INSERT_VERTEX". It will join
C a vertex which is outside of boundary or in a hole to a surface domain
C-------------------------------------------------------------
C CALL ADD_V_OUTSIDE_BD ([*SNER,] X, Y, IV, IB, NWS, NWT)
C
C where SNER is the statement number for error return
C
```

```
C INPUT
C R*4 X, Y are X- and Y-coordinates of vertex in question.
C I*4  IV is vertex index of vertex in question.
C I*4  IB is the boundary index where the vertex outside of.
C
C OUTPUT
C I*4  NWS is number of sides have been created or changed
C I*4  NWT is number of triangles have been created or changed.
C
C A list of side indices which have been created or changed will be
C stored in SCR_ORG (1,ACTIVE_DOMAIN) if any.
C A list of triangle indices which have been created or changed will
C be stored in SCR_ORG (2,ACTIVE_DOMAIN) if any.
C-----------------------------------------------------------------
```

14.File-name: LOCALTRI.FOR

```
C-----------------------------------------------------------------
C LOCAL_TRI: Triangulate a polygon in a database domain.
C
C      This subroutine is used by triangulation database handler to
C re-triangulate a polygon area.  All vertices and boundary sides are
C already existing in database, so the output of this routine only
C contains : 1) interior sides of this polygon 2) triangles after
C triangulation.
C-----------------------------------------------------------------
C                                Usage
C
C CALL LOCAL_TRI ([*SNER,] MAXV, NV, VLIST, NEWS, SLIST, NEWT,
C                         TLIST, XY, VP)
C
C where SNER is the statement number of error return
C VP is address of working space in commom area
C
C INPUT
C I*4  MAXV is total number of vertices in database.
C I*4  IV is number of vertices of polygon.
```

```
C I*4  VLIST(I) (I=1,...NV)  is the list of polygon vertices.
C R*4 XY stores the X and Y values of each index.
C
C OUTPUT
C I*4  NS is number of sides created after triangulation
C I*4  SLIST(J) (J=1,...NS)  is a list of new sides after triangulation.
C I*4  NT is number of triangles created after triangulation
C I*4  TLIST(K)  (K=1,...NT)   is  a  list  of  new  triangles  after
C      triangulation.
C-------------------------------------------------------------
```

15.File-name: LSWAP.FOR

```
C-------------------------------------------------------------
C LOCAL_SIDE_SWAP:  Perform side swapping for some triangles.
C
C      This routine will perform local side-swapping for a list of
C triangles in database domain.  Swapping principle is according to
C minimun length.
C-------------------------------------------------------------
C                                Usage
C
C CALL LOCAL_SIDE_SWAP ([*SNER,] LEN, TLIST)
C
C where SNER is the statement number for error return
C
C INPUT
C I*4  LEN is number of triangles to be swapped.
C I*4  TLIST(I) (I=1,LEN) is a list of triangle indices to be swapped
C
C After side-swapping, database is already updated
C-------------------------------------------------------------
```

16.File-name: LSWAP.FOR

```
C-------------------------------------------------------------
C DISTANCE: To compute the distance between two points.
C-------------------------------------------------------------
```

```
C                              Usage
C
C L = DISTANCE(X1, Y1, X2, Y2)
C
C INPUT
C R*4 X1, Y1 are coordinates of point 1.
C R*4 X2, Y2 are coordinates of point 2.
C
C OUTPUT
C R*4 L is the distance between point 1 and point 2
C-----------------------------------------------------------------
```

17.File-name : PINTDB.FOR

```
C-----------------------------------------------------------------
C FIND_PT_IN_2DTDB: Find a point related to 2-D triangulated surface.
C
C      This subroutine relates a point in 2-space to 2-D triangulated
C surface.  The surface may have holes.  Six relations are
C distinguished: The point may be (1) outside of exterior boundary (2)
C inside a hole (3) on a vertex (4) on a boundary (5) on non-boundary
C side (6) inside of a triangle.
C-----------------------------------------------------------------
C                              Usage
C
C CALL PT_IN_2DTDB ([*SNER,] X, Y, SURFACE_INDEX, POSITION1,
C                   POSITION2)
C
C Where SNER is a statement no. for error return
C
C INPUTS
C R*4 X, Y are the X- and Y-coordinates of the point in question.
C I*4 SURFACE_INDEX is the index of the surface in question.
C
C OUTPUTS
C I*4 POSITION1 relates the point to the surface:
C      0 = outside of exterior boundary
```

