National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

University of Alberta

Identification and Reconstruction of 3D Objects from Gray Value Data

by

Xiaoqing Qu ©

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Doctor of Philosophy

Department of Computing Science

Edmonton, Alberta
Fall 1993

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file   Votre référence

Our file   Notre référence

ISBN 0-315-88222-0

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Xiaoqing Qu
TITLE OF THESIS: Identification and Reconstruction of 3D Objects from Gray Value Data

DEGREE: Doctor of Philosophy
YEAR THIS DEGREE GRANTED: 1993

Permission is hereby granted to UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis not extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) . Xiaoqing Qu .
Permanent Address:
2105 47 Street,
Edmonton,
Canada T6L 3H9

Date: Sep. 20th, 1993

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Identification and Reconstruction of 3D Objects from Gray Value Data** submitted by **Xiaoqing Qu** in partial fulfillment of the requirements for the degree of Doctor of Philosophy .

Wayne A. Davis
Dr. W.A. Davis

Xiaobo Li
Dr. X. Li

Dr. A. Basu

Z.J. Koles
Dr. Z.J. Koles

Dr. M. Green

Wm. Armstrong for
Dr. S.W. Zucker

Date: Sep. 16, 93

Abstract

This thesis is concerned with the conversion of volumetric data to a surface model for display purposes. Previous attempts at this problem either did not convert the volumetric data to a surface model or did it with a conversion routine that has certain limitations. The thesis introduces a more general surface model for the conversion of the volumetric data structure.

The thesis presents an extended cuberille model, a 3D border identification method, a surface tracking algorithm, and a surface closure algorithm for the identification, reconstruction, and display of 3D objects from 3D gray value data. The identification starts with 3D gradient data, and concludes with a surface description of an identified object. The surface can then be displayed by any graphics package.

3D edge elements are gradients, and the orientations of the gradients are quantized to 26 directions. The edge elements are then converted to the extended cuberille model. The extended cuberille model has four volume primitives. Besides a cube, voxels are extended to include three other polyhedra so that each voxel has a face whose orientation is compatible with one of the 26 gradient orientations. This face is termed a face primitive. There are also four types of external voxel faces, hence a surface description in the extended cuberille model consists of the four types of voxel faces.

The three dimensional border identification method is based on the sign of the second derivative of intensity change. For a bright object surrounded by a darker background, the condition for a voxel being on a border is that the second derivative is negative and changes sign for neighbors in the gradient direction. There exists exactly one layer of border voxels that satisfy the condition. This avoids a multiple layer problem. Tracking border voxels could simply be a breadth first search.

The surface tracking algorithm consists of three algorithms: the border face tracking algorithm, face connection testing, and the surface closure algorithm.

The border face tracking algorithm traverses outways instead of 26 face, edge or vertex connected voxels. Adjacent voxels of each outway of four types of face primitives are defined according to a face orientation and the border voxel condition. The definition results in less than 26 adjacent voxels. The border face tracking algorithm is therefore faster than a breadth first search.

Because a voxel is converted to one face, quite few border voxel faces are not connected. During border face tracking, the information of disconnected faces is saved. The surface closure algorithm will use this information to close a surface with four types of external voxel faces in the extended cuberille model.

All algorithms are implemented and tested. Experimental results of 3D identification, border face tracking, and surface closure by the extended cuberille model on test data and medical data are given.

Acknowledgements

I would like to thank my supervisor Dr. W. Davis for his guidance and support throughout this research, and specially for his great patience of correcting the thesis.

I am grateful to my co-supervisor Dr. X. Li for his advise and help over the last few months. I would also like to thank the members of my committee for carefully reading of the thesis and making positive comments.

A special thanks goes to Dr. M. Green for his kindness to allow me to use the Graphics facilities. Appreciation should also be expressed for the financial assistance of the Department of Computing Science, since it would otherwise be impossible to complete the research.

Special thanks also goes to Edith Drummond for her patience, assistance and thoughtfulness throughout the program.

Contents

List of Figures

Chapter 1

Introduction

This thesis is concerned with the conversion of volumetric data to a surface model for display purposes. Previous attempts at this problem either did not convert the volumetric data to a surface model or did it with a conversion routine that has certain limitations. The thesis introduces a more general surface model for the conversion of the volumetric data structure.

The thesis presents an extended cuberille model, a 3D border identification method, a surface tracking algorithm, and a surface closure algorithm for the identification, reconstruction, and display of 3D objects from 3D gray value data. The identification starts with 3D gradient data, and concludes with a surface description of an identified object. The surface can then be displayed by any graphics package.

1.1 The Extended Cuberille Model

In 3D identification and display, thresholding is widely used but restricted. In a variety of applications, not many objects can be identified by simple thresholding. Even the optimal thresholding [GW87] also has some problems. Identification based on 3D edge detection is a more general method. Some 3D edge operators have been proposed to detect edge elements [Liu77, ZH79, MR81] using gradients.

The problem of how to group detected edge elements together to reconstruct an integral object has not been discussed. Integral object representation is a relatively new topic [Man88]. It implies that if an object is represented by its surface, the surface must be closed without missing faces. If the object is represented by a collection of voxels, each voxel is a solid with a thickness so that the space occupied by the object can be measured.

A 3D Edge element detected by a gradient operator is usually described by its location (the center or a corner coordinates), magnitudes (steepness), and orientation (normal). This description has little geometrical information because the edge's shape and size are undefined. As a result, they cannot be used to construct a surface. What is needed are modeling primitives whose shape and size are defined and whose orientations are close to their edge counterparts, i.e., a **model**. For instance, a modeling primitive in a polyhedra surface is a face defined by a list of vertices. Faces are connected through edges to form a closed a surface. A modeling primitive in a cubic B-spline surface is a patch defined by its control net. Patches are connected to form a surface having curvature continuity across edges.

Currently available models in 3D surface identification are: cuberille model [HL79] and marching cubes [LC87]. Both are polyhedra models and both use thresholding to identify objects.

In the cuberille model, the face primitive is a square voxel face. Algorithms tracking a surface of an object in a binary image have been seen in [AFH81] and [GU89]. An object can also be represented as a collection of cube shaped voxels. Volume rendering algorithms have been seen in [Rey85b, FGR85, Rey87a]. Because of the regularity of voxels, an object can be organized as an octree. Octree related algorithms include tree generation algorithms [Sam81, Sam80, YS83], set operation algorithms [HS79], geometric transformation algorithms [JT80, Mea82b], and display algorithms [DT84, Mea82a, ZD91].

The face primitives in the marching cubes [LC87] are triangles. The average normal of the triangles close to a voxel reflects the gradient orientation at that voxel. In marching cubes algorithm, data are divided to cubes. A cube is made up of eight voxels in eight vertices, that are either inside or outside as determined by thresholding. The algorithm marches cube by cube, creating triangle faces to model a piece of surface within each cube. When the algorithm stops, a surface of triangles is completed. The surface is displayed using traditional graphics techniques.

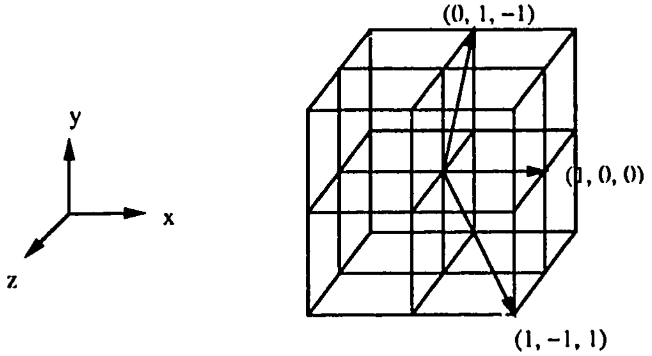
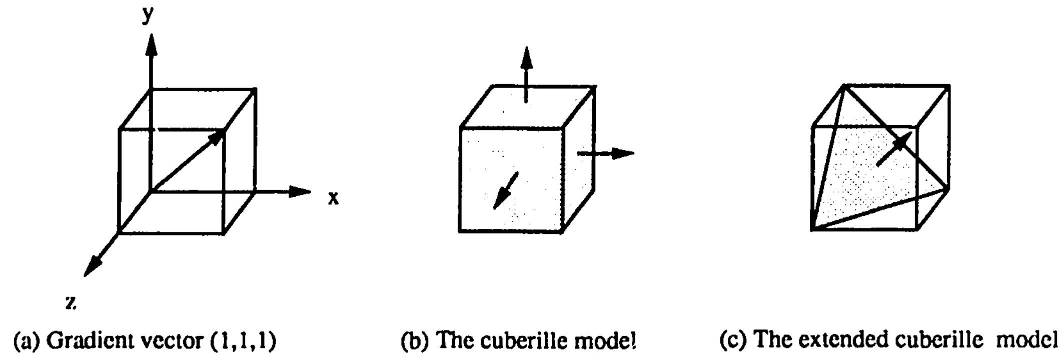


Figure 1.1: Three of the 26 gradient directions.

The following presents the motivation to extend the cuberille model for 3D edge detection and surface construction. Suppose a 3D edge operator, such as the one by [MR81], is applied to gray value data. It detects edges by computing a gradient at each voxel. The gradient at a voxel can be written as a vector $\nabla = (\nabla_x, \nabla_y, \nabla_z)$ with the three components indicating intensity changes along three principal axes. The magnitude of the gradient, approximated by $|\nabla| = |\nabla_x| + |\nabla_y| + |\nabla_z|$, indicates the possibility that the voxel is an edge element. The larger the magnitude, the more likely that it is an edge voxel. The direction of the gradient vector determines the edge orientation. For simplicity, the direction of a gradient is quantized to one of 26 directions (see Fig. 1.1.)

Suppose an edge is detected at a voxel with orientation (1,1,1), as shown in Fig. 1.2(a). The cuberille model would represent the edge by three voxel faces whose average normal is (1,1,1). It is more natural, however, to represent the edge by a face whose normal coincides with the edge orientation. For example, the triangle face in Fig. 1.2(c) would construct a smoother surface than the three voxel faces in Fig. 1.2(b). Of course, another face with the same normal could also be chosen as a face primitive. The problem therefore is to choose a set of modeling primitives whose normals reflect the 26 gradient directions. To solve the problem the **Extended Cuberille** model is introduced in Chapter 3.

The extended cuberille model has four volume primitives (see page 37, Fig. 3.1). Besides a cube, voxels are extended to include three other polyhedra so that each voxel has a face whose orientation is compatible with one of the 26 gradient vectors. This face is called a face primitive. Hence a volume primitive can be referred to as either a voxel or a face. The merits of the three representation schemes: space occupancy enumeration, octree, and surface representation by the extended cuberille model are also briefly discussed in Chapter 3.



(a) Gradient vector (1,1,1) (b) The cuberille model (c) The extended cuberille model

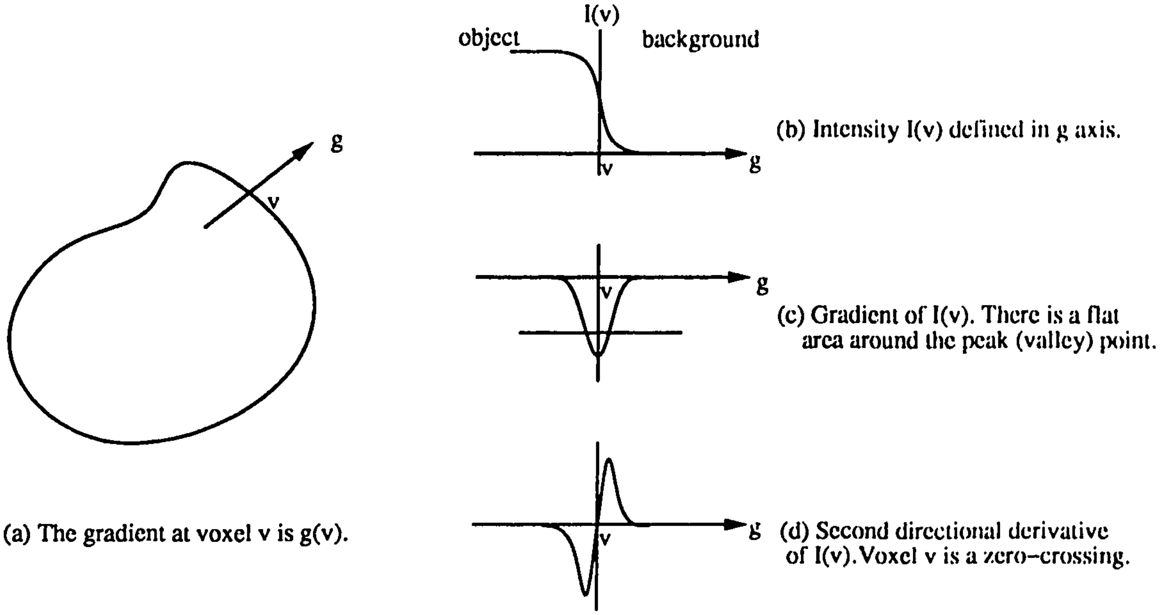
Figure 1.2: Different modeling primitives.

1.2 Three Dimensional Identification

Some results use 3D edge detection to identify a surface. The 3D boundary following algorithm by [CR89] constructs a 3D boundary by stacking 2D boundaries, that have been extracted using a heuristic search algorithm [Mar76]. The heuristic search uses global information to determine boundary elements. To find a boundary is to find a path of minimum cost from a starting edge element to a goal edge element. The sum of the gradient magnitudes along the path could be a cost criterion. The algorithm is basically a 2D method, and the search space is very large.

The method by [SZ87] models the local surface by parameterized patches. The trace points of a patch are thresholded 3D gradient points and further refined using contextual structure. How to connect patches to a closed surface hasn't been discussed.

A problem with identification based on gradient magnitudes is that there is no definition of boundary edges in an edge image. For example, for the step intensity change in Fig. 1.3(b), the detected gradient is shown in Fig. 1.3(c). There is a rather flat area around the peak (valley). All edge elements whose magnitudes are larger than a given threshold are legal boundary edge candidates, because edge operator and magnitude calculation introduce errors. This causes a multiple layer problem. It is difficult to construct a surface from multiple layers of edges.



(a) The gradient at voxel v is g(v).

(b) Intensity I(v) defined in g axis.

(c) Gradient of I(v). There is a flat area around the peak (valley) point.

(d) Second directional derivative of I(v). Voxel v is a zero-crossing.

Figure 1.3: Many edge elements in a gradient image may be chosen as boundary candidates.

Observe in Fig. 1.3(d) that the directional second derivative of an intensity change has a steep slope around a zero-crossing. This directional second derivative corresponds precisely to the peak of the gradient. Marr and Hildreth [MH80] suggested that zero-crossings can be used to detect edge elements. A Gaussian filter is used to smooth the data. Since there is no analytical model for image intensity, the directional second derivative is used for the Gaussian filter. Hence zero-crossings are zero points resulting from convolving the directional second derivative of a Gaussian filter with image intensity. Recent research has been focused on the description of an image by zero-crossings at various scales [Bro88].

A 3D border identification method based on signs of the directional second derivatives of intensity change is proposed in Chapter 4. The condition for a voxel being on a border is given in page 57, i.e. the inequalities (4.26). For a bright object surrounded by a darker background, the condition states that if the second derivative at a voxel is negative and changes sign for neighbors in its gradient direction, the voxel is a border voxel.

To compute the second derivative, a 3D edge operator is applied to compute gradients. The gradient directions are quantized to 26 vectors. Asymmetric Gaussian filters for 3D convolution are designed for the 26 gradient vectors. The long scale of the Gaussian filter coincides with the gradient vector, with short scales in the perpendicular directions. Hence the 3D convolution computation can be speeded up. Although the quantization of gradients to 26 vectors is not smooth, analysis shows that the condition of border voxel identification is not sensitive to quantization errors. From this condition, there exists exactly one layer of border voxels, therefore the multiple layer problem doesn't occur. Tracking border voxels can be as simple as a breadth first search. Compared with the heuristic search [CR89], the search space is greatly reduced.

1.3 The Surface Tracking Algorithm

By the proposed 3D identification method, there exists one layer of border voxels. The breadth-first search algorithm [AHU83] is used to track border voxels. In the algorithm, a gradient image is interpreted as a graph, where nodes are voxels and edges are adjacency relations between pairs of voxels (see Fig. 5.1(d), page 66). A possible way to define the adjacency relation between a pair of voxels is by digital topology [KR89], where two voxels are adjacent each other if they are either face, edge or vertex connected. By the definition, each voxel has 26 adjacent voxels, hence every node has 26 edges in the graph.

The underlying data structure of the algorithm is a queue. The algorithm removes a border voxel from the queue and traverses its 26 adjacent voxels, testing if any satisfies the inequalities (4.26). If so, the adjacent voxel is a border voxel. Meanwhile, if a border voxel is found unmarked, mark it and queue it. The algorithm repeatedly removes border voxels from the queue until the queue is empty.

For a given object, the number of border voxels is determined by the condition (4.26), and is fixed. Since each border voxel is placed in the queue once, and 26 adjacent voxels of every border voxel need to be tested, the time complexity of the algorithm is an order of $O(kn_B)$, where n_B is the number of border voxels of an object, and k is the number of adjacent voxels that a border voxel has, i.e. 26.

In the extended cuberille model, however, every border voxel is converted to a face primitive. Intuitively, a face normal, which is one of the 26, shouldn't have a dramatic change from one border voxel to an adjacent one because the surface is supposed to be smooth. Moreover, only those adjacent voxels on a border are of interest. This suggests that the normal of a face can be used to assist in defining adjacent voxels. In Chapter 5, the adjacent border voxels are defined for each face primitive according to the face orientation and the border voxel conditions (4.26). It results in less than 26 adjacent border voxels for each face.

Since a face primitive is either a square or a triangle, (see page 37 Fig. 3.1), it has at most four edges. There are at most four ways to connect one face to the next, and each possibility crosses an edge. The ways of connection are called outways of a face. In Chapter 5, the set of adjacent voxels is defined for each outway, and it is further split into disjoint subsets. The surface tracking algorithm traverses outways instead of the 26 edge, face, or vertex connected voxels. While traversing an outway, if any adjacent border voxel has been found once a subset is scanned, the traversal breaks and continues to the next outway. This brings the constant k down to about half and therefore speeds up surface tracking.

1.4 The Surface Closure

In the extended cuberille model, a voxel is coverted to one face primitive. As a result, the border voxel faces are not always connected. During surface tracking, the connection between the current face and an adjacent face is tested. If the two faces are not connected, the information about a missing face is saved. The surface closure algorithm uses this information to close a surface with four types of external voxel faces in the extended cuberille model. The algorithm is discussed in Chapter 7.

The surface tracking algorithm therefore consists of three algorithms: the border face tracking algorithm, face connection testing, and the surface closure algorithm. An outline is given in Section 5.1. The algorithms and experimental results are covered in Chapters 5 to 8.

1.5 The Surface Smoothing

Since voxels are very small, the shape and orientation of voxel faces are indiscriminating. But if a zoom-in view is required, the surface appears to be rough. In this circumstance, the surface smoothing algorithm can be used to smooth a surface. The algorithm adjusts a border face normal according to its neighbor face's orientation during border face tracking. The algorithm is given in Chapter 9.

1.6 Overview

Figure 1.4 presents a depiction of the 3D identification and reconstruction process. Topic boxes in the shaded area – the extended cuberille model, border voxel identification, the border face tracking algorithm, border face connection testing, surface closure, and surface smoothing – will be addressed in this thesis. All algorithms have been implemented and tested. An identified and reconstructed surface can be displayed by nearly any graphical package.

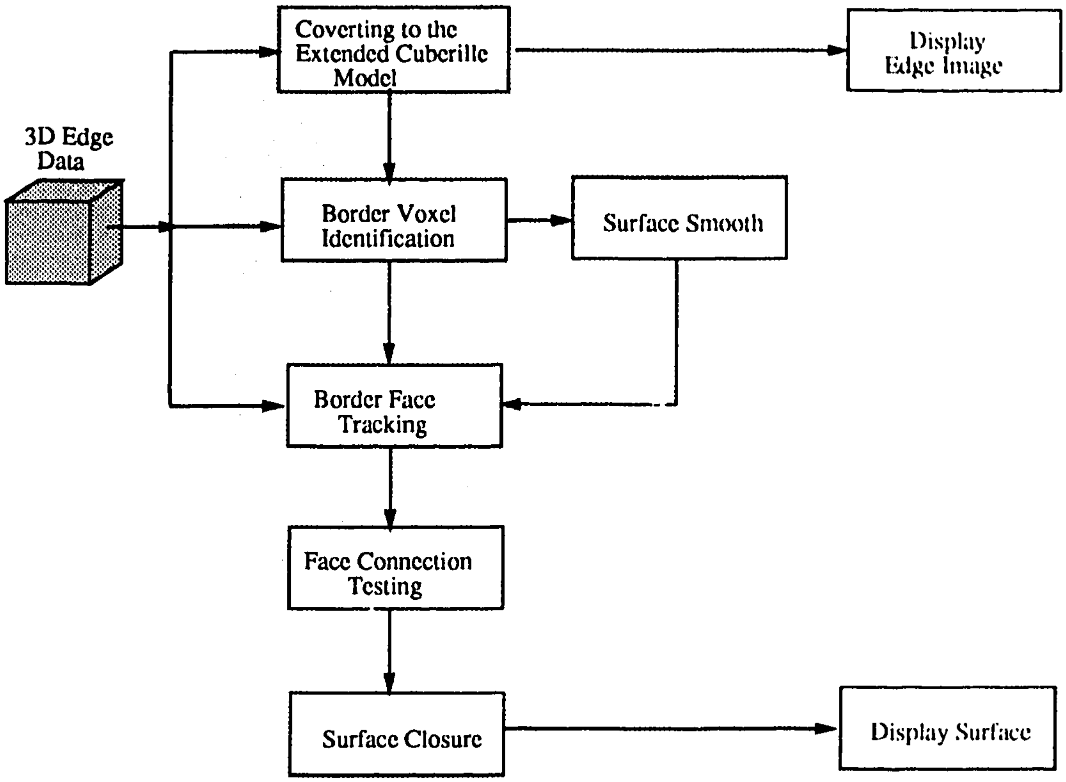


Figure 1.4: The 3D identification and reconstruction process.

The thesis is organized as follows: Chapter 2 is a review of the literature. Since how to guarantee an integral representation is covered in the topic of solid modeling, the mathematical background of solid modeling is given first, then a survey and analysis of the current state-of-the-art for various representation schemes and related algorithms are given. Three 3D edge operators are reviewed next. Specially, the edge detection method by zero-crossings is discussed in detail because it is the starting point for 3D border voxel identification.

Chapter 3 introduces the extented cuberille model that defines four volume primitives. The implementation of converting the gradient to the model by location/direction codes is discussed.

Chapter 4 discusses the border identification method that is based on the sign of the directional second derivative of intensity change. The analysis is first conducted in continuous space. The condition for identifying border voxels in discrete space is then given.

The surface tracking algorithm is covered from Chapters 5 to 8. Chapter 5 discusses the border face tracking algorithm. The method and implementation of face connection testing are discussed in Chapter 6. Chapter 7 discusses the surface closure algorithm. Experiment results are given in Chapter 8.

Chapter 9 discusses the surface smoothing algorithm, and Chapter 10 contains the conclusion.

Chapter 2

Review of the Literature

2.1 Solid Modeling

As depicted in Fig. 2.1, a solid model is a subset of a geometric model. It focuses on creating complete representations of solid objects [Man88]; that is, one representation models only one physical object, hence it permits answers to arbitrary geometric questions such as surface area, volume, etc.

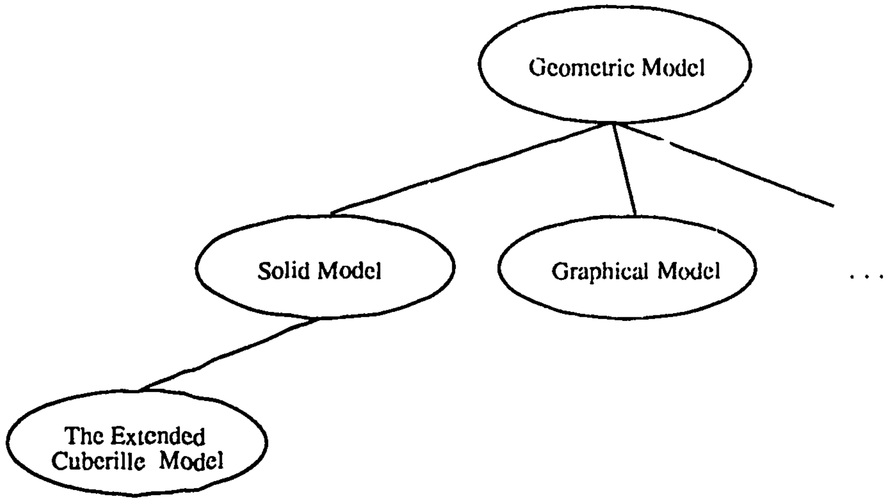


Figure 2.1: The extended cuberille model is a subset of a solid model

The characteristics of **completeness** separates a solid model from a graphical model that does not offer complete shape information. For example, in a wire frame model a representation composed of a collection of lines may correspond to several solids, because some lines might have different interpretations, hence define different surfaces. Since the solids defined by a wire frame representation are not unique, the geometric properties for answering the above questions cannot be calculated automatically. A standard example is given in Fig. 2.2.

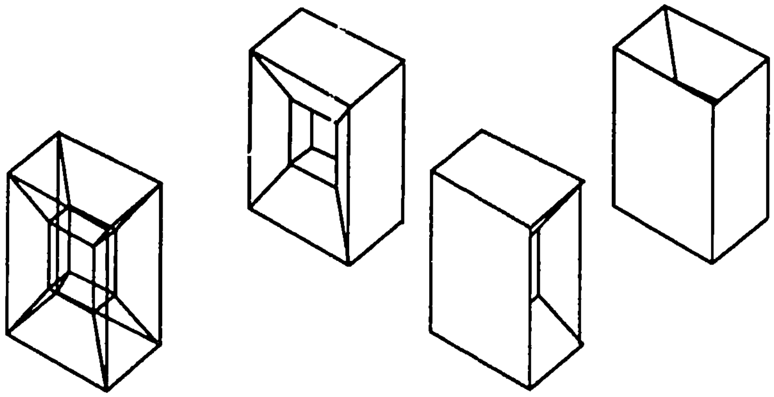


Figure 2.2: An incomplete representation at the left interpreted as any one of the three solids on the right.

To solve this problem a natural step is to upgrade the wire frame model to a polyhedral model so that the hidden part of the surface can be removed. A polyhedral model represents an object by a list of polygons, yet it still cannot guarantee the integrity of a solid. The **integrity** requires that the surface of a solid be closed. It also requires that intersecting polygons not be allowed so that one would be able to tell which side is inside and which side is outside the solid.

Completeness and **integrity** are the main problems faced by solid modeling. A solid model should be able to enforce these conditions automatically so that an incorrect representation is not created. Fortunately, a rigorous mathematical model has been developed to characterize solid objects. In the next section, the mathematical definitions of the three-dimensional **solid** and **regularized** set operations on solid objects are introduced.

2.1.1 Mathematical Model

Starting from the three-dimensional Euclidean space E^3, the abstract entities to model physical objects are subsets of E^3, i.e. sets of points of E^3. But this is too general because only a few of the subsets of E^3 are adequate to model **solid** objects. Intuitively, a solid is closed and bounded. It is also rigid, meaning that its shape is invariant under a rigid transformation, i.e. translation and rotation. A solid must have an interior, and cannot have dangling faces and branches, nor isolated points. A solid must be representable by a finite number of faces. Finally, the surface of a solid is orientable. It is unambiguous which side is inside and which side is outside the solid. The notion of orientable implies that the surface of a solid cannot intersect itself.

Fig. 2.3(a) gives an example of a nonsolid set. The set in Fig. 2.3(a) is closed but not solid because it has a dangling face, a dangling edge, an isolated point and an intersecting face.

The **solid** informally described above can be compactly defined in terms of a point-set topology language as a bounded **regular** set or **r-set**. A regular set is defined in the following definition.

Definition 2.1 *The regularization of a point set A, $r(A)$, is defined by*

$$r(A) = c(i(A)),$$

where $c(A)$ and $i(A)$ denote the closure and interior of A, respectively.

Sets that satisfy $r(A) = A$ are said to be regular.

Informally speaking, the regularization of a set A removes from A all dangling faces, edges, isolated points, then covers it with a tight surface and fills it. Fig. 2.3 shows the process to regularize the nonregular set A given in (a).

Fig. 2.3(b) shows $i(A)$, the interior of A, and 2.3(c) shows $r(A)$, the regularization of A.

Since regularity is widely used to characterize solids, it is also concisely called an r-set.

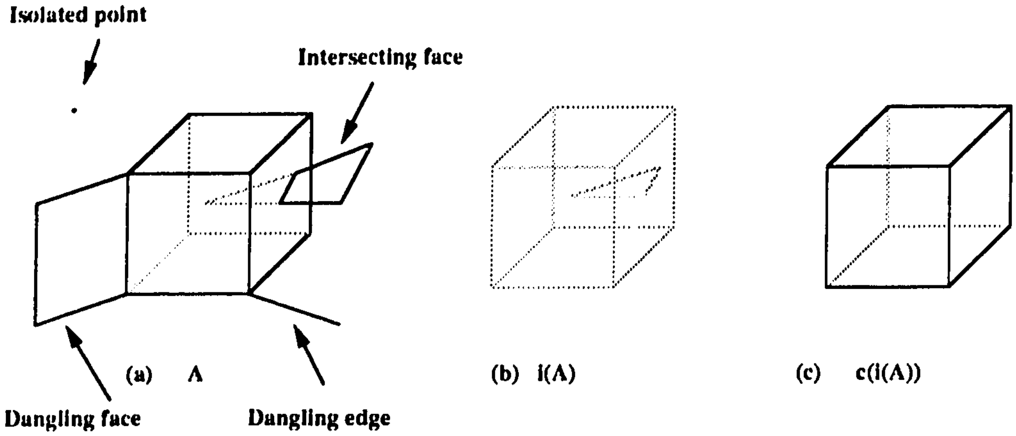


Figure 2.3: Regularize a nonsolid set to an r-set.

Definition 2.2 *A bounded regular set is termed an r-set.*

A regular set need not be connected; it may model two or more solids. It may also have holes.

In conventional Boolean set operations r-sets are not closed; the set operation applied to a solid may not necessarily produce another solid (see Fig. 2.4(b)). To ensure that regularity is closed under set operations, a modified version, called **regularized** set operations, are introduced for r-sets.

Definition 2.3 *The regularized set operations: union, intersection, and set difference, denoted by* $\cup^*$, $\cap^*$, $-^*$, *are defined as*

$$A \cup^* B = c(i(A \cup B))$$

$$A \cap^* B = c(i(A \cap B))$$

$$A -^* B = c(i(A - B))$$

Fig. 2.4(c) gives an example where the regularized intersection of two r-sets is still an r-set.

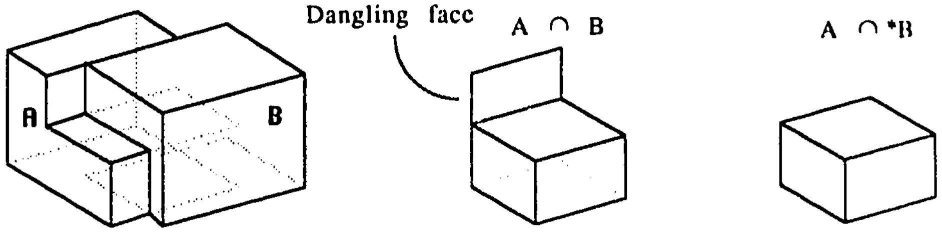


Figure 2.4: Regularized intersection of sets A and B

2.1.2 Representation Schemes

Once the solid objects have been mathematically defined, representation schemes suitable for computer manipulation are needed. There are several methods for representing a solid. For example, a solid may be represented by a 3D enumeration array, or by a surface model, etc. In the next section, various representation schemes are reviewed. Some important properties of representation schemes are expressive power, completeness, conciseness, computational ease and applicability. Not all of these issues are related to this application; however, those properties directly related will be discussed subsequently.

2.2 Various Representation Schemes

There are basically three schemes for representing a 3D object: **spatial occupancy enumeration**, **space subdivision**, and **surface** representations. The properties of various schemes and related algorithms are reviewed.

2.2.1 Spatial Occupancy Enumeration

A spatial occupancy enumeration is a representation that lists all voxels occupied by an object. For example, an object in a 3D binary image is enumerated by the voxels with density value 1. In the literature, the identification of an object from 3D density data is accomplished mostly by **thresholding**, and it can be carried out at display time. Therefore, interest is shown only in display algorithms. As the 3D enumeration array possesses the property of **spatial pre-sortedness** [Mea82b], hidden surface removal can be achieved by back-to-front (BTF) or front-to-back (FTB) readout. This has motivated many 3D display algorithms, the so called **volume rendering** algorithms. Some representatives are:

- back to front,
    recursive BTF [Rey85a],
    slice-by-slice BTF [FGR85],
- front to back,
    slice-by-slice FTB [Far84],
    dynamic screen FTB [Rey87b],
- ray tracing [TT84, Rey85a].

Fig. 2.5 shows the recursive back to front display sequence for the given view point [Rey85a].

In volume rendering algorithms, the modeling primitives are 3D points, i.e. voxels without dimension. Strictly speaking, they cannot be used to model solids. Although

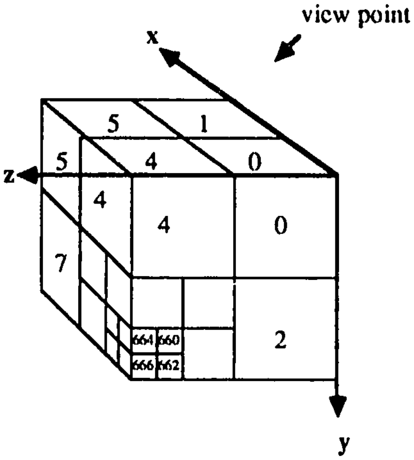


Figure 2.5: The BTF read out sequence for the given view point is 6 4 2 7 0 3 5 1.

display of 3D points is easy because only voxel coordinates are needed, the image has dark holes caused by round off errors in coordinate transformations. Scaling also becomes a problem. As a result, extra steps are suggested to fix holes by scaling down the image or painting a pixel color over a small neighborhood [Rey85a].

The computation is relatively easy because hidden surface removal is simply a back to front readout. The 3D enumerate array, however, is not concise. Since if the view point changes, the back to front sequence also changes. Thus the whole data volume needs to be read out again.

2.2.2 Space Subdivision Schemes

Spatial enumeration arrays are simple, general, and allow the use of a variety of display algorithms. The sizable memory consumption caused by vast data volumes offsets these good points. Observe that in a spatial enumeration array the neighbors of a black voxel are very likely to be black as well. By encoding this coherence information, the space subdivision scheme divides the space adaptively to achieve space saving. A prime example of space subdivision is the **octree** [JT80, Mea82b]. In this section octree and related algorithms are reviewed, together with another scheme, the **linear octree**

[Gar82].

The Octree Representation

The root node of an octree represents the entire space of interest. The space is subdivided in a recursive manner into eight octants, and each octant is represented by one of eight children of the root node. If an octant is fully occupied by the object, the corresponding node is marked **black**; if the octant is empty, the corresponding node is marked **white**. Both black and white nodes are leaf nodes and subdivision of the octant stops. Otherwise the node is marked **gray**, for the octant is partially occupied. It is continuously subdivided into eight octants unless it reaches the minimal resolution (voxel level). An octree encoded object is shown in Fig. 2.6.

Octree related algorithms are listed below. Some of them are quadtree algorithms, but they can be directly extended to octrees.

- **Tree generation** algorithms that create octrees from the data represented by spatial enumeration arrays:

  Raster image to quadtree [Sam81],

  Binary image to quadtree [Sam80],

  Quadtree to octree [YS83].

- **Set operation** algorithms that take two octrees and calculate a new octree resulting from regularized union, intersection, and difference of the two octrees:

  Boolean operations [HS79].

- **Geometric transformation** algorithms that take an octree and calculate the new octree after translating, rotating and scaling:

  90 rotation, scaling and translation [JT80],

  Rotating an arbitrary angle [Mea82b].

- **Display** algorithms that generate the image of an object encoded by an octree:

90 rotation and BTF display [DT84],

FTB display from arbitrary viewpoint [Mea82a],

FTB display [ZD91].

Some of these algorithms are reviewed.

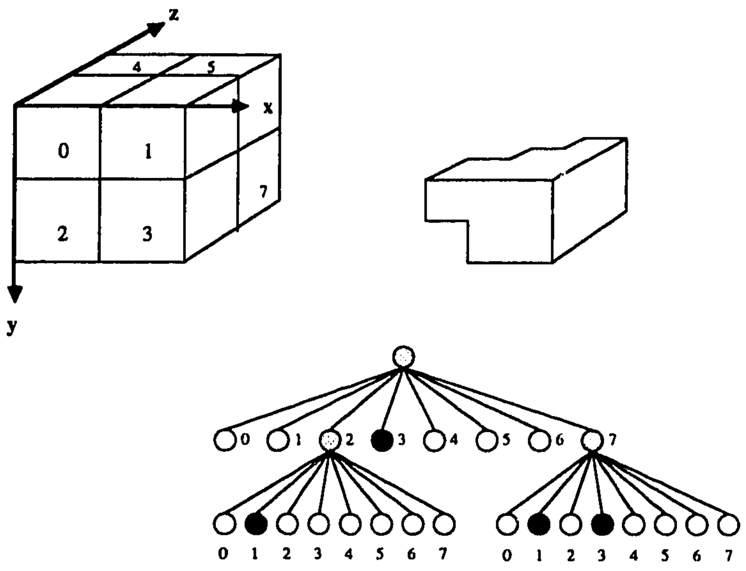


Figure 2.6: An octree encoded object.

