การแปลงแบบจำลองโครงลวดสามมิติให้เป็นแบบจำลองปริภูมิโครงสร้างเซลล์

นาย วรากร อึ้งวิเชียร

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

ปีการศึกษา 2549

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CONVERTING THREE DIMENSIONAL WIREFRAME MODEL TO CELLULAR STRUCTURED SPACE MODEL

Mr. Varakorn Ungvichian

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering Program in Computer Engineering

Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University

Academic Year 2006

Copyright of Chulalongkorn University

Thesis Title	CONVERTING THREE DIMENSIONAL WIREFRAME MODEL TO CELLULAR STRUCTURED SPACE MODEL
By	Mr. Varakorn Ungvichian
Field of Study	Computer Engineering
Thesis Advisor	Pizzanu Kanongchaiyos, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

.... Dean of the Faculty of Engineering

(Professor Direk Lavansiri, Ph.D.)

THESIS COMMITTEE

.... Chairman

(Associate Professor Somchai Prasitjutrakul, Ph.D.)

.... Thesis Advisor

(Pizzanu Kanongchaiyos, Ph.D.)

.... Member

(Atiwong Suchato, Ph.D.)

.... Member

(Chakrit Watcharopas, Ph.D.)

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

วรากร อิ้งวิเชียร : การแปลงแบบจำลองโครงลวดสามมิติให้เป็นแบบจำลองปริภูมิโครงสร้างเซลล์. (CONVERTING THREE DIMENSIONAL WIREFRAME MODEL TO CELLULAR STRUCTURED SPACE MODEL) อ. ที่ปรึกษา : อ. ดร. พิษณุ คนองชัยยศ, 61 หน้า.

ปัจจุบันการใช้ข้อมูลสามมิติในคอมพิวเตอร์มีปัญหาสำคัญข้อหนึ่งคือ คือ ข้อมูลไม่สามารถใช้งานร่วมกันได้อย่างมีประสิทธิภาพ เนื่องจากการแทนข้อมูลสามมิตินั้นทำได้หลายรูปแบบ เช่นโครงลวด ขอบมีปีก แบบจำลองพื้นผิว ฯลฯ ซึ่งแต่ละรูปแบบใช้ลักษณะโครงสร้างข้อมูลต่างกัน ทำให้ไม่สามารถใช้งานร่วมกันได้ และต้องมีวิธีการแปลงรูปแบบข้อมูลสามมิติจากรูปแบบหนึ่งไปยังอีกรูปแบบ ซึ่งอาจทำให้เกิดปัญหาความไม่ถูกต้องหรือไม่สอดคล้องกันของข้อมูลได้ ซึ่งวิธีที่นิยมคือการแปลงเป็นแบบจำลองโครงลวดเป็นข้อมูล เนื่องจากเป็นแบบจำลองที่มีความซับซ้อนน้อย และสามารถแปลงจากแบบจำลองอื่นๆได้ง่ายแต่ก็ยังมีข้อจำกัดคือขาดข้อมูลลักษณะเฉพาะของข้อมูลสามมิติ

งานวิจัยนี้นำเสนอวิธีการแปลงข้อมูลสามมิติให้เป็นแบบจำลองปริภูมิโครงสร้างเซลล์ เพื่อให้ใช้แทนข้อมูลสามมิติได้อย่างมีประสิทธิภาพโดยใช้แบบจำลองโครงลวดเป็นข้อมูลนำเข้าแล้วการค้นหาจุด เส้นขอบ และหน้าต่าง ๆ ที่เป็นไปได้จากแบบจำลองโครงลวด แยกเป็นเซลล์แต่ละมิติ แล้วจัดเรียงให้เป็นโครงสร้างเซลล์ที่เหมาะสมที่สุดที่มีความสัมพันธ์เชิงทอพอโลยีระหว่างส่วนต่าง ๆ

ในงานวิจัยได้นำแบบจำลองปริภูมิโครงสร้างเซลล์ที่นำเสนอมามาทดลองประยุกต์ใช้ หาส่วนคอดที่สุดของวัตถุ อีกทั้งประยุกต์ใช้ในการปรับรูปทรงให้เรียบขึ้นโดยแปลงเส้นขอบจากเส้นตรงเป็นเส้นโค้ง ผลลัพธ์ของการทดลองทั้งสองแสดงให้เห็นประสิทธิภาพของแบบจำลองปริภูมิโครงสร้างเซลล์ว่าสามารถแทนลักษณะเฉพาะเชิงทอพอลอยีของวัตถุได้อย่างถูกต้อง ซึ่งประยุกต์ใช้แก้ปัญหาทางด้านคอมพิวเตอร์กราฟิกส์ได้ด้วยความซับซ้อนเชิงเวลาในระดับที่ยอมรับได้

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

ภาควิชา..............วิศวกรรมคอมพิวเตอร์.............. ลายมือชื่อนิสิต..
สาขาวิชา..............วิศวกรรมคอมพิวเตอร์............ ลายมือชื่ออาจารย์ที่ปรึกษา..
ปีการศึกษา ....2549....................................

4870454621 : MAJOR COMPUTER ENGINEERING
KEY WORD: TOPOLOGY / DATA REPRESENTATION / CELLULAR MODEL

VARAKORN UNGVICHIAN : CONVERTING THREE DIMENSIONAL WIREFRAME MODEL TO CELLULAR STRUCTURED SPACE MODEL. THESIS ADVISOR : PIZZANU KANONGCHAIYOS, PH.D., 61 pp.

An important problem in using three-dimensional data in computer graphics is effective data incompatibility, due to the various methods available to store three-dimensional data, such as wireframe modeling, winged-edge modeling, or surface modeling, etc. Each format uses different data structures, rendering them incompatible, and necessitating conversions from one format to another, which may result in incorrect or inconsistent data. The preferred structure for conversion output is usually the wireframe, because of its low complexity, and it is trivial to be converted from other models. However, it still lacks topological property of the three-dimensional data.

This research proposes a method to convert three-dimensional data into a cellular structured space model, to efficiently store three-dimensional topological information. We use a wireframe as input, and then search for its points, edges, and possible faces for categorizing into cells with various dimensions. Finally, the cells are arranged into the appropriate cellular structure providing the topological relations between each component of the model.

This research also explores possible applications using the proposed structure: we find the thinnest points in a solid, and we also smooth out a solid by converting its straight edges into curves. The experimental results show that the cellular structured space model correctly provides objects' topological properties, and can be applied to correctly solving problems in computer graphics with reasonable time complexity.

Department...Computer Engineering..............Student's signature...........................
Field of study.. Computer Engineering............Advisor's signature...........................
Academic year 2006................

Acknowledgements

First of all, I will thank my thesis advisor, Dr. Pizzanu Kanongchaiyos, for his useful advice and research suggestions, as well as the thesis committee for this project: Assoc. Prof. Dr. Somchai Prasitjutrakul, Dr. Atiwong Suchato, Dr. Chakrit Watcharophas, and Mr. Pinyo Jinuntuya. They have provided me with much useful feedback. For these I am very grateful.

I also thank all the members of the Computer Graphics and Animation laboratory (CG&A), especially Mr. Rungvit Laichuthai, for their additional support.

I also thank the Graduate School of Chulalongkorn University for providing me with financial support to present my research at various conferences, namely the Computer Aided Design and Applications Conference 2006 at Phuket between June 19-23 2006, and the 15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2007 in the Czech Republic between January 29-February 2 2007.

Lastly, I also have to thank my family for their love and support through all my years of education.

Contents

Chapter	Page
Abstract (Thai)	iv
Abstract (English)	v
Acknowledgements	vi
Contents	vii
I. Introduction	1
1.1 The problem	1
1.2 Purpose of the research	2
1.3 Limitations	2
1.4 Expected benefits	2
1.5 Outline	2
II. Related Research and Background	3
2.1 2D Wireframe to 3D Solid Conversion	3
2.2 Finding Curvilinear Structures in Images	4
2.3 Point Sampled Cell Complexes	5
2.4 Modeling Using Cellular Structured Space	6
2.5 Conclusions	7
III. Background	8
3.1 Three-dimensional Cell List	8
3.2 Cellular Design System	9
IV. 3D Cellular Structure Conversion Algorithm	10
4.1 Reading vertices and edges	11
4.2 Face detection	11
4.3 Reducing to unique circuits	13
4.4 Face reduction	15
4.5 Face arrangement	17
4.6 Extra face processing	19
4.7 Object and volume detection	19
4.8 Finding relations	25

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Chapter Page

4.9 Output 27
V. Thin Point Finding Algorithm 28
5.1 Pre-processing and finding central axis 29
5.2 Obtaining and processing selections 31
5.3 Determining thin points from obtained values 35
5.4 Thresholding 37
VI. Edge Smoothing Algorithm 39
6.1 Pre-processing and finding continuous surface 40
6.2 Obtaining data to smooth surface 40
6.3 Find normal vectors 41
6.4 Redefining edges as curves 42
6.5 Readjusting midpoints 46
VII. Experimentation and Results 49
7.1 Tools for experimentation 49
7.2 Wireframe to Three-dimensional Cell List Conversion 49
7.3 Finding Thin Points in 3DCL 54
7.4 Smoothing out 3DCL 57
VIII. Conclusions and future improvement 59
8.1 Conclusions 59
8.2 Future improvement 60
References 62
Vita 63

Figures

Figure Page

1. Examples of data representations ... 1
2. Converting a 2D wireframe drawing into 3D solid ... 4
3. Detecting blood vessels in MRA data ... 5
4. An example of a Point sampled cell complex ... 6
5. A teacup in cellular structured space model ... 6
6. An example of a Three-dimensional cell list ... 9
7. Changing a bag's design in the cellular design system ... 9
8. Flowchart of the 3D cellular structure conversion algorithm ... 10
9. Results from reading in the edges ... 11
10. Tracing paths ... 12
11. Removing paths ... 13
12. Identical circuits from different traces ... 14
13. Two different paths representing the same face ... 14
14. Circuit ordering ... 15
15. Face splitting to find area ... 16
16. Calculating flatness ... 16
17. Storing faces adjacent to each edge and vertex ... 18
18. Tracing an object ... 19
19. Flowchart of finding volumes ... 20
20. Finding the leftmost face ... 21
21. Determining the next face with turn direction ... 22
22. For case 1 ... 23
23. For case 2 ... 23
24. Special case solution ... 25
25. Determining a face's outward pointer from that of an adjacent face ... 26
26. Right hand rotation around edge a ... 27
27. Flowchart of the Thin Point Finding Algorithm ... 28
28. Determining axes using averages ... 30

Figure Page

29. Flowchart of axis finding 31
30. Creating new selections 32
31. Face adjacency 32
32. Keeping track of edges with lists 33
33. Flowchart of selection processing 35
34. Averaging 36
35. Flowchart of edge smoothing algorithm 39
36. Faces, edges, and vertices used to smooth the surface 40
37. Determining the normal vector of an edge from its adjacent faces 41
38. Determining the normal vector of a vertex from its adjacent edges 42
39. Obtaining derivatives from normal vectors 43
40. Determining the new midpoint of an edge 47
41. Simple examples 49
42. Complex examples 49
43. Examples for thin point finding 55
44. Stanford Bunny 57

Tables

Table Page

1. Result of the simple examples 50
2. Visual results of the simple examples 51
3. Results of the complex examples 52
4. Visual results of the complex examples 53
5. Time to find thin points 55
6. Visual results for thin points 56
7. Results for 3DCL smoothing 58

CHAPTER I
INTRODUCTION

1.1 The problem

Computer graphics have developed from two dimensions to three dimensions, allowing for versatile usage, such as for design. There are many methods of data representation in three-dimensional computer graphics. However, different representations are often inherently non-interchangeable, due to the storage of different data to represent the object's shape. For example, a wireframe (Figure 1, upper left) generally stores information on the edges that comprise the structure, while Baumgart's winged-edge structure [1] (upper right) stores the faces and edges that are adjacent to each edge. Other representations include surface models (bottom left) and solid models (bottom right). Therefore, methods to convert between various representations are required.

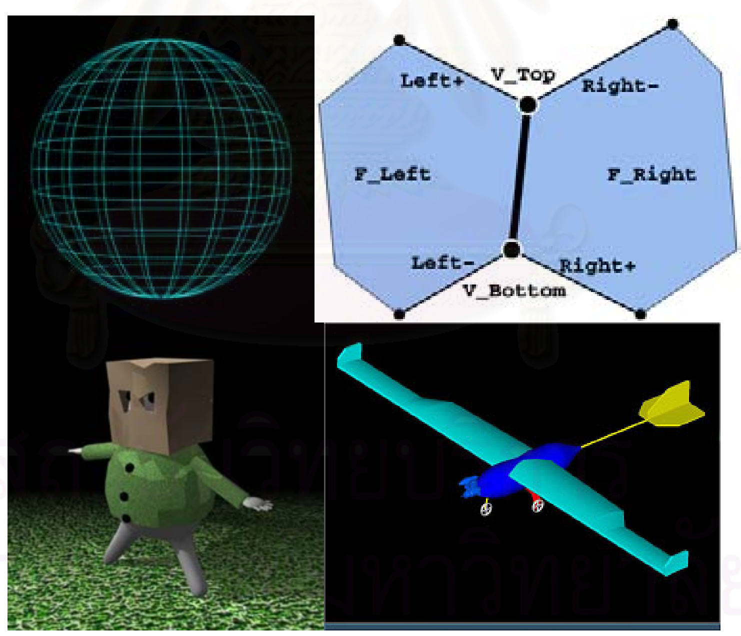


Figure 1. Examples of data representations

Among the many representations of three-dimensional computer graphics that have been developed is the "Three-dimensional Cell List" format by Kovalevsky [2]. This structure has the property of being able to efficiently represent

topological data, which is useful in some applications such as product design. This structure will be the output of this research.

1.2 Purpose of the research

In this research, we will describe a method to create a solid in Three-dimensional Cell List format from a wireframe. We will also describe two algorithms based on this new format, one to determine the thinnest points of a solid represented in the format, and another to make the solid smoother by re-defining the edges as cubic and quadratic curves.

1.3 Limitations

The wireframes used for this algorithm comprise entirely of straight edges, have a genus of 0, and do not represent shapes overlapping each other. We have not accounted for any of these anomalies.

1.4 Expected benefits

1) To learn about finding topological data from wireframe information

2) To provide a method for converting wireframe information into Three-dimensional Cell List format

3) To create applications for the Three-dimensional Cell List format

1.5 Outline

This research is outlined as follows: We describe related research in Chapter II, and background research in Chapter III. We described the algorithms we have developed in Chapters IV-VI. We describe our experiments and results for each algorithm in Chapter VII, and we discuss the results in Chapter VIII.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER II
RELATED WORKS

In this chapter, we will discuss a selection of previous research related to the research on the algorithms presented in this paper. We will describe methods to convert 2D wireframes into 3D solids, find curvilinear structures in 3D images, and model solids with points.

2.1 2D Wireframe to 3D Solid Conversion (Shpitalni and Lipson, etc.)

In 1996, Shpitalni and Lipson [3] developed an algorithm for converting a 2D wireframe drawing into a 3D shape, as shown in Figure 2, with these steps:

1. Converting the drawing into a graph
2. Finding possible faces, by finding every circuit in the graph without any intersecting edges
3. Removing faces comprised of two smaller ones from consideration
4. Using a face adjacency theorem to reduce the faces for searching: two faces can co-exist in a volume only if their common edge has a continuous first derivative
5. Calculating the maximum rank, that is, the maximum number of faces that are adjacent to each edge or vertex, using geometrical principles
6. Finding the best arrangement of faces, using a function consisting of the absolute difference between the calculated values and the actual values
7. Searching in a tree to find the result with the best value from the above function
8. Extra considerations, if necessary, such as skewed orthography

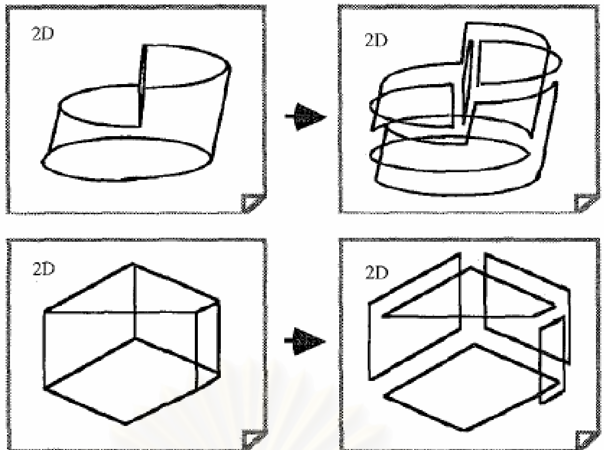


Figure 2. Converting a 2D wireframe drawing into 3D solid

Shpitalni and Lipson's algorithm was improved on by later research. Liu and Lee [4] developed a depth first search algorithm for finding faces, and an algorithm for finding the best arrangement based on finding the maximum weight clique. Oh and Kim [5] enhanced the algorithm by splitting faces into 3 categories, and using "sketch order analysis" (i.e., determining which faces were most likely to have been drawn first). However, while these algorithms produce accurate results, the main limitation of these algorithms is that they were designed to work on 2D drawings, rather than actual 3D wireframes.