```
C       1 = inside  of an interior hole
C       2 = corresponds to a vertex
C       3 = on a boundary side but between its endpoints
C       4 = on a non-boundary side but between its endpoints
C       5 = properly contained by a triangle
C
C I*4 POSITION2  qualifies POSITION1 as follows:
C       POSITION1 = 0 --> POSITION2  is 1 (exterior boundary)
C       POSITION1 = 1 --> POSITION2  is the index of an interior
C                         boundary
C       POSITION1 = 2 --> POSITION2 is the index of the vertex
C       POSITION1 = 3 --> POSITION2 is the index of the boundary side
C       POSITION1 = 4 --> POSITION2 is the index of the side
C       POSITION1 = 5 --> POSITION2 is the index of the triangle
C-------------------------------------------------------------
```

18.File-name: PLTTRI.FOR

```
C-------------------------------------------------------------
C PLTTR  Create an HSR file to plot a triangulation
C
C       This program creates, or adds to, an "HSR" file for plotting a
C triangulation specified in the format as produced, for example, by
C subroutine RADIAL_SWEEP.  It provides for controlled thickness of
C lines in the triangulation and of dots representing vertices.  The
C vertex, side, triangle indexing, and  cut lines may also be displayed.
C
C       This program is modified from Dr. Simpson's program
C-------------------------------------------------------------
C                              Usage
C
C CALL PLTTRI (*SNER, VERTS, NVERTS, ISVP, NSIDES, ITSP, NTRIS,
C                  IFXSD, NFXSD, NCUT, CUT, TFINIT, FILE, TITLE, XMAX,
C                  YMAX, XPLO, XPHI, YPLO, YPHI, XLO, XHI, YLO, YHI,
C                  THKV, THKS, THKBS, THKFS, THKGW, HVN, HSN, HTN)
C
C where SNER is statement number for error return
```

```
C
C INPUTS
C R*4 VERTS(I,J),I=1..2,J=1..NVERTS are the X, Y coordinates of
C       vertices in the present triangulation
C       VERTS(1,J) = X coordinate of vertex J
C       VERTS(2,J) = Y coordinate of vertex J
C       These X, Y values must be in the ranges specified by the
C       arguments XLO, XHI, YLO, YHI described below.
C
C I*4 NVERTS must be .GE. 3
C
C I*4 ISVP(I,K), I=1...2,K=1...NSIDES are SIDE-TO-VECTOR pointers
C       ISVP(1,K) = J index in verts of one end of side K ISVP(2,K) = J
C       index in verts of other end of side K Each ISVP value must be .GE.
C       1 and .LE. NVERTS
C
C I*4 NSIDES must be .GE. 3
C
C I*4 ITSP(I,L),I=1...3,L=1...NTRIS are TRIANGLE-TO-SIDE pointers.
C       Each value must be .GE. 1 and .LE. NSIDES. This argument is only
C       used for triangle numbering.  If triangle numbering is
C       suppressed (HTN=0.0) then ITSP and NTRIS are ignored.
C
C I*4 NTRIS must be .GE. 1
C
C I*4 IFXSD(1...NFXSD) are K indices in ISVP of sides which must not
C       be swapped. Each value must be in the range 1...NSIDES
C
C I*4 NFXSD must be .GE. 0,  and .LE. NSIDES
C
C R*4 CUT(I,J),I=1..2,J=1..NCUT are the X, Y coordinates of cutting
C       points
C
C I*4 NCUT is number of cutting points
C
C L*4 TFINIT is true if program is to initialize the plotter and the
```