Octree Generation [Sam80] The algorithm converts a 3D binary array to an octree. In the algorithm, each node has ten fields: one for the node type, i.e., black, white, or gray, and nine pointers, eight pointing to its sons and one to its father. The voxels are accessed sequentially as the postorder tree traversal and the octree is constructed bottom up. During tree creation each voxel is accessed exactly once, and only maximal nodes are created, that means the created nodes are either gray or black/white leaf nodes so that no further node merges will occur. Hence the algorithm creates a minimum number of tree nodes during tree creation.

Set Operation [HS79] The set operation algorithm is a simple tree traversal. For example, to compute intersection, the two input trees are traversed simultaneously. For the two corresponding nodes n_1 and n_2, if

1. nodes n_1 and n_2 are both leaves, the corresponding node in the output tree is black if both n_1 and n_2 are black; otherwise, it is white;
2. either n_1 or n_2 is a leaf node, and the node is black, the subtree of the nonleaf node is copied to the output tree node; otherwise, the output tree node is white;
3. neither n_1 nor n_2 is a leaf node, the algorithm processes recursively to the next lower level.

The complexity of the algorithm is at most proportional to the size of the smaller tree.

Geometric Transformation [JT80] Rotating an octree encoded space by 90 degrees around an axis is accomplished simply by traversing the octree and permuting the nodes. To scale an octree encoded object by a factor of 2, choose any child node of the current root as the new root and discard its siblings. To scale by a factor of 1/2, create a new root node and link the former root to it as its child, with all its siblings white.

To translate an octree, however, may appear rather difficult. It takes a source tree S, a translation vector $(e,\ f,\ g)$, and creates a target octree T, which represents the translation of S by $(e,\ f,\ g)$. The algorithm traverses two octrees and compares their nodes for overlap in a recursive manner. It has been shown that in the worst case each of the eight children of a target node may be compared to eight source nodes and their 64 children, yielding 576 comparisons. Since at the voxel level no overlap occurs and the maximum number of nodes above the voxel level is bounded by 8^n, where n is the resolution parameter for a $2^n \times 2^n \times 2^n$ array, the number of comparisons is bounded by $567 * N^3$, where $N = 2^n$.

Rotating an octree an arbitrary angle is also difficult, and no details are given here.

FTB Display [Meag82b] The input to the algorithm is an octree encoded object; the output is a quadtree representing a display screen. The projections of an octree node on the display screen are three four-sided polygons, that can be enclosed by a bounding box. The quadtree nodes, called **windows**, are associated with two property values. The first value is **inactive/active**, indicating that the corresponding screen area has been painted or not, and is initialized to active. For those windows marked inactive, the second value is the pixel value painted.

The algorithm traverses an octree in a front to back sequence, checking intersection of octree nodes with the quadtree windows. If a node intersects only inactive windows, it is not visible and is discarded, so are all its descendants, and the next node in the traversal sequence at the same level or higher is processed. If not, for a gray node, the eight children are processed in a like manner; for a leaf node, the children of the windows are examined. Any active window enclosed by the octree node is written with the appropriate intensity value and marked inactive. At the lowest quadtree level the center of a window is checked for enclosure.

Block shading is used instead of computing the local surface normal. In block shading, The surface intensity values for the three visible faces of a cube are calculated. Quadtree windows enclosed by one of the three faces of a terminal node are given the intensity value for that face. An anti-aliasing technique is used to reduce the artifacts introduced by block shading, where intersection and enclosure tests are performed to a higher resolution level than the screen. The displayed intensity value for a window at the pixel level is an average of those windows below the pixel level.

Linear Octree

In a regular octree each node has a unique path address: a string of node numbers designating the nodes traversed from the root to it. Based on this observation, the **linear octree** of [Gar82] is simply a sorted list of path addresses of all black nodes. For example, the linear octree that corresponds to the octree of Fig. 2.6 is

21, 3X, 71, 73

The mark X indicates that the eight children in this level are all black and have been grouped together. Since there are no pointers and only black nodes are encoded, linear octrees save a lot of storage space (approximately 80% according to [Gar82]) over regular octrees.

Boolean operations on linear octrees are straightforward because of their sorted nature. For example, to find the union of two linear octrees that have N_{B_1}, N_{B_2} black nodes respectively needs $O(N_{B_1} + N_{B_2})$ time to scan and merge the input trees to an output tree. The display algorithm, however, appears to be difficult [Gar86]. From descriptions in previous sections it is clear that all display algorithms use a traversal sequence, either BTF or FTB, to get hidden surface removal. Because a linear octree has no pointers and only black nodes are encoded, tree traversal in a certain order becomes awkward.

Another display strategy first determines the 3D border voxels by repeatedly eliminating the non-border nodes from a linear octree [AGR84], then traversing the octree encoded border voxels in BTF sequence. But the $O(n^2(N_L+M))$ time to find the border voxels from a linear octree, where N_L is the length of the tree and M the number of the border voxels, is much longer than the $O(M)$ time to find the border voxels from a 3D array.

2.2.3 Surface Representation Schemes

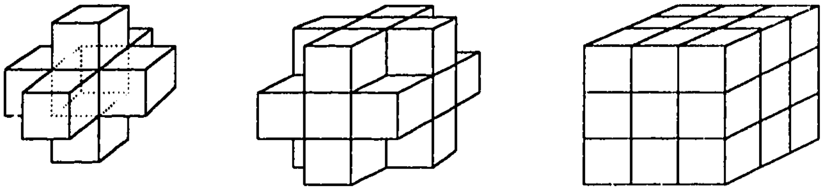
Another way is to represent objects by their surfaces. In this section two algorithms are reviewed. One is **surface tracking** [AFH81, GU89] based on the cuberille model, while the other, **marching cubes** [LC87], constructs a surface by triangle faces.

Some useful topological concepts and definitions are summarized in the next section. Among them are adjacency, neighbor, border and surface [Sri81]. The definitions are given in terms of the cuberille model.

Border and Surface

There are three kinds of **voxel adjacency**: **face** adjacency by which two voxels touch in a common face, **edge** adjacency by which two voxels touch in a common edge, and similarly **vertex** adjacency. Hence a voxel has three kinds of **neighbors**: face adjacent neighbors, edge adjacent neighbors and vertex adjacent neighbors. For two voxels $u(u_1, u_2, u_3)$ and $v(v_1, v_2, v_3)$, if **not** more than n, for $1 \leq n \leq 3$, of their indices differ by 1 and the rest are identical, the two voxels are said to be **n-adjacent**, denoted by the relation R_n. For example, if voxel u and v are face adjacent, uR_1v.

For a voxel u, the set of its n-adjacent neighbors or n-neighbors is denoted by $N_n(u)$. A voxel has six 1-neighbors (face neighbors), eighteen 2-neighbors (face and edge neighbors) and twenty six 3-neighbors (face, edge and vertex neighbors), as shown in Fig. 2.7.



(a) Six 1-neighbors (b) Eighteen 2-neighbors (c) Twenty six 3-neighbors

Figure 2.7: A voxel (in the center) and its n-neighbors.

Let S denote the set of voxels that belong to an object and $\bar{S}$ the background, then the **border** of the object S, denoted by $B(S)$, is defined as the set of voxels in S which have at least one neighbor in $\bar{S}$, i.e.,

$$B(S) = \{u | u \epsilon S \text{ and } N_n(u) \cap \bar{S} \text{ is not empty}\}$$

The **surface** of an object S, denoted by $\delta(S, \bar{S})$, consists of all border voxel faces that are at the interface of S and $\bar{S}$. A voxel face is uniquely defined by two abutting voxels, so the surface can be defined in terms of the ordered pair (u, v):

$$\delta(S, \bar{S}) = \{(u, v) | u \epsilon S,\ v \epsilon \bar{S},\ uR_1v\}.$$

Border and surface are different concepts. A border tracking algorithm has been reported [USH82]. It takes a 3D binary array, a border voxel and outputs a list of all border voxels of an object component. Algorithms to display the border voxels of an object occur in [Gar86]. The algorithm encodes the border voxels in a linear octree and displays three visible faces of each border voxel in a BTF sequence.

Surface Tracking Algorithms [AFH81,GU89]

In the cuberille model, each voxel face has four adjacent voxel faces connected through its four edges. If voxel faces of a surface are mapped to nodes and adjacency to edges, the surface of an object can be interpreted as a connected graph where every node has degree four. If a four dimensional array, whose indices are voxel coordinates and face orientations, is available for marking visited faces, surface tracking is then a standard breadth first or depth first graph traversal. Unfortunately, this marking scheme is not practical. The surface tracking algorithm, originally proposed by [AFH81] and later modified by [GU89], aims at solving the marking problem. Based on certain adjacency definition the algorithm interprets a surface as a digraph with its nodes all having indegree two and outdegree two. A list is used for holding and checking visited faces and it is possible to keep the list short. In the following a summary of related definitions is given first and then the algorithm.

To define the adjacency on the set $\delta(S,\bar{S})$, assign three directed circuits, i, j k, to a border voxel u as shown in Fig. 2.8. Assume its front face, f, is on a surface, and let e_1 and e_2 be two edges of f such that the i-circuit passes from e_1 to e_2. Then e_1 is called the **in-edge** of f and e_2 the **out-edge**. Apparently each face is passed by two circuits, so a voxel face has two in-edges and two out-edges.

The adjacency of voxel faces of a surface is defined as follows: Let $f_1=(u,z)$ and $f_2=(u',z')$ be two faces on a surface, f_1 is said to be **T-adjacent** to f_2, and denoted f_1Tf_2, iff:

1. f_1 and f_2 share a common edge e;

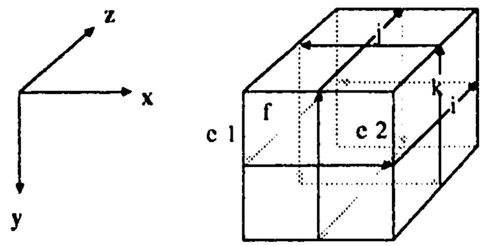


Figure 2.8: The three circuits assigned to a voxel.

2. e is an out-edge of f_1 and an in-edge of f_2;

3. exactly one of the following cases holds (see Fig. 2.9):

    (a) $u = u'$;

    (b) f_1, f_2 are in the same plane and u, u' are on the same side of the plane;

    (c) $z = z'$.

By definition, for any face $f\epsilon\delta(S, \bar{S})$, there are exactly two faces f_1, f_2, such that fTf_i, $1 \leq i \leq 2$, and there are exactly two faces f_1', f_2' such that $f_i'Tf$, for $1 \leq i \leq 2$. It has been shown [HW83] that a surface can be represented by a digraph whose nodes are voxel faces and whose arcs represent T-adjacency of the faces. Each node of the digraph has indegree two and outdegree two. The surface of an edge and face connected component of S corresponds to a connected subgraph of such a digraph. Furthermore, every connected subgraph of the digraph is strongly connected. So there is a binary spanning tree rooted at any node of a subgraph. Hence given a starting face f_0, the surface tracking problem is reduced to traverse a binary spanning tree rooted at f_0.

The input to the algorithm is binary data and a starting face f_0. The output is a list of faces on the surface that contains f_0. The data structures are: a queue Q containing faces to be processed, a list M of marked faces, and an output list L. A face is represented by voxel coordinates and its orientation (one of six). The algorithm is outlined below:

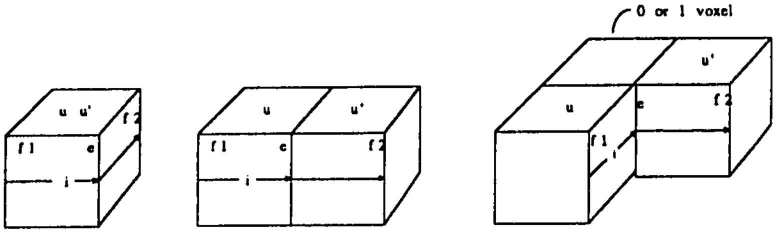
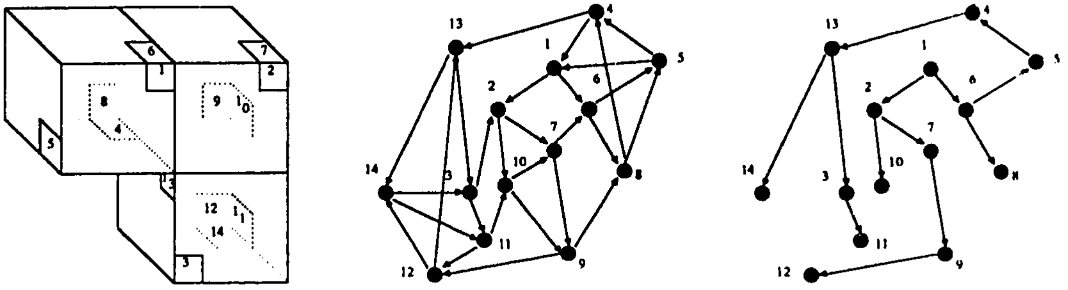


Figure 2.9: Face f_1 is T-adjacent to face F_2 in the three cases.

```
Surface tracking algorithm

Begin

1    queue $f_0$ and put two copies of $f_0$ in $M$;
2    while $Q$ is not empty
3        remove a face $f$ from $Q$;
4        find $f_l$ for $1 \leq l \leq 2$, such that $fTf_l$;
5        output $f$ to $L$;
6        for $l \leftarrow 1$ to 2 do
7            if $f_l \in M$ then delete $f_l$ from $M$;
8            else queue $f_l$ and put $f_l$ into $M$;
9        end for;
10   end while;
end
```

The complexity of the algorithm depends on line 7 checking for a marked node. When a node is visited, it is checked against M. If it is not in M, mark it by putting it into M. If it is in M, remove it. It will never be visited again because every node has indegree two. By this scheme the length of M is kept short.

An example of surface tracking is shown in Fig. 2.10. The body in Fig. 2.10(a) has three voxels and each exterior face is identified by a number. The surface digraph and the binary spanning tree rooted in face 1 is shown in (b) and (c). The digraph for the three voxel body has 14 nodes. One can imagine that for an ordinary object the digraph must be extremely large.



(a) The three voxel body (b) The modeling digraph (c) The binary spanning tree

Figure 2.10: A three voxel body and its digraph.

To improve the performance, one possibility is to redefine face connectivity such that some nodes of the digraph may only have indegree one, and the digraph is still strongly connected. Thus for those nodes which have indegree one no checking for a marked node is necessary. If a considerable number of nodes have indegree one, a great time saving can be achieved. This observation leads to a modified version of the surface tracking algorithm [GU89], which achieved a 30% time saving by making the definition of face connectivity directionally sensitive.

Marching Cubes [LC87]

A **cube** is made up of eight voxels at eight vertices. If the vertices with value one are inside a surface, and the vertices with value zero are outside, the surface intersects those cube edges with one vertex having value one and the other zero. Since there are eight vertices in a cube and two states, inside and outside, there are $2^8 = 256$ ways a surface can intersect a cube.

By inspecting two different symmetries the 256 cases can be reduced to 14 . First, the triangulated surface within a cube is unchanged if the vertex values are reversed.

This reduces the number of cases to 128. Second, all cases which can be reached by 90 degree rotations can be represented by one pattern. This reduces the cases to 14 (see Fig. 2.11).

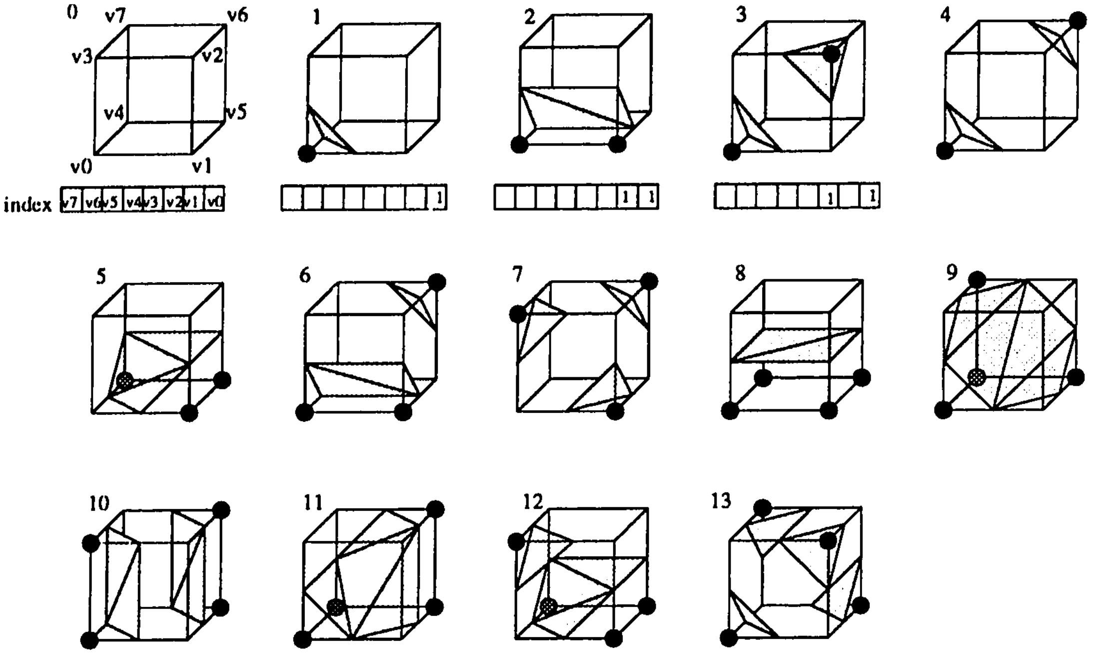


Figure 2.11: Fourteen intersection patterns of the marching cube algorithm.

Pattern 0 occurs if all vertex values are either one or zero, and produces no triangles. Pattern 1 occurs if the surface separates one vertex from the other seven, resulting in one triangle, and so on. Permutation of these 14 patterns using complementary and rotational symmetry produces the 256 cases. A table is created for looking up each case by the index of a given cube configuration.

The triangle face model approximates a surface more accurately than the voxel face model because it gives more face orientations. Since the vertex normals of the triangles are computed explicitly and Gouraud shading is used, the image quality is good. The marching cubes is more time consuming comparing with surface tracking because it marches all data. The experimental results has shown that creating a triangle model and a shaded image for 260 × 260 × 93 CT data (6,286,800 voxels) on a VAX 11/780 needs 30 minutes.

2.3 3D Identification Methods

In various representation schemes surveyed so far, the identification method used is thresholding. There are some 3D edge operators developed in the early 80's. One is a direct extension of the Roberts operator [Liu77]; another is a generalization of the Hueckel method of defining an operator using basis function expansion [ZH79]; the third is a generalization of the Prewitt method of defining an edge operator by fitting a surface to a neighborhood of each point [MR81]. A review of each of these methods is now given.

2.3.1 3D Edge Operators

3D Gradient Operator [Liu77]

Denote the intensity value associated with a voxel $v(i,j,k)$ by $I(i,j,k)$. A boundary element in the three-dimensional space is a line, e.g., lines $LX(i,j,k)$, $LY(i,j,k)$, $LZ(i,j,k)$ (see Fig. 2.12). The gradients at the three lines $LX(i,j,k)$, $LY(i,j,k)$, and $LZ(i,j,k)$ are defined by:

$$|Grad|_{LX(i,j,k)} = |I(i,j,k) - I(i,j+1,k+1)| + |I(i,j+1,k) - I(i,j,k+1)|$$

$$|Grad|_{LY(i,j,k)} = |I(i,j,k) - I(i+1,j,k+1)| + |I(i,j,k+1) - I(i+1,j,k)|$$

$$|Grad|_{LZ(i,j,k)} = |I(i,j,k) - I(i+1,j+1,k)| + |I(i+1,j,k) - I(i,j+1,k)|.$$

The possible boundary elements are those lines which have high gradient values.

The set of neighbors of a boundary element $LX(i,j,k)$ is defined as

$$N_{LX(i,j,k)} = \{LX(l,m,n)|\ l=i \text{ and } |j-m|+|k-n|=1\},$$

and $N_{LY(i,j,k)}$ and $N_{LZ(i,j,k)}$ are defined similarly. The algorithm insists that at least one and at most two neighbors of every boundary element also lie on the boundary. So in

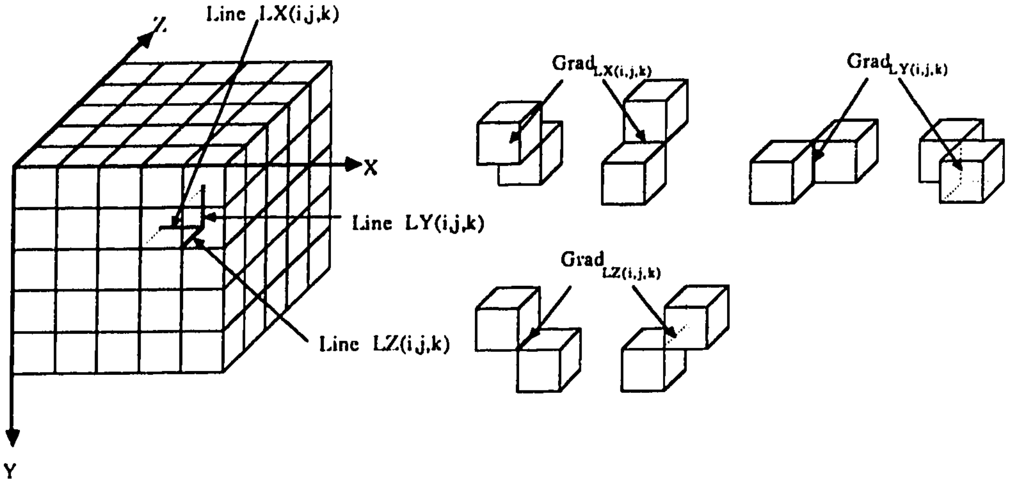


Figure 2.12: Liu's gradient operator.

searching the next boundary element only the four horizontal and vertical neighbors of the current boundary element need to be considered. Some heuristic information such as the boundary is a low curvature, closed surface, etc., is also used to help searching the next boundary element.

An Optimal 3D Edge Operator[ZH79]

A 3D edge element is interpreted as an oriented unit plane that separates voxels of different intensities such that dark voxels lie on one side of the plane, and light voxels lie on the other. An Optimal 3D edge operator finds the best oriented plane at each point in a 3D image.

Mathematically, the ideal 3D edge patterns can by described by the set of parameterized step functions:

$$E_{a,b,c}(x,y,z) = \begin{cases} +1 & \text{if } ax+by+cz \geq 0 \\ -1 & \text{if } ax+by+cz < 0 \end{cases}$$

which are defined on the sphere:

$$S = \{(x,y,z) : x^2+y^2+z^2 \leq 1\},$$

and the vector $\bar{N} = (a, b, c)$ is the unit normal of the plane $ax + by + cz = 0$.

Let $I(x, y, z)$ denote an input image defined on the unit solid sphere S that has been normalized to have zero mean and unit variance. The problem of detecting a 3D edge element is formulated to seek the parameter values (a, b, c) so that the mean square error

$$\iint_S |I(x, y, z) - E_{a,b,c}(x, y, z)|^2 dxdydz$$

is minimized. A practical solution to this minimization problem is to consider a finite-dimensional subspace M and find its orthnormal basis functions

$$\{\psi_1, \psi_2, \ldots, \psi_N\}.$$

The projection $I'(x, y, z)$ of the $I(x, y, z)$ onto M is given by

$$I'(a, b, c) = \sum_{i=1}^{N} c_i \psi_i(x, y, z)$$

where

$$c_i = \iint I'(x, y, z)\psi_i(x, y, z)dxdydz.$$

Similarly let $E'_{a,b,c}(x, y, z)$ denote the projection of patterns $E_{a,b,c}(x, y, z)$ onto M, then the full minimization problem can be replaced by finding (a, b, c) such that

$$\iint_S |I'(x, y, z) - E'_{a,b,c}(x, y, z)|^2 dxdydz \qquad (2.1)$$

is minimized.

The finite-dimensional space M must be selected such that the patterns E' are a good approximation to E. Assume that all patterns E have zero mean and unit variance, then the best basis functions are those functions that minimize the expected mean square error

$$\mathcal{E} \iint_S |E - E'|^2 dxdydz =$$

$$\mathcal{E} \iint_S |E_{a,b,c}(x, y, z) - \sum_{i=1}^{N} a_i \psi_i(x, y, z)|^2 dxdydz.$$

This expectation is taken over the full set of patterns E and is weighted by the probability density of occurrence of the patterns. Since the set of patterns is parameterized by (a, b, c), E can be regarded as a random field with probability density $p(a, b, c)$.

The solution to this minimization problem is given by the Karhunen-Loeve basis functions; i.e., the ψ_i are the solutions to the integral eigenvalue problem:

$$\iint R(x,y,z,x',y',z')\psi_i(x',y',z')dx'dy'dz' = \lambda_i\psi_i(x,y,z), \tag{2.2}$$

where $R(x,y,z,x'y'z')$ is the autocorrelation function of $E_{a,b,c}(x,y,z)$. The autocorrelation function can be modeled as

$$R(x,y,z,x',y',z') = 1 - \frac{2}{\pi}\arccos(xx' + yy' + zz'),$$

which is equal to 1 when (x,y,z) and (x',y',z') are vectors on the same direction; equal to -1 when the two vectors point in opposite direction; and drops off linearly as the angle between the vectors (x,y,z) and (x',y',z') increases from 0 to π. The first three eigenfunctions which correspond to the largest three eigenvalues of (2.2) are

$$\begin{aligned}\psi_1(x,y,z) &= \frac{x}{\sqrt{x^2+y^2+z^2}},\\ \psi_2(x,y,z) &= \frac{y}{\sqrt{x^2+y^2+z^2}},\\ \psi_3(x,y,z) &= \frac{z}{\sqrt{x^2+y^2+z^2}}.\end{aligned}$$

The discrete approximation to the three eigenfunctions defines the 3D edge operators. A $3 \times 3 \times 3$ operator is shown in Fig. 2.13. Since the three operators are rotationally invariant only one is shown. Apply the operators to an input image will produce the surface normal (a,b,c) at each image point, and this normal provides a precise minimum for the equation (2.1).

3D Edge Operator by Surface Fitting [MR81]

The original correspondence is a n-dimensional generalization of the Prewitt edge operator by fitting a hyperplane to a given hyperrectangular neighborhood. For simplicity, only a 3-dimensional edge operator is given below.

Let (x_1,x_2,x_3) be coordinates of the given point v in a three dimensional space. Suppose v is the origin. Then the neighborhood of v of odd side length has the form

$$N_v = \{(x_1,x_2,x_3) | -m_i \le x_i \le m_i,\ 1 \le i \le 3\}.$$

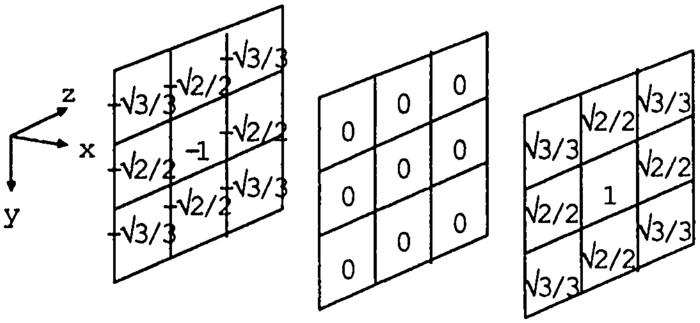


Figure 2.13: The optimal 3D edge operator ψ_1

The 3D edge element over N_v is a plane defined by

$$g(x_1, x_2, x_3) \equiv a_0 + \sum_{i=1}^{3} a_i x_i$$

that best fits the input image $I(x_1, x_2, x_3)$ at the origin such that the mean square error

$$\varepsilon = \sum_{N_v} [g(x_1, x_2, x_3) - I(x_1, x_2, x_3)]^2$$

is minimized.

To find $a's$ that minimize ε, differentiate ε with respect to a_0, a_1, a_2, a_3 and set the results to zero. Taking $\partial\varepsilon/\partial a_0$ gives

$$2\sum_{N_v} [g(x_1, x_2, x_3) - I(x_1, x_2, x_3)],$$

so that at the origin $a_0 = I(0, 0, 0)$. To determine a_1, a_2, a_3 which are the components of the gradient of $g(x_1, x_2, x_3)$ in the principal directions, write $\partial\varepsilon/\partial a_j$ in the form

$$2\sum_{N_v} [h_j(x_{j-1}, x_{j+1}) + a_j x_j - I(x_1, x_2, x_3)]\ x_j,$$

for $1 \le j \le 3$, where $h_j(x_{j-1}, x_{j+1}) \equiv g(x_1, x_2, x_3) - a_j x_j$, so that a_j and x_j do not occur in h_j. Setting this to zero gives

$$\sum_{N_v} x_j^2 a_j = \sum_{N_v} x_j I(x_1, x_2, x_3) - \sum_{N_v} x_j h_j(x_{j-1}, x_{j+1}).$$

In the last term of this expression, since h_j does not involve x_j, it may be reordered. For example, to find a_1, the last summation above can be reordered as

$$\sum_{x_2=-m_2}^{m_2} \sum_{x_3=-m_3}^{m_3} h_1(x_2, x_3) \sum_{x_1=-m_1}^{m_1} x_1.$$

Since

$$\sum_{x_j=-m_j}^{m_j} x_j = 0 \text{ for } 1 \leq j \leq 3,$$

this entire expression reduces to zero. Thus,

$$a_j = \frac{\sum_{N_v} x_j I(x_1, x_2, x_3)}{\sum_{N_v} x_j^2}.$$

Note that $\sum_{N_v} x_j^2$ is a constant, thus a_j is proportional to a linear combination of I values in the neighborhood of the given point, weighted by the j-coordinates of their positions relative to the point.

For a $3 \times 3 \times 3$ neighborhood the 3D edge operator obtained this way are the weights in the numerators (see Fig. 2.14).

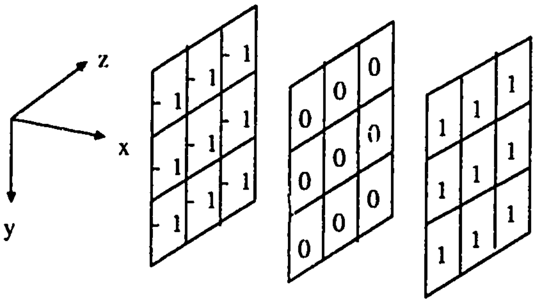


Figure 2.14: The 3D edge operator defined by surface fitting

2.3.2 Edge Detection by Zero Crossing

Edge detection proposed by [MH80] takes a local average of an image, then detects intensity changes in the image.

To average an image, a smoothing filter should be smooth and roughly band-limited in the frequency domain. To eliminate noise, the filter should be smooth and localized in the spatial domain. The two localization requirements in both spatial and frequency domains are conflicting. There is only one distribution, namely the Gaussian:

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{2.3}$$

with Fourier transform

$$\hat{G}(\omega) = e^{-\frac{\sigma^2\omega^2}{2}} \tag{2.4}$$

that can optimize these two conflicting localization requirements.

Since intensity change occurs at the extreme value of the gradient, that corresponds to a zero-crossing of directional second derivative, edge elements are zero-crossings of the second derivative D'' in an appropriate direction:

$$f(x,y) = D''[G(x,y) * I(x,y)] \tag{2.5}$$

$$= G'' * I(x,y), \tag{2.6}$$

where $I(x,y)$ is image intensity, and $*$ is the convolution operator. In one dimension

$$G''(x) = -\frac{1}{\sqrt{2\pi}\sigma^3}(1 - \frac{x^2}{\sigma^2})e^{-\frac{x^2}{2\sigma^2}}. \tag{2.7}$$

The Fourier transform of G'' is $(j\omega)^2 G(\omega)$, and denote it by $\hat{G}_2(\omega)$

$$\hat{G}_2(\omega) = -\omega^2 e^{-\frac{\sigma^2\omega^2}{2}}. \tag{2.8}$$

Fig. 2.15 shows the Mexican hat shaped G'' operator for σ=2 and its Fourier transform.

The direction in which the second derivative should be in is the one in which the zero-crossing has maximum slope.

Similar to edge elements detected by a gradient operator, edge elements detected by G'' can also be characterized by their locations, orientations and amplitudes. The location of an edge is a zero-crossing generated by a G'' operator. The orientation of the edge is perpendicular to the direction in which G'' is in. The amplitude of the edge is the maximum slope of the G''.

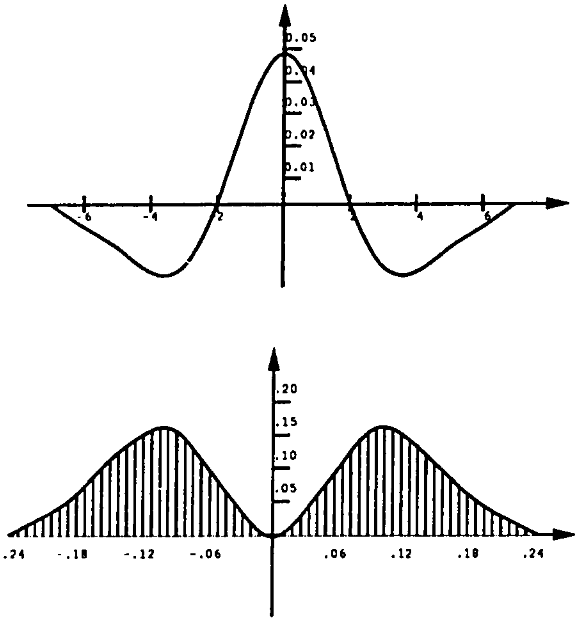


Figure 2.15: The G'' operator for σ=2 and its Fourier transform.

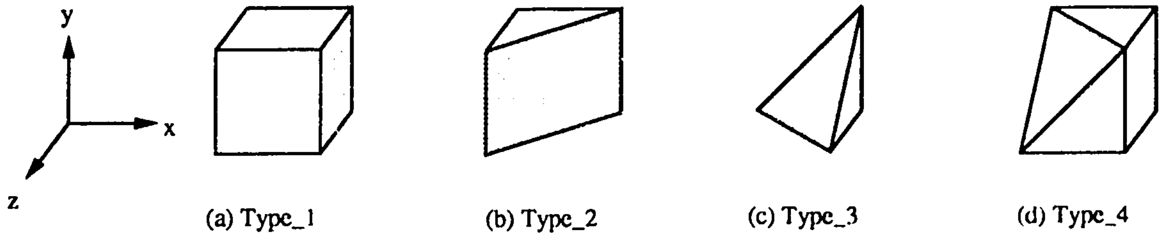
Chapter 3

The Extended Cuberille Model

This chapter introduces the extended cuberille model which has four volume primitives. The mathematical definitions of voxels and objects in the extended cuberille model are proposed. 3D edge elements are converted to the four types of voxels by loc_dir codes. Merits of the three representation schemes: space occupancy enumeration, octree, and surface representation by the extended cuberille model are briefly discussed.

3.1 The Extended Cuberille Model

The modeling volume primitives are extended to include three other polyhedra (see Fig. 3.1) so that each voxel has a face whose orientation is compatible with one of the 26 gradient orientations.



(a) Type_1 (b) Type_2 (c) Type_3 (d) Type_4

Figure 3.1: The set of volume primitives

As shown in Fig. 3.1, cutting a cube voxel through diagonals of top and bottom face gives the second polyhedron with a face orientation $(1,0,1)$. Cutting a cube voxel through three vertices defines the third and the fourth polyhedra, and gives a face oriented at $(-1,1,1)$. Since the last three polyhedra are not symmetric, it is assumed that a volume primitive is invariant under 90 degree rotation about any principal axis. Hence the faces of the volume primitives give 26 orientations. Furthermore, to be able to represent objects hierarchically, it is also assumed that a voxel scaled by a power of two is the same. The formal definition is given in the following.

Let V be the set of four volume primitives shown in Fig. 3.1, i.e., $V=\{type\_1,\ type\_2,\ type\_3,\ type\_4\}$. Let T denote the transformation of 90 degree rotations about the principal axes or power of two scalings. Thus T is an equivalence relation on V and each $type\_k$, $k = 1,2,3,4$, is an equivalence class by T. Let I be the 3D coordinate set, $I \subset Z^3$, Z is the set of integers, and f is a map, $f : I \rightarrow V$, then

Definition 3.1 *A voxel v_i is a pair of (i,f(i)), where $f(i)$ is a volume primitive, i.e., $f(i) \in V$, of minimum scale and i are the coordinates, $i \in I$.*

Arguably, a cube could be cut other ways, as shown in Fig. 3.2, that also results in 26 face orientations. But the set V is better than other choices for it is the smallest set that is closed under the subdivision operation.