2.2 Finding Curvilinear Structures in Images (Koller et al., Danielsson and Lin)

Koller et al. [6] described using multiscale linear filtering to detect curvilinear structures in 2D and 3D images. Danielsson and Lin [7] also described a method to achieve a similar goal using Hessian matrices and spherical harmonics. In both works, the researchers cited detection of blood vessels in MRA (magnetic resonance angiography) data as a possible application for detecting curvilinear structures in 3D, as seen in Figure 3. However, it concentrates solely on curvilinear structures, rather than just finding thin points.

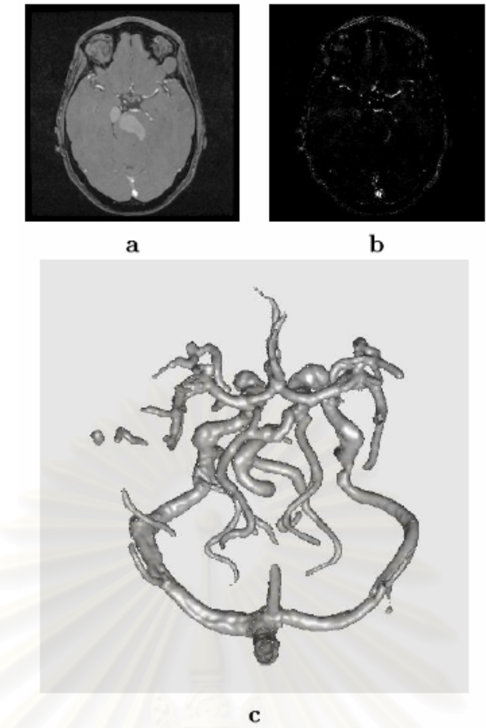


Figure 3. Detecting blood vessels in MRA data

2.3 Point Sampled Cell Complexes (Adamson and Alexa)

Adamson and Alexa [8] proposed a model to define a piecewise smooth surface using point samples, by projecting a surface on to the samples and using connectivity information to glue surface patches to curves and curves to points (vertices). They also described methods for interpolating tangents across cell boundaries to create tangential continuity. Figure 4 shows an example of this model. However, the limitation of this model is that it requires uniform sampling for best results. Also, due to the use of point sampling, the model may be more unstable.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

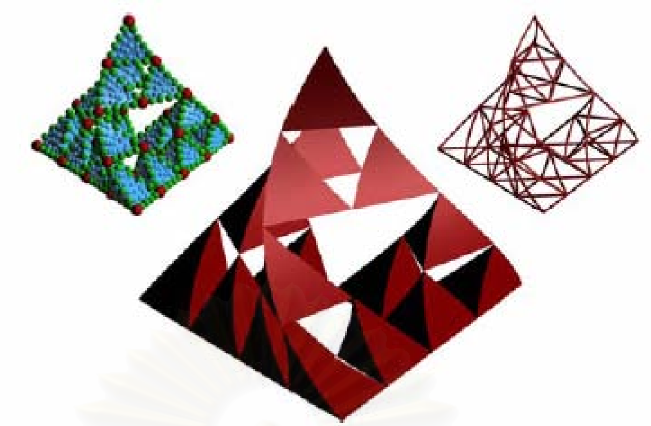


Figure 4. An example of a Point sampled cell complex

2.4 Modeling Using Cellular Structured Space (Charussuriyong and Kanongchaiyos, 2005)

Charussuriyong and Kanongchaiyos [9] described a method to create 3D objects using the cellular structured space model. It uses topology features to check the validity of the model, such that invariant and some topological properties are preserved. The model also lends well to being used to find similarities between 3D objects in a multimedia database. Figure 5 shows a teacup being modeled. However, Charussuriyong and Kanongchaiyos' paper describes mainly methods to create an object in this model from scratch, rather than to convert from other models.

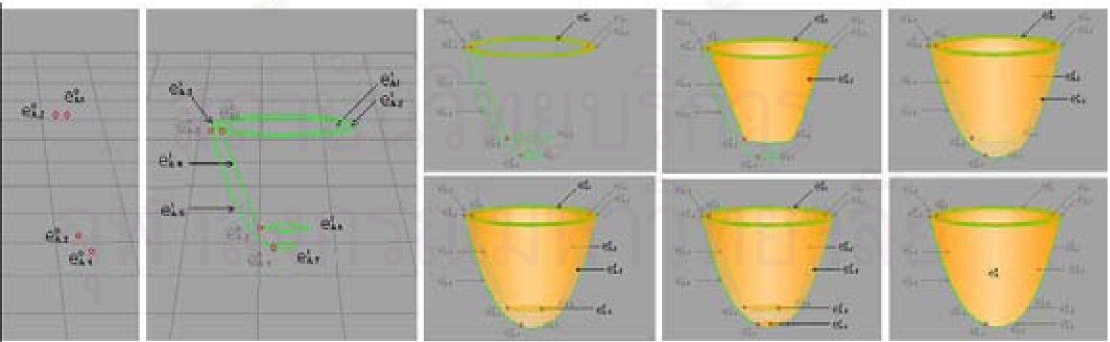


Figure 5. A teacup in cellular structured space model

2.5 Conclusions

The previous research in converting 2D drawings to 3D solids, while useful as an approach, is limited by its use of 2D drawings, rather than 3D wireframes, as input. The previous work in modeling 3D objects using cellular structured space model is designed for creating objects from scratch, rather than converting from other forms (such as wireframe). Lastly, the other research works, while similar in concept to those presented in this paper, are not fully compatible: the curvilinear structure research is designed to search for long thin lines rather than just thin points, while tangent interpolation on point samples to smooth out the represented figure is less stable than interpolation using the method we will describe.

CHAPTER III
THEORETICAL BACKGROUND

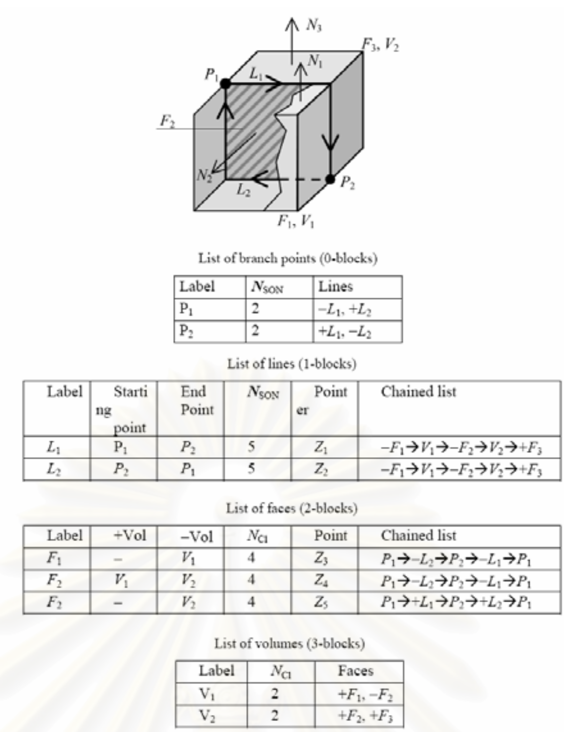
In this chapter, we will discuss previous research that provides a background to the algorithms presented in this paper. We will describe a cell-based data structure, the Three-dimensional Cell List, and an application of a similar structure in product design.

3.1 Three-dimensional Cell List (Kovalevsky)

Kovalevsky [2] explained the requirements for a data structure to efficiently store 3D solid topology: that there is sufficient topological data to find the relationships between various parts of the structure without a search, and that it can correctly represent non-proper complexes, which contain much less elements than the corresponding proper complex, and are often used in topological investigation for that reason.

Kovalevsky demonstrated his concepts for abstract cellular complex data structures that satisfy these requirements, the Two-dimensional Cell List and the Three-dimensional Cell List (abbreviated from this point as 3DCL). The latter is designed for 3D structures, and is comprised of lists of vertices, edges, faces, and volumes, as well as the relations between various elements, e.g., the edges that are adjacent to a given vertex. Figure 6 shows an example of a 3DCL on a figure with two vertices, two edges, two faces, and two volumes, explaining the structure of the 3DCL.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย



List of branch points (0-blocks)

Label	N_{SON}	Lines
P_1	2	$-L_1, +L_2$
P_2	2	$+L_1, -L_2$

List of lines (1-blocks)

Label	Starting point	End Point	N_{SON}	Pointer	Chained list
L_1	P_1	P_2	5	Z_1	$-F_1 \rightarrow V_1 \rightarrow -F_2 \rightarrow V_2 \rightarrow +F_3$
L_2	P_2	P_1	5	Z_2	$-F_1 \rightarrow V_1 \rightarrow -F_2 \rightarrow V_2 \rightarrow +F_3$

List of faces (2-blocks)

Label	+Vol	−Vol	N_{CI}	Point	Chained list
F_1	–	V_1	4	Z_3	$P_1 \rightarrow -L_2 \rightarrow P_2 \rightarrow -L_1 \rightarrow P_1$
F_2	V_1	V_2	4	Z_4	$P_1 \rightarrow -L_2 \rightarrow P_2 \rightarrow -L_1 \rightarrow P_1$
F_2	–	V_2	4	Z_5	$P_1 \rightarrow +L_1 \rightarrow P_2 \rightarrow +L_2 \rightarrow P_1$

List of volumes (3-blocks)

Label	N_{CI}	Faces
V_1	2	$+F_1, -F_2$
V_2	2	$+F_2, +F_3$

Figure 6. An example of a Three-dimensional cell list

3.2 Cellular Design System (Matsumoto and Kunii, 2002)

Matsumoto and Kunii [10] described an application of cellular-based data structures: Using them to design soft and varied-sized objects. One advantage of using cellular-based data structures for designing such objects is that the configuration of the object can be changed by simply changing the attributes needed, as shown in Figure 7. Matsumoto and Kunii used the design of bags to illustrate their new system.

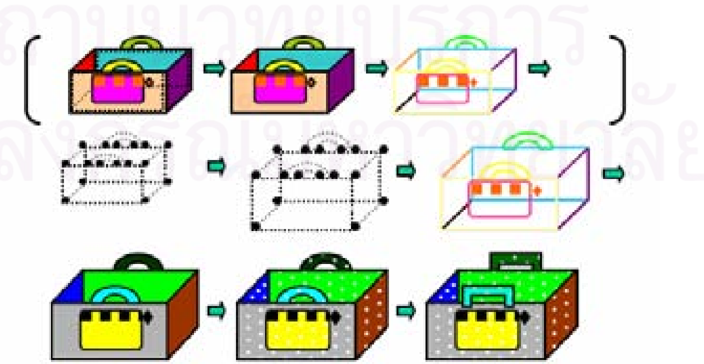


Figure 7. Changing a bag’s design in the cellular design system

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER IV

3D CELLULAR STRUCTURE CONVERSION ALGORITHM

In this chapter, we will explain the proposed algorithms we have devised for converting wireframes to Three-dimensional Cell Lists. The steps of the algorithms are as follows: we read in the vertices and edges (as described in section 4.1), we detect the faces from the data we have read (described in section 4.2), then we reduce the circuits found to unique circuits (described in section 4.3) before reducing to the most plausible faces (described in section 4.4), and then we select the faces that produce the most likely shape (described in section 4.5) and perform some extra processing on the faces (described in section 4.6), before detecting objects and volumes (described in section 4.7), finding the relations between the various elements (described in section 4.8), and finally outputting all the data found (described in section 4.9). Figure 8 describes the process as a flowchart.

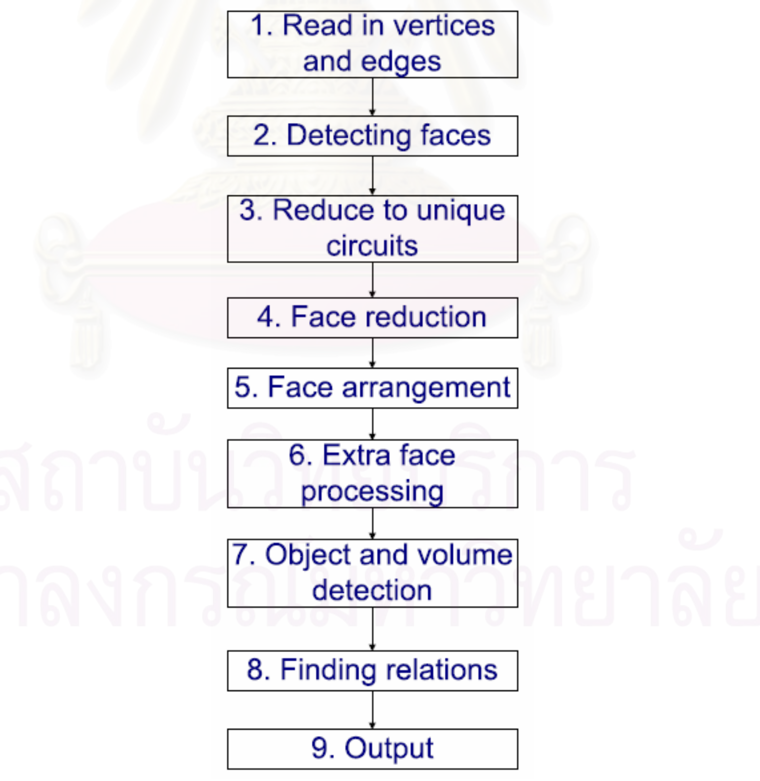
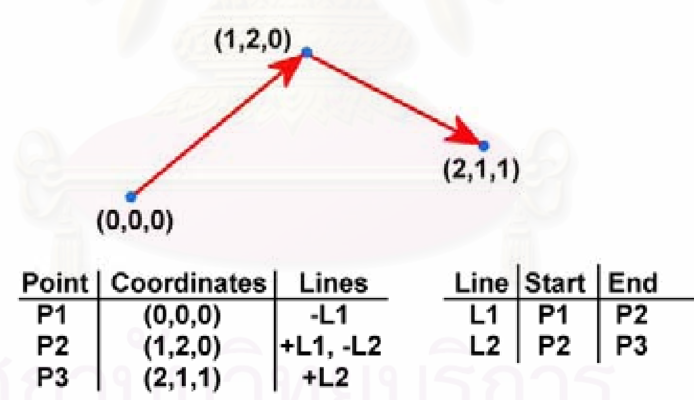


Figure 8. Flowchart of the 3D cellular structure conversion algorithm

4.1 Reading vertices and edges

Converting a wireframe to a Three-dimensional Cell List begins with obtaining the data of each vertex and edge. We read each edge in, and record the coordinates of its two vertices. If a vertex is not already in the list, it is added. At the end of this step, we have a list of vertices in the shape. This list will be sorted according to the *x*, *y* and *z* coordinates of each vertex, that is, a vertex with a lower *x* value will appear earlier in the list, and each vertex will then be labeled according its list position.

Next, we compare the vertices of each edge with the list we have obtained, and store for each vertex the coordinates of the vertex, the edges adjacent to the vertex, and the direction of the edge in relation to the vertex. For each edge, we store the start and end vertices of the edge. Figure 9 shows a simple example with three vertices and two edges.