```
C       output file and to close the output file.  Otherwise the user does
C       these things.
C
C C*(*) FILE is the name of the output HSR file, without an extension.
C       The extension ".HSR" is applied by the program.  This argument is
C       ignored if TFINIT is false.
C C*(*) TITLE is a character title for the plot
C
C R*4 XMAX, YMAX are the window for the plotter in inches
C       Both must be .GE. 0.0
C       XMAX must be .LT. 60.0 (five feet)
C       YMAX must be .LT. 24.0 (two feet)
C
C R*4 XPLO, XPHI are the low and high values of the graph to be located
C       inside of plotter window in inches
C       XPLO must be .GE. 0.0
C       XPHI must be .LT. XMAX
C
C R*4 YPLO, YPHI are the low and high values of the graph to be located
C       inside of plotter window in inches
C       YPLO must be .GE. 0.0
C       YPHI must be .GT. YMAX
C
C R*4 XMAX, YMAXare the window for the plotter in inches
C       Both must be .GE. 0.0
C       XMAX must be .LT. 60.0 (five feet)
C       YMAX must be .LT. 24.0 (two feet)
C
C R*4 XPLO, XPHI are the low and high values of the graph to be located
C       inside of plotter window in inches
C       XPLO must be .GE. 0.0
C       XPHI must be .LT. XMAX
C
C R*4 YPLO, YPHI are the low and high values of the graph to be located
C       inside of plotter window in inches
C       YPLO must be .GE. 0.0
```

```
C       YPHI must be .LT. YMAX
C
C R*4 XLO,XHI are vertex coordinate units to correspond to the graph
C       window XPLO, XPHI
C       XHI must be .GT. XLO
C
C R*4 YLO, YHI are vertex coordinate units to correspond to the graph
C       window YPLO, YPHI
C       YHI must be .GT. YLO
C
C R*4 THKVis dot thickness in inches for plotting vertices
C       must be .GE. zero, and .LE. 1.0
C
C R*4 THKS is line thickness in inches for plotting ordinary sides of
C       triangles.
C       must be .GE. zero, and .LE. 1.0
C       if equal zero, sides are not drawn
C
C R*4 THKBS is line thickness in inches for plotting the boundary sides
C       of the triangulation.
C       must be .GE. zero, and .LE. 1.0
C
C R*4 THKFS is line thickness in inches for plotting the fixed sides of
C       the triangulation.
C       must be .GE. zero, and .LE. 1.0
C
C R*4 THKGW is line thickness in inches for plotting the boundary of
C       the graph window.
C       must be .GE. zero, and .LE. 1.0
C       if equal zero, graph boundary is not drawn
C
C R*4 HVN is height in inches for printing vertex numbers
C       must be .GE. zero, and .LE. 1.0
C       if equal zero, vertex numbering is suppressed
C
C R*4 HSN is height in inches for printing side numbers
```

```
C       must be .GE. zero, and .LE. 1.0
C       if equal zero, side numbering is suppressed
C
C R*4 HTN is height in inches for printing triangle numbers
C       must be .GE. zero, and .LE. 1.0
C       if equal zero, triangle numbering is suppressed
C
C I*4  BUF is working buffer for finding all boundary sides. This buffer
C       is only needed when the thickness of boundary sides differ from
C       the thickness of interior sides.
C
C OUTPUTS:
C The output file is created, or added to, according to TFINIT.
C-------------------------------------------------------------
```

19.File-name: PMAP.FOR

```
C-------------------------------------------------------------
C WRITE_PROFILE: Create a "PRF" file to write a triangulated curve
C
C       This routine will write the data of a triangulated curve to a
C "PRF" file with the same file name as the database file
C-------------------------------------------------------------
C                                Usage
C CALL  WRITE_PROFILE ([*SNER,] ID, NPT, POINT, TFS, TFE, NAME,
C                        FNAME)
C
C where SNER is the statement number for error return
C
C INPUT
C I*4  ID is the index of the domain to be written.
C I*4  NPT is number of points of polyline.
C R*4 POINT(2,NPT)  are X- and Y-coordinates of those points.
C L    TFS indicates the first line segment extends to infinite when it
C       is true
C L    TFE indicates the last line segment extends to infinite when it
C       is true
```

```
C CHAR*(*) NAME is the active domain name
C
C OUTPUT
C CHAR*(*) FNAME is the file name where we wrote it
C-----------------------------------------------------------------
```

20.File-name: PROFILE.FOR