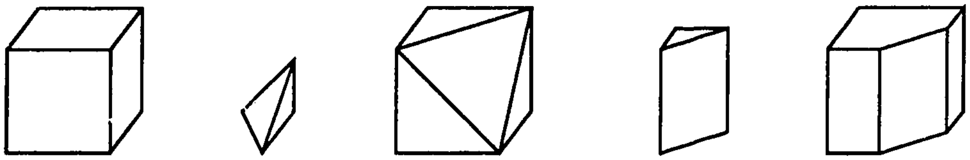
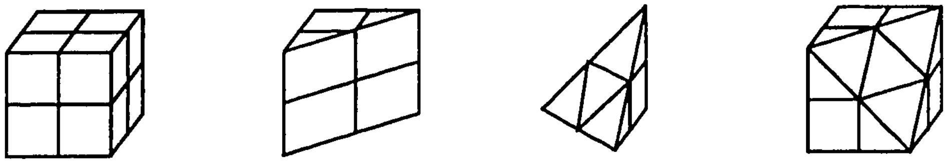


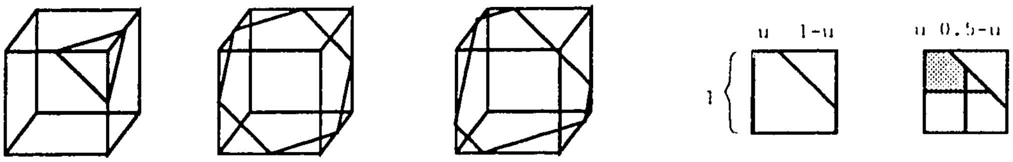
Figure 3.2: Another set of volume primitives

Theorem 3.1 *The set V is closed under the subdivision operation, and it has the least cardinality among those having the same property.*

Proof: It follows directly from subdividing the four volume primitives, as shown in Figs. 3.3(a) to (d), that the set V is closed under subdivision. It remains to be shown that it has the least cardinality. Let U be any set of volume primitives whose type_3 and type_4 polyhedra are obtained by cutting a cube voxel, but not passing through face diagonals. Assume the edge of a face is cut at a ratio $u/(1 - u)$, as shown in Fig. 3.3(f), then subdivision of the cube results in a new polyhedra with edge cut ratio of $u/(0.5 - u)$. If U is closed, it must include at least the two sets of polyhedra from the two ways of cutting to give the same face orientation. The same argument holds for a type_2 polyhedron. It follows $|U| > |V|$.



(a) Type_1 (b) Type_2 (c) Type_3 (d) Type_4



(e) Different cutting to get a face with normal (1,1,1). (f) Subdividing introduces a new polyhedron.

Figure 3.3: The V set of volume primitives is closed under subdivision.

Since the set V is closed under the subdivision, it is possible to represent objects by octrees.

The mathematical definition of an object in the extended cuberille model is proposed below:

Definition 3.2 *An object S is a regularized union of voxels, $S = \cup^*_{i,j} v_i$, where $\cup^*$ denotes the regularized union operation.*

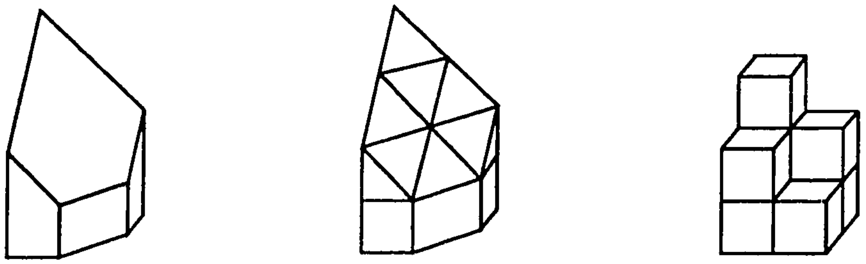
The regularized set operations, see page 14, defines $A \cup^* B = c(i(A \cup B))$, where $c(A \cup B)$ and $i(A \cup B)$ denote the closure and interior of $A \cup B$.

The definition gives the constructive process: because voxels can only touch in faces, edges or vertices, it implies $v_i \cap^* v_j = \phi$ for $i \neq j$. Thus $\cup^* v_i$ removes all internal voxel faces to make a solid object with one interior enclosed by a surface.

There are also several ways to represent objects in the extended cuberille model. In the next Section, the merits of three representation schemes: the enumeration scheme, octrees, and surface representation, are briefly discussed.

3.2 Merits of Representation Schemes

The spatial enumeration scheme in the extended cuberille model lists all voxels whose spaces are either **fully** or **partially** occupied by an object. The interior of an object is filled by type_1 voxels because they are fully occupied. All type_2 to type_4 voxels represent partially occupied space and are therefore on the border of an object. Type_1 voxels could also be on the border if the surface passes through at least one of its faces. For example, Figs. 3.4(a) to (c) show a mathematically defined object, its space enumeration representation, and its space occupied in the cuberille model. Although the representation gives a smoother surface, it is not as storage efficient as its counterpart in the cuberille model, for it needs two arguments - coordinates, type or gradient direction to list a voxel.

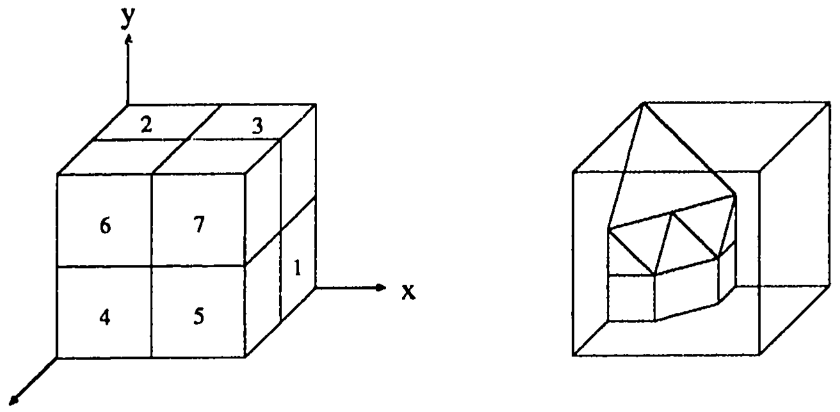


(a) Defined object.

(b) Space enumeration in the extended cuberille model.

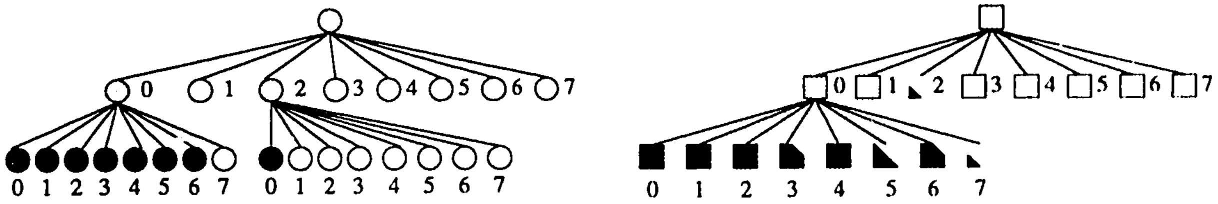
(c) Space enumeration in the cuberille model.

Figure 3.4: A mathematically defined object and its representation.



(a) Space subdivision.

(b) The subdivided object.

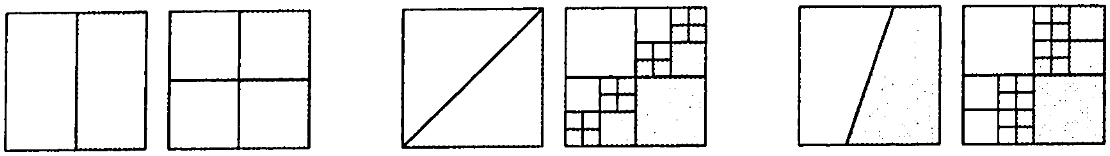


(c) The Octree representation in the cuberill model.

(d) The octree representation in the extented cuberille model.

Figure 3.5: An octree represented object in the extended cuberille model.

An octree in the extended cuberille model may have **five** types of leaf nodes. In addition to the cube shaped black and white leaf nodes, type_2, type_3 and type_4 leaf nodes represent partially occupied space and are all black. An octree representing the object in the previous example is shown in Fig. 3.5(d). Compared with the octree in the cuberille model (Fig. 3.5(c)), the octree in the extended model is more concise because it includes leaf nodes to represent certain partially occupied spaces. Now consider two extreme cases shown in Fig. 3.6. For the object in Fig. 3.6(a), the octrees for the two models would be the same. For the object in Fig. 3.6(b), the subdivision around the border in the cuberille model would reach the voxel level, whereas there is no subdivision in the extended model, because the root node is a leaf node. For the object with a surface slope as shown in Fig. 3.6(c), the subdivision in the extended mode would, in the worst



(a) (b) (c)

Figure 3.6: Comparing tree size of two models.

case, reach the voxel level. The following conclusion results:

Theorem 3.2 *To represent an object, the size of an octree in the extended cuberille model is at most the size of an octree in the cuberille model.*

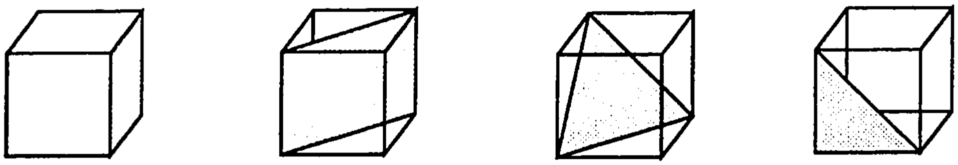


Figure 3.7: Four types of external voxel faces.

A surface representation lists all external voxel faces on a surface. Since there are four types of voxels, and by examing Fig. 3.1, there are only four types of external voxel faces, as shown in Fig. 3.7. It therefore follows:

Theorem 3.3 *The four types of external voxel faces shown in Fig. 3.7 can close the surface of any object.*

Since there are only four types of external voxel faces in the extended cuberille model, it is expected that the implementation of a surface representation is not very complicated. On the other hand, the surface of an object may not always be very smooth.

3.3 Converting to the Extended Cuberille Model

This Section discusses the implementation of converting edge elements to the modeling primitives.

The characteristic of the extended cuberille model is that a voxel has a face whose normal coincides with edge orientation. This face is termed a **face primitive**. A voxel can therefore be referred to as either a volume primitive or a face primitive. The next section gives the implementation using a one-byte code to record edge orientation.

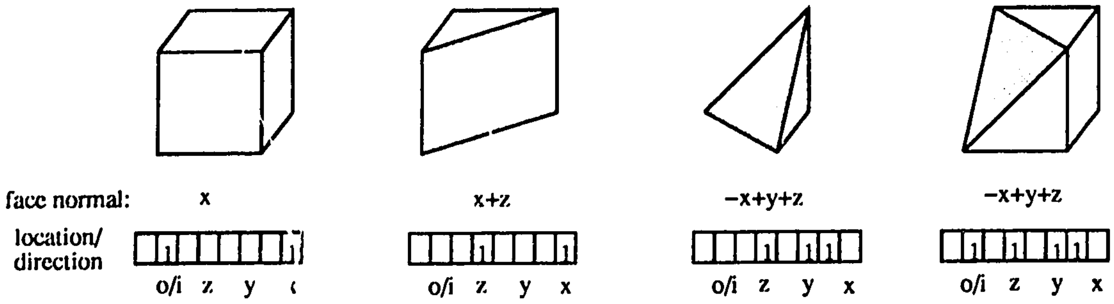
3.3.1 The location/direction code

An edge has both magnitude and orientation. The magnitude is stored as an integer, and the orientation is converted to a one-byte location/direction code, loc_dir code for short. Since there are 26 edge orientations, the orientation of an edge is recorded in the lower six bits of its loc_dir code with one bit for each $x, -x, y, -y, z$ and $-z$ direction. Each location/direction code corresponds to a face primitive. Fig. 3.8 shows the loc_dir code for the four face primitives. Since type_3 and type_4 face have the same normal, they are distinguished by the fact that a type_3 voxel is outside an object and a type_4 voxel is inside. Bit six of the location/direction code is the inside/outside bit. Because a type_4 voxel is inside, bit six of its loc_dir code is set to one.

The inside/outside question is tested with the edge direction as follows: if v_1 and v_2 are the two voxel neighbors in the $3 \times 3 \times 3$ neighborhood along the gradient direction, then v_0, the central voxel, is inside if $I(v_0) \geq [I(v_1) + I(v_2)]/2$, otherwise it is outside. $I(v_i)$, $i = 1, 2$, is the intensity value at voxel v_i and it is assumed that objects have larger intensity values than the background.

A type_1 voxel is always set to inside (an outside type_1 voxel is not a solid). A type_2 voxel can be set to either inside or outside. In the implementation, it is set to outside.

Bit 7 of the loc_dir code is used to mark a voxel visited in the surface tracking algorithm (see Chapter 5).



(a) Type_1 (b) Type_2 (c) Type_3 (d) Type_4

Figure 3.8: The location/direction codes for the four volume primitives.

Hence applying the 3D edge operator to a 3D array results in an array of edge magnitudes and location/direction codes.

3.3.2 Display an Edge Image and a Surface

The location/direction code, along with the voxel coordinates, specifies the type (shape), the normal and the location of a face primitive, but not the size. This section describes how to store and access the geometrical information of faces to display an edge image and a surface.

A data structure, called `TABLE`, is created to save the geometry of the four face primitives. The first two levels of the table necessary for displaying are depicted in Fig. 3.9. The full table structure will be given in Section 5.4.

The table has 128 entries, corresponding to the lower seven bits of the loc_dir code. Each table entry has two fields: a pointer, `face`, pointing to a `BASIC_FACE` structure where the face geometry is stored, and a coordinate transformation `Matrix`. Since the lower seven bits are not fully used, some entries are empty.

A face is stored as three lists of vertex coordinates, `*ix,*iy,*iz`. Other information such as the number of vertices, the center coordinates are also stored in the structure.

To display an edge, the edge loc_dir code is used to index the table entry. The face vertices pointed to by the `face` pointer are read out. The vertex coordinates are transformed by the `Matrix` so that the resulting face normal is consistent with the edge

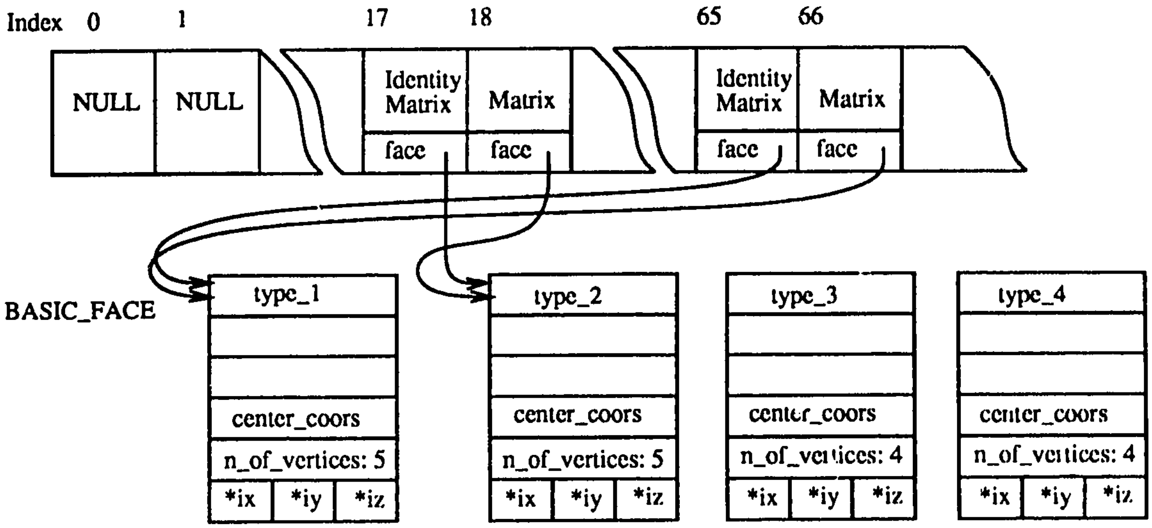


Figure 3.9: The TABLE structure.

orientation.

A graphics package, WINDLIB [GB87], is used to display voxel faces. WINDLIB displays shaded polygons and interactively supports rotating and scaling of displayed objects. Fig. 3.10 shows the edge image of a test object sphere. The test data has volume of 40 × 40 × 40 with 16 gray values. The detected edges whose magnitudes are greater than 8 are displayed. The displayed size of a face is proportional to the edge magnitude. The bigger the edge magnitude, the larger the size of the displayed face.

Fig. 3.10 also showns the surfaces of three test objects reconstructed by the extended cuberille model. The surfaces were extracted by the 3D border identification method discussed in Chapter 4, the surface tracking algorithm in Chapter 5, and the surface closure algorithm in Chapter 7 . The three surfaces clearly show the four types of external voxel faces.

Chapter 4

3D Border Identification

This chapter presents a three dimensional border identification method based on the sign of the second derivative of intensity change. The chapter is organized as follows. Section 4.1 defines the surface as zero-crossings from a step intensity change in continuous space. Section 4.2 analyzes the direction the second derivative should be in. The next two sections discuss two ways of reducing the computational cost, and asymmetric Gaussian filters are proposed for 3D convolutions. In Section 4.5, border voxels in discrete space are defined based on the sign of the second derivatives. The implementation of the asymmetric Gaussian filters and discrete convolution are given in Section 4.6. Procedures to compute the sign of the second derivative and some experimental results are contained in the last two sections.

4.1 Surface of Step Intensity Change

This section will show that zero-crossings correspond to a step intensity change.

Suppose there is a step intensity change across an object surface, as shown in 1.3(b). Let $I(x, y, z)$ be the intensity function, $(x, y, z) \in R^3$, R is the set of real numbers. Without loss of generality, suppose the intensity change occurs at the origin and in the x direction. Clearly, the origin is the desired surface point. It will show in the following that a zero-crossing occurs at the origin.

For a bright object surrounded by a darker background, the intensity around the surface can be modeled as a one dimensional step change, $I(x,y,z) = c(\pi/2 - \arctan(kx))$ for a sufficiently large constant $k > 0$. The constant c describes the change magnitude and k describes the intensity change rate. The smoothing filter $G(x,y,z)$ is a Gaussian filter with variance σ. Let $w(x,y,z)$ be the second derivative in the x-direction of the smoothed function $I(x,y,z)$. $w(x,y,z)$ is given by

$$\frac{\partial^2}{\partial x^2}[G(x,y,z) * I(x,y,z)], \tag{4.1}$$

that is,

$$\frac{\partial^2}{\partial x^2}\int\int\int -\frac{1}{(\sqrt{2\pi}\sigma)^3}e^{-\frac{x'^2+y'^2+z'^2}{2\sigma^2}}\, c\arctan k(x-x')dx'dy'dz'.$$

Exchange the integral with the derivative, and evaluate the second derivative with respect to $I(x,y,z)$ results in

$$w(x) = \frac{1}{\sqrt{2\pi}\sigma}\int_{-\infty}^{\infty} e^{-\frac{x'^2}{2\sigma^2}}\frac{2ck^3(x-x')}{(1+k^2(x-x')^2)^2}dx' \tag{4.2}$$

which is a function of x. Because the integrand is an odd function, $w(x) = 0$ at $x = 0$. To see how $w(x)$ changes sign across $x = 0$, it is necessary to evaluate the sign of $w'(x)$ at $x = 0$. Note that the difference between the derivative of $x - x'$ with respect to x and x' is minus one, hence

$$w'(x) = \frac{1}{\sqrt{2\pi}\sigma}\int_{-\infty}^{\infty} e^{-\frac{x'^2}{2\sigma^2}}\frac{d}{dx'}\left(\frac{-2ck^3(x-x')}{(1+k^2(x-x')^2)^2}\right) dx'.$$

Integrating the right hand side of above equation by parts gives

$$w'(x) = \frac{1}{\sqrt{2\pi}\sigma}\int_{-\infty}^{\infty}\left(-\frac{x'}{\sigma^2}\right)e^{-\frac{x'^2}{2\sigma^2}}\frac{2ck^3(x-x')}{(1+k^2(x-x')^2)^2}\, dx'$$

and at $x = 0$,

$$w'(0) = \frac{2ck^3}{\sqrt{2\pi}\sigma^3}\int_{-\infty}^{\infty}\frac{x'^2e^{-\frac{x'^2}{2\sigma^2}}}{(1+k^2x'^2)^2}dx' > 0. \tag{4.3}$$

Hence $w(x)$ is monotonically increasing in the neighborhood of $x = 0$. In other words, $w(x)$ changes sign from negative to positive as it crosses zero at $x = 0$. $x = 0$ is therefore called a **zero-crossing** and the properties of zero-crossings can be used as a condition to detect surface points.

It is worthwhile to point out that the location of zero-crossings is not sensitive to the direction of the second derivative of an intensity change. Suppose the second derivative is taken in the direction l. The direction cosines of l are $\cos\alpha$, $\cos\beta$ and $\cos\theta$ for some $\alpha, \beta, \theta, < \pi/2$. (4.2) then becomes

$$w(x) = -\frac{1}{\sqrt{2\pi}\sigma}\int_{-\infty}^{\infty} e^{-\frac{x'^2}{2\sigma^2}} \frac{2ck^3(x-x')\cos^2\alpha}{(1+k^2(x-x')^2)^2} dx'.$$

Because $\cos^2\alpha > 0$, the direction of l does not affect the zero location nor the sign of $w(x)$ but only the magnitude of $w(x)$.

To summarize, let $v = (x, y, z) \in R^3$ and l be a directed line at v pointing outside of the surface. If v is on the surface, it must satisfy the following conditions:

$$\frac{\partial^2}{\partial l^2}[G(v) * I(v)] = 0 \tag{4.4}$$

and

$$\frac{\partial^2}{\partial l^2}[G(v-\delta l) * I(v-\delta l)] < 0, \quad \frac{\partial^2}{\partial l^2}[G(v+\delta l) * I(v+\delta l)] > 0 \tag{4.5}$$

for a small $\delta l > 0$ in the direction of l. Alternatively, the conditions above can be written in terms of $w(v)$ as

$$\begin{aligned} &w(v) = 0 \\ &w(v-\delta l) < 0 \\ &w(v+\delta l) > 0. \end{aligned} \tag{4.6}$$

4.2 The Direction of The Second Derivative

As pointed out in the previous section, the direction of the second derivative does not affect the location of zero-crossings. But due to the quantization errors and noise, it is desired to take the second derivative in a direction that has a maximum rate of change. This section shows the direction in which the second derivative of the intensity function has maximum change rate.

Suppose the gradient at v is $g(v)$ with the direction g. Take v as the origin and form a right-handed orthogonal coordinate system $\{v; n, t, b\}$ such that n is in the same

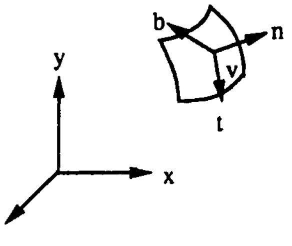


Figure 4.1: Coordina [illegible] $t, b)$ with origin v.

direction as g, as shown in Fig. 4.1 [illegible] directed line, and $\cos\alpha$, $\cos\beta$ and $\cos\gamma$ be the direction cosines of l respect to the (n, t, b)-coordinate system.

The derivative of the intensity function $I(n, t, b)$ in the direction of l is given by

$$\frac{\partial}{\partial l} I(n, t, b) = \left(\frac{\partial}{\partial n}\cos\alpha + \frac{\partial}{\partial t}\cos\beta + \frac{\partial}{\partial b}\cos\gamma\right) I(n, t, b). \tag{4.7}$$

At the origin v, because n is in the gradient direction, $\frac{\partial I}{\partial t}|_{(0,0,0)} = 0$ and $\frac{\partial I}{\partial b}|_{(0,0,0)} = 0$. Therefore,

$$\frac{\partial}{\partial l} I(0, 0, 0) = \frac{\partial}{\partial n} I(0, 0, 0)\ \cos\alpha. \tag{4.8}$$

If $\alpha = 0$, $\frac{\partial}{\partial l} I(0, 0, 0)$ takes its extreme value $\frac{\partial}{\partial n} I(0, 0, 0)$, the gradient value of v.

The second derivative of $I(n, t, b)$ in the direction l is given by

$$\frac{\partial^2}{\partial l^2} I(n, t, b) = \left(\frac{\partial^2}{\partial n^2}\cos^2\alpha + \frac{\partial^2}{\partial n \partial t} 2\cos\alpha\cos\beta + \frac{\partial^2}{\partial n \partial b} 2\cos\alpha\cos\gamma + \right.$$

$$\left. \frac{\partial^2}{\partial t \partial b} 2\cos\beta\cos\gamma + \frac{\partial^2}{\partial t^2}\cos^2\beta + \frac{\partial^2}{\partial b^2}\cos^2\gamma\right) I(n, t, b). \tag{4.9}$$

For a fixed l, $\frac{\partial^2}{\partial l^2} I(n, t, b)$ changes most rapidly in its gradient direction, that is the direction of $(\frac{\partial}{\partial n}\vec{n} + \frac{\partial}{\partial t}\vec{t} + \frac{\partial}{\partial b}\vec{b})\frac{\partial^2}{\partial l^2} I(n, t, b)$. Because at origin v, $\frac{\partial I}{\partial t}|_{(0,0,0)} = 0$ and $\frac{\partial I}{\partial b}|_{(0,0,0)} = 0$, to make the calculation simpler, an approximation is made that in a small neighborhood V of v, $\frac{\partial I}{\partial t}|_{v \in V} = 0$ and $\frac{\partial I}{\partial b}|_{v \in V} = 0$ (see Fig. 4.2). Hence equation (4.9) is reduced to

$$\frac{\partial^2}{\partial l^2} I(n, t, b) = \frac{\partial I^2}{\partial n^2}\cos^2\alpha. \tag{4.10}$$

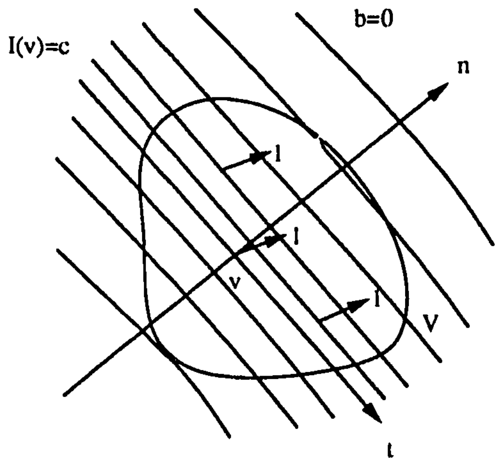


Figure 4.2: The neighborhood V of v.

For a given direction l in V, the gradient of $\frac{\partial^2}{\partial l^2} I(n,t,b)$ is then given by

$$\frac{\partial I^3}{\partial n^3}\cos^2\alpha\vec{n} + \frac{\partial I^3}{\partial n^2\partial t}\cos^2\alpha\vec{t} + \frac{\partial I^3}{\partial n^2\partial b}\cos^2\alpha\vec{b}$$
$$= \frac{\partial I^3}{\partial n^3}\cos^2\alpha\vec{n}. \tag{4.11}$$

Equation (4.11) shows that for any given l, if at every point $(n,t,b) \in V$ the second derivative is taken in the direction of l, the resulting $\frac{\partial^2}{\partial l^2} I(n,t,b)$ always changes rapidly in the direction of $\vec{n}$. Especially when $\alpha = 0$, l lies in the direction of $\vec{n}$, $\frac{\partial^2}{\partial l^2} I(n,t,b) = \frac{\partial^2}{\partial n^2} I(n,t,b)$ and changes most rapidly with the rate $\frac{\partial I^3}{\partial n^3}$. Consequently, the direction in which the second derivative of v should be taken is the gradient direction of v. Conditions (4.4) and (4.5) are then written as

$$\frac{\partial^2}{\partial g^2}[G(v) * I(v)] = 0, \tag{4.12}$$

and

$$\frac{\partial^2}{\partial g^2}[G(v-\delta g) * I(v-\delta g)] < 0, \quad \frac{\partial^2}{\partial g^2}[G(v+\delta g) * I(v+\delta g)] > 0, \tag{4.13}$$

where $\delta g > 0$ and g is in the gradient direction of $I(v)$ at v.

By the symmetric property of convolution, the second derivative in condition (4.12) can be taken with either $G(v)$ or $I(v)$. The left-hand side of (4.12) can therefore be written as $G(v) * \frac{\partial^2}{\partial g^2} I(v)$ or $\frac{\partial^2}{\partial g^2} G(v) * I(v)$. Because the intensity function $I(v)$ is usually unknown, the second derivative is taken with the Gaussian filter $G(v)$. Denote $\frac{\partial^2}{\partial g^2} G(v)$ by G''_g, the above equations become:

$$G''_g(v) * I(v) = 0, \tag{4.14}$$

$$G''_g(v - \delta g) * I(v - \delta g) < 0, \quad G''_g(v + \delta g) * I(v + \delta g) > 0. \tag{4.15}$$

Write above equations in terms of $w(v)$, $w(v) = G''_g(v) * I(v)$, (4.6) becomes:

$$\begin{aligned} w(v) &= 0 \\ w(v - \delta g) &< 0 \\ w(v + \delta g) &> 0. \end{aligned} \tag{4.16}$$

Since the conditions (4.14) and (4.15) involve three dimensional convolutions, the computation is very expensive. The next two sections show how to simplify the computation. It first shows in what circumstance the 3D convolutions can be reduced to 1D convolutions. It then shows that in general, the 3D Gaussian filter can be replaced by an asymmetric one so that it is possible to implement conditions (4.14) and (4.15) with a reasonable cost.

4.3 Evaluation by 1D Convolution

This section will show that if the intensity function $I(v)$ is separable in the $\{v; n, t, b\}$ coordinate system, the 3D convolution in conditions (4.14) and (4.15) can be replaced by a 1D convolution.

Substitute $G(x, y, z)$ into condition (4.14), and its left-hand side becomes

$$w(x, y, z) = \int\int\int \frac{\partial^2}{\partial g^2}\left(\frac{1}{(\sqrt{2\pi}\sigma)^3} e^{-\frac{(x-x')^2+(y-y')^2+(z-z')^2}{2\sigma^2}}\right) I(x', y', z') dx' dy' dz', \tag{4.17}$$

where g is the gradient direction of $I(x, y, z)$ at (x, y, z), and the three integrations are in the interval $(-\infty, \infty)$.

To calculate the convolution at v, transform (x, y, z) to the $\{v; n, t, b\}$ coordinate system and express (4.17) in terms of (n, t, b) as

$$\tilde{w}(n,t,b) = \int\int\int \frac{\partial^2}{\partial n^2}\left(\frac{1}{(\sqrt{2\pi}\sigma)^3} e^{-\frac{(n-n')^2+(t-t')^2+(b-b')^2}{2\sigma^2}}\right) I(n',t',b')dn'dt'db'. \qquad (4.18)$$

At the origin v, $\tilde{w}(0,0,0) = w(x,y,z)$. If $I(n,t,b)$ is separable, i.e. $I(n,t,b) = I_1(n)I_2(t)I_3(b)$, the right hand side of (4.18) can be expressed as

$$\int\int \frac{1}{(\sqrt{2\pi}\sigma)^2} e^{-\frac{(t-t')^2+(b-b')^2}{2\sigma^2}} I_2(t')I_3(b')dt'db' \int \frac{\partial^2}{\partial n^2}\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(n-n')^2}{2\sigma^2}}\right) I_1(n')dn'. \qquad (4.19)$$

Since $I(n,t,b) > 0$, the first 2-dimensional integration of (4.19) is positive. The zero location and the sign of (4.19) are determined by the second integration. The 3D convolutions in conditions (4.14) and (4.15) can therefore be replaced by the following one dimensional convolution in the direction n instead:

$$\begin{aligned}\tilde{w}_n(n,t,b) &= \int \frac{\partial^2}{\partial n^2}\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(n-n')^2}{2\sigma^2}}\right) I(n',t,b)dn' \\ &= G''_n(n) * I(n,t,b).\end{aligned}$$

Hence for a separable intensity function, the condition (4.16) for testing zero-crossings is replaced by:

$$\begin{aligned}&\tilde{w}_n(n,t,b) = 0 \\ &\tilde{w}_n(n-\delta n,t,b) < 0 \qquad\qquad (4.20) \\ &\tilde{w}_n(n+\delta n,t,b) > 0.\end{aligned}$$

4.4 Convolutions for the General Case

In general, however, $I(n,t,b)$ is not separable. But it will soon be seen that the symmetric Gaussian filter in the 3D convolution can be replaced by an asymmetric one with a smaller scale in the t, b axes. As a result, conditions (4.14) and (4.15) can be tested with a reasonable expense.

First, observe from Fig. 4.3 that a Gaussian filter is localized. It approximates zero when its arguments are out of the range $(-3\sigma, 3\sigma)$. Hence finite scale Gaussian filters

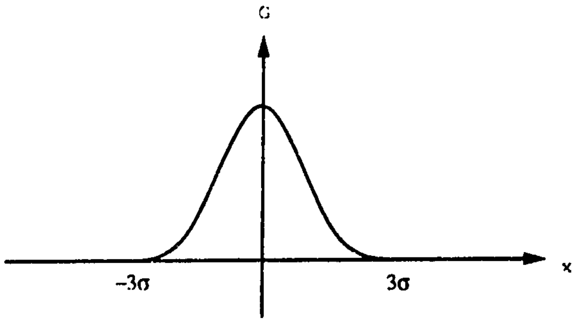


Figure 4.3: A Gaussian filter is localized within $(-3\sigma, 3\sigma)$.

can be used in convolutions. The integral interval in (4.18) therefore takes $(-3\sigma, 3\sigma)$ instead of $(-\infty, \infty)$.

Secondly, let $\tilde{I}(n', t, b)$ be the result of the 2D integration respect to $t'.b'$ in (4.18),

$$\begin{aligned} \tilde{I}(n', t, b) &= \int_{-3\sigma}^{3\sigma} \int_{-3\sigma}^{3\sigma} \frac{1}{(\sqrt{2\pi}\sigma)^2} e^{-\frac{(t-t')^2+(b-b')^2}{2\sigma^2}} I(n', t', b') dt' db' \\ &= G(t, b) * I(n, t, b) \qquad (4.21) \end{aligned}$$

where $G(t, b)$ is the two dimensional Gaussian filter in t, b direction, then (4.18) becomes

$$\tilde{w}(n, t, b) = \int_{-3\sigma}^{3\sigma} \frac{\partial^2}{\partial n^2} \left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(n-n')^2}{2\sigma^2}} \right) \tilde{I}(n', t, b) dn'$$

which is

$$\begin{aligned} \tilde{w}(n, t, b) &= \frac{\partial^2}{\partial n^2} G(n) * \tilde{I}(n, t, b) \\ &= G''_n(n) * G(t, b) * I(n, t, b). \end{aligned}$$

Note from (4.21) that $\tilde{I}(n, b, t)$ is a smoothed version of $I(n, b, t)$ in planes perpendicular to the n axis. Remember that $\vec{n}$ is the gradient direction at v, in whichever direction in the tb plane the intensity change rate at v is zero. It is reasonable to assume that in a small neighborhood of v, $I(n, t, b)$ does not significantly change in the tb direction. Hence the 2D Gaussian filter $G(t, b)$ with a smaller σ can be used in (4.21). This suggests the use of an asymmetric Gaussian filter with different scales, σ_1 in n axis and σ_2 in b, t axes and $\sigma_1 > \sigma_2$, i.e.

$$G(n,t,b) = \frac{1}{(\sqrt{2\pi})^3\sigma_1\sigma_2^2} e^{-\frac{n^2}{2\sigma_1^2}-\frac{t^2+b^2}{2\sigma_2^2}},$$

to smooth the intensity $I(n,t,b)$ at v. Denote $\frac{\partial^2}{\partial n^2}G(n,t,b)$ by $G''_n(n,t,b)$, where

$$G''_n(n,t,b) = \frac{1}{(\sqrt{2\pi})^3\sigma_1\sigma_2^2}(1-\frac{n^2}{\sigma_1^2}) e^{-\frac{n^2}{2\sigma_1^2}-\frac{t^2+b^2}{2\sigma_2^2}}, \tag{4.22}$$

and the 3D convolution in (4.18) becomes:

$$w(n,t,b) = G''_n(n,t,b) * I(n,t,b)$$

$$= \int_{-3\sigma_1}^{3\sigma_1}\int_{-3\sigma_2}^{3\sigma_2}\int_{-3\sigma_2}^{3\sigma_2} \frac{1}{(\sqrt{2\pi})^3\sigma_1\sigma_2^2}(1-\frac{n^2}{\sigma_1^2}) e^{-\frac{(n-n')^2}{2\sigma_1^2}-\frac{(t-t')^2+(b-b')^2}{2\sigma_2^2}} I(n',t',b')\, dn'dt'db'.$$

σ_2 can take a rather small value, and the size of an asymmetric Gaussian filter is smaller than a symmetric one. As a result the computation of 3D convolution can be speeded up.

The analysis so far in continuous space is summarized as follows. The surface points are zero-crossings where the directional second derivatives of the intensity function are zero and change sign across the surface. The direction of the second derivative at a point v is the gradient direction of v. But the location of a zero-crossing is not sensitive to the direction of the second derivative. This property makes zero-crossings very competitive for identifying border voxels in discrete space. Finally, to reduce the computation cost, asymmetric Gaussian filters are used in 3D convolutions.

The border identification in the discrete space is introduced in the following sections.

4.5 A Border in Discrete Space

Because the data to be processed are defined on integer coordinates where they were collected, conditions (4.14) and (4.15) for identifying a surface in a continuous space must be adequately adapted to a discrete space. This section introduces the condition to identify border voxels in a discrete space.