Point	Coordinates	Lines
P1	(0,0,0)	-L1
P2	(1,2,0)	+L1, -L2
P3	(2,1,1)	+L2

Line	Start	End
L1	P1	P2
L2	P2	P3

Figure 9. Results from reading in the edges

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

4.2 Face detection

The first major step is finding the faces, by tracing along various edges and finding circuits. We store the path we trace as a series of edges and direction values (Boolean): if the value associated with a given edge is true, the path runs from the start vertex to the end vertex of that edge; if it is false, the path runs in the opposite

direction. We begin processing each edge by selecting the vertex that is adjacent to the lesser number of edges. If both vertices have the same number of edges, we select the edge's end vertex by default. After one vertex is selected, we initialize the path by starting with the current edge, and setting the direction value so that the path runs from the opposite end to the selected vertex.

In each step, we process each non-circuit path in the array, by finding the latest vertex of the path (from the latest edge and its associated direction value), attaching each edge adjacent to this vertex (besides those already part of the path), and storing the new path in a new array. For example, if there are two edges (besides the latest edge in the current path) adjacent to the latest vertex, we will put two new paths into the new array. Figure 10 shows an example of tracing.

From the new array, we remove paths with circuits within the path (as opposed to being the whole path), and any path starting with the same two edges as a found circuit. Figure 11 shows these two methods of path removal.

We repeat the process with the new array until there are no changes in the array, and then write the array to a file. This method of searching is equivalent to a breadth-first search in a tree.

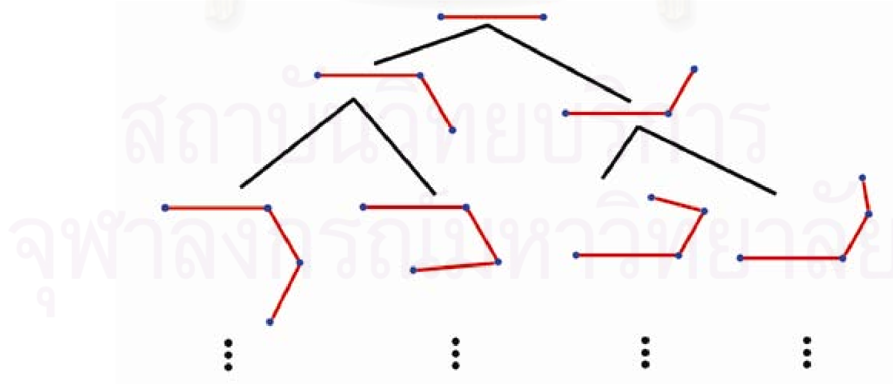


Figure 10. Tracing paths

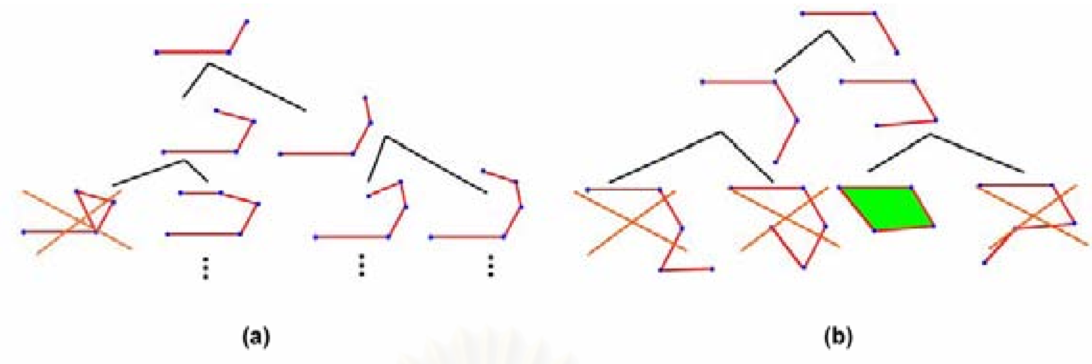


Figure 11. Removing paths

4.3 Reducing to unique circuits

When processing each edge, we will most likely find the same circuit by tracing from each of the different edges that comprise that circuit (see Figure 12 for an example). Therefore, after all edges have been processed, we reduce the file down to unique circuits, by reading in each circuit, and comparing it to the previously read circuits stored in a table. We search each circuit in the table for the first edge of the circuit. If the first edge of the circuit is not in a given circuit in the table, then it can be ignored, since it is definitely different from the current circuit. If the edge is in the circuit in the table, we then check the direction values associated with the edge in both circuits. If the direction values are the same, we traverse both circuits in the same direction. If they are opposite, we traverse the circuits in opposite directions. If we return to the initial edge without finding any different edges, the circuits are the same.

Example (as shown in Figure 13):

$$F_1 : A\,(+\alpha)\,B\,(-\delta)\,D\,(-\beta)\,C\,(+\gamma)\,A$$
$$F_2 : D\,(+\delta)\,B\,(-\alpha)\,A\,(-\gamma)\,C\,(+\beta)\,A$$

The first edge of F_1 is $+\alpha$. We find $-\alpha$ in F_2. Since these are in opposite directions, we traverse these circuits in opposite directions:

- F_1: $+\alpha$ ($\longrightarrow$), F_2: $-\alpha$ ($\longleftarrow$)
- F_1: $-\delta$ ($\longrightarrow$), F_2: $+\delta$ (loop back to the end)

- F_1: $-\beta$ ($\longrightarrow$), F_2: $+\beta$ ($\longleftarrow$)
- F_1: $-\gamma$ (loop back to the start), F_2: $+\gamma$ ($\longleftarrow$)
- F_1: $+\alpha$, F_2: $-\alpha$ (initial edge)

We have returned to the initial edge without finding any different edges. Therefore, $F_1 = F_2$.

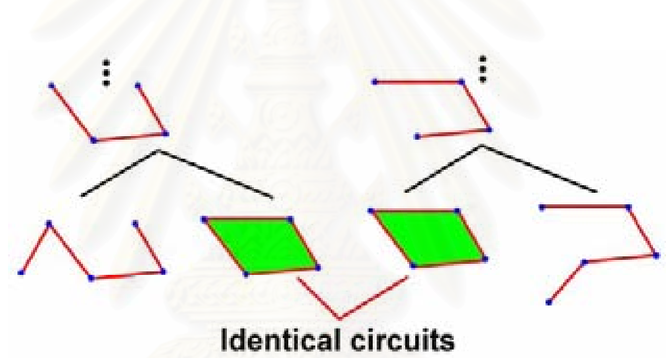


Figure 12. Identical circuits from different traces

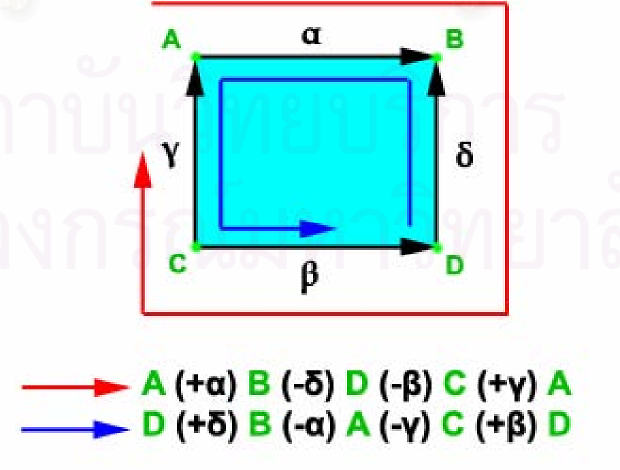


Figure 13. Two different paths representing the same face

If the circuit we have read does not match any previous circuit in the table, it will be added. After all the circuits have been read, we sort the circuits. We index the circuits by the vertex in the circuit with the smallest-numbered label, then the vertex adjacent to this vertex with the smaller-numbered label, and the rest of the vertices in order. Figure 14 shows circuits being sorted.

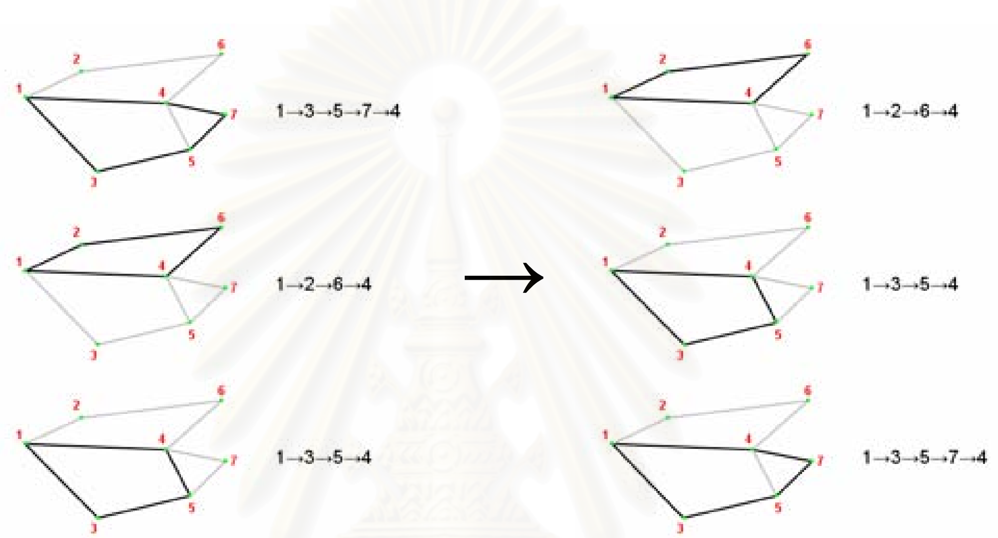


Figure 14. Circuit ordering

4.4 Face reduction

After obtaining all the possible faces, we need to reduce them to just likely faces. To do this, we first find the area of each face by splitting the shape into triangles, and summing their areas. Figure 15 shows two examples. To split a face, we find the axis with the largest range (the difference between the coordinates of any two vertices), and then find the vertex with the smallest coordinate on that axis (the leftmost vertex in both examples). We calculate vectors from this vertex to the other vertices of the face, and search for a vertex with a corresponding vector that lies between the vectors from the selected vertex to the ones adjacent to it. If we find such a vertex, we split the shape along a line from the selected vertex to the found vertex (left). Otherwise, we split between the vertices adjacent to the selected vertex (right).

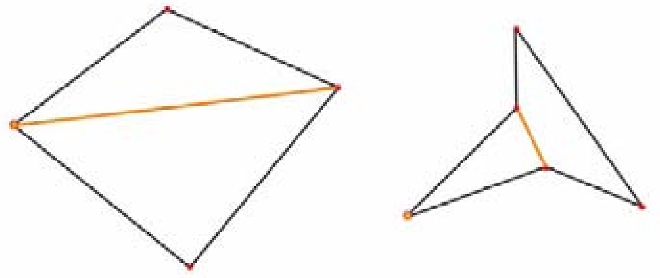


Figure 15. Face splitting to find area

Then, we find the flatness of the face, by considering each of its corners. We take a vector from a corner to the other vertices of the edges adjacent to the corner, and find the cross product. After all the cross products have been obtained, we find the two cross products whose dot product results in the smallest (absolute) cosine value, and use that value as the flatness. Figure 16 shows an example.

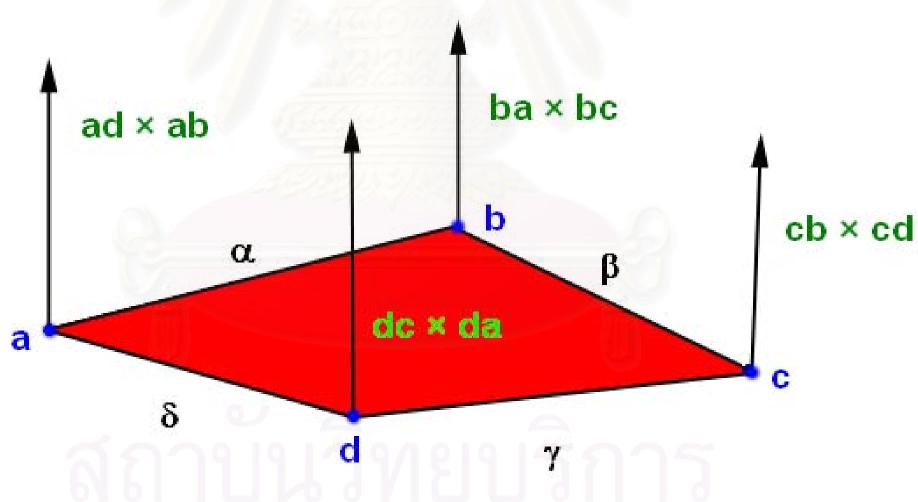


Figure 16. Calculating flatness

With the required information obtained, we then start to reduce the faces down to the most plausible ones. For this purpose, we have designated each face with a number, based on its position in the sorted table.

For each face, we find faces that come after it in the sorted table and are adjacent to its edges. We take each adjacent face and combine it with the current face, and compare the sum with all the other faces in the list. If a face is found that matches this sum, and its area is more than each of the two constituent faces, that face is marked unusable. We also combine the sum with another face that comes after both of the two constituent faces in the table, and do the same.

If two faces share the same 2 edges, the larger face is designated as a secondary face. All other faces are designated primary faces.

4.5 Face arrangement

The next step is to arrange the faces into the most likely shape. We begin by looking for smooth entity chains. Given the limitations of our program in using only straight edges, this will be limited to finding adjacent edges that comprise a single straight line. This is achieved by simply adding together the lengths of the edges, and comparing to the distance between the start and end vertices of the chain.

After finding smooth entity chains, we then calculate the maximum rank. Rank is the number of faces that are adjacent to a given vertex or edge. We use the equations used by Shpitalni and Lipson in their research [3]:

$$R(v) \le \frac{1}{2}\left[d(v) \times \left(d(v) - 1\right)\right]$$

$$R(e) \le \min\left[d(v_1), d(v_2)\right] - 1$$

$$R(v) = \frac{1}{2} \Sigma R(e)$$

$$R(e) \le \min\left[R(v_1), R(v_2)\right]$$

$$R(e) \le \min\left[\Sigma d(v_L) - 2n_L, \Sigma d(v_R) - 2n_R\right] + 1$$

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

In the first four equations:

- $R(v)$ is the number of faces adjacent to the vertex v.
- $R(e)$ is the number of faces adjacent to the edge e.
- $d(v)$ is the number of edges adjacent to the vertex v.
- v_1 and v_2 are the vertices that comprise the edge e.

The last equation is for the smooth entity chains we had found earlier, where v_L is the n_L vertices to the left of edge e, including the left end of e, and v_R is the n_R vertices to the right of edge e, including the right end of e.

We store the faces that are adjacent to each edge and vertex, to use in face arrangement. Figure 17 shows an example.

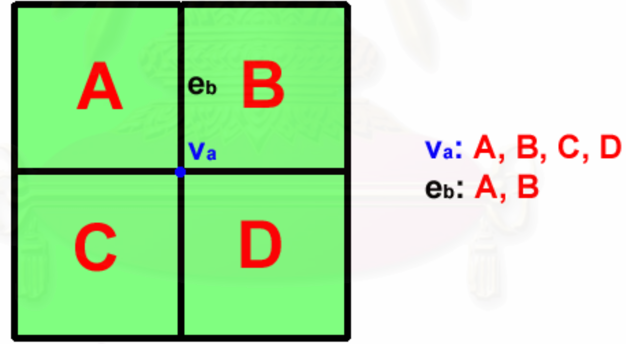


Figure 17. Storing faces adjacent to each edge and vertex

In arranging faces, we start by picking the smallest primary faces, with a flatness value of at least 0.9, adjacent to each edge and vertex. If none exists, we pick the smallest such secondary face, if available.

After the selections are made, if there are more faces adjacent to any edge or vertex than its calculated maximum rank, we will remove the largest face adjacent to it.

We then add the unselected faces in, checking ranks after adding each face, and removing faces as necessary.

4.6 Extra face processing

The next step is to do some extra face processing. First, we find faces that were not found in the first tracing (for example, when tracing a cone, the base of the cone is not found). We find edges adjacent to one face at most (with the edge also being adjacent to less faces than its maximum rank). We use these edges to trace for more faces, and add the faces that can be added.

We also remove overlapping faces. We combine two adjacent faces, and if the combined face has a flatness value of more than 0.9, we find other faces that share at least two edges with this combined face, and remove such faces if the center of the combined face lies within the other face.

4.7 Object and volume detection

The next step is to detect objects and volumes. We will use lists of vertices, edges and faces. We start from a given edge, and put it into the edge list. We check every edge in the edge list, and put every face adjacent to each edge into the face list, as well as putting the edges and vertices of each such face into the edge list and vertex list respectively. We repeat this process until there are no changes in any of the lists, with the faces in the face list comprising one object. If there are faces remaining to be used, the process is repeated with the remaining faces. Figure 18 shows a simplified progress of this algorithm.