```
C-----------------------------------------------------------------
C XYCRV_TRICRV: Convert a x-y curve to a triangulated curve
C
C       This routine will convert polyline in x-y space to a curve lines
C in a triangulated surface domain.  It calls subroutine "Get_Profile" to
C convert the polyline one line segment at a time.
C       The result of this triangulated curve will store in blank common
C area where address starts from NV_ORG(ACTIVE).  NV_LEN(ACTIVE)
C stores the total number of points of this triangulated curve. For each
C point of this new curve has four 4-byte memory space to keep the
C information associated with it and those are type, index, u1 and u2.
C 1. Type is an integer from 0 to 2
C       If point is on a vertex, then type is equal to 0
C       If point is on a side, then type is equal to 1
C       If point is in a triangle, then type is equal to 2
C 2. Index indicates the point is on which vertex or side or in which
C       triangle
C 3. u1 and u2 are a pair of parameters to indicate the relative position
C       of the point to side or triangle.
C       If type = 0 then u1 and u2 are both equal to 0.0
C       If type = 1 then u1 is the ratio of the length from the point to
C       the origin of the side to the length of side, where 0.0 < u1 < 1.0
C       and u2 = 0.0
C       If type = 2 then the u1 and u2 represent two coefficients so that
C       the linear combination of vectors (V1,V2) and (V1,V3) are equal
C       to a single vector (V1,V).  Where V1, V2 and V3 are vertices of
C       the triangle and V is the point of triangulated curve, and (Vi,Vj)
C       means vector from Vi to Vj.  Note that
C               0.0 < u1 < 1.0 and
```

```
C              0.0 < u2 < 1.0 and
C              0.0 < u1 + u2 < 1.0
C-----------------------------------------------------------------
C                                Usage
C
C CALL XYCRV_TRICRV ([*SNER,] ACTIVE, NPT, PT, TFSINF, TFEINF)
C
C where SNER is statement number for error return
C
C INPUT
C I*4  ACTIVE is the index of working domain
C I*4  NPT is number of points of the polyline.  It is .GE. 2
C R*4 PT is a list of X- and Y-coordinates of points
C L    TFSINF indicates the first line segment extend to infinite when
C      it is true
C L    TFEINF indicates the last line segment extend to infinite when
C      it is true
C-----------------------------------------------------------------
```

21.File-name: PROFILE.FOR

```
C-----------------------------------------------------------------
C GET_PROFILE: Convert a line segment in x-y space into triangulated
C      space
C
C      This is a special routine called by subroutine "XYCRV_TRICRV".
C routine will scan the surface domain alomg the finite line segment
C and convert the xy line into triangulated line
C-----------------------------------------------------------------
C                                Usage
C
C CALL GET_PROFILE ([*SNER] ACTIVE, X1, Y1, X2, Y2, EPS)
C
C where SNER is statement number for error return
C
C INPUT
C I*4  ACTIVE is the index of domain to be cut
```

```
C R*4 X1, Y1 is an end point of finite line segment
C R*4 X2, Y2 is another end point of finite line segment
C R*4 EPS is the epsilon used in routine
C-------------------------------------------------------------
```

22.File-name: PROFLIB.FOR

```
C-------------------------------------------------------------
C GET_INTERSECTIONS: Find intersections between boundary and a line
C
C     This is a special routine called by GET_PROFILE to find all
C intersection points between boundary sides and a finite line, but
C doesn't insert new intersection points into the database domain
C-------------------------------------------------------------
C                              Usage
C
C CALL GET_INTERSECTIONS ([*SNER,] POINT, WB, NUM_SEC, B_SEC_XY,
C                          START, CLOCKWISE, COUNT_CLOCK)
C
C where SNER is statement number of error return
C     The first error statement number is used for any kind of error
C     except "NO INTERSECTION FOUND"
C     The second error statement number is only used when no
C     intersection found by this routine
C
C INPUT
C R*4 POINT(2,2) is end points of cutting line
C I*4  WB is the index of working boundary
C
C OUTPUT
C I*4 NUM_SEC is number of intersections between boundary and
C     cutting line
C I*4 B_SEC(2,*) is the type and index of the intersections
C R*4 B_SEC_XY(2,*)  is X- and Y-coordinates of intersection vertices
C I*4 START is an intersection vertex with largest X-value among all
C     intersection points.
C I*4 CLOCKWISE is the next intersection vertex following START in
```

```
C       clockwise direction
C I*4 COUNT_CLOCK  is the previous intersection vertex before START
C       in counter-clockwise direction
C-------------------------------------------------------------
```

23.File-name: PROFLIB.FOR