Let $I(v) \in N$, $N = \{0, 1, 2, \ldots\}$, be an intensity function defined on a discrete domain, $v = (x, y, z) \in Z^3$, $Z = \{0, \pm 1, \pm 2, \ldots\}$. For simplicity, v is called a voxel. Let $G_g''(x, y, z)$ be a finite scale discrete Gaussian filter, resulting from sampling $G_g''(v)$ in (4.14) in a finite interval $(-3\sigma, 3\sigma)$, and $w(x, y, z)$ be the the discrete convolution of $G_g''(x, y, z)$ with $I(x, y, z)$,

$$w(x, y, z) = G_g''(x, y, z) * I(x, y, z). \tag{4.23}$$

Transform the coordinate system $\{0; x, y, z\}$ to $\{v; n, t, b\}$ and express the above convolution in terms of (n, t, b) as

$$\tilde{w}(n, t, b) = \sum_i \sum_j \sum_k G_n''(i, j, k) I(n - i, t - j, b - k), \tag{4.24}$$

where $G_n''(n, t, b)$ is the finite scale discrete Gaussian filter resulting from sampling the one in (4.22),

$$G_n''(n, t, b) = \sum_i \sum_j \sum_k G_n''(i, j, k) \delta(n - i, t - j, b - k). \tag{4.25}$$

At the origin v,

$$\tilde{w}(0, 0, 0) = w(x, y, z).$$

The $w(x, y, z)$ in (4.23) is defined on voxels of integer coordinates and is undefined between voxels. Because of the discrete nature of voxels, not many voxels have $w(x, y, z) = 0$. Condition (4.16) in Section 4.2, however, infers that for a bright object surrounded by a darker background and for those voxels close to a border, $w(x, y, z)$ is negative inside the object and positive outside. There exists exactly one layer of voxels on which $w(x, y, z)$ is negative and changes sign for neighbors in the gradient direction. This layer is called the **negative layer.** Similarly, there exists exactly one **positive layer** of voxels. It is possible to define either the negative layer or the positive layer as the border. Since the negative layer is part of the object, the border is defined as the negative layer of voxels. Zero-crossing voxels can be treated as either positive or negative, and are also included in the border set. Condition (4.14) therefore becomes $w(x, y, z) \leq 0$.

To express condition (4.15) in the discrete space, it is necessary to determine the neighbor voxels on which the condition needs to be tested.

The gradient of $I(v)$ at v is approximated by $\nabla_v = (\nabla_x, \nabla_y, \nabla_z)$ and is quantized to 26 directions, see page 3 Fig. 1.1. In each of the directions, voxels are either face connected, edge connected, or vertex connected. If $\nabla_v = (1, 0, 0)$ and v is on the border (the negative layer), the voxel $(x+1, y, z)$ is on the positive layer, $w(x+1, y, z) > 0$, and the voxel $(x-1, y, z)$ is inside, $w(x-1, y, z) < 0$. For $\nabla_v = (1, 1, 0)$, however, if v is on the border, the voxel $(x+1, y+1, z)$ is not on the positive layer but $(x+1, y, z)$ and $(x, y+1, z)$ are, as shown in Fig. 4.4. Therefore it is necessary to verify that $w(x-1, y, z) < 0$, $w(x, y-1, z) < 0$, $w(x+1, y, z) > 0$, and $w(x, y+1, z) > 0$. These cases reveal that the condition (4.15) should be tested on face connected neighbor voxels for every non-zero component of ∇_v.

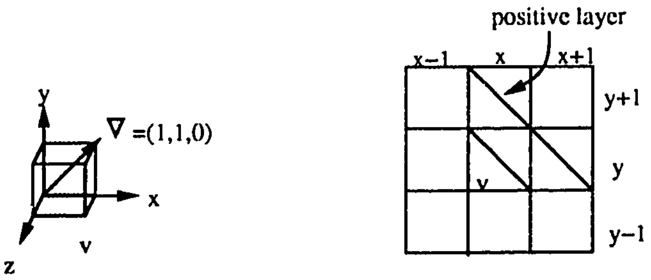


Figure 4.4: If the gradient components, ∇_x, ∇_y, of a border voxel v are non zero, $w(v)$ has to change sign in both x, y directions.

To summarize, for a bright object surrounded by a darker background, if $v(x, y, z)$ is a border voxel, it must simultaneously satisfy the following inequalities:

$$\begin{aligned} & w(x,y,z) \leq 0, \\ & w(x-1,y,z)\, w(x+1,y,z) < 0, \quad \text{if } \nabla_x \neq 0, \\ & w(x,y-1,z)\, w(x,y+1,z) < 0, \quad \text{if } \nabla_y \neq 0, \\ & w(x,y,z-1)\, w(x,y,z+1) < 0, \quad \text{if } \nabla_z \neq 0. \end{aligned} \tag{4.26}$$

Similarly, for a dark object surrounded by a brighter background, the border voxels can be defined as the positive layer of voxels.

The inequalities (4.26) are the condition to identify border voxels in a discrete space. Obviously, there is only one layer of voxels that will satisfy the condition. This makes subsequent surface tracking easy. Computation of $w(x,y,z)$ in a discrete space will be discussed in the next section.

4.6 The Design of the Discrete Filter

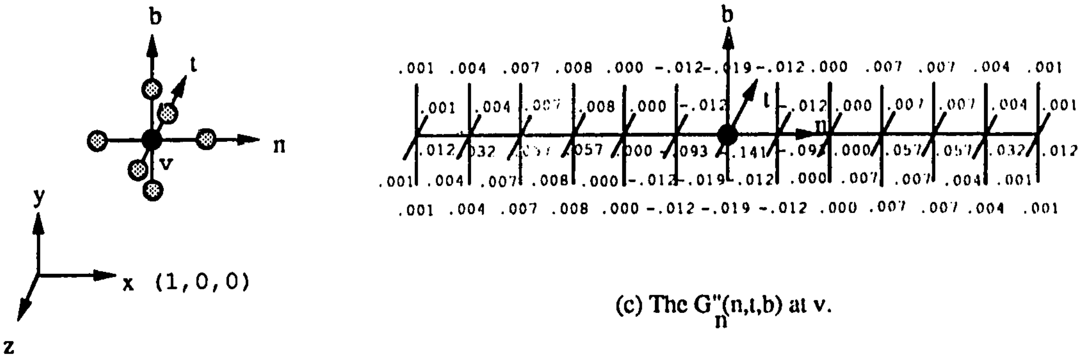
This section gives the setting of $\{v;n,t,b\}$ coordinate system at a voxel and the discrete asymmetric Gaussian filters used in the implementation.

To compute $\tilde{w}(x,y,z)$, a $\{v;n,t,b\}$ coordinate system is established at every voxel v. Figs. 4.5(a) to 4.7(a) show the three $\{v;n,t,b\}$ for three voxels with gradient vectors $(1,0,0)$, $(0,1,-1)$ and $(-1,1,-1)$. The $\{v;n,t,b\}$ for other gradient vectors are just 90 degree rotations of the three. The axis $\vec{n}$ at v is always set in the gradient direction of v, so the system $\{v;n,t,b\}$ changes from voxel to voxel.

Since G''_n is separable, $G''_n(n,t,b) = G''_n(n)G(t)G(b)$, the scale, or length, of G''_n in each dimension is the length of the corresponding one dimensional filter. The length of the one dimensional $G''_n(n)$ is determined by variance $\sigma 1$, the length of one dimensional $G(t)$ and $G(b)$ is determined by variance $\sigma 2$, $\sigma 1 > \sigma 2$. Figs. 4.5(b) to 4.7(b) depict $G''_n(n)$ of $\sigma 1 = 2.0$ and $G(t)$ of $\sigma 2 = 0.5$ for the three gradient vectors. The unit on the n or t axis is the distance between pair of voxels along the axis. For the gradient vector $(1,0,0)$ shown in Fig. 4.5, $G''_n(n)$ spans approximately 13 voxels, $G(t)$ and $G(b)$ spans approximately 3 voxels. For the gradient vector $(0,1,-1)$, however, the distance between pairs of voxel on the n axis is longer than that for $(1,0,0)$, so the $G''_n(n)$ in Fig. 4.6(b) is narrower, and the $G''_n(n)$ in Fig. 4.7(b)is even narrower. Consequently, the sum of the discrete convolution in (4.24) is over the range:

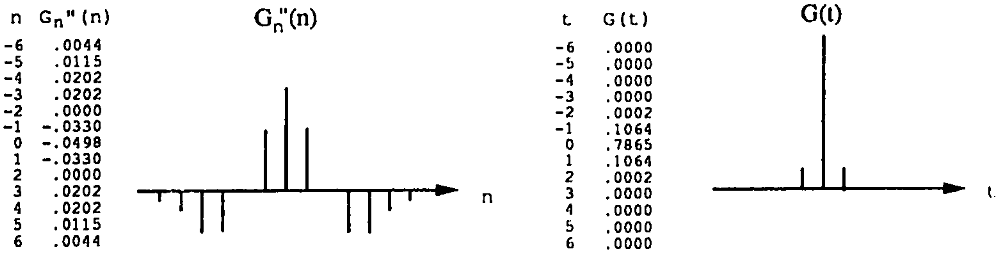
$$\tilde{w}(n,t,b) = \sum_{i=-6}^{6} \sum_{j=-1}^{1} \sum_{k=-1}^{1} G''_n(i,j,k) I(n-i, t-j, b-k).$$

Figs. 4.5(c) to 4.7(c) show the three discrete asymmetric $G''_n(n,t,b)$ filters of scales $\sigma_1 = 2.0$ and $\sigma_2 = 0.5$ for the three gradient vectors. The line segments in the Figures



(a) The {v;n,t,b}.

(c) The $G''_n(n,t,b)$ at v.



n	$G_n''(n)$
-6	.0044
-5	.0115
-4	.0202
-3	.0202
-2	.0000
-1	-.0330
0	-.0498
1	-.0330
2	.0000
3	.0202
4	.0202
5	.0115
6	.0044

t	G(t)
-6	.0000
-5	.0000
-4	.0000
-3	.0000
-2	.0002
-1	.1064
0	.7865
1	.1064
2	.0002
3	.0000
4	.0000
5	.0000
6	.0000

(b) The one dimensional $G_n''(n)$ and G(t).

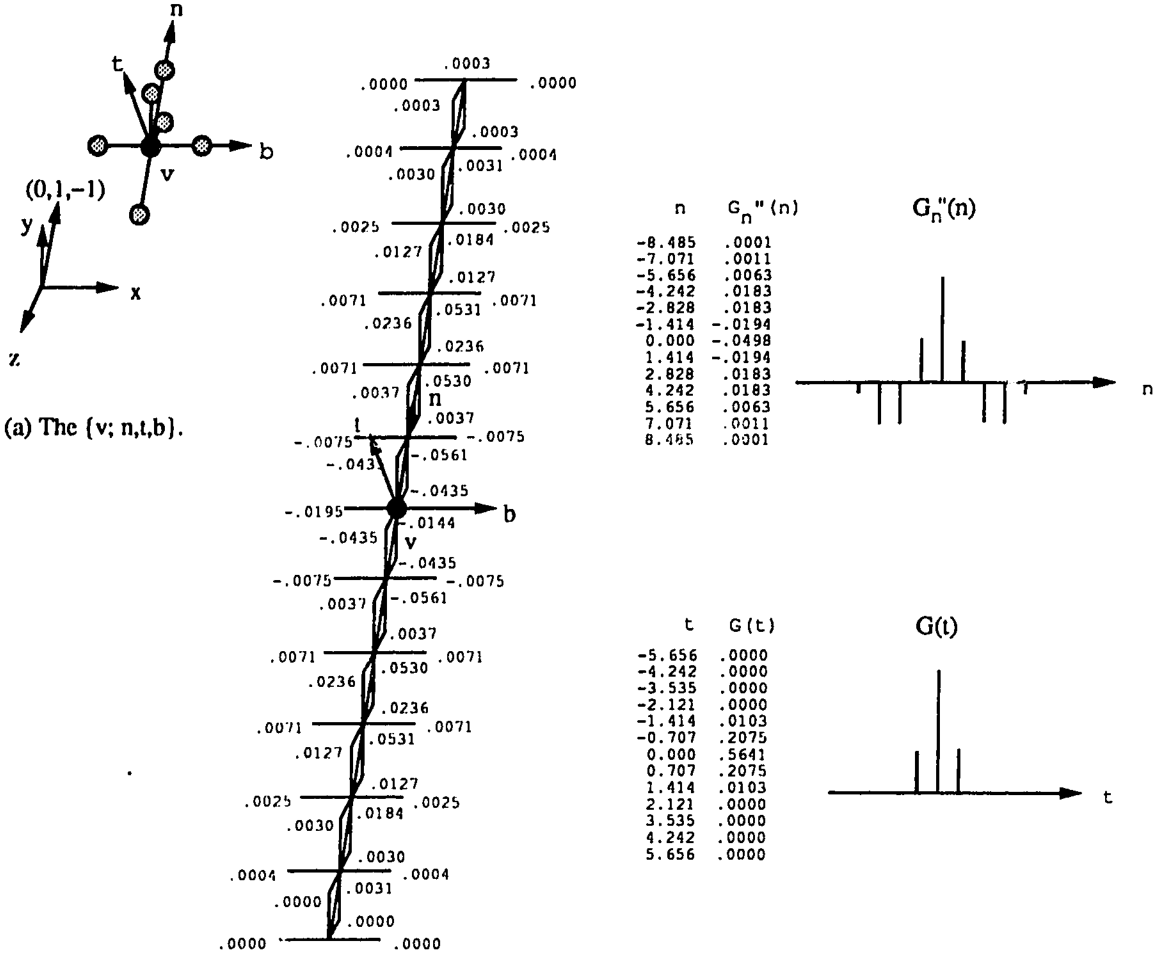
Figure 4.5: The $\{v; n, t, b\}$ system and the G''_n filter for the gradient vector $(1, 0, 0)$.

represent connections between voxels. There is a voxel located in every line intersection, join and end point. A weight, $G''_n(i, j, k)$, is labeled beside every such point.

The $G''_n(n, t, b)$ in Fig. 4.5(c) spans 6 face connected voxels each side along n axis, one voxel each side along t and b axes. So the filter size is 65.

The $G''_n(n, t, b)$ in Fig. 4.6(c) spans 6 edge connected voxels each side along n axis, one voxel each side along t axis, and one voxel each side along b axis. The filter size is 63.

The $G''_n(n, t, b)$ in Fig. 4.7(c) spans 6 vertex connected voxels each side along n axis, 3 voxels in tb plane, and totally 52 voxels.



n	$G_n''(n)$
-8.485	.0001
-7.071	.0011
-5.656	.0063
-4.242	.0183
-2.828	.0183
-1.414	-.0194
0.000	-.0498
1.414	-.0194
2.828	.0183
4.242	.0183
5.656	.0063
7.071	.0011
8.485	.0001

t	G(t)
-5.656	.0000
-4.242	.0000
-3.535	.0000
-2.121	.0000
-1.414	.0103
-0.707	.2075
0.000	.5641
0.707	.2075
1.414	.0103
2.121	.0000
3.535	.0000
4.242	.0000
5.656	.0000

(a) The {v; n,t,b}.

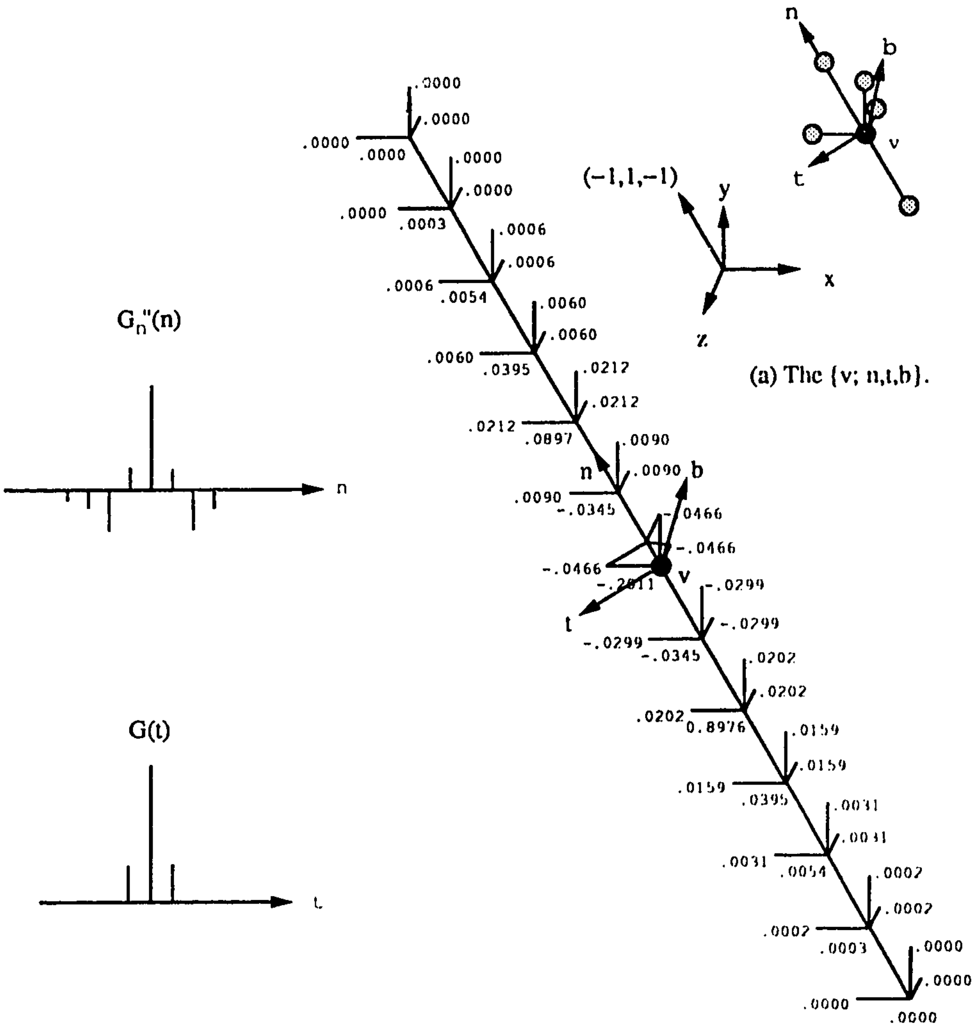
(b) The one dimensional $G_n''(n)$ and G(t).

(c) The $G_n''(n,t,b)$.

Figure 4.6: The $\{v; n, t, b\}$ system and the G_n'' filter for the gradient vector $(0, 1, -1)$.

n	$G_n''(n)$
-10.39	.0000
-8.660	.0000
-6.928	.0013
-5.191	.0098
-3.464	.0222
-1.732	-.0085
0.000	-.0498
1.732	-.0085
3.464	.0222
5.191	.0098
6.928	.0013
8.660	.0000
10.39	.0000

t	G(t)
-4.898	.0000
-4.082	.0000
-3.265	.0000
-2.449	.0000
-1.632	.0031
-0.816	.1715
0.000	.6506
0.816	.1715
1.632	.0031
2.449	.0000
3.265	.0000
4.082	.0000
4.898	.0000



(a) The {v; n,t,b}.

(b) The one demensional $G_n''(n)$ and G(t).

(c) The $G_n''(n,t,b)$.

Figure 4.7: The $\{v; n, t, b\}$ system and the G_n'' filter for the gradient vector $(-1, 1, -1)$.

4.7 The Implementation and Results

As shown in Fig. 4.8, there are basically two steps before computing $w(x, y, z)$. Procedure `read_data()` reads the data file named `object` into an unsigned char array `scene[][][]`. Procedure `edge_detector()` applies a 3D edge operator to the data array `scene[][][]`. The results are an unsigned char array `grad_loc_dir[][][]` which stores the gradient direction/location code of every voxel, and a short integer array `grad_mag[][][]` recording gradient magnitudes at every voxel. Procedure `compute_w()` computes $w(x, y, z)$. The results are stored in a short integer array `w[][][]`.

```
#define         clen            128
#define         rlen            128
#define         zlen            32

unsigned char scene[zlen][rlen][clen];
unsigned char grad_loc_dir[zlen][rlen][clen];
short    grad_mag[zlen][rlen][clen],w[zlen][rlen][clen];

extern  void read_data(),edge_detector(),compute_w();

preprocess(char *object)
{
     read_data(object);              /* read data into memory      */
     edge_detector();                /* apply 3d edge operator     */
     compute_w();                    /* compute w(x,y,z)           */
}
```

Figure 4.8: Steps to compute $w(x, y, z)$.

The procedure `compute_w()`, see below, calls `gaussian_1()`, `gaussian_2()`, and `gaussian_3()` to compute the three Gaussian filters. The resulting filters are stored in three double arrays `G1_3[][][]`, `G2_3[][][]`, and `G3_3[][][]`. Next, for every voxel `(i,j,k)` with non zero gradient magnitude, the micro `get_increaments()` is called to compute gradient components along each axis. In the micro, arguments `i,j,k` are voxel coordinates, arguments `ii,jj,kk` are gradient components in axis `i,j,k`, respectively and are either 1, -1, or 0. Argument `n` is the number of non zero gradient components. The procedure then computes `w[i][j][k]` by calling either `convolution_1()`, `convolution_2()`, or `convolution_3()` depending on `n`. If n is one, `convolution_1()` is called to compute convolution with filter `G1_3`. Else if n is two, `convolution_2()` is called to compute convolution with `G2_3`. Else if n is three, `convolution_3()` is called to compute with `G3_3`. The result is rounded to a short integer and stored in `w[i][j][k]`.

```
#define         N       21
#define         T       5

double G1_3[T][T][N],G2_3[T][T][N],G3_3[T][T][N];/*Gaussian filters*/

int R_LEN,C_LEN,Z_LEN;                                  /* data size     */
int xml, yml, zml;                                      /* left margins  */
int xmr, ymr, zmr;                                      /* right margins */

void gaussian_1(),gaussian_2(),gaussian_3();

void compute_w(void)
{
short   i,j,k,ii,jj,kk,n;
double  convolution_1(),convolution_2(),convolution_3();

gaussian_1();gaussian_2();gaussian_3(); /*compute G2_1,G3_2,G3_3*/

for(i=zml; i<=Z_LEN-zmr; i++)
 for(j=yml; j<=R_LEN-ymr; j++)
  for(k=xml; k<=C_LEN-xmr; k++)
    if(grad_mag[i][j][k]){
      get_increments(i,j,k,ii,jj,kk,n); /* gradient components  */
      if(n==0)      continue;
      else if(n==1) w[i][j][k]=(short)convolution_1(i,j,k,ii,jj,kk);
      else if(n==2) w[i][j][k]=(short)convolution_2(i,j,k,ii,jj,kk);
      else if(n==3) w[i][j][k]=(short)convolution_3(i,j,k,ii,jj,kk);
    }
}
```

Figure 4.9: The procedure compute_w().

Fig. 4.10 shows one slice of an experimental result of $w(x, y, z)$ on test data of volume $40 \times 40 \times 40$. The object is a sphere of radius 7 with 16 values. The sphere has gray value 0 and the background 16. Since the sphere is darker than the background, the border is chosen to be the positive layer of voxels.

```
w(x,y,z) slice y = 21

\ x  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
z
10    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
11    0  0  0  0  0  0  0  0  0  0 -4  0  0  0  0  0  0  0  0  0  0
12    0  0  0  0  0 -5 -5 -8 -7 -7 -7 -7 -7 -8 -5 -5  0  0  0  0  0
13    0  0  0  0 -6 -6 -4 -4 -3 -2 -2 -2 -3 -4 -4 -6 -6  0  0  0  0
14    0  0  0 -6 -6 -4 -1  1  3  4  4  4  3  1 -1 -4 -6 -6  0  0  0
15    0  0 -5 -6 -4 -1  2  5  8  8  8  8  8  5  2 -1 -4 -6 -5  0  0
16    0  0 -5 -4 -1  2  5  6  6  0  0  0  6  6  5  2 -1 -4 -5  0  0
17    0  0 -8 -4  1  5  6  6  0  0  0  0  0  6  6  5  1 -4 -8  0  0
18    0  0 -7 -3  3  8  6  0  0  0  0  0  0  0  6  8  3 -3 -7  0  0
19    0  0 -7 -2  4  8  0  0  0  0  0  0  0  0  0  8  4 -2 -7  0  0
20    0 -4 -7 -2  4  8  0  0  0  0  0  0  0  0  0  8  4 -2 -7 -4  0
21    0  0 -7 -2  4  8  0  0  0  0  0  0  0  0  0  8  4 -2 -7  0  0
22    0  0 -7 -3  3  8  6  0  0  0  0  0  0  0  6  8  3 -3 -7  0  0
23    0  0 -8 -4  1  5  6  6  0  0  0  0  0  6  6  5  1 -4 -8  0  0
24    0  0 -5 -4 -1  2  5  6  6  0  0  0  6  6  5  2 -1 -4 -5  0  0
25    0  0 -5 -6 -4 -1  2  5  8  8  8  8  8  5  2 -1 -4 -6 -5  0  0
26    0  0  0 -6 -6 -4 -1  1  3  4  4  4  3  1 -1 -4 -6 -6  0  0  0
27    0  0  0  0 -6 -6 -4 -4 -3 -2 -2 -2 -3 -4 -4 -6 -6  0  0  0  0
28    0  0  0  0  0 -5 -5 -8 -7 -7 -7 -7 -7 -8 -5 -5  0  0  0  0  0
29    0  0  0  0  0  0  0  0  0  0 -4  0  0  0  0  0  0  0  0  0  0
30    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Figure 4.10: Sphere slice y = 21.

Fig. 4.11 shows the border voxels of the sphere slice identified by the condition (4.26). The positive layer of border voxels is labeled /, —, \\, and |, which signify the gradient orientations. For instance, for the voxel $x = 23$, $y = 21$, $z = 14$, its $w(23, 21, 14) = 1$ which is positive. Its gradient is directed at $\swarrow$. To be a border voxel, its neighbor'w has to change sign in both ← and ↓ direction. Examining the horizontal neighbors shows that the w values change sign from negative to positive. The vertical neighbors'w also changes sign from negative to positive. Since the $w(23, 21, 14) > 0$ and its neighbor changes sign in the gradient direction, the voxel (23,21,14) satisfies inequalities (4.26). It is a border voxel. For those border voxels with non-zero ∇_y, the w values of their y-neighbors have to change sign as well.

Except the labeled border voxels layer, none of the remaining voxels with positive w values having neighbors change sign in the gradient direction. So none of them are on the border. From this experimental result it clearly shows that there is only one positive layer of border voxels.

```
w(x,y,z) slice y = 21

\ x 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
z
10   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
11   0  0  0  0  0  0  0  0  0  0 -4  0  0  0  0  0  0  0  0  0  0
12   0  0  0  0  0 -5 -5 -8 -7 -7 -7 -7 -7 -8 -5 -5  0  0  0  0  0
13   0  0  0  0 -6 -6 -4 -4 -3 -2 -2 -2 -3 -4 -4 -6 -6  0  0  0  0
14   0  0  0 -6 -6 -4 -1  \  |  |  |  |  |  / -1 -4 -6 -6  0  0  0
15   0  0 -5 -6 -4 -1  \  5  8  8  8  8  8  5  / -1 -4 -6 -5  0  0
16   0  0 -5 -4 -1  \  5  6  6  0  0  0  6  6  5  / -1 -4 -5  0  0
17   0  0 -8 -4  \  5  6  6  0  0  0  0  0  6  6  5  / -4 -8  0  0
18   0  0 -7 -3  -  8  6  0  0  0  0  0  0  0  6  8  - -3 -7  0  0
19   0  0 -7 -2  -  8  0  0  0  0  0  0  0  0  0  8  - -2 -7  0  0
20   0 -4 -7 -2  -  8  0  0  0  0  0  0  0  0  0  8  - -2 -7 -4  0
21   0  0 -7 -2  -  8  0  0  0  0  0  0  0  0  0  8  - -2 -7  0  0
22   0  0 -7 -3  -  8  6  0  0  0  0  0  0  0  6  8  - -3 -7  0  0
23   0  0 -8 -4  /  5  6  6  0  0  0  0  0  6  6  5  - -4 -8  0  0
24   0  0 -5 -4 -1  /  5  6  6  0  0  0  6  6  5  \ -1 -4 -5  0  0
25   0  0 -5 -6 -4 -1  /  5  8  8  8  8  8  5  \ -1 -4 -6 -5  0  0
26   0  0  0 -6 -6 -4 -1  /  |  |  |  |  |  \ -1 -4 -6 -6  0  0  0
27   0  0  0  0 -6 -6 -4 -4 -3 -2 -2 -2 -3 -4 -4 -6 -6  0  0  0  0
28   0  0  0  0  0 -5 -5 -8 -7 -7 -7 -7 -7 -8 -5 -5  0  0  0  0  0
29   0  0  0  0  0  0  0  0  0  0 -4  0  0  0  0  0  0  0  0  0  0
30   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Figure 4.11: Border voxels of sphere slice y = 21

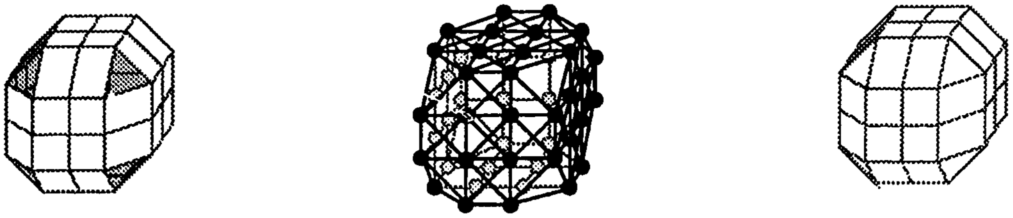
Chapter 5

The Surface Tracking Algorithm

The surface tracking algorithm traverses border voxels, converts border voxels to face primitives as defined in the extended cuberille model, and connects the faces in a closed surface.

(a) An object defined by thresholding.

(b) The object border voxels defined by conditions (4.26).



(c) The border voxels converted to face primitives defined in the extended cubrille model.

(d) The embedded graph where nodes are border voxels and edges are adjacency between pairs of voxels.

(e) Surface closure.

Figure 5.1: An example to show the surface tracking steps.

An example is given in Fig. 5.1 (a) to (e) to demonstrate how an object is converted to a surface. The object is a 4 × 4 × 4 cube defined by thresholding. According to condition (4.26), however, the object border is the set of voxels shown in (b). The border voxels are converted to the extended cuberille model in (c), where a border voxel is represented by a face. Because each voxel is converted to one face primitive, the resulting surface in (c) is not closed. There is therefore one more step to fill missing faces to close the surface, as shown in (e).

5.1 The Tasks of Surface Tracking

The surface tracking algorithm is outlined in Fig. 5.2. The algorithm starts with three sets of data: the loc_dir codes, the gradient magnitudes, and the w values, and concludes with a surface description of an identified object. It has basically three tasks: calling procedure `Border_face_tracking()` in line 4 to traverse border voxel faces and test face connections, and calling procedure `Close_surface()` in line 5 to connect border voxel faces to close a surface.

```
    Surface_tracking()
    {
1       read_data();                    /* read data into memory       */
2       init_lists();                   /* initialize the queue, lists */
3       make_table();                   /* initialize the table        */

4       Border_face_tracking();         /* track border voxel faces    */
5       Close_surface();                /* close surface               */
    }
```

Figure 5.2: The surface tracking algorithm

The procedure `read_data()` in line 1 reads three sets of data into memory. The loc_dir codes are read into a 3D array `grad_loc_dir` of type `unsigned char`. The gradient magnitudes, used as a rough threshold to assist in border voxel identification, are read into a `short` integer array `grad_mag`. These two data sets were results of calling procedure `edge_detector()` (see page 62). $w(x, y, z)$ values, resulting from calling procedure `compute_w()`, are read into a short integer array `w`. These procedures have been discussed in Chapter 4.

In lines 2 to 3, procedures `init_lists()` and `make_table()` initialize all associated data structures, such as queue, lists, tables, and trie used by the border face tracking and the surface closure algorithms.

The surface tracking algorithm consists of several algorithms, and there are many data structures to implement the algorithm. Therefore, discussion of the algorithm undergoes several chapters. In this chapter, the border face tracking algorithm is discussed. In Chapter 6, the method and implementation to test voxel face connection are discussed. Chapter 7 will discuss the surface closure algorithm. Experimental results are given in Chapter 8.

5.2 The Border Face Tracking Algorithm

As shown in Fig. 5.1(d), if border voxels and adjacency relations between voxels can be interpreted as a graph, where nodes are border voxels and edges are adjacency relations between pairs of voxels, a standard breadth-first search algorithm can be used to track border voxels. The procedure is outlined in Fig. 5.3.

The underlying data structure is a queue, where each queue cell has three integers, `x,y,z`, for recording the coordinates of a border voxel. The type definition, `Q_CELL`, is given in page 83.

In the procedure, the micro `get_start_face` in line 1 reads the coordinates of a start border voxel, and the procedure `span_start_face()` searches its adjacent border voxels, marks them and queues them.

The `while` loop in line 4 starts tracking border voxels. A cell is obtained from the queue as the current voxel. In line 9, the `Neighbor_face()` is called to search for the current voxel's adjacent border voxels. It scans all adjacent voxels, testing if any satisfies the condition (4.26). Meanwhile if an adjacent border voxel is unmarked, mark it and queue it. After the procedure `Neighbor_face()` returns, `add_to_display_table()` is

```
int      x,y,z;                        /*current voxel coordinates*/
Q_CELL   *current_voxel;

Border_face_tracking()
{
  unsigned char   index;               /* the table index          */

  get_start_face;                      /* start voxel coordinates */
  span_start_face();                   /* span the start voxel    */

  while(current_voxel = remove_q_head() {
    x=current_voxel->x;
    y=current_voxel->y;
    z=current_voxel->z;
    index=grad_loc_dir[x][y][z] & ~MARK;
    Neighbor_face(index);               /* search neighbor voxels */
    add_to_display_table(x,y,z,index);  /* add to display table   */
  }
}
```

Figure 5.3: The border face tracking procedure.

called to add the current voxel coordinates, **x y z**, and its loc_dir code, **index**, to a structure array for displaying the face later. While the queue is not empty, the loop continues to the next cell in the queue. The **while** loop stops once the queue is empty.

The time complexity of **Border_face_tracking()** depends on the **while** loop and the procedure call **Neighbor_face()**. It is analyzed below. The data to start tracking are three arrays, **grad_loc_dir**, **grad_mag**, and **w**. Bit 7 of **grad_loc_dir** is designated for marking, hence checking and marking a voxel visited can be done in constant time. As a result, the time to execute **Neighbor_face()** depends on the number of adjacent voxels that a border voxel could have. Since each unmarked border voxel is placed in the queue once, the **while** loop is executed only once for every border voxel. Denote the number of adjacent voxels that a border voxel has as k, and the total number of the border voxels as n_B, the time complexity of **Border_face_tracking()** is therefore $O(kn_B)$.

For a given object, n_B is determined by condition (4.26) and is fixed. But k varies with the number of adjacent voxels that a border voxel could have. A way to define the adjacency relations between pairs of voxels is by the digital topology [KR89], where two voxels are adjacent to each other if they are either face, edge or vertex connected. By this definition, each voxel has 26 adjacent voxels. Hence the constant k is about 26.

Observe from Fig. 5.1 (c), however, that in the extended cuberille model, every border voxel can be converted to a face, and all the faces are on a surface. Intuitively, a face normal shouldn't have a dramatic change from one border voxel to an adjacent one because the surface should be smooth. This suggests that the face normal can be used to assist in searching adjacent voxels. In the next section, adjacent voxels and related definitions based on a given face normal will be introduced. It results in less than 26 adjacent voxels. and the algorithm will track border voxel **faces** instead of border voxels

Since face primitives are either square or triangular, see Fig. 3.1 page 37, a face primitive has at most four edges. There are at most four ways to connect the current face to the next face. The neighbor connections are called **outways** of a border voxel face. The border face tracking algorithm traverses outways instead of adjacent voxels. Whenever a border voxel of an outway is found, the search breaks and proceeds to the next outway. This reduces the constant k to about half, and hence speeds up border face tracking.

5.3 Related Definitions

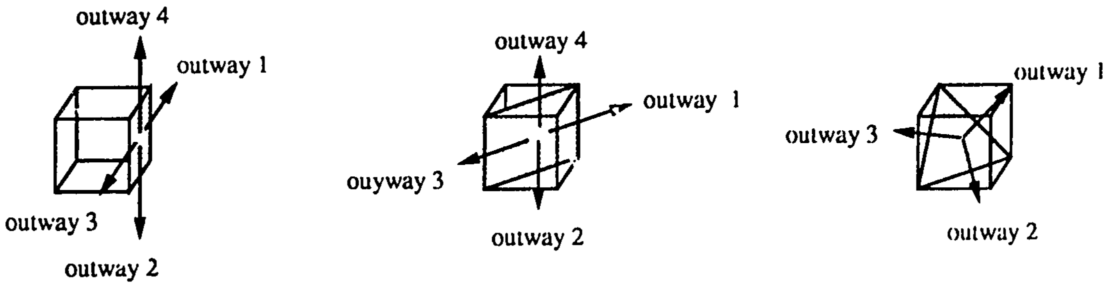
This section introduces definitions of the current voxel and the current face, outways, adjacent voxels, and neighbors for `three` face primitives used in the border face tracking algorithm.