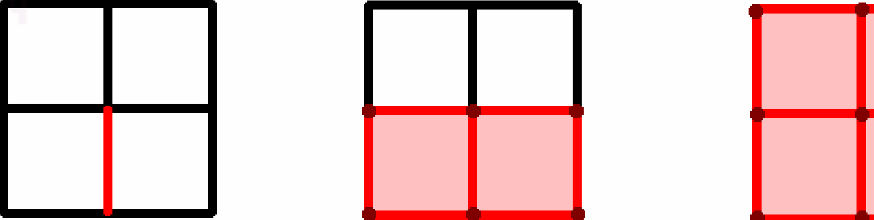


Figure 18. Tracing an object

The next step is to find the volumes. Objects may consist of many volumes. For example, two cubes sharing one face, while detected as one object, enclose two volumes. Therefore, finding the volumes uses a more complex procedure. We will also use lists of vertices, edges, and faces for the procedure. Figure 19 explains the process as a flowchart.

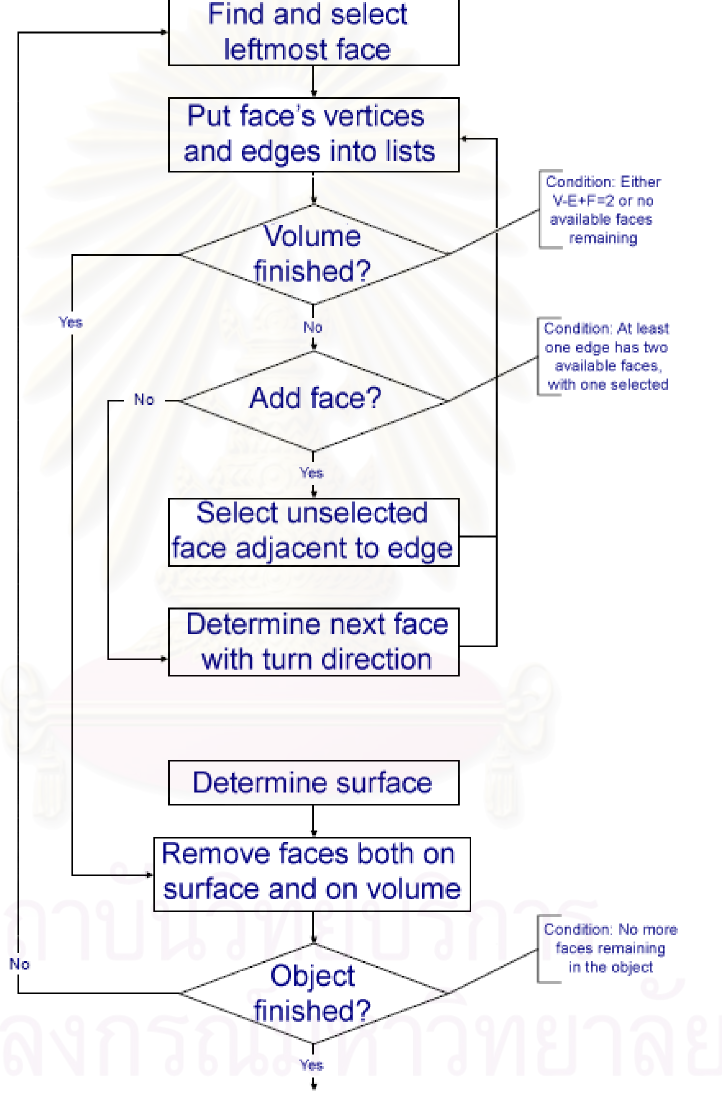


Figure 19. Flowchart of finding volumes

To find volumes, we start by finding the leftmost face, as shown in Figure 20. To find this face, we find the vertex with the smallest (x, y, z) coordinates, and then find the edge adjacent to this vertex that has the least angle with the z-axis. From the edge, we calculate vectors from the middle of the edge to the centers of the faces

adjacent to the edge. The face that produces the vector with the least angle with the *y*-axis is the leftmost face.

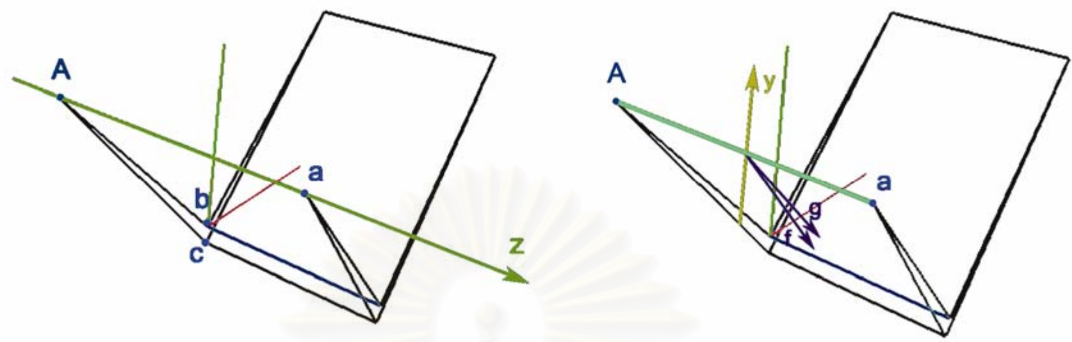


Figure 20. Finding the leftmost face

After obtaining the leftmost face (as well as its edges and vertices), we put it into a list of the volume's faces, and put its edges and vertices into the appropriate lists.

As per Baumgart's winged-edge structure [1], each edge is adjacent to two faces in a volume. Therefore, in finding the volumes, if we find an edge that is adjacent to one selected face, and two available faces are adjacent to the edge (i.e., there is only one adjacent face remaining to be added), we will pick the other available face (along with its edges and vertices). We repeat the process until no faces can be added, or the Euler-Poincaré equation $V - E + F = 2$ has been satisfied.

If no faces can be added with the above method, the equation $V - E + F = 2$ has not yet been satisfied, and there are faces available to add, another method for adding faces is used. Starting from an edge e with at least three available faces attached (with one already selected), we create a chain of edges from edges adjacent to one selected face. To obtain the "turn direction", we calculate the vector v from the center of e to the center of the chain of edges. For convenience, we then calculate a vector that lies on the same plane as e and v, and perpendicular to e:

$$v_n = (e \times v) \times e$$

For each face F adjacent to the edge, we take two vectors f_s and f_e from the center of face F to the start and end vertices of e respectively, and calculate a vector that lies on the same plane as f_s and f_e, and perpendicular with e:

$$v_f(F) = (f_s \times f_e) \times e$$

All of the vectors produced are then scaled down to a length of 1, for ease of calculation.

One of the faces adjacent to e (notated here as F_S) has already been selected. Based on v_n and the v_f calculated for each face, we select the face with the smallest dihedral angle from the selected face in the turn direction we have determined. On the left of Figure 21, we have selected face *a*, and we then pick face *b* if the center of the edge chain is at β, and face *d* if the center of the edge chain is at α. On the right, we have selected face *a*, and with the edge chain center at α, we pick face *b* as the next face.

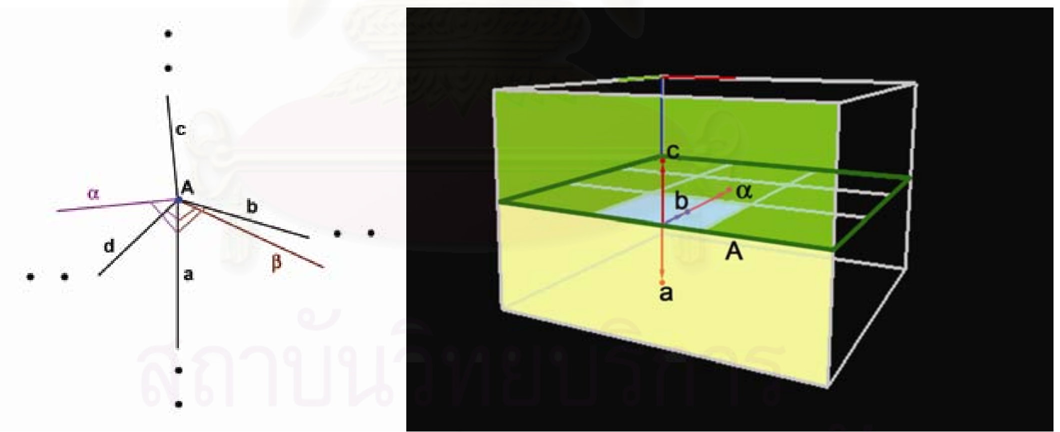


Figure 21. Determining the next face with turn direction

To determine the correct faces to pick, there are two cases that we need to consider. Each case is illustrated in Figures 22 and 23.

<u>Case 1:</u> $\exists F : \left(v_f(F_S) \times v_n\right) \cdot \left(v_f(F_S) \times v_f(F)\right) > 0$

Find the face F with the above property that produces the maximum value for $\left(v_f(F_S) \times v_f(F)\right)$.

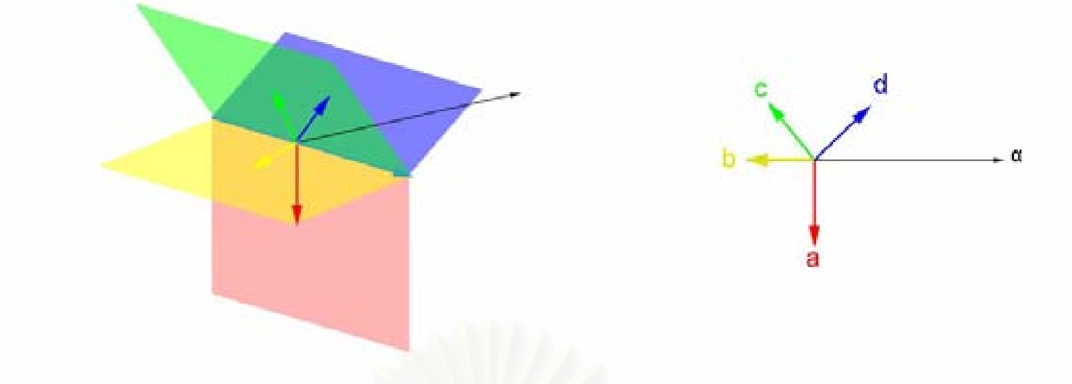


Figure 22. For case 1

Explanation: In this case, there exists at least one face where, in the direction of the rotation from the selected face (a) to the center of the edge chain (α), the angle from a's v_f to the given face's v_f is less than 180 degrees. Therefore, from such faces, we will select the one with the largest cosine value (i.e., smallest angle) between itself and a.

Case 2: $\forall F : (v_f(F_S) \times v_n) \cdot (v_f(F_S) \times v_f(F)) < 0$

Find the face F with the above property that produces the *minimum* value for $(v_f(F_S) \times v_f(F))$.

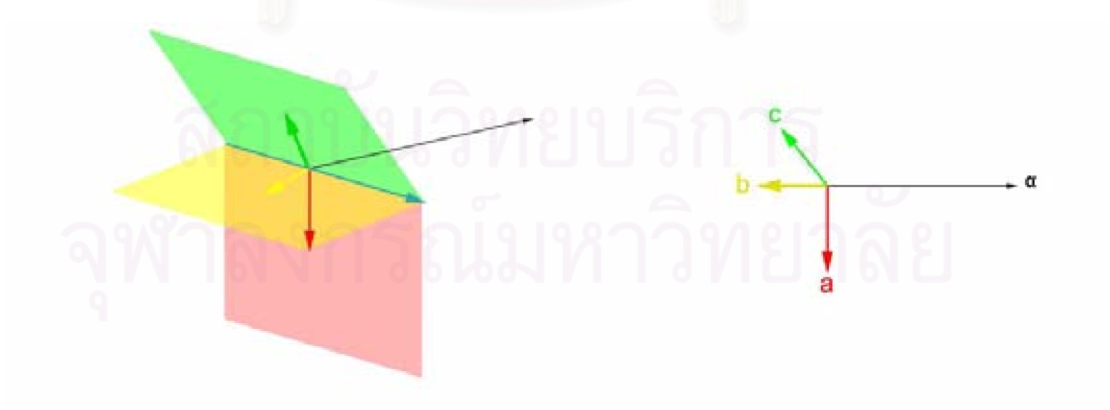


Figure 23. For case 2

Explanation: In this case, in the direction of the rotation from a to α, the angles from a's v_f vector to all the other faces' v_f vectors are more than 180 degrees.

Therefore, we will select the face with the smallest cosine value (i.e., largest angle) between itself and a.

We use this method in conjunction with the above method, and repeat until $V - E + F = 2$ has been satisfied, or no available faces remain.

After we have obtained the faces that comprise the volume, we need to remove some of the faces from future consideration. For this purpose, we then also search for the outside surface of the object using the same procedure (i.e., finding the leftmost face and adding adjacent faces), except selecting the face with the largest dihedral angle from the selected face in the turn direction, when using the edge chain method to add faces. After the results has been obtained, we remove the faces that are both part of the volume and part of the outside surface, and repeat the process as necessary.

A potential flaw in both the closed volume and surface finding algorithms is that the algorithm calls for calculating a vector that is perpendicular to the currently selected edge and lies on the same plane as the edge's adjacent face, as well as calculating another vector between the centre of the selected edge and the centre of the chain of edges starting from that edge. There is a possibility of those two vectors being in the same direction (resulting in a zero vector as their cross product), which would create difficulties in properly tracing surfaces and closed volumes, since the program cannot determine the proper turn direction in this case. Currently, the program solves this issue by testing different edges instead, and if all the edges produce this same result (which is most likely when there is just one selected face, or when the selected faces form a single plane), the program modifies the vector from the centre of the edge to centre of the edge chain, by adding to the *x* (and *y*, if necessary) values of the actual vector. The use of this special case solution is due to how both finding algorithms start at the leftmost face. This solution is shown in Figure 24.

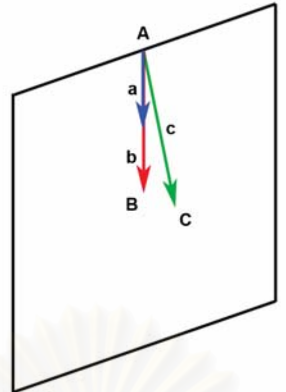


Figure 24. Special case solution

4.8 Finding relations

The next step is to find the relations between the faces, volumes and edges. We start at the leftmost face again, and calculate the normal vector of the face (by averaging the cross products between 2 vectors from the center of the face and the two vertices of each edge). If the *x* coordinate of the normal vector is more than 0 (i.e. the vector is pointing to the right), the face is facing towards the volume. If the *x* coordinate is less than 0 (pointing to the left), then it is facing away.

We store the normal vector, or its inverse, in an array, so that the vector stored in the array (hereby referred to as the “outward pointer”) points away from the volume.

To find the outward pointer of each face, we consider a face with a known outward pointer (at the beginning, this will just the leftmost face) and an adjacent face without a known outward pointer. We connect the centers of the two faces with a vector c. With a as the face with the known outward pointing vector, we can determine the unknown outward pointing vector from b, with 3 cases to consider, as illustrated in Figure 25.

(a) $a \cdot c < 0 \rightarrow b \cdot c > 0$
(b) $a \cdot c > 0 \rightarrow b \cdot c < 0$
(c) $a \cdot c = 0 \rightarrow a \cdot b > 0$

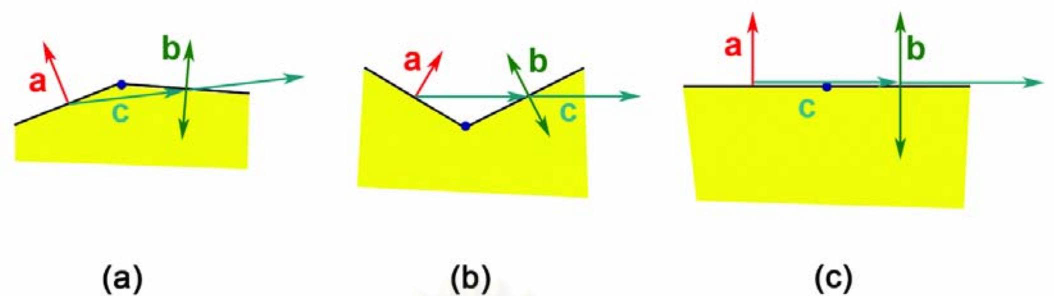


Figure 25. Determining a face's outward pointer from that of an adjacent face

The next step is to analyze the edges. This is because Kovalevsky's data structure calls for each edge to contain the information on the faces and volumes adjacent to the edge in right-hand rotation order. Figure 26 shows a few examples of this.