```
C-------------------------------------------------------------
C GET_NEXT_POINT: To find the next crossing point along a line.
C
C       This is a special routine used by subroutine "GET_PROFILE".
C handler.  Given the location of last cut point, routine will find the
C next cutting point no matter what the point is interior of the
C endpoints of a side or an existing vertex.  Note that this routine
C won't insert the new point into surface domain.
C-------------------------------------------------------------
C                                 Usage
C
C CALL GET_NEXT_POINT ([*SNER,] CLINE, LPT, LTYPE, LINDEX, LXY,
C                    MAXV, VLIST, SLIST, TLIST, KT, NTYPE, NINDEX, X, Y)
C
C where SNER is statement number for error return and VLIST,
C SLIST and TLIST are working buffers.
C
C INPUT
C R*4 CLINE(2,2) defines the endpoint of current cutting line segment
C I*4 LPT(2) gives you the information of the ending of cut line.
C       LPT(1) is defined the same as LTYPE and NTYPE, and LPT(2) is
C       the index where the last point in.
C I*4 LTYPE indicates the location of previous cutting point is on an
C       existing vertex, if LTYPE is equal to zero, on a side if LTYPE is
C       equal to one.
C I*4 LINDEX is the index of a vertex or a side where the last
C       cutting point located.
C R*4 LXY(2)is the X- and Y-coordinates where last cutting point
C       located.
C I*4 MAXVis maximum number of vertices in working domain.
```

```
C I*4 KT is the index of a triangle where the last triangle passed
C       through by cutting, it will be updated during this routine.
C
C OUTPUT
C I*4 NTYPE indicates the location of next cutting point is on an
C       existing vertex, if NTYPE is equal to zero, on a side if NTYPE is
C       equal to one.
C I*4 NINDEX is the index of a vertex or a side where the next
C       cutting point located.
C R*4 X, Y are X- and Y- coordinates of next cutting point.
C-------------------------------------------------------------
```

24.File-name: REMS.FOR

```
C-------------------------------------------------------------
C BOUNDARY_SIDE: Check a given side is on a boundary or not.
C-------------------------------------------------------------
C                              Usage
C
C CALL BOUNDARY_SIDE ([*SNER,] IS, IB)
C
C where SNER is a statement number for error return
C
C INPUT
C I*4 IS is the side index of the given side.
C
C OUTPUT
C I*4 IB  is boundary index where the given side is on, otherwise IB is
C       zero.
C-------------------------------------------------------------
```

25.File-name: REMS.FOR

```
C-------------------------------------------------------------
C REMOVE_SIDE_ENTRY: Remove the entry of sides from SVP list
C-------------------------------------------------------------
C                              Usage
C
```

```
C CALL REMOVE_SIDE_ENTRY ([*SNER,] LEN, LIST)
C
C where SNER is a statement number for error return
C
C INPUT
C I*4  LEN is number of sides to be removed
C I*4  LIST(I=1,LEN)  is a list of side indices to be removed
C
C Information about SVP and TSP block, side function value blocks
C and missing side function value flag blocks will be updated.
C-------------------------------------------------------------
```

26.File-name: REMS.FOR

```
C-------------------------------------------------------------
C COPY_DOMAIN: will duplicate a given domain in memory.
C
C       The duplicated one has its own domain name and domain index
C       and memory spaces
C-------------------------------------------------------------
C                                Usage
C
C CALL COPY_DOMAIN ([*SNER,] ORGD, NEWD)
C
C where SNER is the statement number for error return
C
C INPUT
C I*4  ORGD is the domain index to be copied
C
C OUTPUT
C I*4  NEWD is the domain index of the new copied domain
C-------------------------------------------------------------
```

27.File-name: STRSORT.FOR

```
C-------------------------------------------------------------
C STR_SORT_INT: Sort a list of integers
C
```

```
C       This is a sorting routine to sort a list of integers in ascending
C order by "STRAIGHT-INSERTION" algorithm.
C-------------------------------------------------------------
C                              Usage
C
C CALL STR_SORT_INT (LEN, LIST)
C
C INPUT
C I*4  LEN is the length of the integer list to be sorted.
C I*4  LIST(I=1,LEN)  is a list of integer to be sorted.
C
C OUTPUT
C I*4  LIST(I=1,LEN)  is a list of integers in ascending order.
C-------------------------------------------------------------
```

28.File-name: STRSORT.FOR