5.3.1 The Current Voxel and the Current Face

As shown in Fig. 5.3, the border face tracking algorithm is a breadth first search and the data structure is a queue of border voxels. While tracking, the algorithm obtains a voxel from the queue, and this voxel is called the **current voxel**. The current voxel is converted to a face by its loc_dir code. This face is called the **current face**. The algorithm then traverses the adjacent border voxels of the current face by calling procedure `Neighbor_face()`.

5.3.2 Outways of the Current Voxel

As shown in Fig. 3.1, face primitives in the extended cuberille model are either square or triangular, and a face primitive has at most four edges. As a result, there are at most four ways to connect the current face to the next face. Equivalently, there are at most four ways to traverse from the current voxel to the next voxel, and each possibility crosses an edge. The ways of traversing are called **outways** of the current voxel. Fig. 5.4 shows the four outways of a type_1 face, the four outways of a type_2 face, and the three outways of a type_3 outside face.



(1) The four outways of a type_1 face.

(2) The four outways of a type_2 face.

(3) The three outways of a type_3 outside face.

Figure 5.4: The outways of a type_1 face, a type_2 face, and a type_3 outside face.

5.3.3 Adjacent Voxels of an Outway

Those voxels located in front of an outway may be defined as the **adjacent voxels** of the outway. Searching adjacent voxels can therefore be divided to searching outways. Observe Fig. 5.4 that outways of a type_1 face are symmetric. It is enough to define adjacent voxels for the first outway. For the other three outways, adjacent voxels can be obtained by 90 degree rotations of those of the first outway. Similar reasoning applies for a type_3 face. For a type_2 face, outway 1 and 3 are symmetric, and outway 2 and 4 are symmetric. It is necessary to define adjacent voxels for outways 1 and 2.

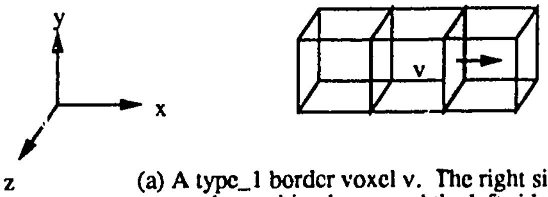
Defining adjacent voxels for three face primitives used in the border face tracking algorithm is based on the following assumption of a smooth surface:

Assumption 5.1 *The face normal difference between the current border voxel and an adjacent voxel is less than 90 degrees.*

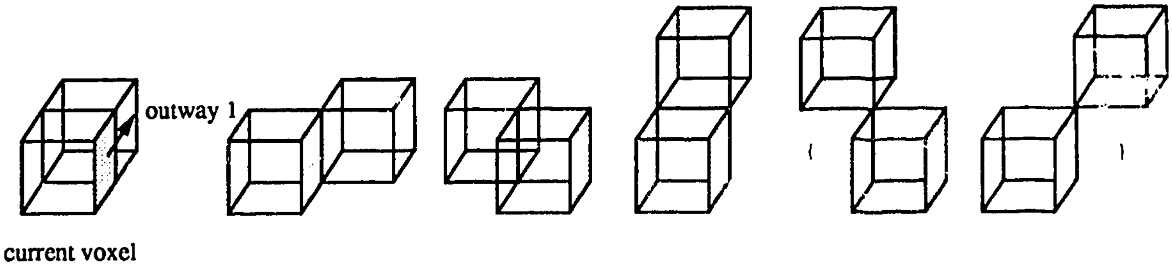
Since searching border voxels is the main concern of the border face tracking algorithm, among the 26 face, edge, or vertex connected voxels, only those voxels possible on a border are defined as adjacent voxels. It will be shown in the following that a type_1 face has 24 adjacent voxels, a type_2 face has 22 adjacent voxels, and a type_3 face has 18 adjacent voxels.

Suppose voxel v is a type_1 border voxel as shown in Fig. 5.5(a). If the border is defined as the negative layer, according to condition (4.26), the voxel on the right hand side is on the positive layer. The voxel on the left hand side has negative w value. If it is on the border, under Assumption 5.1, the voxel v would have $w(v) > 0$. But this contradicts the fact that voxel v is on the border. Hence the left voxel is inside. This leaves 24 face, edge, and vertex connected voxels. Because four outways of the type_1 face are symmetric, the 24 voxels are divided to 4 disjoint sets, with one for each outway. Fig. 5.5(b) shows the adjacent voxels defined for outway 1 of the type_1 face.

For a type_2 border voxel v, see Fig. 5.6(a), if v is on a border of a negative layer, Voxels $v1$ and $v2$ are on the positive layer. Voxels $v3$ and $v4$ are not on the border under Assumption 5.1. It leaves 22 face, edge, and vertex connected voxels. Dividing them



(a) A type_1 border voxel v. The right side voxel of v is on the positive layer, and the left side voxel is inside.



(b) Six adjacent voxels defined for outway 1 of a type_1 face.

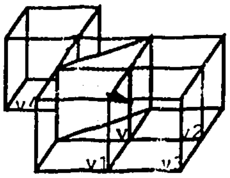
Figure 5.5: The adjacent voxels defined for outway 1 of a type_1 face.

into four disjoint sets for four outways results in four adjacent voxel sets. Figs. 5.6(b) and (c) show the adjacent voxels defined for outway 1 and outway 2 of the type_2 face.

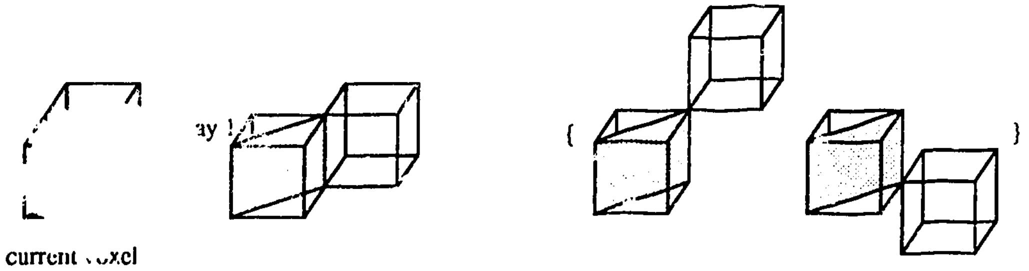
For a type_3 border voxel v, see Fig. 5.7(a), if v is on a border of a negative layer, v_1 v_2 and v_3 are on the positive layer. By the same argument, voxels v_4 to v_8 are not on the border. It leaves 18 face, edge, and vertex connected voxels that are divided into three adjacent voxel sets, with one for each outway. Fig. 5.7(b) shows the six adjacent voxels for outway 1 of the type_3 face.

Adjacent voxels of each outway of the three faces are in turn divided to subsets. A subset with more than one voxel is enclosed by braces. Duing border face tracking, adjacent voxels of a subset will be searched as an unit.

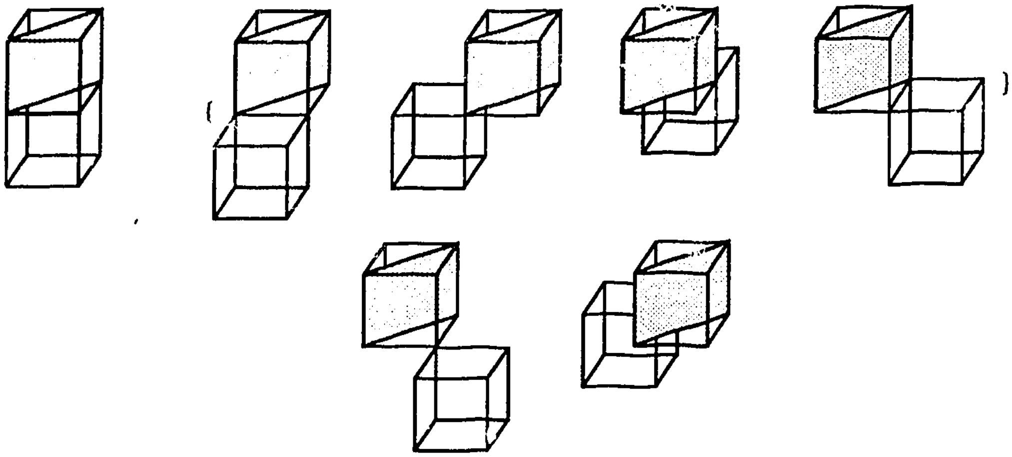
A type_1 face have six orientations. A type_2 face have 12 orientations, and a type_3 face 8 orientations. The adjacent voxels for any face orientation can be obtained by 90 degree rotations of the three given in Fig. 5.5 to Fig. 5.7. The rotation matrices, the adjacent voxel coordinates, etc. are stored in a table of structure type TABLE. The TABLE structure will be given in Section 5.4.



(a) A type_2 border voxel v. Voxels v1 and v2 are on the positive layer. Voxel v3 and v4 are not on the border.

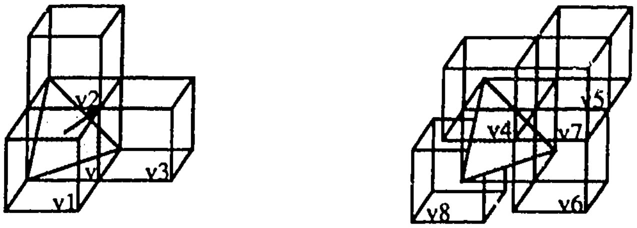


(b) Four adjacent voxels of outway 1 of the type_2 face.

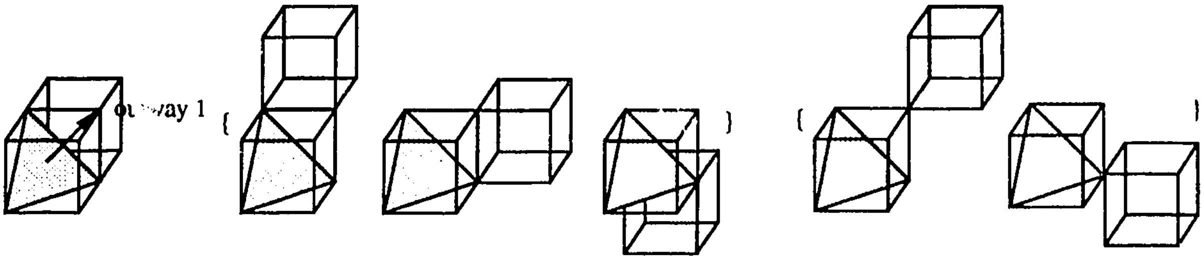


(c) Seven adjacent voxels for outway 2 of the type_2 face.

Figure 5.6: The adjacent voxels defined for a type_2 face.



(a) A type_3 border voxel v. Voxels v1, v2 and v3 are on the positive layer. Voxels v4 to v8 are not on the border.



(b) Six adjacent voxels for outway 1 of the type_3 face.

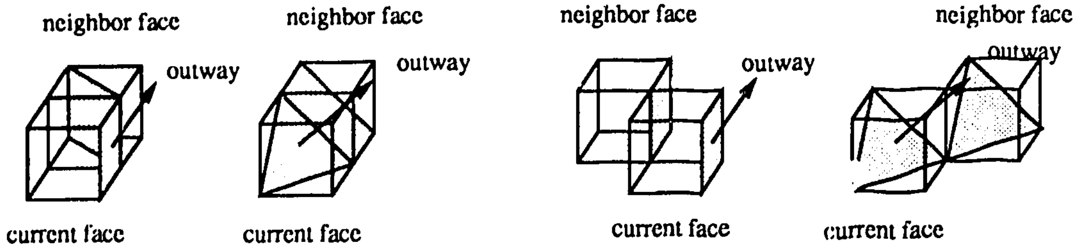
Figure 5.7: Adjacent voxels defined for a type_3 face.

5.3.4 Neighbors and Neighbor Faces of an Outway

Among the adjacent voxels of an outway, those voxels located on a border are called **neighbors** of the outway. A neighbor voxel is converted to a face by its loc_dir code. This face is called a **neighbor face** of the outway.

5.3.5 Connected Faces

In the extended cuberille model, each voxel is converted to one face primitive. As a consequence, a neighbor face may not be connected to the current face. If the current face and a neighbor face share a common edge, they are said to be **connected** to each other. Otherwise, the two faces are not connected. Fig 5.8 shows some examples. In case (a), neighbor faces are connected to the current face. In case (b), neighbor faces are not connected to the current face. The close surface algorithm will solve the surface closure problem.



(a) The neighbor faces are connected to the current face.

(b) The neighbor faces are not connected to the current face.

Figure 5.8: A neighbor face may not be connected to the current face.

5.4 The TABLE structure

Outways and adjacent voxels are stored in a `table` of type `TABLE`, and are accessible by a face's loc_dir code. The `TABLE` type definition and the `table` structure are given in Fig. 5.9 and Fig. 5.10.

As shown in Fig 5.9, the `table` has 128 entries. The seven bit table index corresponds to the loc_dir code. A `table` entry has three fields: a pointer `face` pointing to the `BASIC_FACE` structure of the face, a 4 × 4 rotation `matrix`, and the normal components `nx, ny, nz`. Only the three face primitives oriented at the direction shown in Fig 5.4 are stored in the `BASIC_FACE` structures. Other face orientations can be obtained by multiplying with the rotation matrices in the table entries.

The `BASIC_FACE` structure has: an integer `n_of_ways` indicating the number of outways of a face, and a pointer `outways` pointing to an array of `OUTWAY` structures. It also includes the number of vertices, and three arrays of the vertex coordinates, `*ix, *iy, *iz`, that are necessary for display a face.

The `OUTWAY` structure includes a rotation `matrix`, the number of subsets, and a pointer `subset` pointing to an array of `subsets` structures. Again, the `subsets` structure has an integer indicating the number of adjacent voxels in the subset, and a pointer `adj_voxels` pointing to an `ADJ_VOXEL` structure array where the adjacent voxels in the subset are stored, etc. For a type_1 and a type_3 face, all outways are symmetric. Therefore, only the subsets of the first outway are stored and pointed to by the `subset`

```
typedef         float                           MATRIX[4][4];

typedef struct lookup_table   {
        MATRIX          matrix;         /* rotation matrix      */
        BASIC_FACE      *face;
        double          nx,ny,nz;       /* normal components    */
} TABLE table[128];

typedef struct basic_face     {
        int             type;           /* face type            */
        int             n_of_ways;      /* number of outways    */
        OUTWAY          *outways;
        int             n_of_vertices;  /* number of vertices   */
        float           *ix,*iy,*iz;    /* vertex coordinates   */
        float           cx,cy,cz;       /* center coordinates   */
} BASIC_FACE;
typedef struct outcoming_way  {
        MATRIX          matrix;         /* rotation matrix      */
        int             edge_type;
        int             n_of_subsets;   /* number of subsets    */
        struct subsets  *subset;
        COORDINATES     *bounding_voxel;
} OUTWAY;
typedef struct subsets        {
        int             n_of_adj_voxels;/* number of adj voxels */
        ADJ_VOXEL       *adj_voxels;
};
typedef struct adjacent_voxel {
        float           ix,iy,iz;       /* voxel coordinates    */
        int             n_of_con_faces; /* number of con faces  */
        CON_FACE        *con_faces;
        int             n_of_dis_faces; /* number of discon faces*/
        DIS_FACE        *dis_faces;
        int             b_voxel_type;
        COORDINATES     bounding_voxel;
} ADJ_VOXEL;
```

Figure 5.9: The TABLE type definition.

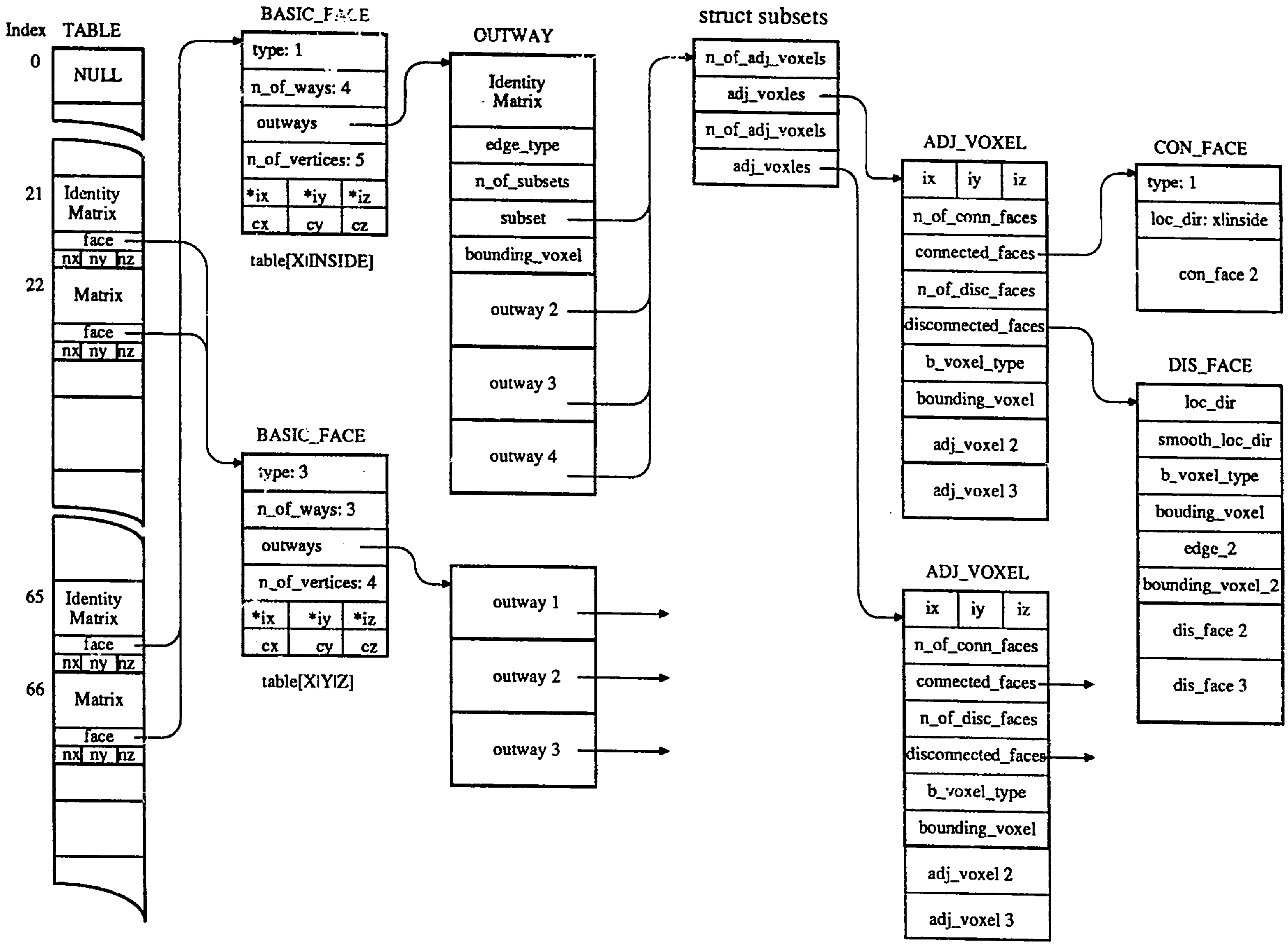


Figure 5.10: The table structure.

pointers of other outways. The adjacent voxel coordinates of other outways can be obtained by multiplying the rotation `matrix` in the corresponding `OUTWAY` entry. For a type_2 face, adjacent voxels for the first and the second outway are stored in two `subsets` structures .

Besides voxel coordinates, the `ADJ_VOXEL` structure includes two more structures, `CON_FACE` and `DIS_FACE`, storing possible neighbor faces that are connected or disconnected to the current face. Their usage will be discussed with the close surface algorithm in Chapter 7.

5.5 Tracking Neighbor Faces

This section discusses the `Neighbor_face()` procedure, which is called by the border face tracking algorithm in page 69. The procedure is outlined in Fig 5.11.

The purpose of the procedure is to search outways of the current voxel for neighbors and to check connections between the current face and neighbor faces. Two `for` loops in line 6 and line 9 serve the two purposes.

In the procedure, the argument `index` is the current voxel's loc_dir code, which is used to index the table entry in line 1. Line 2 allocates enough memory to hold the biggest subset of each outway, and initializes associated flags. In line 6, procedure `Search_neighbor()` is called for each outway. It searches adjacent voxel subsets for neighbors. The neighbor information is saved for checking face connection in the second `for` loop. The procedure `Face_connection()` in the second `for` loop will be discussed with the close surface algorithm in Chapter 7.

The procedure `Search_neighbor()` is outlined in Fig 5.12. It has four arguments: `outway` is the specified outway; `index` is the current voxel's loc_dir code; `w` is a pointer pointing to the corresponding outway structure in the `table`; `n_w` is a pointer in the calling procedure, pointing to a piece of memory for saving neighbor information of the outway.

In the procedure, line 1 copies the rotation matrix to `final_m`, which is a concate-

```
typedef          ADJ_VOXEL                    NEIGHBOR;

BASIC_FACE       *f;

void Neighbor_face(unsigned char index)
{
    int              i;
    OUTWAY           *w;
    NEIGHBOR         *n,*n_w;

    f = table[index].face;

    /* allocate  memory for neighbors, initialize flags        */

    n_w = (NEIGHBOR *)calloc(f->n_of_ways*4,sizeof(NEIGHBOR));
    for(n=n_w, i=0; i<f->n_of_ways; n++,i++){
        init_neighbors_and_flags;
    }

    /* search every outway of the current voxel for neighbors */

    for(n=n_w,w=f->outways,i=0; i<f->n_of_ways; w++,n+=4,i++){
        Search_neighbor(i,index,w,n);


    /* test face connections for every outway                  */

    for(n=n_w,w=f->outways,i=0; i<f->n_of_ways; w++,n+=4,i++){
        Face_connection(i,index,w,n);
    }
}
```

Figure 5.11: The Neighbor_face() procedure.

```
extern int gr_threshold;                 /* a rough threshold  */
int        n_n[4][4],n_set[4],n_num[4];
MATRIX     final_m[4];                   /* rotation matrix    */

Search_neighbor(int outway,unsigned char index,OUTWAY *w,NEIGHBOR *n_w)
 {
   int             j,k,l;
   int             tx,ty,tz;             /* adjacent voxel coord*/
   struct subsets  *s;
   NEIGHBOR        *n_tab,*n;

   /* final_m concatenates table[index].matrix and w->matrix */
   copy_matrix(outway_matrix[index*4+outway],final_m[outway],d);

   for(s=w->subset,n=n_w,j=0; j<w->n_of_subsets; s++,j++){
    for(n_tab=s->adj_voxels,k=0,l=0;k<s->n_of_adj_voxels;n_tab++,k++){
       calc_adj_voxel_coords(tx,ty,tz);

       /* test if the adjacent voxel satisfies conditions (4.26) */
       if(grad_mag[tx][ty][tz] < gr_threshold || w[tx][ty][tz] > 0
                               || !( cross_zero(tx,ty,tz,-1)) ) continue;
       n_n[outway][l++] = k; n_num[outway] += 1;

       /* save neighbor coordinates of the outway                 */
       save_neighbor_coordinates(n);

       /* if the neighbor is unmarked, mark it and queue it       */
       if(!(grad_loc_dir[(int)n->ix][(int)n->iy][(int)n->iz] & MARK)){
            grad_loc_dir[(int)n->ix][(int)n->iy][(int)n->iz] |= MARK;
            (void)append_queue();
            n++; }
     }
     if(n_num[outway]) {       /* as least one neighbor is found */
            n_set[outway] = j; break;
     }
   }
}
```

Figure 5.12: The procedure Search_neighbor().

nation of two matrices: the matrix in the indexed table entry and the matrix in the outway entry. Rotation matrices for every outway of every face orientation were precalculated during initialization, and stored in a global matrix array `outway_matrix`. All matrices are 4 × 4, indicated by an externally declared variable d in the `copy_matrix()` procedure. Since all matrices are 90 degrees rotation matrices, the matrix elements are either 0 , 1, or -1.

In lines 2 and 3, for each subset of the `outway` and for each adjacent voxel of a subset, the function `calc_adj_voxel_coords` is called to multiply the rotation matrix `final_m[outway]` with the voxel coordinates read from the `table`. The results are saved in three integers, `tx,ty,tz`.

Line 5 tests if the adjacent voxel, `tx,ty,tz`, is a border voxel. If the voxel's gradient magnitude is less than a rough threshold, `gr_threshold`, or its `w` value is positive but a border is defined as a negative layer, the adjacent voxel is not a neighbor. The for loop of line 3 continues to test the next adjacent voxel. Otherwise it tests if the voxel satisfies condition (4.26) by calling function `cross_zero()`. If the function returns false, the for loop continues. If the function returns true, the adjacent voxel is a neighbor. Subsequently, from lines 6 to 11, the neighbor information is saved for testing face connections later, and if it is unmarked, mark it and queue it. The `for` loop of line 3 stops after a subset is scanned. In line 13, if at least one neighbor has been found, the `for` loop of line 2 breaks and the procedure `Search_neighbor()` returns.

The function `cross_zero(tx,ty,tz,border)` tests if the voxel, `tx,ty,tz`, satisfies condition (4.26). The argument `border` accepts two values: `-1` if the border is defined as a negative layer of voxels, or `1` if the border is defined as a positive layer of voxels. The function accesses the global array `w[][][]` by voxel coordinates. Therefore, testing condition (4.26) for a given voxel can be done in constant time. Its implementation is straightforward. No details are given here.

The procedure `append_queue()` appends an unmarked neighbor to the queue. The queue is a linked list of queue cells, defined by type `Q_CELL`, and pointed to by a global pointer, `queue`, of type `QUEUE`. The type definitions are given in Fig 5.13. The pointer queue has two pointers: head pointing to the first cell of the queue, and **rear** pointing to the last cell of the queue. Hence either removing or appending a queue cell can be done in constant time.

```
typedef struct  queue_cell  {
    int                   x,y,z;      /* a border voxel coordinates */
    struct  queue_cell    *next;
} Q_CELL;

typedef struct  queue    {
    Q_CELL    *head;                  /* pointing to the queue head */
    Q_CELL    *rear;                  /* pointing to the queue rear */
} QUEUE  *queue;
```

Figure 5.13: The queue cell and queue type definition.

The time complexity of the procedure `Search_neighbor()` depends on two `for` loops in lines 2 and 3. Similar to calculating adjacent voxel coordinates in line 4, testing `if` conditions in line 5, marking a neighbor in line 9, etc., all can be done in time $O(1)$, the `for` loop of line 3 depends on the number of adjacent voxels in a subset, `s->n_of_adj_voxels`. Once the `for` loop of line 3 stops and at least one neighbor has been found, the `for` loop of line 2 breaks. Therefore, the time to execute the two `for` loops is about half of the length of all subsets of an outway. Detailed analysis will be given in the next Section.

5.6 Complexity Analysis

In previous sections, the definitions of adjacent voxels for three face primitives, the `table` structure to store and access this information, and the `Neighbor_face()` procedure for searching neighbors have been discussed. This section discusses the performance of the border face tracking algorithm by this implementation.

The analysis given in page 69 has shown that the time complexity of the border face tracking algorithm is $O(kn_b)$. n_b is the the number of border voxels, and for a given object, it is fixed. The constant k depends on the number of adjacent voxels that a border voxel has and the way that the procedure `Neighbor_face()` searches the adjacent voxels for neighbors. As discussed in the previous section, the search neighborhood of the current voxel is split to search outways. Adjacent voxels of an outway are separated into subsets. Subsets are searched in a given order. Whenever a subset has been searched and at least one neighbor has been found, the search stops and proceeds to the next outway.

If the possibility of an adjacent voxel to be a neighbor is uniformly distributed, for the type_1 face whose adjacent voxels are shown in Fig 5.5, the average time to search an outway is approximately $\frac{1}{6}(1 + 2 + 3 + 4) + \frac{2}{6} \times 6 = 3\frac{2}{3}$. The total time to search four outways is approximately 14.33. For the type_2 face whose adjacent voxels are shown in Fig 5.6, the average search time is approximately 15.50. For a type 3 face, it's approximately 12.50. Therefore, the constant k is about $(14.33 + 15.50 + 12.50)/3$, i.e., 14.

In conclusion the time complexity of the `Border_face_tracking()` procedure is $O(kn_B)$, where n_B is the number of border voxels of a given object and constant k is approximately 14.

5.7 Correctness of the Algorithm

This section will show that tracking outways is equivalent to tracking adjacent voxels. In other words, no border voxel will be missed. The analysis is based on the following theory:

Theorem 5.1 *If an object surface is closed, neighbors of any border voxel are edge connected voxels.*

Proof: Consider voxels as a tessellation of the space, every border voxel contains a piece of surface, or a patch. If a border voxel has no edge connected neighbor but a vertex connected neighbor voxel, the two patches in the two border voxels are not connected, because two patches can not be connected through one vertex. The surface is therefore not closed. It concludes that for an object with a closed surface, tracking 18 edge connected voxels is enough. The theory is also correct for a polyhedral surface where each voxel contains one or more faces. The property of a polyhedral surface will be given in Chapter 6.

If an object surface is not closed because of noise or some unknown reasons, vertex connected voxels should be searched as a natural extension. This is the reason that vertex connected voxels are also included in adjacent voxel sets. Missing faces between two vertex connected voxel faces can be tiled arbitrarily. In the extended cuberille model, they should be tiled with the four types of external voxel faces.

Adjacent voxels defined for type_1, type_2 and type_3 faces in Fig. 5.5 to 5.7 are ordered so that a face connected voxel is the first one to be searched, and vertex connected voxels are the last ones to be searched, basically as a search extension. The following will show that if more than one adjacent voxel are neighbors, they are in turn adjacent and therefore reachable in the given order.

For outway 1 of the type_1 face, see Fig. 5.5, under Assumption 5.1, a neighbor face should has normal component $\nabla_x > 0$. Clearly, among the first three adjacent voxels, only one could possibly satisfy condition (4.26). Also, among the rest three adjacent voxels, only one could possibly be a neighbor. If Theory 5.1 holds, and if there exists two neighbors, the second is the edge connected neighbor, adjacent voxel 4, and is reachable from adjacent voxel 1 because they are face connected. If adjacent voxel 1 is not a neighbor, but either adjacent voxel 2 or 3 is, adjacent voxel 4 is also reachable. This can be seen by enumerating all possible face patterns as shown in the follows.

There are six possible neighbor face patterns for adjacent voxel 2, see Fig. 6.9(a) (b) and (d), page 99. The second and the last face in (d) are impossible for adjacent voxel 4 to be a neighbor. Among the rest four faces, adjacent voxel 4 is the first neighbor of outway 4 of the face in (b), and the first neighbor of outway 1 of the first face in (d). It is in the second subset of outway 4 of the face in (a), also in the second subset of the third face in (d), and is reachable because the face connected voxel is not a neighbor.

A similar analysis applies to adjacent voxel 3.

For outway 1 of the type_2 face, see Fig. 5.6, by the same argument, only one among the first two adjacent voxels could possibly be a neighbor. If none of them are a neighbor, both adjacent voxel 3 and 4 will be searched as an extension.

For outway 2 of the type_2 face, if adjacent voxel 1 is a neighbor, adjacent voxels 2 to 5 of the second subset are face connected to it and are therefore reachable from adjacent voxel 1. If adjacent voxel 1 is not a neighbor, however, the second subset of four adjacent voxels will be searched. If none are a neighbor, the two vertex connected voxels will be searched as an extension, because only one of them could possibly be a neighbor.

For type_3 voxel, see Fig. 5.7, If adjacent voxel 1 is a border voxel, adjacent voxels 2 to 4 of the second subset are face connected to adjacent voxel 1 and are therefore reachable. If adjacent voxel 1 is not a neighbor, the subset of adjacent voxels 2 to 4 will be searched. If none are a neighbor, both vertex connected voxels will be searched.

It concludes that for an object with a closed border, tracking outways is equivalent to tracking adjacent voxels.

5.8 The Result of Border Face Tracking

Fig. 5.14 shows the result of border face tracking of the two test objects given in Fig. 3.10. The surface of the sphere has few holes on it, and the surface of the cylinder has few disconnected pieces. This is because the algorithm tracks border voxels and a border voxel is converted to one face primitive. Careful observing the images, however, reveals that the border voxels are connected, for the disconnected faces are just one unit apart. The next two chapters will discusses the face connection testing and surface closure.

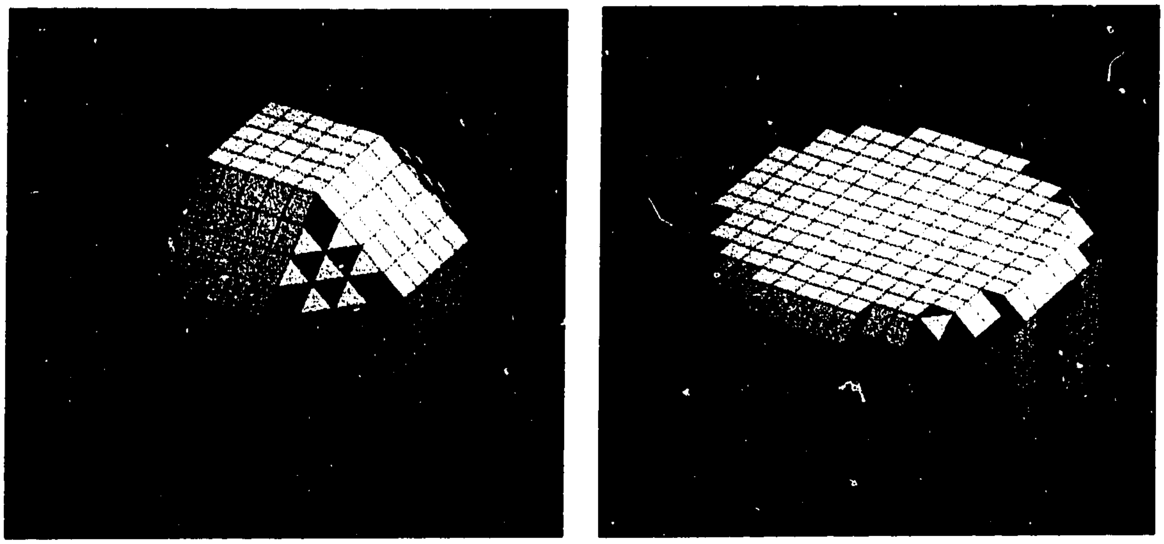


Figure 5.14: The result of tracking border faces of two test objects.

Chapter 6

Face Connection Testing

In the extended cuberille model, each voxel is converted to one face primitive. As a result, a quite few of border voxel faces are not connected. It is necessary to test face connection during border face tracking.

The first two sections of this chapter give background knowledge about a polyhedral surface, and measurements to test and to close a polyhedral surface. The rest of this chapter will discuss data structures and procedures to implement these measurements in the surface tracking algorithm.

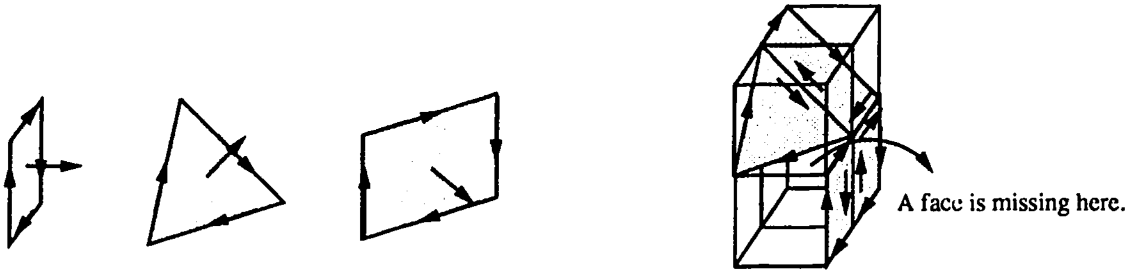
6.1 Polygon Faces

The face primitives in the extended cuberille model are polygon faces. An edge of a polygon face is directed, from its start vertex to its end vertex, and the edges of a face form a directed edge ring. The normal of a polygon face could point to either side of the face. In this implementation, it is assumed that the face normal points in a direction so that the normal and the edge ring form a left handed coordinate system, as shown in Fig. 6.1(a).

6.2 A Polyhedral Surface

A surface in the extended cuberille model is a polyhedral surface. A polyhedral surface consists of polygon faces. The normal of a face is always pointing to outside of the surface. In other words, the edge ring of a polygon face is in the clockwise direction if looked at the face from outside. If two faces are connected, they share a common edge. Both edge rings pass the common edge, once in its forward and once in its backward direction, as shown in Fig. 6.1(b). If an edge is passed with only one edge ring, a face is missing. It therefore concludes that a polyhedral surface is closed if and only if every edge is shared by exactly two faces. This property of a polyhedral surface can be used as a measurement to test surface closure. It also offers a way to close a surface. Obviously, the cuberille model doesn't possess this property.

In the following sections, the data structures for face connection testing are discussed. The procedures are discussed in Section 6.7



(a) The edge ring and face normal of three face primitives in the exteded cuberille model.

(b) If two faces are connected, the two edge rings share a common edge.

Figure 6.1: Polygon faces and polyhedral surface.

6.3 Related Definitions

This section gives the related definitions for testing face connection in the surface tracking algorithm. They are: connected face, disconnected face, bounding voxel and missing edge.