For each edge, we select a face that is not adjacent to two volumes (or just pick a random face if none is available), and put it into a list. To pick the faces and volumes in order, we will generally pick an element that is adjacent to the previous element and has not been put into the list yet, but there is a special case where there are no other volumes adjacent to the most recently-added face, with faces and volumes still remaining to be added. In this case, we will then measure the angles between the various faces, and select the one with the least angle in the same turning direction for the list (while indicating that there is no volume between the two faces). When all the faces and volumes have been picked, we will invert the list if necessary, and find the relationship between the right hand rotation and the normal vector of each involved face, and store that information as well. This is because Kovalevsky's original structure calls for this.

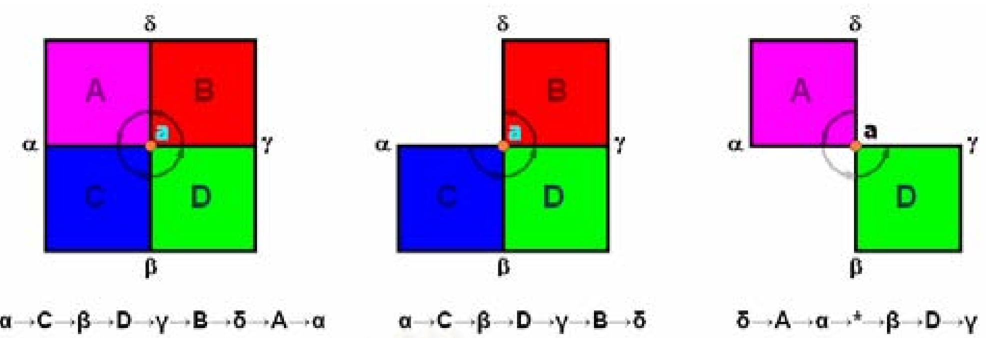


Figure 26. Right hand rotation around edge a

4.9 Output

The last step is storing the data that has been obtained in the previous steps to create the finished 3DCL. Since we have obtained all the data that we require, we will output it in this order:

- Vertex: Coordinates, adjacent edges (with direction indicated)
- Edge: Start and end vertices, faces and volumes adjacent to the edge in right-hand rotation order
- Face: The adjacent volumes (with direction indicated), its vertices and edges (in right hand rotation order around its normal vector)
- Volume: The volume’s faces (with direction indicated)

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER V

THIN POINT FINDING ALGORITHM

After the development of a method for converting a wireframe into a Three-dimensional Cell List, we began to develop algorithms that utilize the 3DCL as its input, in order to prove that the proposed converting algorithm is correct. This chapter describes the first algorithm, finding the thinnest points in a solid represented as a 3DCL. This differs from Koller and Danielsson's works, which emphasize on searching for long, thin, curvilinear structures, rather just thin points.

To define mathematically, a thin point is a point where a sphere inscribed in a solid tangent to the point has the smallest radius in the locality. Here, we will actually search for a series of connected edges which produce the smallest total in the locality.

Finding the thin points starts by pre-processing and then finding the central axis of the solid (as described in section 5.1), obtaining and processing selections (described in section 5.2), and using the obtained values to determine the thin points (described in section 5.3). We also describe the thresholding method that allows the user to set the accuracy of the algorithm (in section 5.4). Figure 27 shows the flowchart of the process.

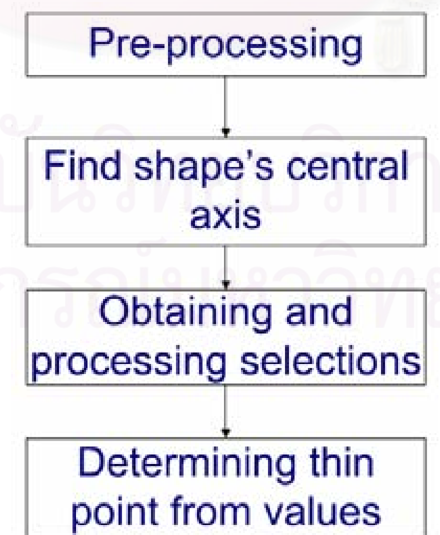


Figure 27. Flowchart of the Thin Point Finding Algorithm

5.1 Pre-processing and finding central axis

The program starts with reading in the Three-dimensional Cell List as input. While reading in the edges, we pre-calculate the lengths of each edge. This will be useful later.

To find the thinnest point(s) of each 3D solid, we start by finding its central axis. We find the average of the coordinates of all the volume's vertices, to find its approximate center, and the center point of every face of the volume.

After that, we split the shape into a number of parts (we have decided to use 20), by splitting the vertices into groups according to the *x*-value of the vertex. To do this, we find the vertices with the largest *x*-value (x_{max}) and the smallest *x*-value (x_{min}), and find the difference of those two values. We then determine which group each vertex falls under using this equation:

$$F(x) = \min\left(\left\lfloor 20 \times \frac{(x - x_{\min})}{(x_{\max} - x_{\min})} \right\rfloor, 19\right)$$

We then find the averages of all vertices in each group, and then find the straight line that fits the 20 averages best using linear regression.

We repeat this process with the *y*- and *z*-values instead of the *x*-value. We obtain 3 lines, and we determine which line is the best, by finding the distance between the line and the average values.

After we have obtained the best line, we split the shape's vertices into groups with this line as the axis. To split the vertices into groups using an arbitrary line (a, b, c) as the axis, we take each vertex's coordinates (x, y, z), and insert it in to this function:

$$G(x, y, z) = ax + by + cz$$

We determine which vertices produce the minimum and maximum values of $G(x, y, z)$, and then substitute $G(x, y, z)$ for the *x* values in the equation for $F(x)$.

Having split the vertices into groups, we average the coordinates of the vertices in each group, and find a new line with linear regression. Figure 28 shows an example of this. We repeat until the new axis deviates from the old axis by less than 0.1 degrees. In the case that the axis does not converge, we will average the most recent values after a certain amount of adjustments have been made. Figure 29 shows the process as a flowchart.

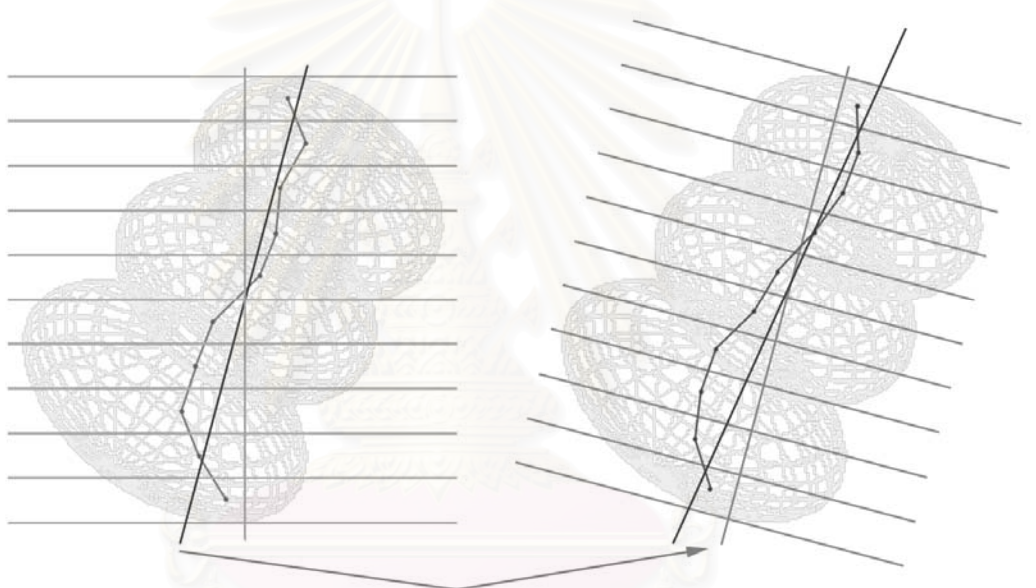


Figure 28. Determining axes using averages

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

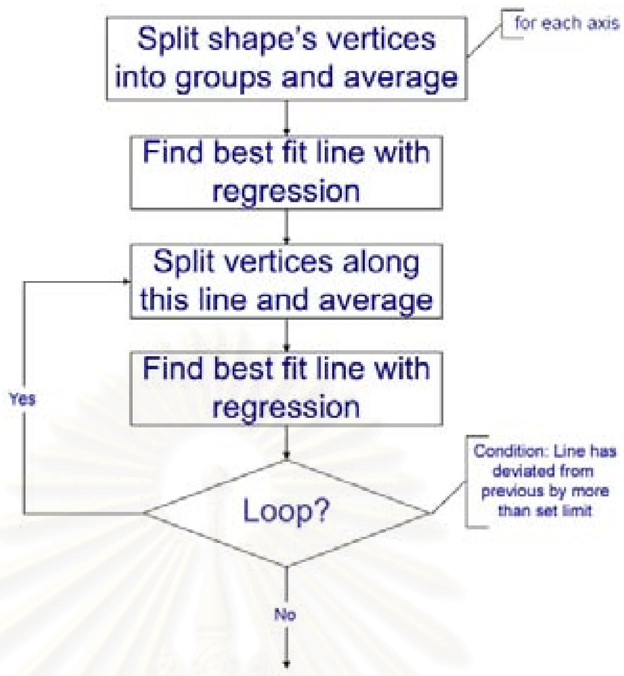


Figure 29. Flowchart of axis finding

After obtaining the axis, we calculate the centers of each face, and pick the one that is furthest from the center of the solid. We do this by finding the face whose center produces the highest absolute value of the $G(x, y, z)$ function.

With the face we have picked, we create a “selection”, stored as an array of Boolean values where the faces that have been selected have a value of *true*.

5.2 Obtaining and processing selections

We now process the selections iteratively to determine thin points. We begin by taking each face adjacent to each selection to create new selections. At first, there is just one selection, consisting of the face picked in the previous step. We create a new selection by taking the current selection, and adding a face in the volume adjacent to the selection (setting that face's value to *true*). Figure 30 shows a selection being used to generate new selections.

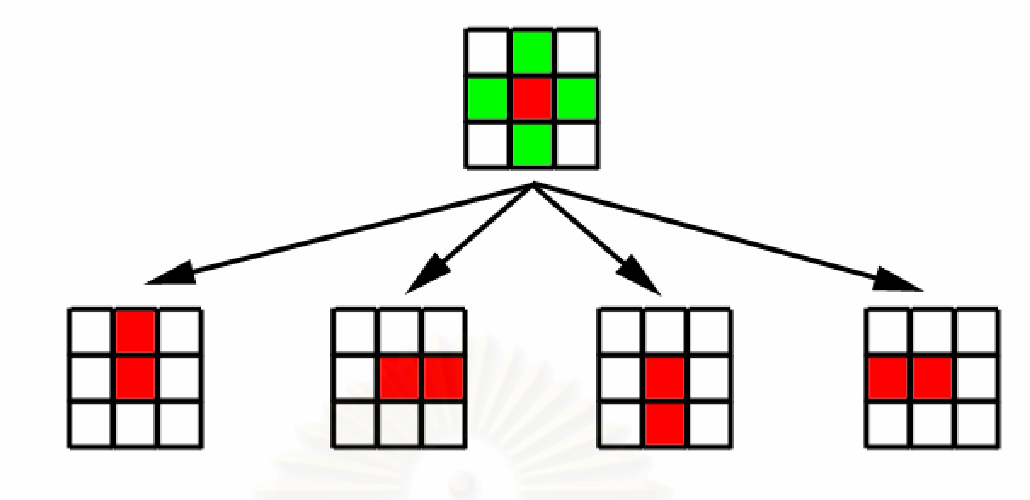


Figure 30. Creating new selections

To save processing time, we pick only the faces that are adjacent to the most faces in the current selection. For example, in Figure 31, out of the unselected faces adjacent to the selection, there are faces adjacent to 3 faces in the selection, and none adjacent to 4, so we pick only those faces adjacent to 3 selected faces.

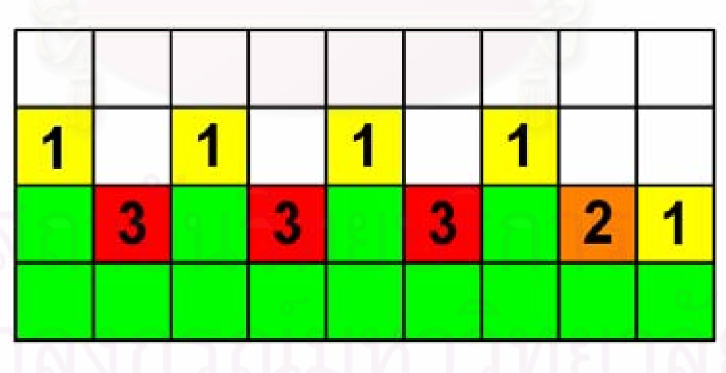


Figure 31. Face adjacency

We then compare the result to the selections that have been previously measured and stored. If it is not the same as any of the previous selections, we total up the lengths of each edge that is adjacent to one face in the selection (by summing the previously measured lengths). To determine which edges are adjacent to one face in the

selection, we store lists of edges adjacent to one face (L1) and two faces (L2). We insert the faces' edges into the lists as necessary. That is, if an edge is not in either list, we insert it into L1. If it is already in L1, we move it to L2. These arrays are stored with the selection, along with the total sums of the lengths of the edges in L1 (hereby referred to as the "total length" of the selection). Figure 32 illustrates this.

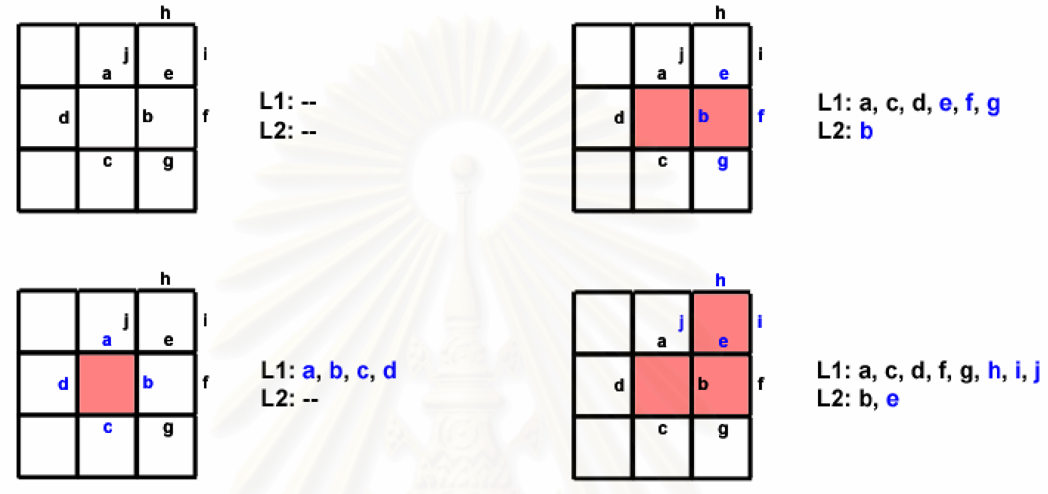


Figure 32. Keeping track of edges with lists

To determine the total length of the edges in L1, we use two separate lists to keep track of the edges that are added and removed from the list. We retrieve the total length we stored during the previous execution. We subtract from the total length the lengths of the edges that have been removed from L1, and add the lengths of the edges that have been added to L1, before storing the total length with the selection. This uses less time than simply directly summing up the lengths of each edge in L1, especially when there are thousands of edges in the list, because it requires less arithmetic operations.

For example, $L1_a$: {*a*, *c*, *d*, *e*, *f*, *g*} and $L1_b$: {*a*, *c*, *d*, *f*, *g*, *h*, *i*, *j*}. We have already stored *x* as the sum of the edges in $L1_a$. We compare the two methods of calculating $L1_b$:

Direct: We retrieve the lengths of each edge in $L1_b$ and sum them. This requires 8 values to retrieve, and 7 arithmetic operations to perform.