```
C-------------------------------------------------------------
C BIN_SEARCH: is a binary search routine for a list of integers.
C-------------------------------------------------------------
C                              Usage
C
C CALL BIN_SEARCH (KEY, LEN, LIST, INDEX)
C
C INPUT
C I*4  KEY is the given integer to be searched.
C I*4  LEN is the length of integer list.
C I*4  LIST(I=1,LEN)  is a list of sorted integers in ascending order
C       where searching is performed.
C
C OUTPUT
C I*4  INDEX is the index number of list where LIST(INDEX) = KEY.  If
C       not found, INDEX will return -999.
C-------------------------------------------------------------
```

29.File-name: STRSORT.FOR

```
C-------------------------------------------------------------
```

```
C SET_DOMAIN_NAME: Set new surface domain name by default
C
C       This routine will set domain name of the new domain which was
C cut out off a given domain. The new domain name is the same as the
C original domain name except the version number. If program knew
C the version number of original domain, the version number will
C increased by one from that number. If we don't know the version
C number, then we will start from version 100.
C-------------------------------------------------------------
C                              Usage
C
C CALL SET_DOMAIN_NAME (OLD, NUM_ND, NEW)
C
C INPUT
C CHAR OLD*(*) is the name of original domain.
C I*4 NUM_ND indicates how many new domain have been cut out from
C       this original domain.
C
C OUTPUT
C CHAR NEW*(*) is the name of new domain.
C-------------------------------------------------------------
```

30.File-name: TILEF.FOR

```
C-------------------------------------------------------------
C TILEF_PROFILE: Get the profile of a vertex function along a
C       triangulated curve.
C
C-------------------------------------------------------------
C                              Usage
C CALL TILEF_PROFILE ([*SNER,] IXVF)
C
C where SNER is the statement number for error return
C
C INPUT
C I*4 IXVF is the index number of a vertex function which to be
C       projected.
```

```
C
C OUTPUT
C The output will store in common area start from address
C RMEM(OV_ORG(ACTIVE_DOMAIN)) with length of 3*OV_LEN
C (ACTIVE_DOMAIN). Each point has three values, the first two
C values are X and Y coordinates, the third one is the function value
C of POINT(X,Y).
C-------------------------------------------------------------
```

31.File-name: TILEF.FOR

```
C-------------------------------------------------------------
C WRITE_TILE: Create a TLP file to write the profile of a triangulated
C      curve
C
C-------------------------------------------------------------
C                              Usage
C
C CALL WRITE_TILE ([*SNER,] ID, NAME, FNAME)
C
C where SNER is the statement number for error return
C
C INPUT
C I*4  ID is the index of domain to be written.
C CHAR*(*) NAME is the active domain name.
C
C OUTPUT
C CHAR*(*) FNAMEis the file name where we wrote it.
C-------------------------------------------------------------
```

32.File-name: TRIPT.FOR

```
C-------------------------------------------------------------
C XYPT_SIDEPT: Convert a x-y point to the side coordinate (u1)
C
C      This routine convert a point V on a directed side S to the ratio
C of length from V to the origin of S to the length of S
C-------------------------------------------------------------
```

```
C                                Usage
C
C CALL XYPT_SIDEPT (X, Y, XY, U1)
C
C INPUT
C R*4 X, Y are the X- and Y- coordinates of the point.
C R*4 XY(2,2) are the X- and Y-coordiantes of the endpoint of side S.
C
C OUTPUT
C R*4 U1 is side-coordinate of point (X,Y) on side S.
C-----------------------------------------------------------------
```

33.File-name: TRIPT.FOR

```
C-----------------------------------------------------------------
C SIDEPT_XYPT: Convert side coordinate (u1) of a point to its x and y
C
C      Converts a side-coordinated point (U1) of a given side S to its
C X-Y coordinate.
C-----------------------------------------------------------------
C                                Usage
C
C CALL SIDEPT_XYPT ([*SNER,] XY, U1, X, Y)
C
C where SNER is the statement number for error return
C
C INPUT
C R*4 XY(2,2) are the X- and Y-coordinates of the endpoint of side S.
C R*4 U1 is side-coordinate of point (X,Y) on side S.
C
C OUTPUT
C R*4 X, Y are the X- and Y-coordinates of the point.
C-----------------------------------------------------------------
```

34.File-name: TRIPT.FOR

```
C-----------------------------------------------------------------
C XYPT_TRIPT: Converts a point (X,Y) to triangle-coordinate (u1,u2)
```