6.3.1 Connected and Disconnected Faces

As mentioned before, if the current face and a neighbor face share an edge, they are said to be connected. The neighbor face is called a **connected** face. Otherwise, the two faces are not connected and the neighbor face is a **disconnected** face. Possible connected and disconnected neighbor faces are stored in the **table** structure.

6.3.2 Bounding Voxel and Missing Edge

While the surface tracking algorithm traverses an outway of the current face to a neighbor face, it crosses an edge. This edge is called the **current edge**. If the current face and the neighbor face are connected, the current edge is passed by two edge rings, the current face edge ring and the neighbor face edge ring. If the current face is not connected to the neighbor face, however, the face between them is missing. This face is called the **missing face**. The voxel where the missing face is located is called the **bounding voxel** of the missing face. In this case, the current edge should be shared by the edge ring of not the neighbor face but the missing face, if the surface is eventually closed. The edge in the missing face is directed backward to the current edge. This edge is called the **current missing edge**, as shown in Fig. 6.2.

There are two types of missing edges. One is a voxel edge of unit length, labeled as type '1'. The other is diagonal of a square voxel face, labeled as type '2'. The two types of missing edges are also shown in Fig. 6.2.

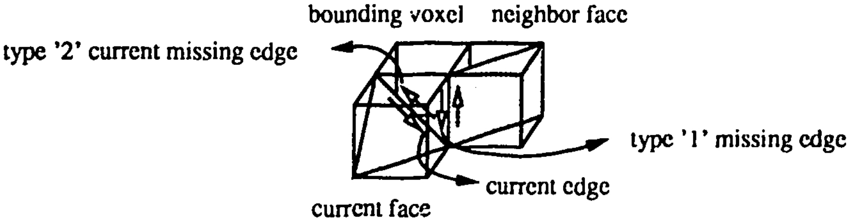


Figure 6.2: The current edge, the current missing edge and the bounding voxel.

Since a missing face can be determined by its bounding voxel and missing edges, this information is saved during face connection testing.

6.4 The Data Structure

This section discusses the part of TABLE type definitions and data structures for storing connected faces, disconnected faces, and bounding voxels, etc.

6.4.1 Connected and Disconnected Face Structure

The possible connected and disconnected neighbor faces are stored in the adjacent voxel structure. The ADJ_VOXEL type definition is shown again in Fig. 6.3. The structure has two pointers: a pointer `con_faces` pointing to an array of CON_FACE structures where possible neighbor faces connected to the current face are stored, and a pointer `dis_faces` pointing to an array of DIS_FACE structures where possible disconnected neighbor faces are stored.

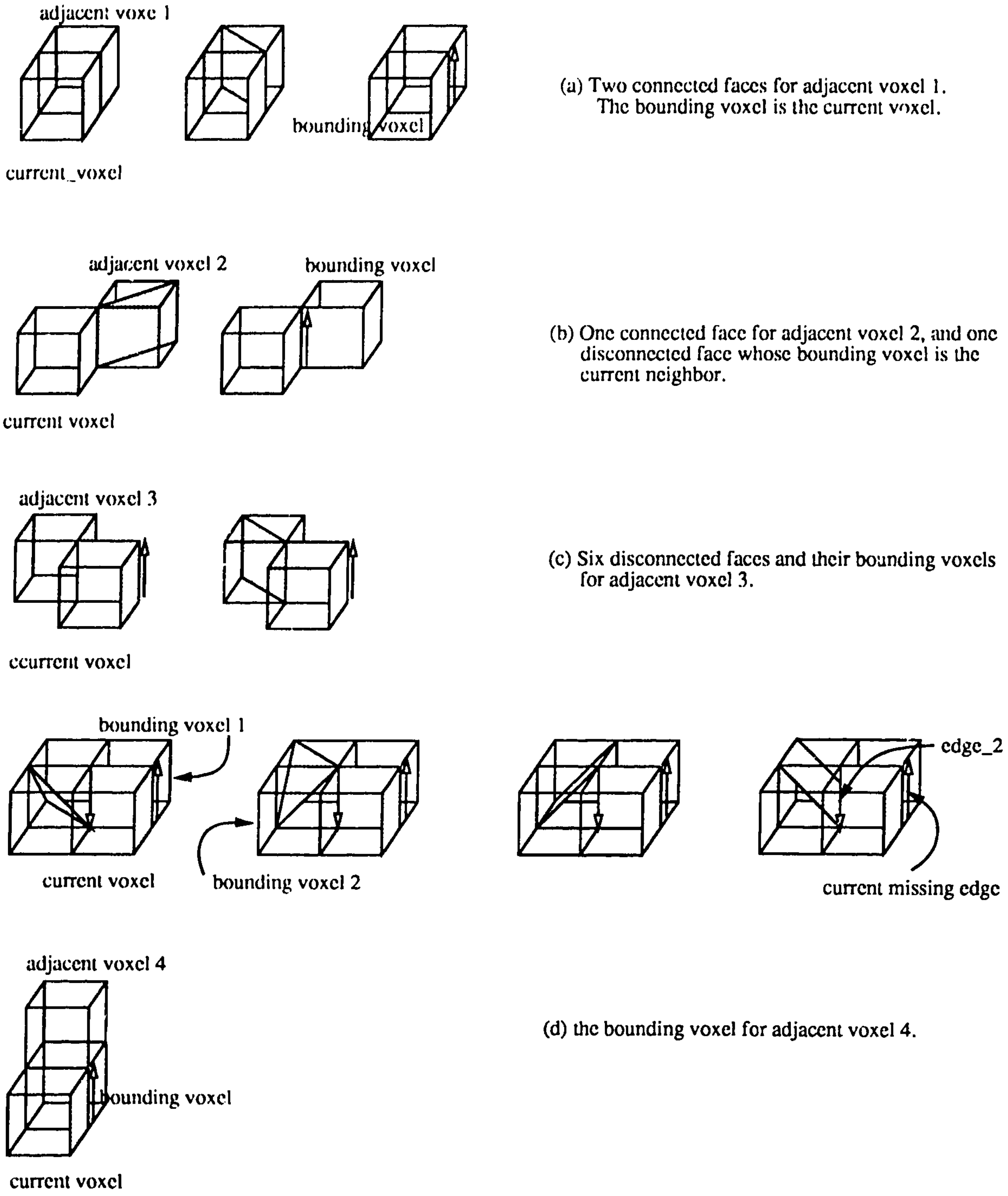
The CON_FACE structure has an unsigned char, `loc_dir`, recording the loc_dir code of a connected neighbor face. The DIS_FACE structure also has a field `loc_dir`. The reason to store a disconnected face is to locate a bounding voxel where a missing face resides. The bounding voxel structure is given in the next section.

While tracking border faces, the algorithm tests if a neighbor face's loc_dir code matches one of the loc_dir codes in the CON_FACE structure. If so, the neighbor face is a connected face, do nothing. Else there is a face missing between the current face and the neighbor face. The algorithm continues to test if the neighbor face's loc_dir code matches one of the loc_dir codes in the DIS_FACE structure. If so, the missing face's bounding voxel coordinates are obtained from the table and saved. This information will be used to find the missing face in the close surface algorithm in Chapter 7.

Figs. 6.4 to 6.7 show connected faces, disconnected faces and bounding voxels for the adjacent voxels of a type_1, a type_2, and a type_3 face. Vertex connected adjacent voxels are not included in the figures.

```
typedef struct  coordinates      {
        float           x,y,z;
} COORDINATES;

typedef struct  connected_face   {
        int             type;
        unsigned char   loc_dir;
} CON_FACE;

typedef struct  disconnected_face {
        unsigned char   loc_dir;
        unsigned char   smooth_loc_dir;
        int             b_voxel_type;
        COORDINATES     bounding_voxel;
        EDGE            edge_2;
        COORDINATES     bounding_voxel_2;
} DIS_FACE;

typedef struct  adjacent_voxel   {
        float           ix,iy,iz;
        int             n_of_con_faces;
        CON_FACE        *con_faces;
        int             n_of_dis_faces;
        DIS_FACE        *dis_faces;
        int             b_voxel_type;
        COORDINATES     bounding_voxel;
} ADJ_VOXEL;

typedef struct outcoming_way    {
        ......
        struct subsets  *subset;
        COORDINATES     *bounding_voxel;
} OUTWAY;
```

Figure 6.3: The TABLE type definition continues: connected and disconnected faces.



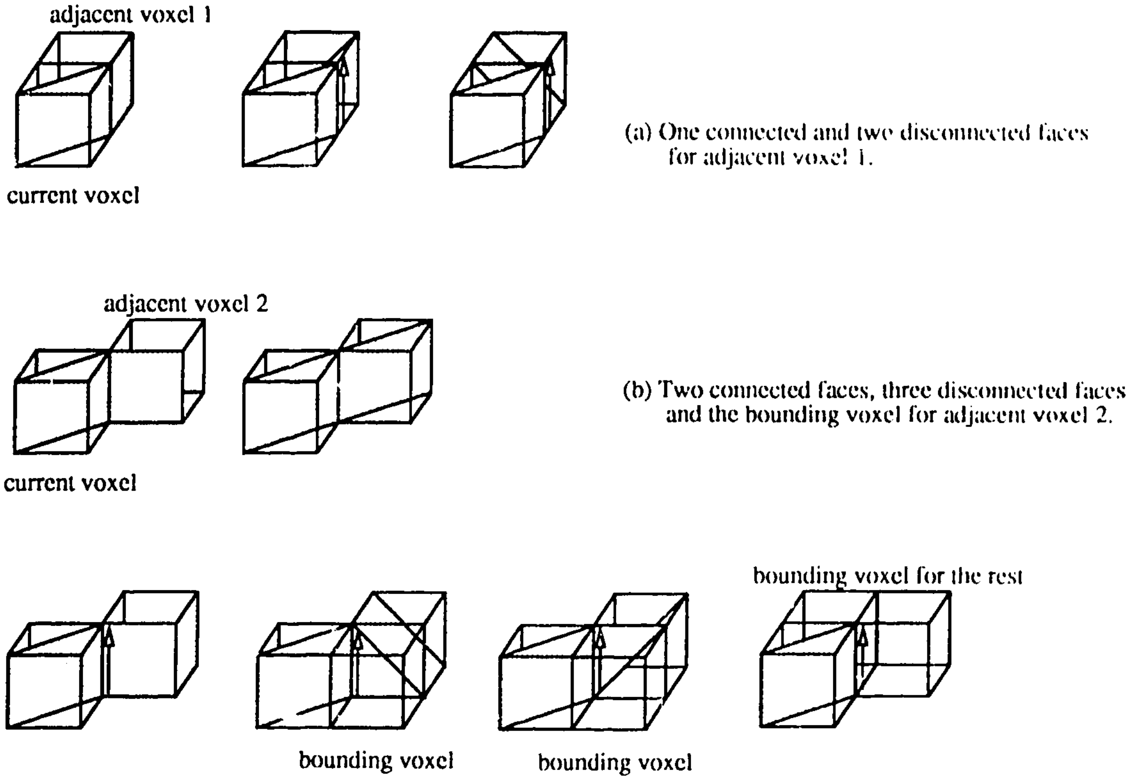
(a) Two connected faces for adjacent voxel 1. The bounding voxel is the current voxel.

(b) One connected face for adjacent voxel 2, and one disconnected face whose bounding voxel is the current neighbor.

(c) Six disconnected faces and their bounding voxels for adjacent voxel 3.

(d) the bounding voxel for adjacent voxel 4.

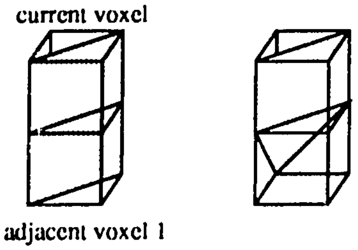
Figure 6.4: The connected faces, disconnected faces, and bounding voxels of a type_1 face.



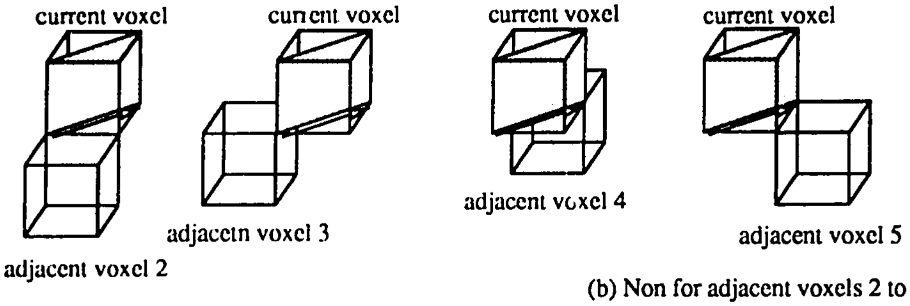
(a) One connected and two disconnected faces for adjacent voxel 1.

(b) Two connected faces, three disconnected faces and the bounding voxel for adjacent voxel 2.

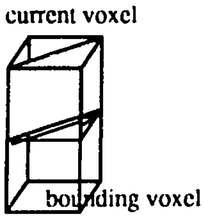
Figure 6.5: The connected faces, disconnected faces and bounding voxels for outway 1 of a type_2 face.



(a) Two connected faces for adjacent voxel 1.

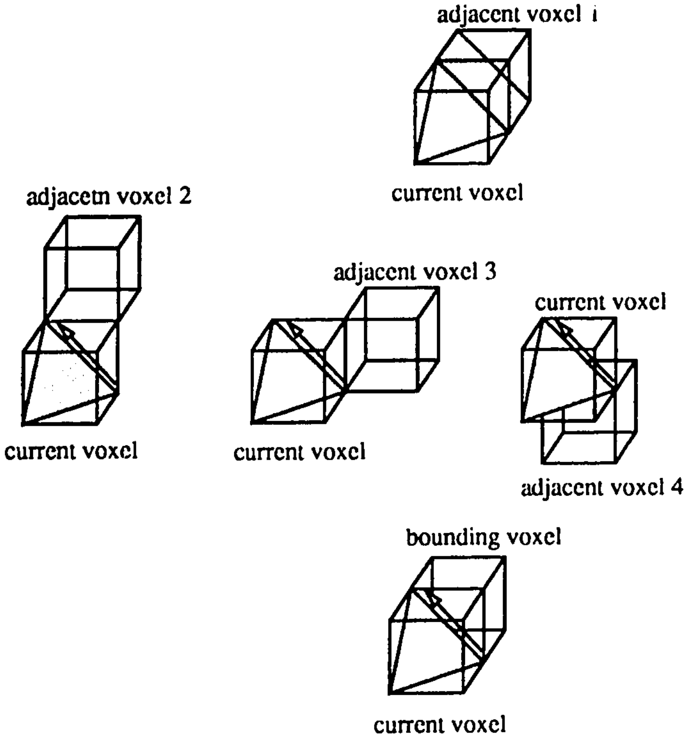


(b) Non for adjacent voxels 2 to 5.



(c) The bounding voxel for outway 2.

Figure 6.6: The connected faces, disconnected faces and bounding voxels for outway 2 of the type_2 face.



(a) One connected face for adjacent voxel 1.

(b) None for adjacent voxels 2 to 4.

(c) The bounding voxel for outway 1.

Figure 6.7: The connected faces, disconnected faces and bounding voxels of a type-3 face.

6.4.2 Bounding Voxel Structure

Bounding voxel coordinates may be stored in one of the three structure levels in the `table`. As shown in Fig. 6.3, there is a `bounding_voxel` pointer in the `OUTWAY` structure, a `bounding_voxel` field in the `ADJ_VOXEL` structure, and a `bounding_voxel` in the `DIS_FACE` structure.

The bounding voxel stored in an outway entry relates to those missing edges whose bounding voxel is unique and obvious. For example, for the type_3 face shown in Fig. 6.7, the bounding voxel of adjacent voxels 1 to 4 is unique. As a missing edge is always of type '2', the bounding voxel is obviously the one that shares the edge. Since there is only one bounding voxel for an outway, the bounding voxel coordinates are stored in the outway structure, as shown in Fig. 6.7(c). No disconnected face is stored. Connection testing can be speeded up.

For a type '1' missing edge, however, its bounding voxel is not obvious. Observe from Figs. 6.4 to 6.6 that the current face and a disconnected neighbor face can determine the location of a missing face's bounding voxel. Therefore, the bounding voxels of disconnected faces are stored in `DIS_FACE` structures. During face connection testing, if a neighbor face's loc_dir code matches a disconnected face's loc_dir code, the bounding voxel coordinates are read out. The bounding voxel that is common for the rest of disconnected faces is stored in the `ADJ_VOXEL` structure so that it can be read out without matching loc_dir codes.

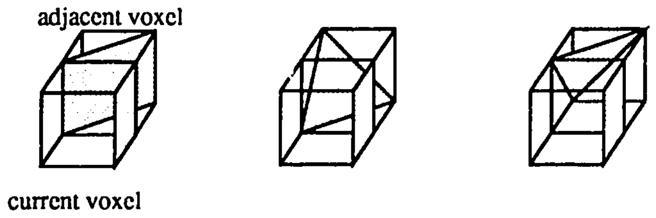
Storing all disconnected faces makes the `table` lengthy and checking inefficient. For some type '1' missing edges, no disconnected face is stored. During face connection testing, if the bounding voxel of the current missing edge isn't in the `table`, the current missing edge is temperately placed in a table, called `open_edge_table`. Its bounding voxels will be located at the beginning of the close surface algorithm. The `open_edge_table` type definition is given in the next section. The disconnected faces that are chosen to store in the table and those that are not is discussed in the next section.

6.5 Neighbor Faces Stored in the Table

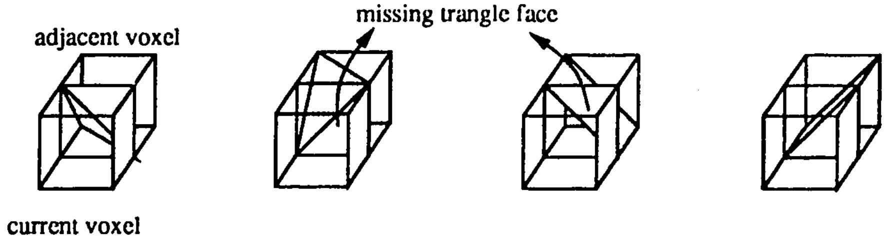
This section discusses which connected or disconnected neighbor faces are chosen to be stored in the `table`.

6.5.1 Neighbor Faces for a Type_1 Face

For the adjacent voxel 1 of a type_1 face, under Assumption 5.1, there could be nine adjacent voxel faces whose normal component $\nabla_x > 0$. Two are connected faces stored in two `CON_FACE` structures, as shown in Fig. 6.4(a). Seven are disconnected faces, as shown in Fig. 6.8. Among them, the three in (a) cannot be a neighbor face if the current voxel is on a border. The four in (b) are possible, and their missing faces are obviously those white triangle faces. Since the missing edges of the neighbor faces are all of type '2', whose bounding voxels could be the same one, only one bounding voxel, the current voxel, is stored in the `ADJ_VOXEL` structure as shown in Fig. 6.4(a), no disconnected face is stored.



(a) These faces are impossible to be a neighbor face.



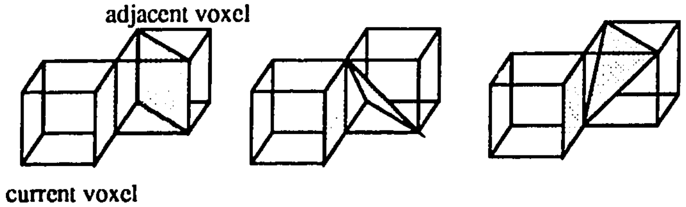
(b) These neighbor faces have missing edges of type '2'.

Figure 6.8: The disconnected faces for adjacent voxel 1 of a type_1 face.

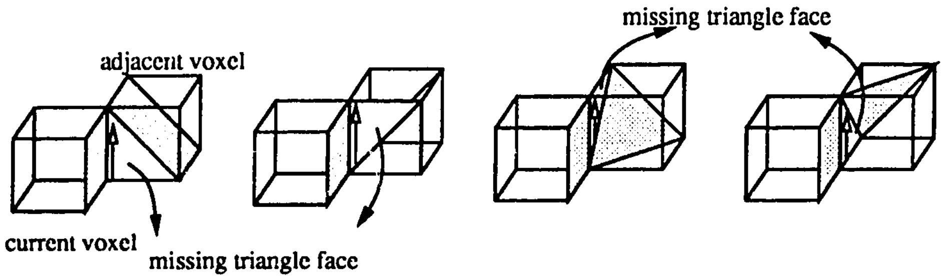
A similar analysis applies to adjacent voxel 2 of the type_1 face. There could be nine adjacent voxel faces whose normal component $\nabla_x > 0$. The nine faces are shown in Figs. 6.9(a) to (d). Those faces in (c) cannot be a neighbor face because the voxel on the right hand side of the current voxel is on the positive layer. The faces in (d) are possible neighbor faces, and are all disconnected faces. The missing faces are obviously those white triangle faces. Since missing edges of the neighbor faces are all of type '2' whose bounding voxels are stored in the outway structure, the four faces in (d) are not stored. During face connection testing, the current missing edge is placed in the open_edge_table. The bounding voxel coordinates were/will be read out when the neighbor face was/is the current face.

(a) One connected face.

(b) One disconnected face. Its bounding voxel is the current neighbor.



(c) These neighbor face orientations are impossible.



(d) These disconnected faces are not stored. Their bounding voxels are obvious.

Figure 6.9: Connected and disconnected faces for adjacent voxel 2 of the type_1 face.

It leaves two faces, a connected face and a disconnected face shown in Figs. 6.9 (a) to (b), stored in the table, (see also Fig. 6.4(b)). The missing face in (b) is obviously the white square face, whose bounding voxel is the neighbor voxel.

For adjacent voxel 3, however, things are different. None of the nine faces are connected to the current face. Three cannot be a neighbor face. Hence it leaves six disconnected faces, as shown in Fig. 6.4(c), and each has its own bounding voxels. The first case in (c) is the same as the second one in (b) but traversing in the inverse direction from the neighbor face to the current face. Since the bounding voxel in (b) is the neighbor voxel, the bounding voxel in (c) is the current voxel so that traversing from either voxel will place missing edges in the same bounding voxel. The bounding voxel for the second face in (c) is similarly defined by comparing with the disconnected face in Fig. 6.5(b).

For the remaining four possible neighbor faces in Fig. 6.4(c), the missing face is obviously the white square face, and the missing edges are all of type '2 , whose bounding voxel is the one, labeled `bounding voxel 2`, that shares the edge. But the current missing edge isn't shared by bounding voxel 2. Only if the missing face is located in another bounding voxel, that could be the one labeled `bounding voxel 1`, could the current face be connected to the neighbor face. Therefore, two bounding voxels, **bounding_voxel** and **bounding_voxel_2**, are stored in the DIS_FACE structure, see Fig. 6.3. To form a square missing face, the edge which is shared by the two bounding voxels is also stored in the **edge_2** field of the DIS_FACE structure.

For adjacent voxel 4, all nine faces are possibly neighbor face, and none of them is connected to the current face. The nine faces could have the same bounding voxel. Therefore, there is only one bounding voxel stored in the ADJ_VOXEL structure, as shown in Fig. 6.4(d). No disconnected face is stored.

6.5.2 Neighbor Faces for a Type_2 Face

Outway 1 Neighbor faces for outway 1 of a type_2 face is shown in Fig. 6.5. For adjacent voxel 1 shown in Fig. 6.5(a), there exist nine faces whose face normal changes less than 90 degrees from the current face. Among them, the six faces shown in Fig. 6.10 cannot be a neighbor face. It leaves three possible neighbor faces, one connected face and two disconnected faces, as shown in Fig. 6.5(a).

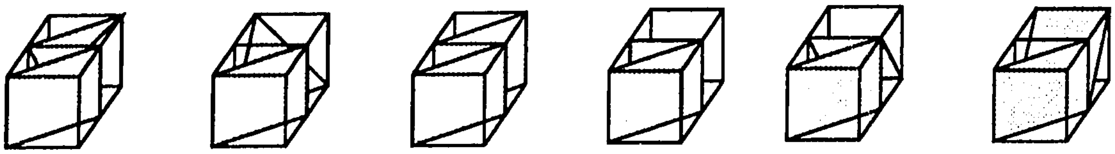


Figure 6.10: These neighbor face orientations are impossible.

For adjacent voxel 2, all nine faces are possible neighbor faces. Two connected faces, three disconnected faces and their bounding voxel are shown in Fig. 6.5(b). One bounding voxel for the remaining four faces is stored in the `ADJ_VOXEL` structure.

Outway 2 Neighbor faces for outway 2 of a type_2 face are shown in Fig. 6.6. Since the current missing edge is always a type '2' edge, one bounding voxel is stored in the `OUTWAY` structure for all disconnected faces. Two connected faces are stored for adjacent voxel 1.

6.5.3 Neighbor Faces for a Type_3 Face

Neighbor faces for outway 1 of a type_3 face are shown in Fig. 6.7. There is one connected face for adjacent voxel 1. Since the current missing edge is always of type '2', one bounding voxel is stored in the `OUTWAY` structure for all disconnected faces.

6.6 The Open Cell Structure

During border face tracking, if a neighbor face is not connected to the current face, the missing face's bounding voxel and missing edges are obtained from the `table`, transformed to the correct orientation, and saved in a data structure called `open_cell`.

The open cells are organized as a hash table, as shown in Fig. 6.11. Each `open_cell` has a field, `center`, of type `COORDINATES`, recording a bounding voxel's coordinates, an integer, `layer`, indicating that the voxel is in the positive or negative layer, a linked list of missing edges, and a pointer pointing to the next open cell in the bucket.

The **hash_table** has 100 buckets. The hash function is a linear combination of the voxel coordinates, see the following, representing a group of 100 parallel planes of an arbitrary orientation. Each plane corresponds to a bucket. If a bounding voxel center, or the module of its coordinate combination, falls into one of the planes, it will be inserted to the corresponding bucket.

```
int hash(COORDINATES center)
{
    return((int)(center.x+2.0*center.y+3.5*center.z) % n_of_buckets);
}
```

A missing edge is saved in a structure called **open_edge**, or **EDGE**. The **open_edge** has an integer, **type**, recording the type of the missing edge, which is either '1' or '2'. The **start_v** and **end_v** are the missing edge's start and end vertex coordinates. Missing edges in a bounding voxel are organized as a linked list and sorted by edge vertices.

6.7 Testing Face Connection

This section discusses the procedure for face connection testing using the data structures given in previous sections.

Recall that the border face tracking procedure in page 69 traverses border voxel faces by calling procedure `Neighbor_face()`. The `Neighbor_face()` procedure, see page 80, has two loops: the first loop searches the neighbors of outways; the second loop tests if the current face is connected to the neighbor faces of each outway by calling procedure `Face_connection()`. The `Face_connection()` procedure is outlined in Fig. 6.12.

The procedure has four arguments: **outway** indicates the current outway; **index** is the current face's loc_dir code; **w** points to the corresponding outway structure in the table; and **n** points to a structure where neighbor coordinates, etc., of the outway were saved during border face tracking. The procedure works as follows: if a neighbor face of the **outway** is connected to the current face, do nothing; else save the current missing edge in the missing face's bounding voxel.

```
#define         n_of_buckets                        100
typedef         COORDINATES                         VERTEX;

typedef struct open_edge {
        int             type;                      /* '1' or '2'   */
        VERTEX          start_v;
        VERTEX          end_v;
} EDGE;

typedef struct  missing_edge    {
        EDGE            edge;
        MISSING_EDGE    *next;
} MISSING_EDGE;

typedef struct open_cell {
        COORDINATES     center;
        int             layer;
        MISSING_EDGE    *missing_edge;
        OPEN_CELL       *next;
} OPEN_CELL;

OPEN_CELL *hash_table[n_of_buckets];
```

Figure 6.11: The bounding voxel, missing edges and the open cell definitions.

In line 1, if no neighbor of the `outway` was found, the current missing edge is calculated and saved in the `open_edge_table`, and the procedure returns. If neighbors do exist, in line 5, the pointer `s` points to the subset the neighbors belong to, and in line 8, the `n_tab` points to the corresponding adjacent voxel of the subset in the `table` structure.

From lines 6 to 31, the `for` loop tests if a neighbor face, whose coordinates are `n->ix,n->iy,n->iz`, is a connected or disconnected face.

Line 7 initializes two flags: `f_flag`, connected face flag, and `b_voxel_flag`, bounding voxel flag, to `NOTFOUND`.

Lines 9 to 14 test if the neighbor face's loc_dir code matches the loc_dir code of a connected face, which is pointed to by the `n_tab->con_faces` pointer. In line 10, the

```
void
Face_connection(int outway, unsigned char index, OUTWAY *w, NEIGHBOR *n)
{
    int               i,l,f_flag,b_voxel_flag;
    struct subsets    *s;
    ADJ_VOXEL         *n_tab;
    ADJ_FACE          *af_tab;
    DIS_FACE          *df_tab;
    COORDINATES       center;
    EDGE              edge;

    if(n_num[outway]==NULL){        /* if no neighbor for the outway  */
      calc_edge_coors(edge);        /* calc the current missing edge  */
      add_to_o_e_table(edge);       /* add the edge to open_edge_table*/
      return;
    }
    s=w->subset+n_set[outway];          /* s: the subset structures    */
    for(i=0; i<n_num[outway]; n++,i++){ /* for each neighbor face      */
       f_flag=NOTFOUND; b_voxel_flag=NOTFOUND;
       n_tab=s->adj_voxels+n_n[outway][i];/* n_tab: the adjacent voxel*/

       /* if the neighbor face is a connected face  */
       for(af_tab=n_tab->con_faces,l=0; l<n_tab->n_of_con_faces;
                                                           af_tab++,l++){
          calc_face_loc_dir(n->con_faces->loc_dir,af_tab->loc_dir);
          if(!((grad_loc_dir[(int)n->ix][(int)n->iy][(int)n->iz] &
                                    ~MARK) ^ n->con_faces->loc_dir)){
              f_flag=FOUND;
              break; }
       }
       /* if the neighbor face is a disconnected face */
       if(f_flag == NOTFOUND)
          for(df_tab=n_tab->dis_faces,l=0; l<n_tab->n_of_dis_faces;
                                                       df_tab++,l++){
              calc_face_loc_dir(n->dis_faces->loc_dir,df_tab->loc_dir);
              if(!((grad_loc_dir[(int)n->ix][(int)n->iy][(int)n->iz] &
                                    ~MARK) ^ n->dis_faces->loc_dir)){
                   b_voxel_flag = FOUND;
                   switch_voxel(&center,&edge);
                   break; }
        }
```

```
       /* if a bounding voxel is in the adj_voxel entry */
 23    if(f_flag==NOTFOUND && b_voxel_flag==NOTFOUND)
 24       if(n_tab->b_voxel_type) {
 25         b_voxel_flag = FOUND;
 26         switch_voxel(&center,&edge); }

       /* if a bounding voxel is in the outway entry */
 27    if(f_flag==NOTFOUND && b_voxel_flag==NOTFOUND)
 28       if(w->bounding_voxel) find_bounding_voxel(w);
 29       else {
 30          calc_edge_coors(edge);
 31          add_to_o_e_table(edge); }
    }  /* end of for */
}
```

Figure 6.12: The Face_connection() procedure.

function `calc_face_loc_dir` calculates a connected face's loc_dir code by multiplying it with the precomputed rotation matrix and saves the result in `n->con_faces->loc_dir`. Line 11 tests if the neighbor face's loc_dir code matches the connected face's loc_dir code. If the two match, the neighbor face is a connected face. The `for` loop breaks and the procedure continues to test the next neighbor face. If the neighbor face doesn't match any one of the connected faces, there is a face missing. The rest of the procedure tries to obtain the bounding voxel of the missing face.

From lines 16 to 22, the `for` loop tests if the neighbor face's loc_dir code matches one of the disconnected face's loc_dir codes in the table. If so, in line 20, the bounding voxel coordinates and the missing edge vertices are computed by calling function `switch_voxel()`, and saved in the `hash_table`. The `for` loop breaks and the procedure continues to test the next neighbor face.

If none of them match, line 24 tests if a bounding voxel exists in the table's `ADJ_VOXEL` structure. This bounding voxel is for those disconnected faces not stored in the table's `DIS_FACE` structures. If it exists, again the function `switch_voxel()` is called to compute the the bounding voxel's coordinates and the missing edge's vertices. Otherwise, line 28 tests if a bounding voxel exists in the table's `OUTWAY` structure. Until line 29, if a bounding voxel has not found, the current missing edge is appended to the `open_edge_table`.

The function `switch_voxel()`, see the following, does basically three things. First, the bounding voxel coordinates are read from the `table`, trarsformed by the precomputed rotation matrix, and saved in a structure `center`. Next, the current missing edge vertices are read from the `BASIC_FACE` structure, multiplied by the matrix in the table entry, and saved in a structure `edge`. Then the function `insert_to_hash_table()` is called to insert the bounding voxel coordinates and the current missing edge vertices in the `hash_table`.

```
void switch_voxel(COORDINATES *center,EDGE *edge)
{
   calc_bounding_voxel_coors(center);
   calc_edge_coors(edge);
   if(insert_to_hash._table(center,edge));
   else add_to_o_e_table(edge);
}
```

The function `insert_to_hash_table()`, see the following, works as follows. If the bounding voxel is already in the hash table, the edge is inserted in the bounding voxel's missing edge list. Otherwise, both the bounding voxel and the edge are inserted in the `hash_table`. It is known that hashing has, on average, constant time for a membership test, an insert or delete operation.

```
OPEN_CELL *insert_to_hash_table(COORDINATES *center, EDGE *edge)
{
    int             index;
    OPEN_CELL       *oc;

    index = hash(center);
    for(oc=hash_table[index]; oc!=NULL; oc=oc->next){
      if(same_box_center)   {
         if(insert_edge(edge,oc,INSERT)) return(oc);
         else   return(NULL); }
    }
    oc = insert_to_new_box(center,edge);
    return(oc);
}
```

As discussed, calling `Face_connection()` results in two tables: the hash table and the open edge table. When the border face tracking algorithm stops, missing edges are either in the `hash_table`, or in the `open_edge_table` if their bounding voxels haven't been found. In other words, the procedure `Face_connection()` guarantees every edge appears in pairs, both forward and backward copy, no matter whether the current face is connected to neighbor faces or not.

Chapter 7

The Close Surface Algorithm

As discussed in the previous chapter, when the border face tracking algorithm stops, border voxel faces have been placed in the display table, and missing edges and their bounding voxels have been placed in the `hash_table`, or in the `open_edge_table` if their bounding voxels haven't been found. Next, the close surface algorithm is called to replace missing edges with external voxel faces to close the surface. This chapter will discuss the close surface algorithm and data structures.

As shown below, the close surface algorithm has two tasks. It calls procedure `add_oe_to_hash_table()` to find bounding voxels for the missing edges in the open edge table, and adds the missing edges to the hash table. It then calls procedure `Fill_edges()` to replace the missing edges in the hash table with four types of external voxel faces.

```
void Close_surface(void)
{
    add_oe_to_hash_table();
    Fill_edges();
}
```

A `open_edge_table` entry has two fields: missing **edge** vertices, and the **center** coordinates of its current voxel.

```
#define n_of_open_edges          500
```

```
struct  open_edge_cell          {
        EDGE            edge;
        COORDINATES     center;
} open_edge_table[n_of_open_edges];
```

The procedure `add_oe_to_hash_table()` is shown below. For every missing edge in the open edge table, it tries to find a bounding voxel already in the hash table for it. Missing edges in the open edge table are all type '1' edges. The bounding voxel of a type '1' edge could be one of the four voxels that share the edge. The procedure tests the four voxels, with the missing edge's current voxel the last, checking if any is contained in the `hash_table`. If a bounding voxel exists and the missing edge is successfully added to the bounding voxel, the table entry of the missing edge is erased. When the `add_oe_to_hash_table()` returns, if every missing edge has been added to the hash table, the `open_edge_table` should be empty.

```
void add_oe_to_hash_table(void)
{
    struct open_edge_cell   *oe;

    for(oe=oe_list->first; oe<oe_list->last; oe++){
        /* if is one of the three voxels sharing the edge */
        if(add_edge_to_hash_table(oe->edge,oe->center))
              ;
        else    /* if is the current voxel     */
        if(add_oe_to_bounding_voxel(oe->center,oe->edge))
              ;
        else  continue;
        /* erase an open_edge table entry */
        *oe = *(oe_list->last-1);
        oe_list->last--;
        oe--;
    }
}
```

For every bounding voxel in the hash table, the `Fill_edges()` procedure fills edges to make a missing edge list a closed edge ring. The missing edge ring is then replaced by external voxel faces, that are added to a table called `o_f_table` for displaying, and removed from the bounding voxel. If removing a missing edge ring results in an empty missing edge list, the bounding voxel itself is deleted from the hash table. When the procedure `Fill_edges()` returns, if missing edges in all bounding voxels have been replaced by external voxel faces, the hash table should be empty.

After `Close_surface()` is called, if both the `open_edge_table` and the `hash_table` are empty, the surface is closed.

The data structures and the algorithm for replacing missing edges in the hash table with external voxel faces are discussed in the following sections.