Quick: We take *x*, retrieve the length of *e* and subtract it from *x*, and retrieve the lengths of *h*, *i*, and *j* and add them to *x* - *e*. This requires 5 values to retrieve, 4 arithmetic operations to perform.

In practice, this method saves a noticeable amount of time when large numbers of loops are being processed.

After obtaining the selections, we insert them into an array, and then we remove the selections with a total length longer than a given threshold. This threshold is determined from the selection with the least total length, with the threshold at 1.1 times of this total length. This makes the algorithm a greedy algorithm.

This process will have the same number of cycles as the number of faces in the volume, and in each cycle, the number of faces that have been selected in each selection is the same as the number of cycles that have been completed. After each cycle, we store the least total length in an array, along with the corresponding edges in L1 that produce that length (hereby referred to as a “loop”). Figure 33 explains the whole process as a flowchart.

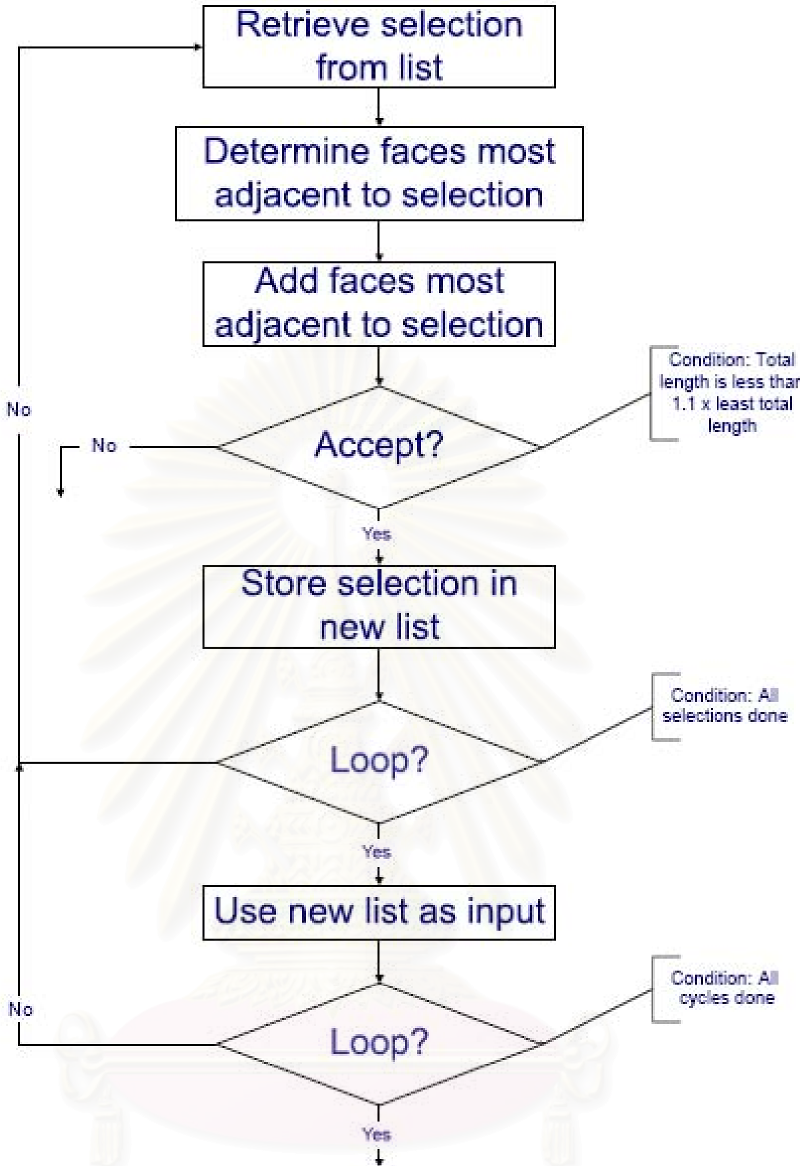


Figure 33. Flowchart of selection processing

5.3 Determining thin points from obtained values

After this step has finished, we adjust the least length values we have stored so that the selections that are not too near the start and end of processing have an advantage. The equation used for adjustment is:

$$v_i = \frac{\min(i, f-i)}{l_i}$$

i is the number of the current loop, *f* is the number of the faces in the volume, and l_i is the length of the current loop.

In some cases, we may need to “smooth out” the values obtained from the above equation, by averaging adjacent values. This is only done if the values have a tendency to alternate between going up and down, such as in volumes with triangular faces.

Next, we create a new array of averages. We average the values in a range of 5 (at most) values in either direction. For example, as shown in Figure 34, the 100th value in the new array is the average of the 95th to 105th values in the original array, and the 3rd value in the new array is the average of the 1st to 8th values.

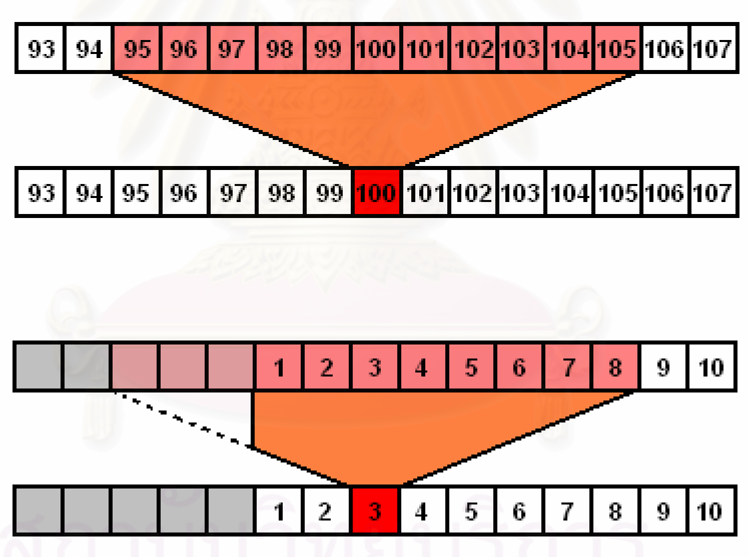


Figure 34. Averaging

With these new values, we compare the values in the original array, with the average of the 5 values on either side in the new array, as well as the adjacent values in the original. For example, we compare the 100th value of the original array with the average of the 95th to 99th values in the new array, the average of the 101st to 105th

values in the new array, the 99th value in the original array, and the 101st value in the original array.

If the value is higher than all of the values we compare it with, we then store the product of the two averages of adjacent new array values and the corresponding new array value.

Lastly, we output the loops where the value obtained from the process is more than 1/50 of the largest value from the process, and there are no higher values within 10% of the number of total cycles.

5.4 Thresholding

The algorithm allows the user to set the number of selections that remain after each cycle, with a minimum of 1, and a maximum of 1000. After the selections with a total length exceeding 1.1 times the length of the selection with the least total length are cut, if there are still more selections than the set value, the program sets a new length limit:

$$t_0 = 0.1 \times \min(\frac{n_t}{n_c}, 0.95)$$

with n_t as the set value, and n_c as the number of selections remaining. The program will remove selections with a length longer than $(1+t_0)$ times the least total length.

If the number of selections that remain are still higher than the set value, we re-adjust the length limit:

$$t_i = t_{i-1} \times \min(\frac{n_t}{n_c}, 0.95)$$

After the length limit has been adjusted 20 times, if there are still more selections remaining than the set value, the program sorts the total length of each selection to find the selections with the least total length. For example, if we require 40 selections, we take the first 40 selections after sorting to obtain the least length.

A large value for the number of selections remaining produces more accuracy, at the cost of more execution time. A small value for this number uses less execution time, at the cost of potentially less accuracy.

CHAPTER VI

EDGE SMOOTHING ALGORITHM

Inspired by Adamson and Alexa's method of tangential interpolation to smooth out point sampled models, we developed another algorithm for the 3DCL. The algorithm is designed to smooth out a solid represented as a 3DCL comprised entirely of straight edges, by converting edges into cubic or quadratic curves. We determine the continuous surface of each volume, and from the normal vectors of the surface's faces, we interpolate the normals of the surface's edges, and then those of the vertices. We use these values to redefine the surface's edges as curves, by repeatedly readjusting all the values until they converge. Figure 35 shows the flowchart of this algorithm.

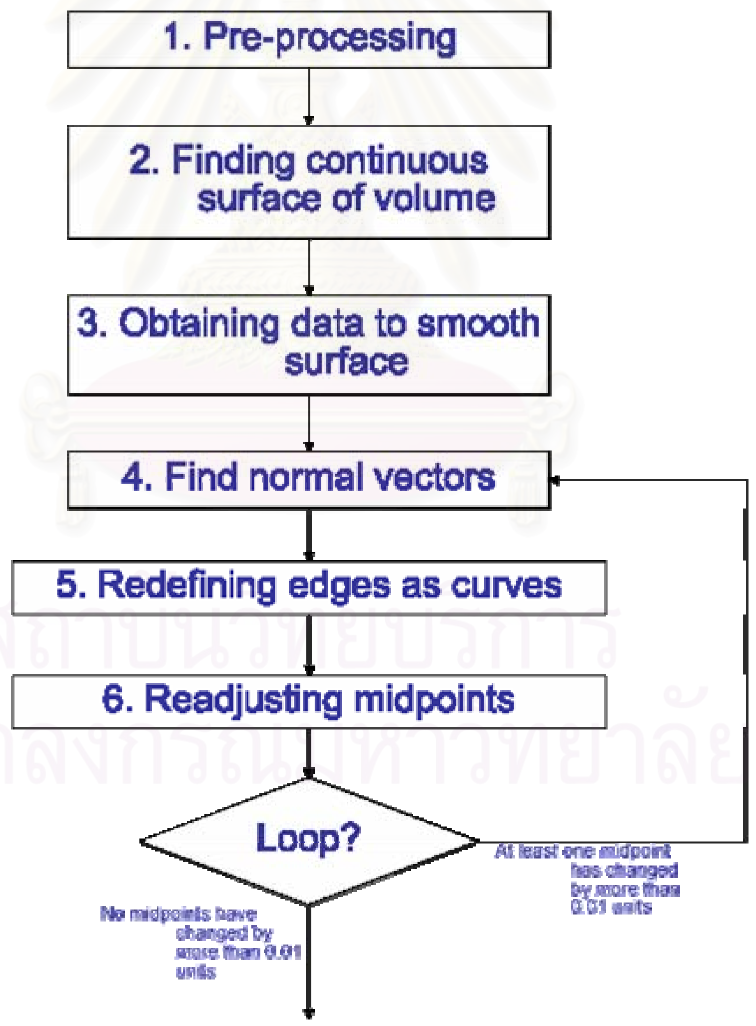


Figure 35. Flowchart of edge smoothing algorithm

6.1 Pre-processing and finding continuous surface

We begin by taking the 3DCL as input, and, for each volume, find the normal vector of each face of the volume, and use the 3DCL data to determine the vector that points away from the volume, and then unitize the vector.

After obtaining the outward pointing vector of each face, we find the continuous surface of each volume, starting at a given face in the volume, and then considering the faces adjacent to it. We maintain a list of faces F, and two lists of edges: L1 for edges adjacent to one face in F, and L2 for edges adjacent to two faces in F. We calculate the angles between the outward pointers of adjacent faces. If the angle is less than a value set by the user (between 0 and 90 degrees), we add that face into F, and update the edge lists. We repeat the process by considering the edges in L1 in the same manner, until all the faces in the volume have been listed, or no more new faces can be added to F.

6.2 Obtaining data to smooth surface

Having obtained faces of the continuous surface (Figure 36, left) as well as the edges adjacent to two such faces (middle), we then find the vertices where every edge adjacent to the vertex is in L2 (right). Such vertices are adjacent only to faces that are part of the current surface, and thus will be absolutely unaffected by other surfaces.

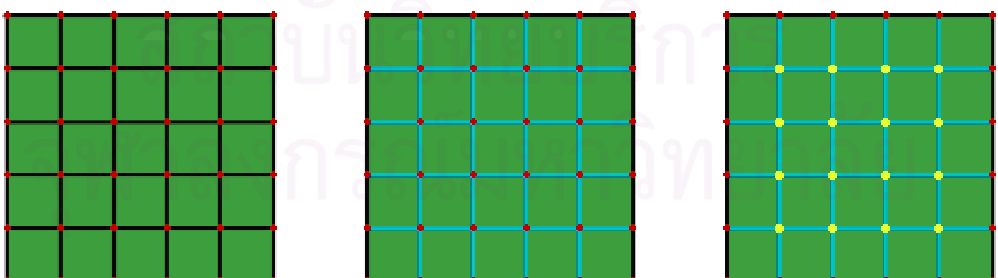


Figure 36. Faces, edges, and vertices used to smooth the surface

The data to be used to smooth out the continuous surface are: the faces of the surface (F), the edges adjacent to two faces in the surface (L2), and the vertices exclusively adjacent to edges in L2.

6.3 Find normal vectors

With all the necessary data obtained, we begin smoothing out the surface by with determining the normal vectors of the edges. For each edge, we calculate the weighted average of the normal vectors of its two adjacent faces, using the distance from the midpoint of the edge to the center of each face. In Figure 37, A is the distance between the center of face a to the midpoint of edge x, and B is the distance between the center of face b to the midpoint of edge x. Therefore, the normal vector of edge x is defined thusly:

$$\vec{N}(x) = \frac{\vec{N}(a)B + \vec{N}(b)A}{A+B}$$

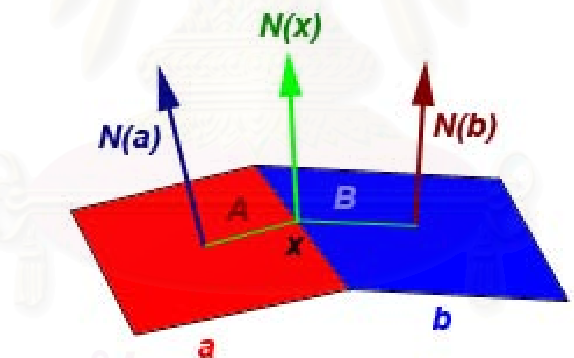


Figure 37. Determining the normal vector of an edge from its adjacent faces

After determining the normal vectors of each edge, we also find the normal vectors of the vertices adjacent to such edges, also using weighted averages, as illustrated in Figure 38. Where e_i is an edge adjacent to vertex p and d_i is the distance between the midpoint of e_i and p, the normal vector of p is defined thusly:

$$\vec{N}(p) = \frac{\sum_i \frac{\vec{N}(e_i)}{d_i}}{\sum_i \frac{1}{d_i}}$$

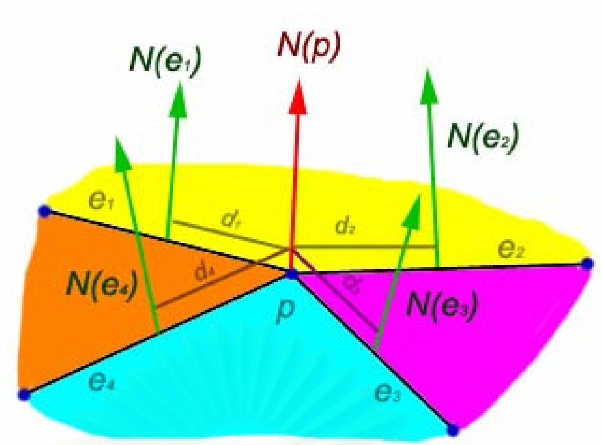


Figure 38. Determining the normal vector of a vertex from its adjacent edges

6.4 Redefining edges as curves

The next step is to smooth out the edges of the surface by redefining them as curves, by considering the start and end vertices of each edge, and the associated normal vectors of these two vertices, where available. There are three possible cases, depending on how many of the edge's vectors have associated normal vectors.

First case: Both vertices have normal vectors

The edge between the two vertices is a cubic curve, as a function of the axis with the largest range (i.e., difference between the start and end vertices).