```
C
C       This routine convert a x-y point V inside of a triangle T to the
C  coordinate system (u1,u2) of triangle T
C-------------------------------------------------------------------
C                                  Usage
C
C  CALL XYPT_TRIPT ([*SNER,] X, Y, XY, U1, U2)
C
C  where SNER is the statement number for error return
C
C  INPUT
C  R*4 X, Y are the X- and Y-coordinates of a given point V.
C  R*4 XY(2,3) are X- and Y-coordinates of three endpoints (V1, V2 and
C       V3) of the given triangle.
C
C  OUTPUT
C  R*4 U1, U2 are coordinates of point V(X,Y) associated with the
C       given triangle
C       (V - V1) = U1*(V2 - V1) + U3*(V3 - V1)
C       0 <= U1 < 1.0  and  0 <= U2 < 1.0  and
C       0 < U1+U2 < 1.0
C-------------------------------------------------------------------
```

35.File-name: TRIPT.FOR

```
C-------------------------------------------------------------------
C  TRIPT_XYPT: Convert a triangulated point (U1,U2) to its x, y values
C-------------------------------------------------------------------
C                                  Usage
C
C  CALL TRIPT_XYPT ([*SNER,] XY, U1, U2, X, Y)
C
C  where SNER is the statement number for error return
C
C  INPUT
C  R*4 XY(2,3) are X- and Y-coordinates of three endpoints (V1, V2
C       and V3) of the given triangle.
```

```
C R*4 U1, U2 are coordinates of point V(X,Y) associated with the
C       given triangle
C       (V - V1) = U1*(V2 - V1) + U3*(V3 - V1)
C       0 <= U1 < 1.0  and  0 <= U2 < 1.0  and
C       0 < U1+U2 < 1.0
C
C OUTPUT
C R*4 X, Y are the X- and Y- coordinates of a given point V.
C-------------------------------------------------------------
```

36.File-name: VMAP.FOR

```
C-------------------------------------------------------------
C WRITE_VERTEX_MAP: Create a "VMP" file to store the vertex mapping
C       tables
C
C       Writes vertices mapping data between the new surface domain
C and the original domain to a file after cut operation.  The file
C specification of this mapping file will be the same as domain name
C except the extension.  The extension of mapping file is ".VMP"
C-------------------------------------------------------------
C                                Usage
C CALL WRITE_VERTEX_MAP ([*SNER,] ID, NAME, FNAME)
C
C where SNER is the statement number for error return
C
C INPUT
C I*4  ID is the index of domain to be written.
C CHAR*(*) NAME is the active domain name.
C
C OUTPUT
C CHAR*(*) FNAME is the file name where we wrote it.
C-------------------------------------------------------------
```

37.File-name: XYCRV.FOR

```
C-------------------------------------------------------------
C TRICRV_XYCRV: Convert a triangulated curve back to X-Y curve
```

```
C
C      This routine convert a triangulated curve to its corresponding
C curve in x-y coordinate system
C-----------------------------------------------------------------
C                              Usage
C
C CALL TRICRV_XYCRV ([*SNER,] DOMAIN, NPT, XY)
C
C where SNER is the statement number for error return
C
C INPUT
C I*4  DOMAIN is the index of the working domain.
C
C OUTPUT
C I*4  NPT is number of points in X-Y curve.
C R*4 XY(2,NPT) is the X- and Y-coordinates of each point.
C-----------------------------------------------------------------
```

38.File-name: XYCRV.FOR

```
C-----------------------------------------------------------------
C CONVERT_POINT: Convert a triangulated point to its X-Y values
C
C      This routine convert a point in triangulated system to its
C corresponding coordinates in x-y system
C-----------------------------------------------------------------
C                              Usage
C
C CALL CONVERT_POINT (ITYPE, INDEX, U1, U2, X, Y)
C
C INPUT
C I*4  ITYPE is the type of the point in triangulated system.
C      If ITYPE = 0, then it is a vertex.
C      If ITYPE = 1, then it is a side.
C      If ITYPE = 2, then it is a triangle.
C I*4  INDEX is the index of a vertex, side or triangle.
C R*4 U1, U2 is the coefficient of the point in triangulated system.
```

```
C
C OUTPUT
C R*4 X, Y are X-Y value of the point.
C-------------------------------------------------------------
```