7.1 The Trie Structure

Recall that the extended cuberille model has four types of external voxel faces: a square, a rectangle, a triangle, and an asymmetric triangular face, as shown in Fig. 3.7, page 42. For any external voxel face, there are only two types of edges: type '1' or type '2'. Since edges are directed, a voxel face can be characterized by a ring of edge types, or a list of numbers. For example, a square voxel face can be listed as 1111, because it has four type '1' edges. A triangular voxel face can be listed as 222, because it has three type '2' edges. Since the other two voxel faces are not symmetric, their edge type rings are not unique. But a list of edge types will uniquely specify a voxel face. For example, either 1212 or 2121 specifies a rectangular voxel face, while 121, 211, or 112 specifies an asymmetric triangular face.

The missing edges in a bounding voxel are therefore directed and of two types, and a list of edge types corresponds to voxel faces. There are a few possible lists of missing edge types in a bounding voxel. The possible edge type lists are stored in a data structure called **trie** [AHU83] as shown in Fig. 7.1.

A trie node has three pointers: a left node pointer corresponding to an edge type '1', a right node pointer corresponding to an edge type '2', and a code pointer. The type definition for a trie node is given in the following, where `trie` is a global pointer initialized to point to the root node.

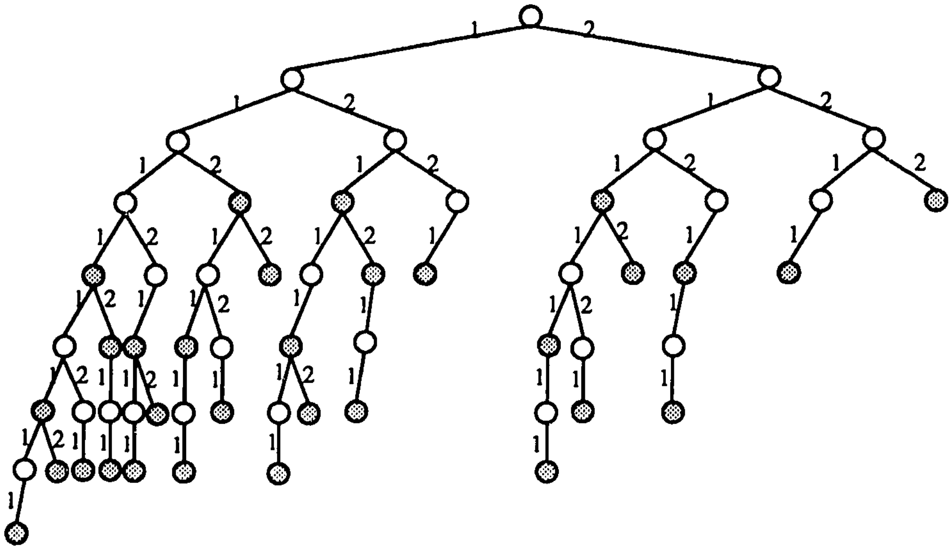


Figure 7.1: The trie for the existing edge type lists.

```
typedef struct  trie_node      {
    NODE    *node[2];
    void    (*procedure)();
} NODE *trie;
```

As shown in Fig. 7.1, a path from the root node to a gray node corresponds to a list of edge types. The code pointer in a gray node points to a procedure that converts the missing edge list to voxel faces, whereas the code pointer in a white node is null. Since some edge type lists correspond to the same voxel faces, their code pointers point to the same procedure. The procedures for initializing the trie, `init_trie()`, inserting a node to the trie, etc., are given in Appendix A.1.

7.2 Lists of Edge Types

There are a total of 31 lists of edge types, therefore 31 gray nodes, in the trie. Some of them are circular shiftings of the same edge ring. For example, 112, 121, and 211 are circular shifts of a triangular face edge ring, and all correspond to the triangle. The 31 lists of edge types correspond to nine different edge type rings, therefore there are nine procedures, see below, converting an edge type list to voxel faces.

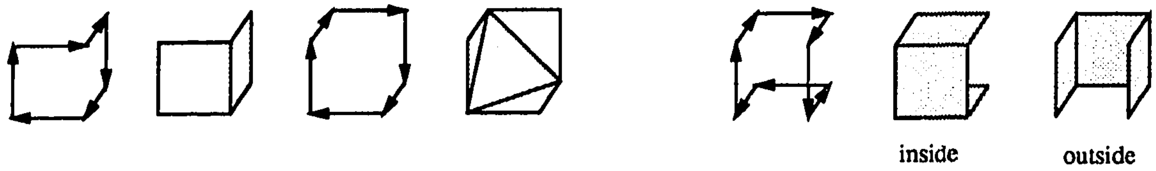
```
void init_procedures(void)
{
    procedures[0] = square_face;
    procedures[1] = triangle_face;
    procedures[2] = face_1122;
    procedures[3] = face_1212;
    procedures[4] = face_11112;
    procedures[5] = face_eight_1;
    procedures[6] = face_six_1;
    procedures[7] = face_six_1_2;
    procedures[8] = face_121112;
}
```

The simple ones are edge type 1111, that is a square face, and edge types 112, 121, 211, and 222, that correspond to a triangular face. For the edge type 1111, the code pointer in the gray node points to a procedure called `square_face()`. The `square_face()`, see the following, takes four vertices of a square face, computes the face normal, and adds the face to the `o_f_table` for displaying, then empties the missing edge list.

```
void square_face(OPEN_CELL *oc)
{
    MISSING_EDGE    *mp;
    VECTOR          nor;

    mp = oc->missing_edge;
    get_square_face_vertices(mp);
    get_square_face_normal(nor);
    add_to_o_f_table(oc->center,nor,SQUARE);
    oc->missing_edge = NULL;
}
```

Similarly, the code pointers in the gray nodes of paths 112, 121, 211, and 222 point to a procedure, `triangle_face()`, that replaces the edge rings with a triangle.

For other edge types, things are little more complicated. Since there are more edges, a list of edge types usually corresponds to more than one voxel face, and may have different face configurations. For example, six type '1' edges can form either two squares or four triangles, as shown in Fig 7.2(a). But the two cases can be distinguished by testing if there are three connected missing edges in one plane.



(a) Two different situations for six type '1' edges.

(b) Eight type '1' edges can be converted to three square voxel faces, depending on which layer the bouding voxel is located in.

Figure 7.2: Six type '1' edges and eight type '1' edges.

Another example is a ring of eight type '1' edges, which can be replaced by three square voxel faces, as shown in Fig 7.2(b). There are two different situations, depending on whether the bounding voxel is inside or outside the surface. The two situations can be distinguished by testing the sign of an integer `layer` in the current bounding voxel structure, as shown in the following. If `layer` is positive and a surface is defined as the negative layer of border voxels, the bounding voxel is outside the surface. Otherwise the bounding voxel is inside.

```
if(oc->layer > 0) i_o = outside;
else i_o = inside;
```

The nine procedures are listed in Appendix A.2. No further discussion will be presented.

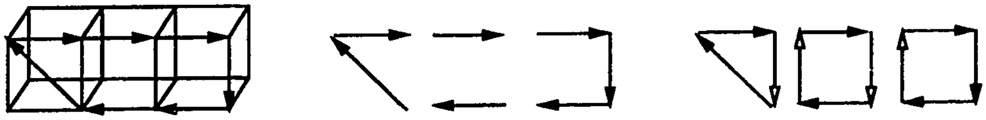
7.3 The Matching Algorithm

The algorithm for converting a missing edge list to voxel faces works as follows. For every bounding voxel in the hash table, the algorithm takes the list of missing edge types in the bounding voxel to traverse the trie by calling procedure `Match_edges()`, see Fig. 7.3 line 3. Once the traversal stops at a gray node so that the edge list matches the path, the procedure, which is pointed to by the gray node code pointer, is called to convert the missing edge list to voxel faces, and to add the voxel faces to the `o_f_table`. Then the bounding voxel is deleted from the hash table.

```
void Fill_edges(void)
{
  int              i;
  MISSING_EDGE     *mp;
  OPEN_CELL        *oc,*pre;

  for(i=0; i<n_of_buckets; i++)
      for(pre=oc=hash_table[i]; oc != NULL; oc=oc->next){
          if(Match_edges(oc,i,ADD) ){/* if edges matches a path   */
              if(oc==hash_table[i]){ /* delete the bounding voxel */
                     hash_table[i]=oc->next;
                     pre=hash_table[i];
              }else  pre->next=oc->next;
          }else pre=oc;
      }
}
```

Figure 7.3: The Fill_edges() procedure.

While the procedure `Match_edges()` matches a missing edge list with a trie path, the missing edges might not be a closed edge ring. An example is shown in Fig. 7.4. The missing edges in the left bounding voxel is short a type '1' edge to form a closed edge ring, which is corresponding to a triangle. Two type '1' edges are absent from the middle bounding voxel to form a square voxel face. One type '1' edge is absent from the right bounding voxel to form a square voxel face. Apparently, if there is an edge absent from a bounding voxel, its backward copy must be also absent from another bounding voxel if the surface is eventually closed.



(a) Three bounding voxels. (b) Their missing edges. (c) Edges inserted in pairs.

Figure 7.4: Insert edges to missing edge list in pairs.

Therefore, if there is an edge absent during matching, the `Match_edges()` procedure inserts an edge to the missing edge list. It also inserts a backward copy of the edge to another bounding voxel. In other words, edges are always inserted in pairs. The procedure is given in Fig 7.5. Its argument, `oc`, is a pointer pointing to an `open_cell` structure in the hash table.

The `for` loop of line 3 traverses a trie path and matches the missing edge list. When the matching starts, the pointer `mp` points to the first missing edge in the list, and `tn` is a node pointer in the trie root node whose index matches the missing edge type. In line 4 the function `match_vertex()` is called to match the missing edge's end vertex with the next missing edge's start vertex. The return value is assigned to a variable `flag`.

The function `match_vertex()` may return four values. If it returns 0, the current missing edge is connected to the next one. The pointer `mp` moves to the next missing edge in the list, and the node pointer `tn` moves to a child node whose index matches the next missing edge type. The `for` loop continues.

If the function returns 1 or 2, however, there is an absence of type '1' or type '2' edge between the current missing edge and the next one. In line 7, the backward copy of the edge is inserted in the hash table by calling `add_edge_to_hash_table()`, and in line 8, the forward copy is inserted in the missing edge list between the current missing edge and the next one. The `for` loop continues.

If the function returns neither 0, 1 nor 2, there are at least two edges lacking between the current missing edge and the next one. The missing edge list cannot be converted to voxel faces at this time. The procedure `Fill_edges()` returns.

When the `for` loop of line 3 stops, the pointer `mp` points to the last edge in the missing edge list.

In line 11, the procedure matches the end vertex of the last edge with the start vertex of the first edge by calling function `match_vertex()`. If the function returns 0, the two are the same vertex. The missing edge list is now a closed edge ring and matches a trie path. In line 12, the procedure pointed to by the code pointer in the path's gray node is called to replace the missing edge ring with voxel faces.

If the function returns 1 or 2, an edge of type '1' or type '2' is absent between the

```
int Match_edges(OPEN_CELL *oc)
{
  int               flag;
  NODE              *tn;
  VERTEX            start_v;
  MISSING_EDGE      *mp,*pre;

  mp = oc->missing_edge;
  start_v = mp->edge.start_v;

  /* match a missing edge list with a trie path */

  for(tn=trie->node[mp->edge.type-'1']; mp->next!=NULL && tn!=NULL;
                  pre=mp,mp=mp->next,tn=tn->node[mp->edge.type-'1']) {
      flag = match_vertex(mp->edge.end_v,mp->next->edge.start_v);
      if(flag == 0)  continue;
      else if(flag == 1 || flag == 2)        {
           add_edge_to_hash_table(re_edge,oc->center);
           insert_edge_to_mp_next(oc,mp,pre);}
      else return(0);
  }

  /* match the end vertex of the edge list with the start vertex */

 if((flag=match_vertex(mp->edge.end_v,start_v)) == 0){
       (*(tn->procedure))(oc);
       return(1);}
 else(flag == 1 || flag == 2) {
       add_edge_to_hash_table(re_edge,oc->center);
       insert_edge_to_mp_next(oc,mp,pre);
       (*(tn->node[flag-1]->procedure))(oc);
       return(1); }
 else return(0);
}
```

Figure 7.5: The Match_edges() procedure.

last missing edge and the first one. Similarly, in lines 15 and 16, a pair of edges are inserted and a procedure is called to replace the missing edge ring with voxel faces.

If the function returns neither 1, 2 nor 0, the missing edge list cannot match a trie path at this time, and the procedure `Fill_edges()` returns.

Because of cross insertion of edges, some missing edge lists may not be a close edge ring after the first time the `Fill_edges()` is called. Therefore, the `Fill_edges()` procedure is called several times.

7.4 Complexity Analysis

The time to traverse a trie path and to match a missing edge list is proportional to the path length to a gray node. The average path length to a gray node in the trie is about 5.3. Hence the time complexity of the close surface algorithm is $O(kn_{bv})$, where n_{bv} is the number of bounding voxels in the hash table and the constant k is about 5.3. The number of bounding voxels depends on the complexity of the surface. Experimental results on test data and medical data have shown that the ratio of the number of border voxels to the number of bounding voxels is ranged from 25.3 to 1.6. Because the number of bounding voxels is less than the number of the border voxels, it expects that the close surface algorithm is faster than the border face tracking algorithm. In other words, the time complexity of the surface tracking algorithm depends on the border face tracking.

7.5 Conclusion of Surface Closure

The surface tracking algorithm guarantees every edge of a surface appears in a pair, a forward copy and a backward copy, no matter if the surface is closed or not. This can be seen from the `Face_connection()` and the `Match_edge()` procedures.

During face connection testing, if the current face is connected to a neighbor face, the current edge appears twice, once in the current face and once in the neighbor face. If the current face is not connected to the neighbor face, the current edge is in the current face, the current missing edge is saved in either the hash table or the open edge table. During matching missing edge lists with trie paths, edges are always inserted in pairs. If a missing edge list matches a trie path, the list will be removed from the hash table and replaced with voxel faces. In other words, if a backward copy of an edge is acquired by a face, it will be deleted from the hash table, for it is shared by two faces. The conclusion follows:

Conclusion 7.1 *After the close surface algorithm is called, if the open edge table and the hash table are both empty, the surface is closed.*

If Theory 5.1 holds, i.e., if there exist an edge connected neighbor for every outway of a border voxel face, the surface can be closed after calling the close surface algorithm. This is because all necessary information to close a surface: the connected faces, disconnected faces and bounding voxels is stored in the `table`, and there exist a small number of possible missing edge list in a bounding voxel.

Unfortunately, due to noise and other reasons, a neighbor of an outway doesn't always exist. The surface therefore can't always be closed. A simple solution to solve this problem is to search more voxels further down the outway. In other words, extending an adjacent voxel set to include more voxels. Some examples are shown in the next section.

7.6 Extension of Adjacent Voxels

A typical example is to extend outway 1 of a type_2 face to include two more adjacent voxels shown in Fig. 7.6.

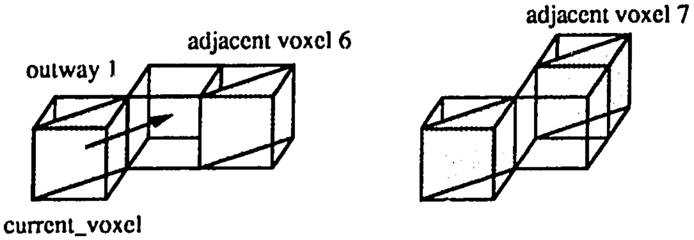


Figure 7.6: Two more adjacent voxels for outway 1 of a type_2 face.

The two voxels are neither face, edge nor vertex connected to the current voxel. But these face patterns happen quite often. It is reasonable to include the two voxels in the adjacent voxel set.

Similar extensions were made for most outways of the three face primitives used in the surface tracking algorithm. No further details will be given here.

Chapter 8

Experimental Results of the Surface Tracking

This chapter gives experimental results of the surface tracking algorithm on medical data. The performance results of call graph profile is given in Section 8.2.

8.1 Experimental Results

Fig. 8.1 shows the surface of a piece of medical object. The data size of the piece is $208 \times 208 \times 26$, resulting from linearly interpolating 6 CT slices with 5 in between every pair. The following gives the code segment for interpolating 5 slices between slices $k-1$ and k.

```
#define R_LEN   208                     /* row size     */
#define C_LEN   208                     /* column size  */

short   n_of_inter = 6;
unsigned char buffer[R_LEN][C_LEN];       /* slice buffer */
unsigned char data[Z_LEN][R_LEN][C_LEN]; /* data  buffer */

for(l=1; l<n_of_inter; l++){     /* interpolate 5 slices */
  for(i=0; i<C_LEN; i++)
    for(j=0; j<R_LEN; j++)
        data[k-1+l][i][j] =
       (data[k-1][i][j]*(n_of_inter-l)+buffer[i][j]*l)/n_of_inter;
```

```
}
k += n_of_inter-1;
for(i=0; i<R_LEN; i++)                    /* copy slice k */
    for(j=0; j<C_LEN; j++)
           data[k][i][j] = buffer[i][j];
k++;
```

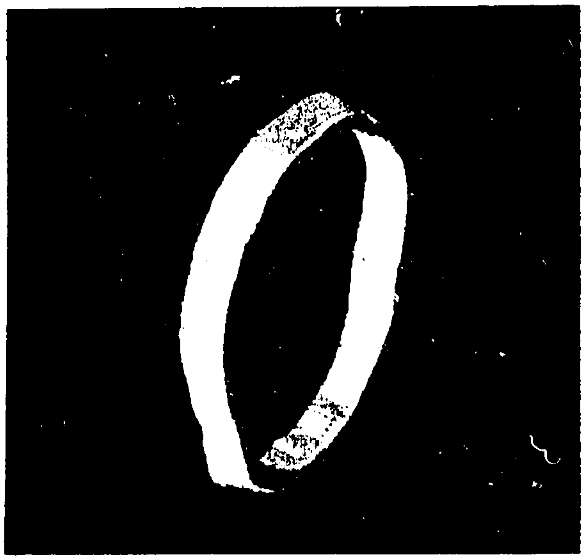


Figure 8.1: The surface of a piece of medical object.

The gray values of the data set, originally ranged from -1024 to 1660, were linerly mapped to the range of 0 to 255. The object has a brighter color than the background, therefore, its border is the negative layer of voxels.

Executing the surface tracking algorithm on the piece results in 17,742 border voxels, and total of 26,409 voxel faces after the close surface algorithm was called. The image was displayed on a Silicon Graphics Iris station 4D/35. Display of 26409 faces takes only seconds.

Fig. 8.3 shows the surface of a skull viewed from three different angles. The surface was reconstructed from 14 CT slices. The 14 CT slices, shown in Fig. 8.2, were linearly interpolated to produce a data set of 208 × 208 × 79 voxels. Executing the surface tracking algorithm on the data results in 84,708 border voxels, and total of 132,022 voxel faces on the surface. The image was displyed on a Silicon Graphics Iris station 4D/35.

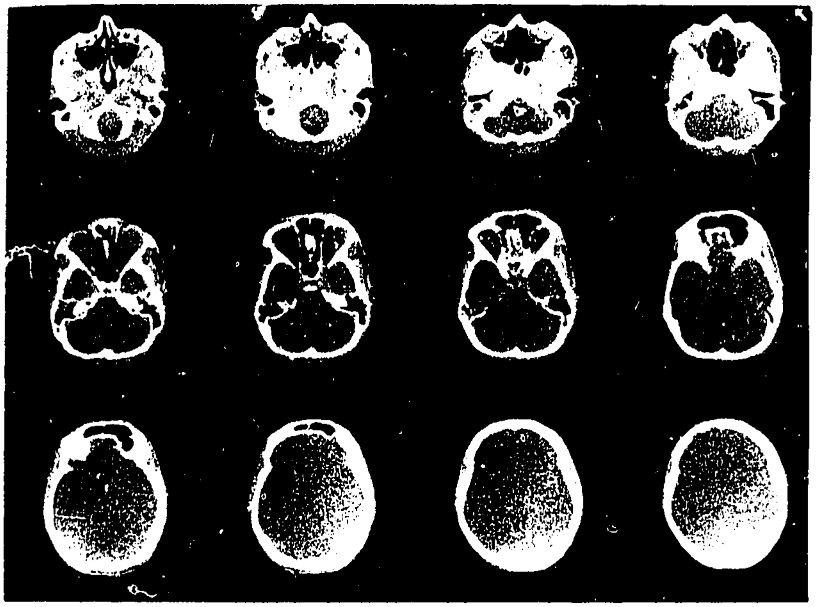


Figure 8.2: 12 out of 14 CT slices.

8.2 Performance Results

The following lists the call graph profile, the results of running `gprof`, of the surface tracking algorithm on the data of Fig. 8.1. The call profile was cut to show only those procedures discussed in previous chapters. The first 22 procedures that spend most of the execution time are listed in Appendix B.

The first column in the call graph is the `index` of a procedure. The second column is the `time` percentage used by the procedure and its descendents. The `self` column shows the seconds of time spent in the procedure itself. The `descendents` column shows the seconds of time spent in the descendents of the procedure. The `called+self` gives the number of times the procedure was called and the number of times the procedure called itself recursively. The `name` is the name of a procedure. The call graph is sorted by time, with the most time consuming procedure on top.

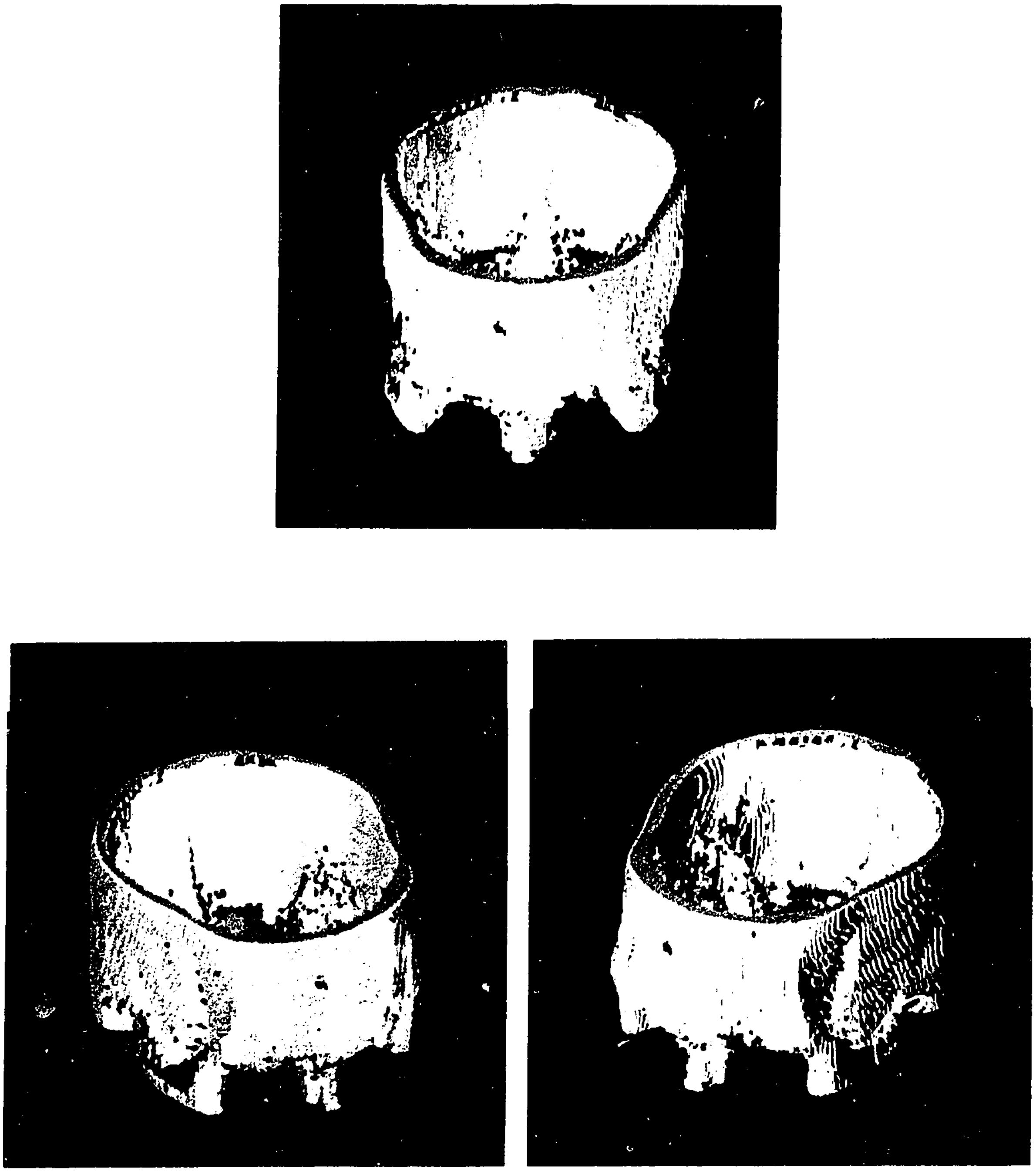


Figure 8.3: The surface of a skull from different view points.

```
                                  called/total       parents
index  %time    self descendents  called+self    name           index
                                  called/total       children

-----------------------------------------------

                0.00      54.06        1/1           _main [3]
[2]     95.3    0.00      54.06        1         _Surface_tracking [2]
                0.53      38.39        1/1           _Border_face_tracking [4]
                8.25       2.59        1/1           _read_all_data [7]
                0.00       4.20        1/1           _close_surface [12]
                0.00       0.09        1/1           _make_table [53]
                0.00       0.00        1/1           _init_lists [103]

-----------------------------------------------

                0.53      38.39        1/1           _Surface_tracking [2]
[4]     68.6    0.53      38.39        1         _Border_face_tracking [4]
                1.85      36.39    17742/17743       _neighbor_face [5]
                0.13       0.00    17743/17743       _remove_q_head [49]
                0.01       0.00        1/1           _span_start_face [86]

-----------------------------------------------

                0.00       0.00        1/17743       _span_start_face [86]
                1.85      36.39    17742/17743       _Border_face_tracking [4]
[5]     67.4    1.85      36.40    17743         _neighbor_face [5]
                4.88      16.17    70583/70583       _search_neighbor [6]
                4.52       4.98    70579/70579       _face_connection [8]
                0.33       2.62   158909/171371      _calloc [16]
                1.31       1.59   158900/159373      _free [18]

-----------------------------------------------

                4.88      16.17    70583/70583       _neighbor_face [5]
[6]     37.1    4.88      16.17    70583         _search_neighbor [6]
                4.87       0.00    79581/79582       _cross_zero [11]
                4.66       0.00   198551/319683      _transform_3d [9]
                0.41       2.94    70583/70583       _untranslate_matrix [15]
                2.85       0.09    70583/141173      _copy_matrix [10]
                0.08       0.26    17742/17742       _append_queue [34]

-----------------------------------------------

                4.52       4.98    70579/70579       _neighbor_face [5]
```

```
[8]     16.7    4.52        4.98   70579         _face_connection [8]
                2.85        0.00  121132/319683    _transform_3d [9]
                1.11        1.03   14447/14447     _insert_to_hash_table[22]

----------------------------------------------

                0.00        4.20       1/1          _Surface_tracking [2]
[12]    7.4     0.00        4.20       1         _close_surface [12]
                0.09        4.04       1/1         _fill_edges [13]
                0.01        0.07       1/1         _add_oe_to_hash_table[54]
```

Among the five procedures of the `surface_tracking` algorithm, see index [2], the `Border_face_tracking` used 0.53 + 38.39 = 38.92 seconds, accounted for 68.6% of the total time. Its descendent, the `neighbor_face`, used 1.85 + 36.39 = 38.24 seconds, accounted for 67.4% of the total time. In turn, 21.05 seconds were used in its descendent `search_neighbor`, see index [5], that accounts for 37.1% of the total time, and 9.5 seconds, 16.7%, were used in its descendent `face_connection`. While the `close_surface` algorithm used only 4.2 seconds, 7.4% of the total time. The result verifies the performance analysis in page 117, and concludes that the time complexity of the surface tracking algorithm depends on the border face tracking.

Chapter 9

Surface Smoothing

Since voxels are very small, the shape and orientation of voxel faces are indiscriminating. But if a zoom-in view is required, the surface appears to be rough. Under this circumstance, the surface smoothing algorithm can be used to smooth a surface.

A border face normal can be adjusted according to its neighbor face's orientation during border face tracking. Recall that the normal of a border voxel face is recorded in its loc_dir code, and the tracking algorithm traverses the border face's outways. If a neighbor of an outway exists, the possible neighbor face's loc_dir codes are read from the `table`. If the neighbor face is a connected face, nothing needs to be changed. If the neighbor face is a disconnected face, the current face's loc_dir code might need to be changed to make smoother connection. Some examples are shown in Fig. 9.1.

A face normal is recorded in the lower six bits of its loc_dir code. Six bit masks, `NZ`, `Z`, `NY`, `Y`, `NX`, `X`, is used to set the six bits. For example, if a face has normal x, the statement `loc_dir=X` will set bit 0 of its loc_dir code to 1. If a face normal is $y -- z$, the statement `loc_dir=Y|NZ` will set bit 2 to 1 and bit 5 to 1. Obviously, bit combination `X` and `NX` shouldn't appear in the same loc_dir code. Neither `Y` and `NY`, `Z` and `NZ`.

Fig. 9.1 shows the adjustment of the current face's normal. This can be implemented by including a field `smooth_loc_dir`, initialized to the desired normal component, in the disconnected face structure, see Fig. 6.3 page 92. For the first case in Fig. 9.1, `dis_faces->smooth_loc_dir` is initialed to `NZ`. During tracking, this information is read

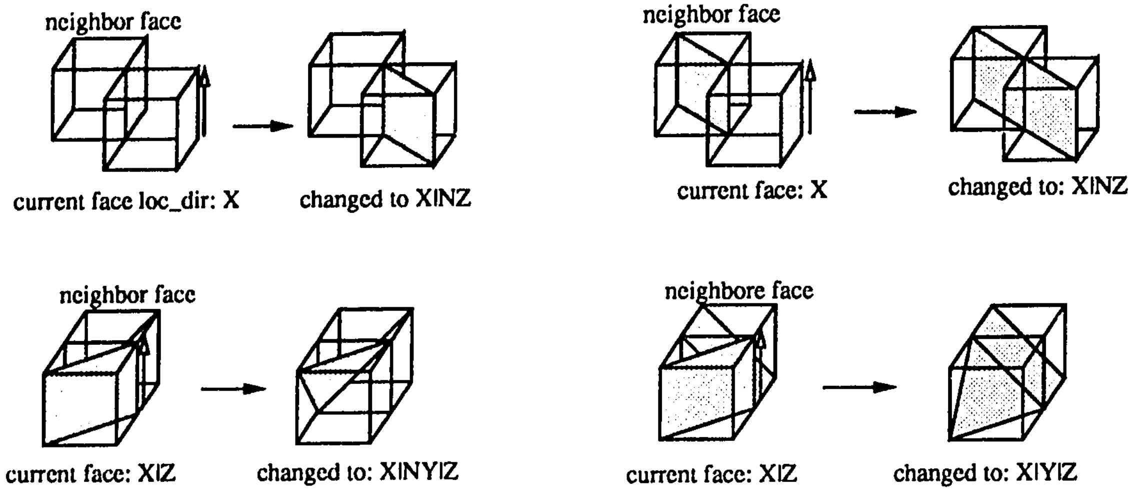


Figure 9.1: The current face normal is adjusted to connect to the neighbor face.

out, transformed, and saved in a global variable `smooth_loc_dir`. The following code segment shows the process of one outway:

```
if(f_flag[outway] == NOTFOUND){
   clear(smooth_loc_dir);

   /* if the neighbor is a disconnected face */
   for(df_tab=n_tab->dis_faces,l=0; l<n_tab->n_of_dis_faces;
                                                       df_tab++,l++){
       calc_face_loc_dir(n->dis_faces->loc_dir,df_tab->loc_dir);

       /* if the neighbor face matches the disconnected face loc_dir */
       if(!((grad_loc_dir[(int)n->ix][(int)n->iy][(int)n->iz] &
                                  ~MARK) ^ n->dis_faces->loc_dir)) {
          if(df_tab->smooth_loc_dir != NULL){
             /* calc the desired normal component  */
             calc_face_loc_dir(n->dis_faces->smooth_loc_dir,
                                               df_tab->smooth_loc_dir);
             /* save it in a global variable  */
             smooth_loc_dir |= n->dis_faces->smooth_loc_dir;
          }
          break;
       }
   }
}
```

The above code segment is included in a procedure `face_smooth()`, which adjusts the current face's normal based on neighbor face's loc_dir code. The surface smoothing procedure is shown below.

The argument `index` is the current face's loc_dir code. The first 7 lines do exactly the same as in `Neighbor_face()` procedure: allocate memory, initialize flags, and search neighbor voxels of each outway.

```
  int smooth_surface(unsigned char index)
  {
        int             i;
        OUTWAY          *w;
        NEIGHBOR        *n,*n_w;

1       f = table[index].face;

        /* allocate  memory for neighbors, initialize flags       */
2       n_w = (NEIGHBOR *)calloc(f->n_of_ways*4,sizeof(NEIGHBOR));
3       for(n=n_w,i=0; i<f->n_of_ways; n++,i++) {
4            init_neighbors_and_flags;
        }

        /* search every outway of the current voxel for neighbors */
5       for(n=n_w,w=f->outways,i=0; i<f->n_of_ways; w++,n+=4,i++){
6            Search_neighbor(i,index,w,n);
7       }

        /* adjust the normal components of the current face      */
8       for(n=n_w,w=f->outways,i=0; i<f->n_of_ways; w++,n+=4,i++){
9            face_smooth(i,index,w,n);
10      }
11      if(smooth_loc_dir) {     /* if the smooth_loc_dir adjusted */

            /* if both X and NX were set, clear the two bits */
12          if(smooth_loc_dir & X && smooth_loc_dir & NX)
13                   smooth_loc_dir &= ~X; smooth_loc_dir &= ~NX;
14          if(smooth_loc_dir & Y && smooth_loc_dir & NY)
15                     smooth_loc_dir &= ~Y; smooth_loc_dir &= ~NY;
16          if(smooth_loc_dir & Z && smooth_loc_dir & NZ)
17                     smooth_loc_dir &= ~Z; smooth_loc_dir &= ~NZ;
       }
       /* adjust the current face's loc_dir */

18     if(smooth_loc_dir)  grad_loc_dir[x][y][z] |= smooth_loc_dir;
  }
```

The `for` loop of line 8 adjusts the normal of the current face by calling function `face_smooth`, that sets the corresponding bits of the `smooth_loc_dir` variable If `smooth_loc_dir` was set, starting from line 12, the `if` statement tests if both bit `X` and `NX` were set. If so, both bits are cleared. Bits `Y` and `NY`, `Z` and `NZ` are also tested. Finally, in line 18, the `smooth_loc_dir` is set to the current voxel's loc_dir code, `grad_loc_dir[x][y][z]`.

The surface smooth algorithm has raised some problems. First, change loc_dir code in one outway may break face connection in other outways. Therefore, the face connection cannot be tested at this local stage, because there is no guarantee that an edge is shared by two faces. It is not possible to retrack border voxel faces either, because all border voxels have been marked after the smooth surface algorithm is called. Second, change of loc_dir code results in a few border voxels that are no longer satisfy condition (4.26), and are therefore not on a border. As a consequence, the surface smooth algorithm operates alone. It revises the `grad_loc_dir` array, and writes it to a disk file. The surface tracking algorithm can read in either the original `grad_loc_dir` array or the revised one to start tracking.

Chapter 10

Conclusion

10.1 Summary

This thesis is concerned with the conversion of volumetric data to a surface model for display purposes. The thesis has presented and implemented an extended cuberille model, a 3D border identification method, a surface tracking algorithm, and a surface closure algorithm.

The 3D edge elements are gradients, and orientations of gradients are quantized to 26 directions. The edge elements are converted to the extended cuberille model by loc_dir codes. The model has four types of voxels so that each voxel has a face whose orientation is compatible with one of the 26 gradient directions. This face is termed a face primitive. There are also four types of external voxel faces, hence a surface in the extended cuberille consists of four types of faces.

Merits of the three representation schemes: space occupancy enumeration, octree, and surface representation by the extended cuberille model were briefly discussed. Analysis has shown that the octree representation is more concise than that in the cuberille model.

The three dimensional border identification method is based on sign of the second derivative of intensity change. For a bright object surrounded by a dark background, the condition for a voxel being on a border is that the second derivative is negative and changes sign for neighbors in the gradient direction. There exists exactly one layer of voxels that satisfies the condition, so the multiple layer problem doesn't occur.

The surface tracking algorithm consists of three algorithms: border face tracking, face connection testing, and the surface closure algorithm.

The border face tracking algorithm traverses outways instead of 26 face, edge or vertex connected voxels. Adjacent voxels for every outway of every face primitive have been defined according to the face orientation. The definition results in less than 26 adjacent voxels. Moreover, adjacent voxels of an outway are further split to disjoint subsets. The algorithm is therefore faster than a breadth first search.

Since a voxel is converted to one face primitive, many border voxel faces are not connected. During border face tracking, the algorithm tests face connection and saves the disconnected face information for the surface closure algorithm to close the surface.

The Algorithms were tested on medical data. The time complexity analysis and the execution call graph profile have shown that the border face tracking algorithm is most time consuming among the three, accounting for 37.2% of total execution time.

The surface tracking algorithm starts with three 3D arrays – the loc_dir codes, the gradient magnitudes, and the w values. Combining the memory consumption, it is a volume based algorithm after all. But once the border voxel faces of a surface have be saved in a display table, all subsequent operations such as display, rotation and scaling, etc., work on border voxel faces. The time complexity is therefore an order of the number of the border voxels of a displayed object.