Suppose that axis in question is *x*. We obtain the equations:

$$y = Ax^3 + Bx^2 + Cx + D$$
$$z = Ex^3 + Fx^2 + Gx + H$$
$$y' = 3Ax^2 + 2Bx + C$$
$$z' = 3Ex^2 + 2Fx + G$$

We have:

- Start vertex *s*: (s_x, s_y, s_z)
- End vertex *e*: (e_x, e_y, e_z)
- Vector from *s* to *e*: v
- Normal vector of *s*: $\vec{n}_s = (ns_x, ns_y, ns_z)$
- Normal vector of *e*: $\vec{n}_e = (ne_x, ne_y, ne_z)$

Since the preceding equations require the use of the derivative of the curve, we determine the derivative from the normal vector with these equations:

$$\vec{v}'(s) = (\vec{n}_s \times \vec{v}) \times \vec{n}_s = (v_x'(s), v_y'(s), v_z'(s))$$
$$\vec{v}'(e) = (\vec{n}_e \times \vec{v}) \times \vec{n}_e = (v_x'(e), v_y'(e), v_z'(e))$$

Figure 39 illustrates the logic behind these equations.

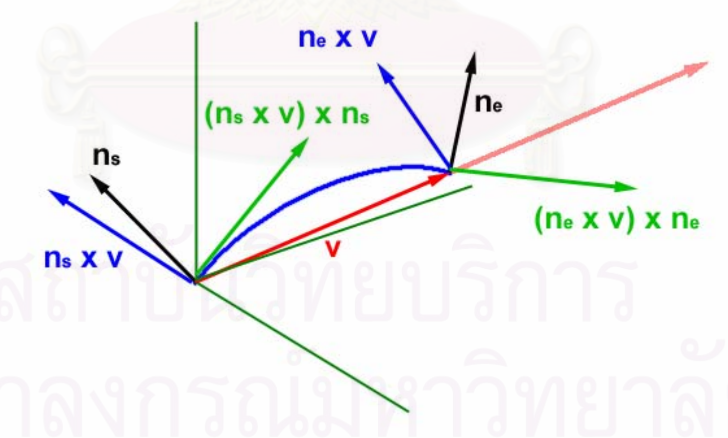


Figure 39. Obtaining derivatives from normal vectors

These equations result from the above steps:

$$s_y = As_x^3 + Bs_x^2 + Cs_x + D$$

$$e_y = Ae_x^3 + Be_x^2 + Ce_x + D$$

$$\frac{v_y'(s)}{v_x'(s)} = 3As_x^2 + 2Bs_x + C$$

$$\frac{v_y'(e)}{v_x'(e)} = 3As_x^2 + 2Bs_x + C$$

$$s_z = Es_x^3 + Fs_x^2 + Gs_x + H$$

$$e_z = Ee_x^3 + Fe_x^2 + Ge_x + H$$

$$\frac{v_z'(s)}{v_x'(s)} = 3Es_x^2 + 2Fs_x + G$$

$$\frac{v_z'(e)}{v_x'(e)} = 3Es_x^2 + 2Fs_x + G$$

<u>Second case: Only one vertex has a normal vector</u>

The edge between these two vertices will be a quadratic curve, once again as a function of the axis with the largest range.

Using *x* as the axis, we obtain:

$$y = Bx^2 + Cx + D$$
$$z = Fx^2 + Gx + H$$
$$y' = 2Bx + C$$
$$z' = 2Fx + G$$

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

If the start vertex has the normal vector, these equations are obtained:

$$s_y = Bs_x^{\;2} + Cs_x + D$$

$$e_y = Be_x^{\;2} + Ce_x + D$$

$$\frac{v_y'(s)}{v_x'(s)} = 2Bs_x + C$$

$$s_z = Fs_x^{\;2} + Gs_x + H$$

$$e_z = Fe_x^{\;2} + Ge_x + H$$

$$\frac{v_z'(s)}{v_x'(s)} = 2Fs_x + G$$

If the end vertex has the normal vector, these equations are substituted for the 3rd and 6th equations above:

$$\frac{v_y'(e)}{v_x'(e)} = 2Be_x + C$$

$$\frac{v_z'(e)}{v_x'(e)} = 2Fe_x + G$$

Third case: Neither vertex has a normal vector

The edge between these two vertices is straight, and can be rendered as linear functions. No calculation of derivatives is required.

$$s_y = Cs_x + D$$

$$e_y = Ce_x + D$$

$$s_z = Gs_x + H$$

$$e_z = Ge_x + H$$

We solve the equations obtained above with a matrix, so as to find the best curve that fits the given data. We consider the two remaining axes separately, resulting in a curve (or straight edge), represented as the start and end points on the axis with the largest range, and equations for the other axes.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

6.5 Readjusting midpoints

If the edge is not a straight line, the next step is to adjust the midpoint of each edge (which originally is the average of the start and end vertices' coordinates). For each edge being considered, we start determining the midpoint by finding the point on the edge between the start and end vertices which is closest to a line L passing through the previous midpoint and parallel to the normal vector. We do this by placing 7 points on the curve, spacing them equally on the axis with the largest range. We next determine which of these points are closest to L. To find the distance between each point and L, we use a simple method.

P (x_p, y_p, z_p) is the point, M (x_m, y_m, z_m) is the midpoint, and $\vec{N}(\vec{x}_n, \vec{y}_n, \vec{z}_n)$ is the normal vector of the edge. We desire to find the distance d between P and a line passing through M parallel to $\vec{N}$.

First we create a third point M': $(x_m + \vec{x}_n, y_m + \vec{y}_n, z_m + \vec{z}_n)$. MM' is a line segment of L. The three points P, M and M' form a triangle. Finding the distance between P and L (here, MM') now simply involves finding the area of the triangle PMM' (which we will call A):

$$s = \frac{PM + MM' + PM'}{2}$$
$$A = \sqrt{s(s - PM)(s - MM')(s - PM')}$$

Thus, from the more conventional method of finding the area of a triangle:

$$A = \frac{1}{2} MM' \times d$$
$$d = \frac{2A}{MM'}$$

After finding the distance for each point, we find which point produces the smallest result. If that particular point is the start or end vertex of the curve, we recursively search between that vertex and the point next to it. For the other spots, we recursively search between the two points next to it (in Figure 40, the 3rd point is closest; therefore, we search between the 2nd and 4th points). When the search area is below a set limit, the new midpoint has been obtained.

Figure 40. Determining the new midpoint of an edge

Having obtained the new midpoints, we adjust the vertices' normal vectors using the same equations as earlier, due to the change of distance between the vertex and the midpoint of the curve. After obtaining the new normal vectors of the vertices, we adjust the curves and their midpoints again to match the new vectors. We repeat this process until none of the edges' midpoints change by more than .001 units (when an edge's midpoint changes by less than .001 units, it is no longer adjusted), or until a number of cycles have passed where the same number of midpoints continues to change by more than this threshold (in that case, we simple average the most recent results). Having completed the current surface, we then search for more surfaces from the unused faces, and repeat the process.

After all the surfaces have been processed, we consider the remaining unprocessed edges as straight edges, and we output a file to define each curve in the terms of the axis with the largest range, the limits of the edge on that axis, and the equations of the other two axes. For example, an edge represented in the file as "0, 0,

0.38, -0.0123, 0.0021, -0.1043, 1.95, 0.0384, 0.0121, -0.0102, -9.81” has the information thusly:

- The axis with the largest range: x (0 = x, 1 = y, 2 = z)
- Limits of edge on x: 0 to 0.38
- Edge’s y coordinates as a cubic equation: $-0.0123x^3 + 0.0021x^2 + -0.1043x + 1.95$
- Edge’s z coordinates as a cubic equation: $0.0384x^3 + 0.0121x^2 + -0.0102x + -9.81$

(For display purposes, we also redefine each curve as 15 short line segments in a separate file.)

CHAPTER VII

EXPERIMENTAL RESULTS

In this section, we describe the experimentation for each of the algorithms, and explain the results. We also analyze the execution time.

7.1 Tools for experimentation

For the implementation of the various algorithms, we use Microsoft Visual Basic .NET. For testing and timing the execution, we use a computer running on a Pentium with 996 MHz.

7.2 Wireframe to Three-dimensional Cell List Conversion

For testing this algorithm, we prepared 6 examples, three simple examples (Figure 41), and three complex examples (Figure 42).

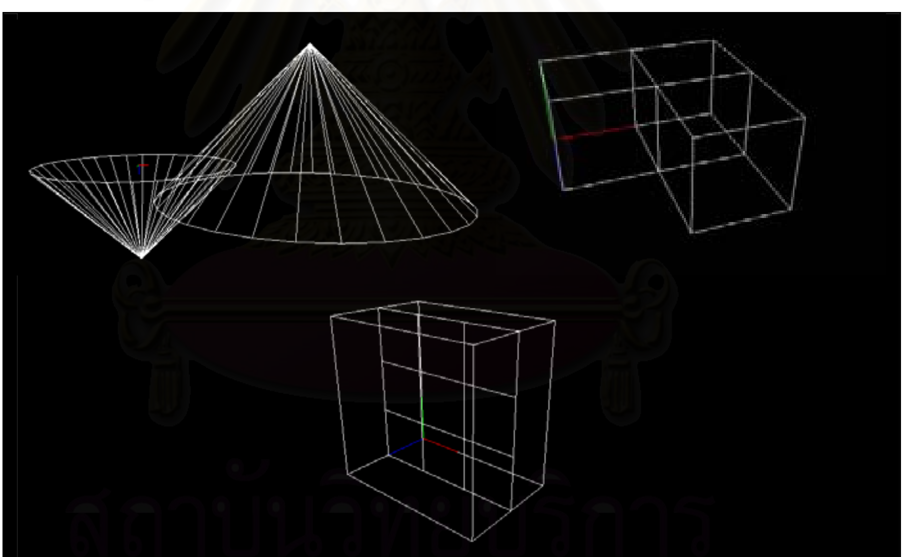


Figure 41. Simple examples

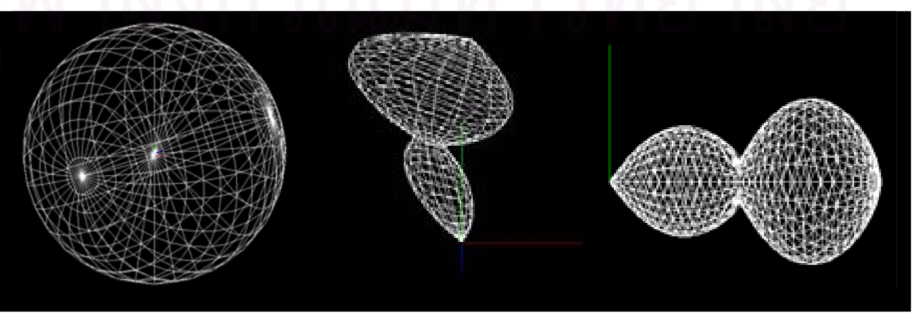
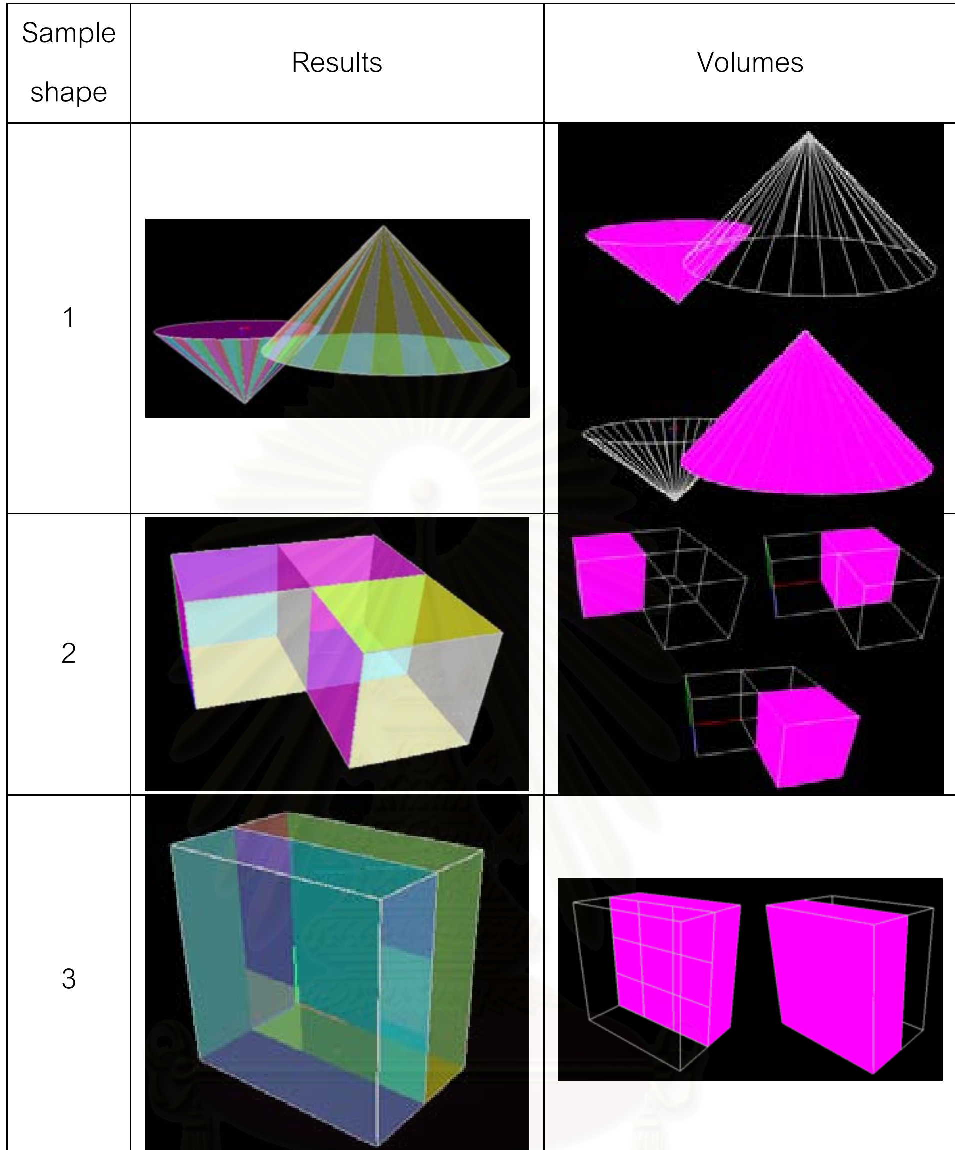


Figure 42. Complex examples

These are the results from the three simple examples:

Sample shape	1	2	3
Initial read:			
Time	7520 ms	761 ms	1221 ms
Vertices	66	16	24
Edges	128	28	40
Face finding:			
Finding faces	22422 ms	4155 ms	5337 ms
Reducing to unique faces	1842 ms	3725 ms	1211 ms
Faces found	128	16	23
Face reduction:			
Area calculation	1742 ms	370 ms	430 ms
Processing	3214 ms	640 ms	771 ms
Remaining faces	64	16	19
Secondary faces	0	0	0
Face selection:			
Processing	420 ms	811 ms	340 ms
Faces selected	64	16	19
Face analysis:			
Analysis	480 ms	731 ms	370 ms
Faces	66	16	19
Volume analysis:			
Manifold analysis	1972ms	1602 ms	1061 ms
Processing	54358ms	6909 ms	11997 ms
Total volumes	2	3	2
Cell list assembly	12287 ms	2293 ms	1812 ms