10.2 Comparison with Other Surface Models

The extended cuberille model in this thesis will be compared with the cuberille model, the marching cubes, and other algorithms in the following aspects: completeness, representation schemes, identification method, time complexity, and surface smoothness.

The extended cuberille model and the triangle model of marching cubes are information complete. But the surface tracking algorithm in this thesis tests surface closure, so it is more complicated than marching cubes. The cuberille model is ambiguous in some sense, because an edge could be shared by four faces. In some cases, there is no way to differentiate the inside from the outside of a surface. The surface tracking algorithm by [AFH81, GU89] works on the cuberille model. There is a theoretical difficulty to prove a surface closure. The other two 3D gradient based algorithms by [CR89] and [SZ87] did not discuss the surface modeling problem.

In the cuberille and the extended cuberille model, an object can easily be represented by an octree because of the regularity of voxels. It is known that the octree is the best data structure for set operations, which is merely a tree traversal. Set operations in the marching cubes, however, are polygon clipping of two surfaces, that is a quadratic algorithm.

There are basically two ways to identify a surface: by thresholding or by gradients. Thresholding has limitations. Even the optimal thresholding [GW87] also has problems, where the thresholded boundary from subimages may not be connected. Identification by gradients or zero-crossings is more general. Especially in 3D display, it is necessary to know face normals for shading. The surface tracking algorithm by [AFH81, GU89] and the marching cubes by [LC87] both use thresholding. Compared with the zero-crossing based identification method in this thesis, the identification of those algorithms is restricted.

The time complexity of surface tracking depends on the number of voxels on a surface. Since surface size is usually less than data size, surface tracking is faster than volumetric data scanning. Marching cubes is a volume scanning algorithm. The surface tracking in this thesis traverses one face per border voxel. Comparing with the one by [AFH81, GU89] which traverses six faces of a border voxel, the surface tracking algorithm in this thesis is faster. However, it needs testing surface closure.

The cuberille model has one face primitive and 6 face orientations. The extended cuberille model has four face primitives and 26 face orientations. The triangle model of marching cubes has 14 face patterns and much more face orientations. Therefore, the extended cuberille model constructs a smoother surface than the cuberille model, while the marching cubes constructs the smoothest surface. However, surface smoothness is not a big issue, since Gouround shading can generate a smooth shaded display for a surface model with much less face orientations.

10.3 Future Research

There are some problems unsolved. In Chapter 3, merits of three representation schemes: space occupancy enumeration, octree, and surface representation by the extended cuberille model have been briefly discussed. Identifying an object and converting it to a surface representation has been implemented in the thesis. But octree related algorithms such as octree generation algorithm, etc., have not been discussed.

In Chapter 4, the Gaussian filters of $\sigma_1 = 2.0$ and $\sigma_2 = 0.5$ for 26 gradient vectors have been designed. But how to chose σ values so that the Assumption 5.1 will hold has not been discussed.

In Chapter 5 Section 5.7, the correctness of the border face tracking algorithm has been shown under Theory 5.1. For situations where Theory 5.1 doesn't hold, tracking adjacent voxels of an outway has been extended to one or two more steps, see Chapter 7 Section 7.6. But if there exist big holes on a border, how to close the surface has not been considered.

All these issues could be future research topics, and it is expected that some of these topics could be quite difficult.

Appendix A

Procedures of the Close Surface Algorithm

A.1 Procedures of Trie Initialization, Insertion

```
#include        <stdio.h>
#include        <math.h>
#include        "trie.h"

FILE    *fopen(), *fp;

int getword(void)
{
        int     i,c;
        char    *w;

        wordlen=0;
        for(w=word,i=0; i<MAXLEN; i++) *w++ = 0;
        w = word;
        if((c = *w++ = getc(fp)) != '1' && c != '2') return(c);
        wordlen++;
        while((c = *w++ = getc(fp)) == '1' || c == '2') wordlen++;
        *(w-1) = ' ';
        return('L');
}

int codename(void)
{
        int     i,c;
        char    *w,code[MAXLEN];

        for(w=code,i=0; i<MAXLEN; i++) *w++ = 0;
```

```
        while((c = getc(fp)) == ' ' || c == '\t' || c == '\n');

        w = code;
        *w++ = c;
        while((c = *w++ = getc(fp)) != ' ' && c != '\t' && c != '\n');
        *(w-1) = ' ';

        return(atoi(code));
}

NODE *makenode(void)
{
        NODE    *tn;

        tn=(NODE *)calloc(1, sizeof(NODE));
        return(tn);

}

insert_to_trie(NODE *tnode,int i,void (*procedure)())
{

        int     index;
        NODE    *tn;

        if(word[i] == ' '){
            tnode->procedure = procedure;
        }else {
            index = word[i] - '1';
            if(tnode->node[index] == NULL){
                tnode->node[index] = makenode();
            }
            insert_to_trie(tnode->node[index],i+1,procedure);
        }
}

print_trie(NODE *tnode,int i)
{
        if(tnode->node[0] != NULL) {
            word[i] = '1';
            print_trie(tnode->node[0],i+1);
        }
        if((tnode->procedure) != NULL) {
                word[i] = '*';
                fputs(word,stdout);
                putchar(' ');
        }
        if(tnode->node[1] != NULL) {
            word[i] = '2';
            print_trie(tnode->node[1],i+1);
        }
```

```
        word[i] = 0;
}

#define         len             20
void            (*procedures[len])();

void init_procedures(void)
{
        extern void square_face();
        extern void triangle_face();
        extern void face_1122();
        extern void face_1212();
        extern void face_11112();
        extern void face_eight_1();
        extern void face_six_1();
        extern void face_six_1_2();
        extern void face_121112();

        procedures[0] = square_face;
        procedures[1] = triangle_face;
        procedures[2] = face_1122;
        procedures[3] = face_1212;
        procedures[4] = face_11112;
        procedures[5] = face_eight_1;
        procedures[6] = face_six_1;
        procedures[7] = face_six_1_2;
        procedures[8] = face_121112;
}

NODE *init_trie(void)
{
        int     c;
        NODE    *tn;

        init_procedures();
        tn = makenode();

        fp=fopen("/usr/sunevere/grad/xiaoqu/surface/trie.data","r");
        if(fp==NULL) {
                fprintf(stderr, "cannot read the file\n");
                exit(1);
        }
        while((c=getword()) != EOF){
          if(c == 'L') {
                fputs(word,stdout);
                insert_to_trie(tn,0,procedures[codename()]);
          }
        }
        fclose(fp);

        printf("\n");
        print_trie(tn,0);
```

```
        return(tn);
}
```

A.2 Procedures to Convert Missing Edge Lists to Voxel Faces

```
void square_face(OPEN_CELL *oc)
{
        int             i;
        MISSING_EDGE    *mp;
        VECTOR          v1,v2,nor;

        mp=oc->missing_edge;
        get_square_vertices;
        get_square_face_normal;
        add_to_o_f_table(oc->center,nor,SQUARE);
        oc->missing_edge = NULL;
}

void triangle_face(OPEN_CELL *oc)
{
        int             i;
        MISSING_EDGE    *mp;
        VECTOR          v1,v2,nor;

        mp=oc->missing_edge;
        get_triangle_vertices;
        get_triangle_face_normal;
        add_to_o_f_table(oc->center,nor,TRIANGLE);
        oc->missing_edge = NULL;
}

void face_1212(OPEN_CELL *oc)
{

        MISSING_EDGE    *mp,*pre;
        VECTOR          v1,v2,nor;
        int             flag=0;

        mp=oc->missing_edge;
        mp_edge_is_vector_v1;
        mp_next_next_edge_is_vector_v2;

        mp=mp->next->next;

        if(v1.x == v2.x && v1.y == v2.y && v1.z == v2.z){
```

```
        square_face(oc);
    }else {
     mp=oc->missing_edge;
     if(mp->next->edge.end_v.x != mp->edge.start_v.x) flag++;
     if(mp->next->edge.end_v.y != mp->edge.start_v.y) flag++;
     if(mp->next->edge.end_v.z != mp->edge.start_v.z) flag++;

     if(flag != 1) {
        /* the two edges are not on the bounding_voxel surface */
        pre=mp;
        mp=mp->next;
     }
     get_triangle_vertices;
     get_triangle_face_normal;
     add_to_o_f_table(oc->center,nor,TRIANGLE);

     if(mp == oc->missing_edge) /* delete these two edges */
            oc->missing_edge = mp->next->next;
     else
            pre->next = mp->next->next;

     mp=oc->missing_edge;
     /* if the left two edges are not connected */
     if(!(ep_end_v_is_ep_next_start_v(mp))){
        oc->missing_edge = oc->missing_edge->next;
        mp->next=NULL;
        oc->missing_edge->next = mp;
     }
     triangle_face(oc);
    }
  }

void face_1122(OPEN_CELL *oc)
{
    int             flag=0;
    MISSING_EDGE    *mp;
    VECTOR          v1,v2,nor;
    EDGE            re_edge;

    mp=oc->missing_edge;
    if((mp->edge.type) != (mp->next->edge.type)) ;
    else mp=mp->next;

      (mp->next->edge.end_v.x != mp->edge.start_v.x) flag++;
    if(mp->next->edge.end_v.y != mp->edge.start_v.y) flag++;
    if(mp->next->edge.end_v.z != mp->edge.start_v.z) flag++;

    if(flag==1){
        get_triangle_vertices;
        get_triangle_face_normal;

        add_to_o_f_table(oc->center,nor,TRIANGLE);
```

```
                if(mp == oc->missing_edge)      /* delete these two edges */
                                oc->missing_edge = mp->next->next;
                else if(mp == oc->missing_edge->next)
                                oc->missing_edge->next = mp->next->next;
                else {
                        fprintf(stderr,"\nstrange! face_1122 flag=1");
                        exit(0);
                }
                re_edge.type = '1';
                get_re_triangle_edge_vertices;

                if(insert_edge(re_edge,oc,ADD)) {
                        triangle_face(oc);
                }else {
                        fprintf(stderr,"\nre_edge not insert");
                        exit(0);
                }

        }else if(flag==3){
                mp=oc->missing_edge;
                if((mp->edge.type) == (mp->next->edge.type)) ;
                else    mp=mp->next;

                get_triangle_vertices;
                get_triangle_face_normal;

                add_to_o_f_table(oc->center,nor,TRIANGLE);

                if(mp == oc->missing_edge)      /* delete these two edges */
                                oc->missing_edge = mp->next->next;
                else if(mp == oc->missing_edge->next)
                                oc->missing_edge->next = mp->next->next;
                else {
                        fprintf(stderr,"\nstrange! flag=3");
                        exit(0);
                }
                re_edge.type = mp->edge.type;
                get_re_triangle_edge_vertices;

                if(insert_edge(re_edge,oc,ADD)){
                        triangle_face(oc);
                }else{
                        fprintf(stderr,"\nre_edge not insert");
                        exit(0);
                }
        }else {
                fprintf(stderr,"\nstrange!");
                exit(0);
        }
}
```

```
void face_11112(OPEN_CELL *oc)
{

        MISSING_EDGE    *mp,*pre;
        VECTOR          v1,v2,nor;
        EDGE            re_edge;
        int             flag=0;

        mp=oc->missing_edge;
        if(mp->edge.type == '1') {
            for(; mp->next->edge.type != '2'; pre=mp,mp=mp->next);
            if(mp->next->next == NULL){                /* mp points to 121 */
                mp->next->next = oc->missing_edge;
                oc->missing_edge = oc->missing_edge->next;
                mp->next->next->next=NULL;
             }
        }else {
                                                       /* mp points to 121 */
            for(; mp->next->next != NULL; pre=mp,mp=mp->next);
            mp->next->next=oc->missing_edge;
            oc->missing_edge = mp->next;
            mp->next = NULL;
            mp=oc->missing_edge;
        }

        /* test if the current edge and the next next edge are parallel */
        mp_edge_is_vector_v1;
        mp_next_next_edge_is_vector_v2;

        if(v1.x == v2.x && v1.y == v2.y && v1.z == v2.z){

            /* first choice: three triangles */
            pre= mp->next;
            mp = mp->next->next;
            if(mp->next == NULL) {
                mp->next = oc->missing_edge;
                oc->missing_edge = oc->missing_edge->next;
                mp->next->next=NULL;
            }
            get_triangle_vertices;
            get_triangle_face_normal;

            add_to_o_f_table(oc->center,nor,TRIANGLE);

            if(mp == oc->missing_edge)          /* delete these two edges */
                        oc->missing_edge = mp->next->next;
            else
                        pre->next = mp->next->next;

            re_edge.type = '2';
            get_re_triangle_edge_vertices;
```

```
             if(insert_edge(re_edge,oc,ADD)){
                 face_1122(oc);
             }else{
                 fprintf(stderr,"\nre_edge not insert");
                 exit(0);
             }
         }else {
                 /* second choice: one square face and one triangle */
                 if(mp->next->edge.end_v.x != mp->edge.start_v.x) flag++;
                 if(mp->next->edge.end_v.y != mp->edge.start_v.y) flag++;
                 if(mp->next->edge.end_v.z != mp->edge.start_v.z) flag++;

                 if(flag!=1){
                     flag=0;
                     pre=mp;
                     mp=mp->next;            /* mp: type '2' edge    */
                     if(mp->next->edge.end_v.x != mp->edge.start_v.x) flag++;
                     if(mp->next->edge.end_v.y != mp->edge.start_v.y) flag++;
                     if(mp->next->edge.end_v.z != mp->edge.start_v.z) flag++;
                 }

                 if(flag==1){
                     get_triangle_vertices;
                     get_triangle_face_normal;

                     add_to_o_f_table(oc->center,nor,TRIANGLE);

                     if(mp == oc->missing_edge)/* delete these two edges */
                             oc->missing_edge = mp->next->next;
                     else
                             pre->next = mp->next->next;

                     re_edge.type = '1';
                     get_re_triangle_edge_vertices;

                     if(insert_edge(re_edge,oc,ADD)){
                             square_face(oc);
                     }else{
                             fprintf(stderr,"\nre_edge not insert");
                     }
                 }else ;
        }
}

void    face_eight_1(OPEN_CELL *oc)
{
        MISSING_EDGE    *mp,*pre;
        VECTOR          v1,v2,out_nor,nor;
        EDGE            re_edge;
        int             flag=0, i_o=inside;
        float           dot_pro;
        void            face_six_1();
```

```
        if(oc->layer > 0) i_o = outside;

        mp=oc->missing_edge;
        mp_edge_is_vector_v1;
        mp_next_next_edge_is_vector_v2;

        if(!(v1.x == v2.x && v1.y == v2.y && v1.z == v2.z)) {
                pre= mp;
                mp = mp->next;
        }
        /* Now, mp, mp->next, mp->next->next are in the same plane */
        get_square_vertices;
        get_square_face_normal;

        /* out_nor is a vector pointing at the center of the voxel */
        out_nor.x = oc->center.x - mp->edge.start_v.x;
        out_nor.y = oc->center.y - mp->edge.start_v.y;
        out_nor.z = oc->center.z - mp->edge.start_v.z;

        dot_pro = nor.x*out_nor.x + nor.y*out_nor.y + nor.z*out_nor.z;
        if((i_o==outside && dot_pro<0.0) || (i_o==inside && dot_pro>0)){
                pre= mp->next;
                mp = mp->next->next;
        }

        /* Now, mp, mp->next, mp->next->next form a legal square face. */
        get_square_vertices;
        get_square_face_normal;
        add_to_o_f_table(oc->center,nor,SQUARE);

        if(mp == oc->missing_edge)
                oc->missing_edge = mp->next->next->next;
        else
                pre->next = mp->next->next->next;

        re_edge.type = '1';
        get_re_square_edge_vertices;
        if(insert_edge(re_edge,oc,ADD)){
                face_six_1(oc);
        }else{
                fprintf(stderr,"\nre_edge not added 1");
                exit(0);
        }
}

void    face_six_1(OPEN_CELL *oc)
{
        MISSING_EDGE    *mp,*pre;
        VECTOR          v1,v2,out_nor,nor;
        EDGE            re_edge;
        int             flag=0;
```

```
extern  void    add_to_single_cell_list();

mp=oc->missing_edge;
mp_edge_is_vector_v1;
mp_next_next_edge_is_vector_v2;

/* test if the current edge and the next next edge are in a plane */
if(!(flag=(v1.x == v2.x && v1.y == v2.y && v1.z == v2.z) ) ) {
  pre=mp;
  mp=mp->next;

  mp_edge_is_vector_v1;
  mp_next_next_edge_is_vector_v2;

  if(!(flag=(v1.x == v2.x && v1.y == v2.y && v1.z == v2.z))) {
    pre=mp;
    mp=mp->next;

    mp_edge_is_vector_v1;
    mp_next_next_edge_is_vector_v2;
    flag = (v1.x == v2.x && v1.y == v2.y && v1.z == v2.z);
  }
}
switch(flag){
 case 1:        /*    two square faces    */
    get_square_vertices;
    get_square_face_normal;
    add_to_o_f_table(oc->center,nor,SQUARE);

    if(mp == oc->missing_edge)
        oc->missing_edge = mp->next->next->next;
    else
        pre->next = mp->next->next->next;

    re_edge.type = '1';
    get_re_square_edge_vertices;
    if(insert_edge(re_edge,oc,ADD)){
        square_face(oc);
    }else{
        fprintf(stderr,"\nre_edge not added 2");
        exit(0);
    }
    break;

case 0:         /* remove a triangle face, left whatever */
    get_triangle_vertices;
    get_triangle_face_normal;
    add_to_o_f_table(oc->center,nor,TRIANGLE);

    if(mp == oc->missing_edge)
        oc->missing_edge = mp->next->next;
    else
```

```
                        pre->next = mp->next->next;

                re_edge.type = '2';
                get_re_triangle_edge_vertices;
                if(insert_edge(re_edge,oc,ADD)){
                        add_to_single_cell_list(oc,NULL);
                }else{
                        fprintf(stderr,"\nre_edge not added 3");
                        exit(0);
                }
                break;
        }
}

void    face_six_1_2(OPEN_CELL *oc)
{
        MISSING_EDGE    *mp,*pre;
        VECTOR          v1,v2,nor;
        EDGE            re_edge;
        int             i_o=inside;

        if(oc->layer > 0) i_o = outside;
        mp=oc->missing_edge;

        switch (i_o) {
          case outside:
                for(; mp->edge.type != '2'; pre=mp, mp=mp->next);
                /* mp points to 21 */
                if(mp->next == NULL){
                  mp->next = oc->missing_edge;
                  oc->missing_edge = oc->missing_edge->next;
                  mp->next->next = NULL;
                }
                get_triangle_vertices;
                get_triangle_face_normal;
                add_to_o_f_table(oc->center,nor,TRIANGLE);

                /* delete these two edges */
                if(mp == oc->missing_edge)
                        oc->missing_edge = mp->next->next;
                else
                        pre->next = mp->next->next;

                re_edge.type = '1';
                get_re_triangle_edge_vertices;

                if(insert_edge(re_edge,oc,ADD)) {
                        face_six_1(oc);
                }else{
                        fprintf(stderr,"\n re_edge not added");
                }
                break;
```

```
case inside:
    if(mp->edge.type == '2'){
        for(; mp->next != NULL;  pre=mp, mp=mp->next);
        /* mp points to the last */
        mp->next = oc->missing_edge;
        oc->missing_edge = mp;
        pre->next = NULL;
        /* mp points to 12, the first */
        get_triangle_vertices;
        get_triangle_face_normal;

        add_to_o_f_table(oc->center,nor,TRIANGLE);

        /* delete these two edges */
        oc->missing_edge = oc->missing_edge->next->next;

        re_edge.type = '1';
        get_re_triangle_edge_vertices;

        if(insert_edge(re_edge,oc,ADD)){
            face_six_1(oc);
        }else{
            fprintf(stderr,"\nre_edge not added 5");
            exit(0);
        }

    }else{
        for(; mp->next->edge.type != '2'; pre=mp, mp=mp->next);
        /* mp points to 12 */
        get_triangle_vertices;
        get_triangle_face_normal;

        add_to_o_f_table(oc->center,nor,TRIANGLE);

        /* delete these two edges */
        if(mp == oc->missing_edge)
            oc->missing_edge = mp->next->next;
        else
            pre->next = mp->next->next;

        re_edge.type = '1';
        get_re_triangle_edge_vertices;

        if(insert_edge(re_edge,oc,ADD)){
            face_six_1(oc);
        }else{
            fprintf(stderr,"\nre_edge not added 6");
        }
    }
    break;
```

```
          default:
                fprintf(stderr,"\nno such case for six_1_2");
        }
}

void    face_121112(OPEN_CELL *oc)
{
        MISSING_EDGE    *mp,*pre;
        VECTOR          v1,v2,nor;
        EDGE            re_edge;
        int             i_o=inside;

        if(oc->layer > 0) i_o = outside;
        mp=oc->missing_edge;
        switch (i_o) {
          case outside:
                for(; mp->edge.type != '2'; pre=mp, mp=mp->next);
                /* mp points to 21          */
                if(mp->next == NULL){
                  mp->next = oc->missing_edge;
                  oc->missing_edge = oc->missing_edge->next;
                  mp->next->next = NULL;
                }
                get_triangle_vertices;
                get_triangle_face_normal;
                add_to_o_f_table(oc->center,nor,TRIANGLE);

                /* delete these two edges   */
                if(mp == oc->missing_edge)
                        oc->missing_edge = mp->next->next;
                else
                        pre->next = mp->next->next;
                re_edge.type = '1';
                get_re_triangle_edge_vertices;
                if(insert_edge(re_edge,oc,ADD)) face_11112(oc);
                else  fprintf(stderr,"\nre_edge not added");
                break;

          case inside:
                break;

          default:
                fprintf(stderr,"\nno such case for 121112");
        }
}
```

Appendix B

Call Graph Profile Listing

```
                                 called/total       parents
index  %time    self descendents  called+self    name           index
                                 called/total       children

                                                    <spontaneous>
[1]     95.3    0.00       54.06                 start [1]
                0.00       54.06       1/1          _main [3]
                0.00        0.00       1/1          _on_exit [109]
                0.00        0.00       1/1          _exit [223]

-----------------------------------------------

                0.00       54.06       1/1          _main [3]
[2]     95.3    0.00       54.06       1         _Surface_tracking [2]
                0.53       38.39       1/1          _Border_face_tracking [4]
                8.25        2.59       1/1          _read_all_data [7]
                0.00        4.20       1/1          _close_surface [12]
                0.00        0.09       1/1          _make_table [53]
                0.00        0.01       2/3          _fprintf [76]
                0.00        0.00       1/1          _init_lists [103]
                0.00        0.00       1/1          .div [111]

-----------------------------------------------

                0.00       54.06       1/1          start [1]
[3]     95.3    0.00       54.06       1         _main [3]
                0.00       54.06       1/1          _Surface_tracking [2]
                0.00        0.00       3/34         _atoi [205]

-----------------------------------------------

                0.53       38.39       1/1          _Surface_tracking [2]
```

```
[4]     68.6    0.53       38.39          1        _Border_face_tracking [4]
                1.85       36.39    17742/17743        _neighbor_face [5]
                0.13        0.00    17743/17743        _remove_q_head [49]
                0.01        0.00          1/1          _span_start_face [86]
                0.00        0.00          1/2          _printf [85]
                0.00        0.00          1/1          _scanf [93]
                0.00        0.00          1/79582      _cross_zero [11]

-----------------------------------------------

                0.00        0.00          1/17743      _span_start_face [86]
                1.85       36.39    17742/17743        _Border_face_tracking [4]
[5]     67.4    1.85       36.40    17743          _neighbor_face [5]
                4.88       16.17    70583/70583        _search_neighbor_for_outway [6]
                4.52        4.98    70579/70579        _face_connection_for_outway [8]
                0.33        2.62   158909/171371       _calloc [16]
                1.31        1.59   158900/159373       _free [18]

-----------------------------------------------

                4.88       16.17    70583/70583        _neighbor_face [5]
[6]     37.1    4.88       16.17    70583          _search_neighbor_for_outway [6]
                4.87        0.00    79581/79582        _cross_zero [11]
                4.66        0.00   198551/319683       _transform_3d [9]
                0.41        2.94    70583/70583        _untranslate_matrix [15]
                2.85        0.09    70583/141173       _copy_matrix [10]
                0.08        0.26    17742/17742        _append_queue [34]

-----------------------------------------------

                8.25        2.59          1/1          _Surface_tracking [2]
[7]     19.1    8.25        2.59          1        _read_all_data [7]
                0.00        2.56         78/78         _fread [19]
                0.00        0.03          3/4          _fopen [62]
                0.00        0.00         80/163603     .mul [45]
                0.00        0.00          3/4          _fclose [102]
                0.00        0.00          2/171371     _calloc [16]
                0.00        0.00          2/159373     _free [18]

-----------------------------------------------

                4.52        4.98    70579/70579        _neighbor_face [5]
[8]     16.7    4.52        4.98    70579          _face_connection_for_outway [8]
                2.85        0.00   121132/319683       _transform_3d [9]
                1.11        1.03    14447/14447        _insert_to_hash_table [22]

-----------------------------------------------
                2.85        0.00   121132/319683       _face_connection_for_outway [8]
                4.66        0.00   198551/319683       _search_neighbor_for_outway [6]
[9]     13.2    7.51        0.00   319683          _transform_3d [9]

-----------------------------------------------
```

```
                0.00        0.00       7/141173       _make_table [53]
                2.85        0.09   70583/141173       _untranslate_matrix [15]
                2.85        0.09   70583/141173       _search_neighbor_for_outway [6]
[10]    10.4    5.71        0.17  141173          _copy_matrix [10]
                0.17        0.00  141173/163603       .mul [45]

-----------------------------------------------

                0.00        0.00       1/79582        _Border_face_tracking [4]
                4.87        0.00   79581/79582        _search_neighbor_for_outway [6]
[11]     8.6    4.87        0.00   79582          _cross_zero [11]

-----------------------------------------------

                0.00        4.20       1/1            _Surface_tracking [2]
[12]     7.4    0.00        4.20       1          _close_surface [12]
                0.09        4.04       1/1            _fill_edges [13]
                0.01        0.07       1/1            _add_o_e_to_hash_table [54]

-----------------------------------------------

                0.09        4.04       1/1            _close_surface [12]
[13]     7.3    0.09        4.04       1          _fill_edges [13]
                0.41        2.81    6068/7611         _match_edges [14]
                0.00        0.82       2/2            _close_single_cells [28]
                0.00        0.00       1/3            _fprintf [76]
                0.00        0.00       1/9            _fflush [208]

-----------------------------------------------

                0.10        0.71    1543/7611         _close_single_cells [28]
                0.41        2.81    6068/7611         _fill_edges [13]
[14]     7.1    0.51        3.52    7611          _match_edges [14]
                0.26        1.52    5980/6203         _add_edge_to_hash_table [23]
                0.19        0.32     890/891          _face_11112 [31]
                0.12        0.21     850/876          _face_six_1 [35]
                0.22        0.00   23615/23615        _match_vertex [41]
                0.08        0.11    1998/2911         _triangle_face [39]
                0.09        0.09    1642/3026         _square_face [37]
                0.04        0.09     392/860          _face_1122 [38]
                0.05        0.07    5077/5077         _insert_edge_to_mp_next [50]
                0.02        0.02     174/174          _face_1212 [59]
                0.01        0.01      23/23           _face_eight_1 [66]
                0.00        0.01     640/676          _add_to_single_cell_list [84]
                0.00        0.00       3/3            _face_six_1_2 [95]
                0.00        0.00       1/1            _face_121112 [98]

-----------------------------------------------

                0.41        2.94   70583/70583        _search_neighbor_for_outway [6]
[15]     5.9    0.41        2.94   70583          _untranslate_matrix [15]
```

```
                2.85        0.09   70583/141173       _copy_matrix [10]

-----------------------------------------------

                0.00        0.00        2/171371      _read_all_data [7]
                0.00        0.00       12/171371      _out_type_3_face [96]
                0.00        0.00       24/171371      _in_type_1_face [90]
                0.00        0.00       26/171371      _type_2_face [89]
                0.00        0.00       58/171371      _makenode [97]
                0.00        0.00      204/171371      _add_edge_to_hash_table [23]
                0.03        0.20    12136/171371      _insert_to_hash_table [22]
                0.33        2.62   158909/171371      _neighbor_face [5]
[16]     5.6    0.36        2.83   171371         _calloc [16]
                1.44        1.06   171371/217824      _malloc [17]
                0.17        0.00   171371/171838      .umul [46]
                0.16        0.00   171371/171371      _bzero [47]

-----------------------------------------------

                0.00        0.00        1/217824      _make_e_face_list [104]
                0.00        0.00        1/217824      _make_f_face_list [105]
                0.00        0.00        1/217824      _make_o_face_list [107]
                0.00        0.00        1/217824      _make_new_q [106]
                0.00        0.00        1/217824      _make_open_edge_list [108]
                0.00        0.00        1/217824      _on_exit [109]
                0.00        0.00        6/217824      __findbuf [77]
                0.01        0.00      676/217824      _add_to_single_cell_list [84]
                0.04        0.03     5077/217824      _insert_edge_to_mp_next [50]
                0.05        0.04     6367/217824      _add_edge_to_hash_table [23]
                0.14        0.10    16579/217824      _insert_edge [26]
                0.15        0.11    17742/217824      _append_queue [34]
                1.44        1.06   171371/217824      _calloc [16]
[17]     5.6    1.83        1.34   217824         _malloc [17]
                0.49        0.22   156280/159366      _delete [30]
                0.40        0.00    61544/61544       _demote [33]
                0.01        0.23      467/467         _morecore [40]

-----------------------------------------------

                0.00        0.00        2/159373      _read_all_data [7]
                0.00        0.00        4/159373      _fclose [102]
                0.00        0.00      467/159373      _morecore [40]
                1.31        1.59   158900/159373      _neighbor_face [5]
[18]     5.1    1.31        1.59   159373         _free [18]
                1.25        0.33   159373/159373      _insert [24]
                0.01        0.00     3086/159366      _delete [30]

-----------------------------------------------

                0.00        2.56      78/78           _read_all_data [7]
[19]     4.5    0.00        2.56      78          _fread [19]
                0.00        2.55    1100/1103         __filbuf [20]
```

```
                0.01        0.00     1170/1170         _memcpy [81]
                0.00        0.00       78/163603       .mul [45]

-----------------------------------------------

                0.00        0.00        1/1103         _number [91]
                0.00        0.00        2/1103         _getword [87]
                0.00        2.55     1100/1103         _fread [19]
[20]     4.5    0.00        2.56     1103          __filbuf [20]
                2.55        0.00     1103/1103         _read [21]
                0.01        0.00        5/6            __findbuf [77]
                0.00        0.00        1/9            _fflush [208]

-----------------------------------------------

                2.55        0.00     1103/1103         __filbuf [20]
[21]     4.5    2.55        0.00     1103          _read [21]

-----------------------------------------------

                1.11        1.03    14447/14447        _face_connection_for_outway [9]
[22]     3.8    1.11        1.03    14447          _insert_to_hash_table [22]
                0.60        0.12     8379/16579        _insert_edge [26]
                0.03        0.20    12136/171371       _calloc [16]
                0.07        0.01    14447/25765        _hash [48]

-----------------------------------------------
```

Bibliography

[AFH81] E. Artzy, G. Frieder, and G. T. Herman. The theory, design, implementation and evaluation of a three-dimensional surface detection algorithm. *Computer Vision, Graphics, and Image Processing*, 15:1–24, 1981.

[AGR84] H. H. Atkinson, I. Gargantini, and M. V. S. Ramanath. Determination of the 3d border by reperted elimination of internal surfaces. *Computing*, pages 279–295, 1984.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, 1983.

[Bro88] C. Brown, editor. *Advances in Computer Vision*, volume 2. Lawrence Erlbalum Associates, Publishers, 1988.

[CR89] J. D. Cappelletti and A. Rosenfeld. Three-dimensional boundary following. *Computer Vision, Graphics, and Image Processing*, 48:80–92, 1989.

[DT84] L. I. Doctor and J. G. Torborg. Display techniques for octree-encoded objects. *IEEE Computer Graphics and Applications*, pages 5–16, 1984.

[Far84] E. J. Farrell. Color 3-d imaging of normal and pathologic intracranial structures. *IEEE Computer Graphics and Applications*, pages 5–16, September 1984.

[FGR85] G. Frieder, D. Gordon, and R. A. Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics and Applications*, pages 52–60, January 1985.

[Gar82] I. Gargantini. Linear octtrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing*, 20:365–374, 1982.

[Gar86] I. Gargantini. Viewing transformation of voxel-based objects via linear octrees. *IEEE Computer Graphics and Applications*, pages 12–21, 1986.

[GB87] M. Green and N. Bridgeman. *WINDLIB Programmer's Manual.* Department of Computing Science, University of Alberta, Edmonton, Alberta, September 1987.

[GU89] D. Gordon and J. K. Udupa. Fast surface tracking in three-dimensional binary images. *Computer Vision, Graphics, and Image Processing*, 45:196–214, 1989.

[GW87] Raface C. Gongaalez and Paul Wintz. *Digital Image Processing.* Addison-Wesley Publising Company, 1987.

[HL79] G. T. Herman and H. K. Liu. Three-dimensional display of human organs from computed tomograms. *Computer Vision, Graphics, and Image Processing*, 9:1–21, 1979.

[HS79] G. M. Hunter and K. Steiglitz. Operations on images using quad tres. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):145–153, 1979.

[HW83] G. T. Herman and D. Webster. A topological proof of a surface tracking algorithm. *Computer Vision, Graphics, and Image Processing*, 23:162–177, 1983.

[JT80] C. L. Jackins and S. L. Tanimoto. Oct-treeess and their use in representing three-dimensional objects. *Computer Vision, Graphics, and Image Processing*, 14:249–270, 1980.

[KR89] T. Y. Kong and A. Rosenfeld. Digital topology: Introduction and survey. *Computer Vision, Graphics, and Image Processing*, 48:357–393, 1989.

[LC87] W. E. Lorensen and H. E. Cline. Marching cubes: a high resolution 3d surface construction algorithm. *ACM Computer Graphics*, 21(4):163–169, 1987.

[Liu77] H. K. Liu. Two- and three-dimensional boundary detection. *Computer Vision, Graphics, and Image Processing*, 6:123–134, 1977.

[Man88] M. Mantyla. *An Introduction to Solid Modeling.* Computer Science Press, 1988.

[Mar76] A. Martelli. An application of heuristic search methods to edge and contour detection. *Communication of the ACM*, 19(2):73–83, February 1976.

[Mea82a] D. J. Meagher. Efficient synthetic image generation of arbitrary 3-d objects. *Proceedings of IEEE Computer Science Conference on Pattern Analysis and Image Processing*, pages 473–478, 1982.

[Mea82b] D. J. Meagher. Geometric modeling using octree encoding. *Computer Vision, Graphics, and Image Processing*, 19:129–147, 1982.

[MH80] D. Marr and E. Hildreth. Theory of edge detection. *Proceeding of the Royol Society of London*, 207:187–217, 1980.

[MR81] D. G. Morgenthaler and A. Rosenfeld. Multidimensional edge detection by hypersurface fitting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3(4):187–217, July 1981.

[Rey85a] R. A. Reynolds. *Fast Methods for 3D Display of Medical Objects.* PhD thesis, University of Pennsylvania, 1985.

[Rey85b] R. A. Reynolks. *Fast Methods for 3D Display of Medical Objects.* PhD thesis, University of Pennsylvania, 1985.

[Rey87a] R. A. Reynolds. A dynamic screen technique for shaded graphics display of slice-represented objects. *Computer Vision, Graphics, and Image Processing*, 38:275–298, 1987.

[Rey87b] R. A. Reynolds. A dynamic screen technique for shaded graphics display of slice-represented objects. *Computer Vision, Graphics, and Image Processing*, 38:275–298, 1987.

[Sam80] H. Samet. Region representation: Quadtrees from binary arrays. *Computer Vision, Graphics, and Image Processing*, 13(1):88–93. 1980.

[Sam81] H. Samet. An algorithm for converting raster to quadtres. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3(1):93–95, 1981.

[Sri81] S. N. Srihari. Representation of three-dimensional images. *Computing Surveys*, pages 401–422, 1981.

[SZ87] P. T. Sander and S. W. Zucker. Tracing surface for surface traces. *IEEE International Conference on Computer Vision*, pages 241–249, 1987.

[TT84] H. K. Tuy and L. T. Tuy. Direct 2-d display of 3-d objects. *IEEE Computer Graphics and Applications*, pages 29–33, October 1984.

[USH82] J. K. Udupa, S. N. Srihari, and G. T. Herman. Boundary detection in multidimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4:41–47, 1982.

[YS83] M. Yau and S. N. Srihari. A hierarvhical data structure for multidimensional digial images. *Communication of the ACM*, 26(7):504–515, 1983.

[ZD91] J. Zhao and W. A. Davis. Fast display of octree representations of 3d objects. *Proceedings of Graphics Interface'91*, pages 160–167, 1991.

[ZH79] S. W. Zucker and R. A. Hummel. An optimal three-dimensional edge operator. *Proceedings of IEEE Computer Science Conference on Pattern Analysis and Image Processing*, pages 162–168, 1979.