Table 1. Results of the simple examples



Sample shape	Results	Volumes
1		
2		
3		

Table 2. Visual results of the simple examples

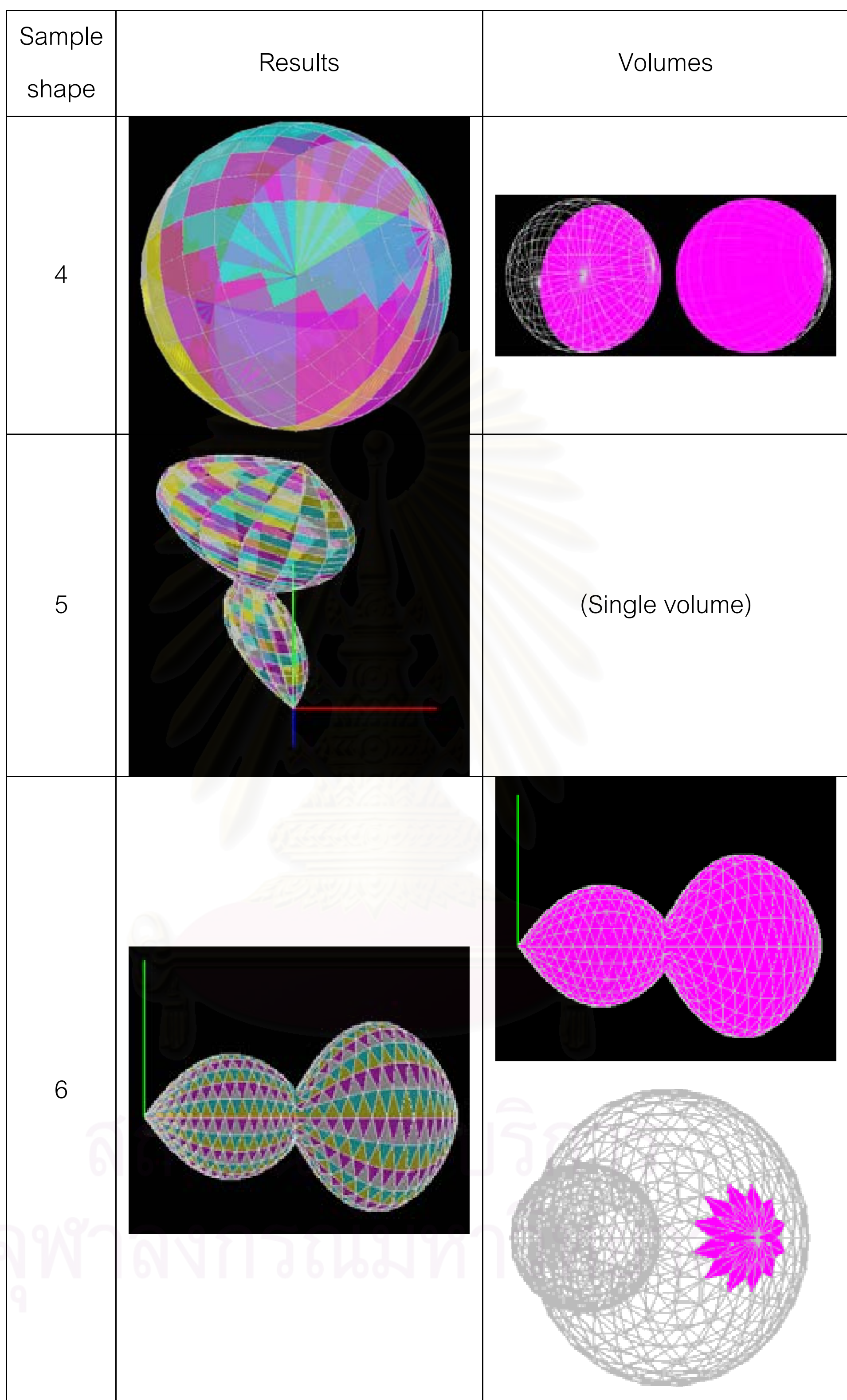
สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

These are the results from the three complex examples:

Sample shape	4	5	6
Initial read:			
Time	51333 ms	47788 ms	152258 ms
Vertices	483	466	710
Edges	1024	944	2124
Face finding:			
Finding faces	294072 ms	326148 ms	1332035ms
Reducing to unique faces	18806 ms	15652 ms	87765 ms
Faces found	1533	1412	7584
Face reduction:			
Area calculation	24044 ms	19938 ms	121945 ms
Processing	76630 ms	56320 ms	366256 ms
Remaining faces	597	488	1464
Secondary faces	51	8	24
Face selection:			
Processing	27579 ms	10595 ms	413394 ms
Faces selected	544	480	1440
Face analysis:			
Analysis	4576 ms	5698 ms	84872 ms
Faces	544	480	1428
Volume analysis:			
Manifold analysis	24795 ms	30103 ms	68768 ms
Processing	349031 ms	329613 ms	1497543 ms
Total volumes	2	1	13
Cell list assembly	79514 ms	75037 ms	225804 ms

Table 3. Results of the complex examples

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย



Sample shape	Results	Volumes
4		
5		(Single volume)
6		

Table 4. Visual results of the complex examples

By plotting the execution times against the size of the input and analyzing the algorithms themselves, these most plausible time complexity results for average cases are obtained in relation to the numbers of edges (e) and faces (f):

Initial read: $O(e \log e)$, due to the sorting algorithm.

Face finding: Due to the use of tree searching, the worst case is exponential time complexity. However, on well-defined wireframes, we have observed a time complexity of $O(e \log e)$, due to most faces being comprised of a low number of edges (3-4), and the removal non-plausible alternatives during tracing.

Area calculation: $O(f)$, due to faces usually having 3 or 4 edges, and thus requiring at most one split.

Face processing: $O(f \log f)$, due to the use of sorting to determine the order of faces to remove or add.

Volume analysis processing: $O(f \log^3 f)$ has been observed. However, the execution time for this algorithm is mostly dependent on whether there are extra faces to be found.

Cell list assembly: $O(f \log f)$ has been observed. However, the execution time for the algorithm is also dependent on the number of volumes found.

7.3 Finding Thin Points in 3DCL

For this experiment, we have prepared 6 solids, each in two versions: quadrangular faces (Figure 43, top) and triangular faces (bottom). We also tested on two thresholds, 40 and 100.

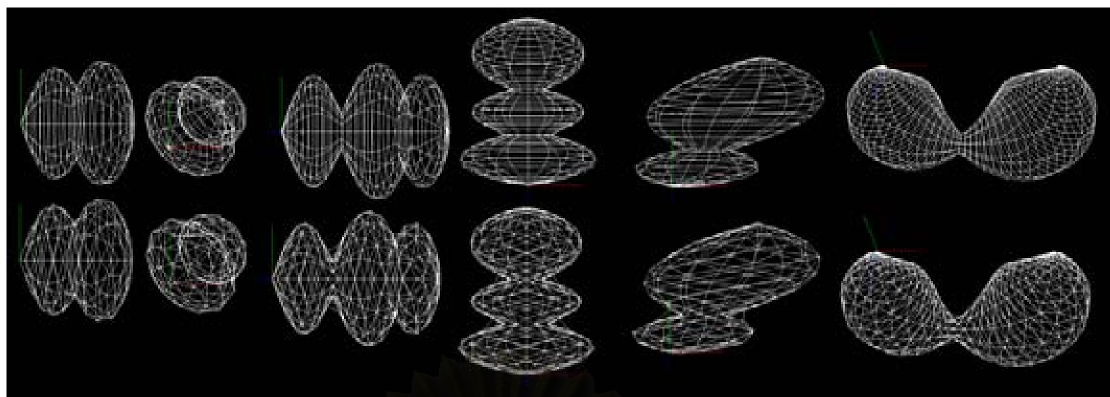


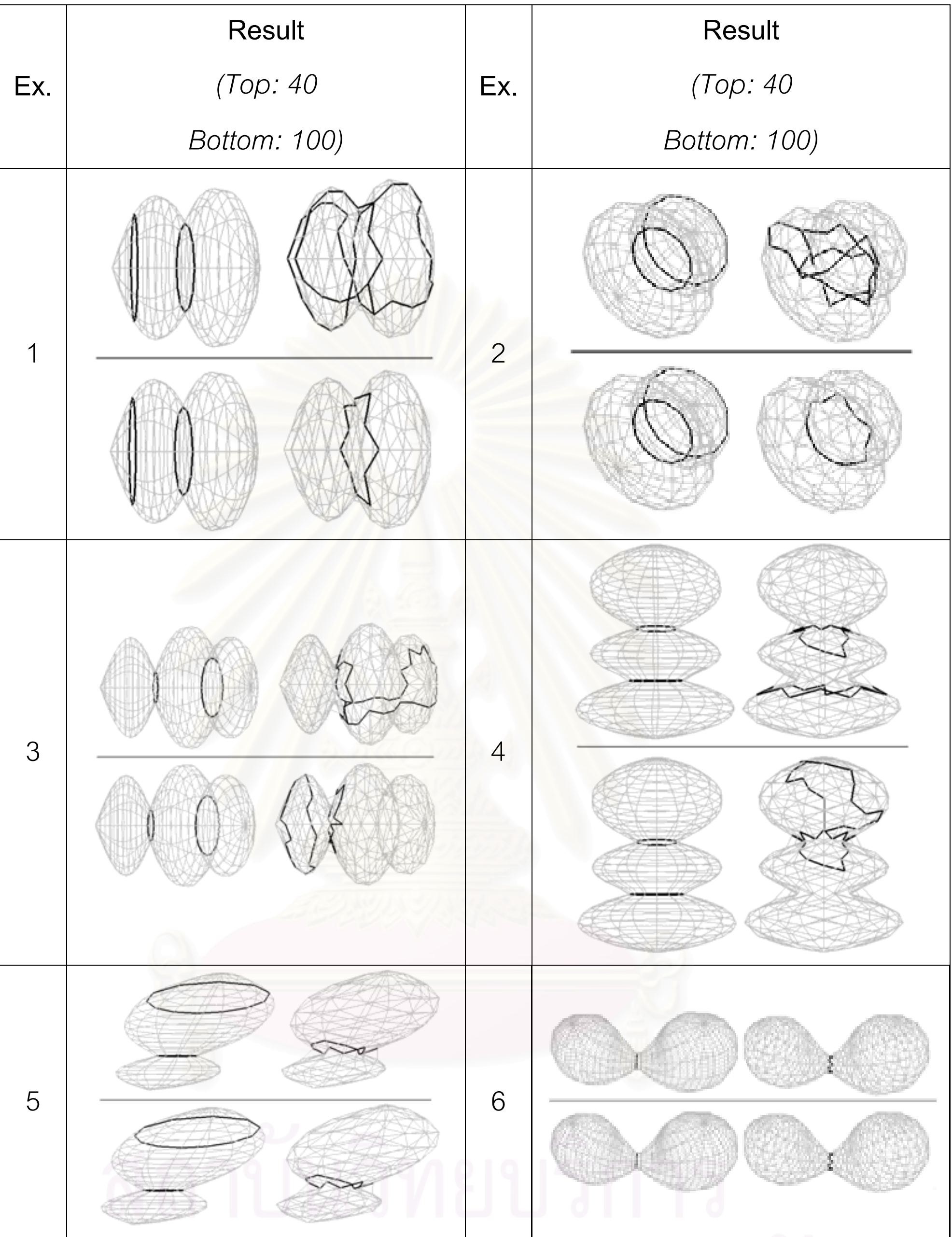
Figure 43. Examples for thin point finding

Ex.	Faces (Quad.)	Time (Threshold 40)	Time (Threshold 100)	Faces (Tri.)	Time (Threshold 40)	Time (Threshold 100)
1	240	12.648 s	11.807 s	216	26.298 s	182.532 s
2	240	12.318 s	12.748 s	216	56.511 s	105.732 s
3	480	47.759 s	27.810 s	456	317.176 s	1711.090 s
4	480	34.309 s	33.809 s	456	160.731 s	584.390 s
5	240	12.167 s	19.708 s	222	23.864 s	58.795 s
6	720	34.089 s	31.325 s	696	284.699 s	1676.851 s

Table 5. Time to find thin points

In Table 6, the result from a threshold of 40 is above the line, while the result from a threshold of 100 is below the line.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย



Ex.	Result (Top: 40 Bottom: 100)	Ex.	Result (Top: 40 Bottom: 100)
1		2	
3		4	
5		6	

Table 6. Visual results for thin points

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

We also tested the program with a standard model, the Stanford Bunny, using the smallest threshold possible (a 1 threshold, i.e., taking only the selection with the smallest length in each cycle for the next cycle's processing). The execution took 83629223 ms. The result of the processing is shown in Figure 44. While the algorithm has found the area around the ears, the other results are still not very good. This is

possibly a result of using such a low threshold, and even with such a low threshold, the execution time is still high, and would be even more so with a higher threshold.

Figure 44. Stanford Bunny

It should be noted that quadrangular faces take advantage of the face adjacency time-saver more than triangular faces (and thus are practically unaffected by the increase in threshold), and are also somewhat less prone to inaccurate results. However, the program still produces some extraneous results, even on quadrangular faces, and more so triangular faces. There are also some cases where it still does not detect thin points well, especially with the Stanford Bunny example.

Also, for solids with triangular faces, the execution time tends towards exponential time complexity in relation to the number of faces (although the actual execution time depends on the solid).

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

7.4 Smoothing out 3DCL

We have tested three different single solids represented as a Three-dimensional Cell List on a 996 MHz Pentium, using 45° as the cut-off for continuous surface detection. The results in Table 7 suggest that the program produces reasonably

smooth solids which are an improvement on the original. Analysis of the execution time suggests a $O(n \log^2 n)$ time complexity relative to the number of edges in the shape. However, execution time is dependent on the number of surfaces found in the shape.

Test figure	Edges	Time	Original	Smoothed
1	120	24515 ms		
2	468	100234 ms		
3	336	40298 ms		
4	944	186558 ms		

Table 7. Results for 3DCL smoothing

CHAPTER VIII

CONCLUSIONS AND FUTURE IMPROVEMENT

In this chapter, we discuss the conclusions we have obtained from the experiments. We also outline possible improvements for further research.

8.1 Conclusions

<u>Wireframe to Three-dimensional Cell List Conversion</u>

As tested, the algorithm produces reasonably accurate results. The algorithm has potential applications in areas where 3D topological information is useful, such as product design.

However, there are some cases where extra faces are found and included, from close and adjacent edges. This skews the volume computation, and results in additional spurious volumes (usually in the form of pyramids). Also, there are some processes that could be improved. For example, reading in the edges and vertices requires two passes of the file currently.

<u>Finding Thin Points in 3DCL</u>

The method for determining which loops are suitable as thin points still needs improvement, as it produces good results on some sets of data and bad results on others. Other methods are currently being tested.

Also, the execution time for finding thin points is high (exponential in the worst case), especially with solids with triangular faces, due to the fact that such solids do not take as much advantage of the face adjacency time-saver as much as quadrangular faces.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

Smoothing out 3DCL

The method for smoothing out the solids by rendering them as curves produces reasonably realistic results, even on less-faceted figures, and the execution time, dependent on the number of surfaces found, also seems to be reasonable.

8.2 Future improvement

Wireframe to Three-dimensional Cell List Conversion

Possible future improvement to the algorithm includes reducing the spurious faces detected and thus the extra volumes found (as seen in example 6 in Table 4), and finding a better solution to the potential flaw mentioned in section 4.7. Other improvements include streamlining processes to reduce execution time without negatively affecting the algorithm's accuracy (for example, in section 4.1, we read the input file twice).

Finding Thin Points in 3DCL

Further research is necessary in order to determine the best method to accurately determine thin points from the data. Another part of the research that also needs improvement is the speed of the algorithm, as its execution time is still unreasonably high in some cases. One possible method of speed improvement would be to consider faces in clusters, rather than individually as we have done in this research. This would prove useful especially with figures with a very high number of faces, such as the Stanford Bunny model in Figure 44.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

<u>Smoothing out 3DCL</u>

The algorithm currently considers edges that were not analyzed during the processing (i.e., edges that border multiple surfaces) as straight edges. Therefore, objects like cylinders will not be smoothed well, as it will consider the top, bottom, and sides as separate surfaces, and edges that border either the top or bottom and the sides will not be processed. Future improvement may include analyzing these edges as well, to improve the smoothness of the shape.

References

[1] Baumgart, B. Winged edge polyhedron representation. Artificial Intelligence Project Memo AIM-179 (CS-TR-74-320), Stanford University, 1972.

[2] Kovalevsky, V. Algorithms and Data Structures for Computer Topology. Digital and Image Geometry, 38-58. Springer, 2000.

[3] Shpitalni, M. and Lipson, H. Identification of Faces in a 2D Line Drawing Projection of a Wireframe Object. IEEE Transactions On Pattern Analysis And Machine Intelligence, 18, 10 (1996): 1000-1012.

[4] Liu, J. and Lee, Y. T. A Graph-Based Method for Face Identification from a Single 2D Line Drawing. IEEE Transactions on Pattern Analysis and Machine Intelligence 23, 10 (2001): 1106-1119.

[5] Oh, B. S. and Kim, C. H. Progressive reconstruction of 3D objects from a single free-hand line drawing. Computers and Graphics 27, 4 (2003): 581-592.

[6] Koller, Th. M., Gerig G., Székely, G. and Dettwiler, D. Multiscale detection of curvilinear structures in 2-D and 3-D image data. ICCV '95: Proceedings of the Fifth International Conference on Computer Vision, 864-869. Washington: IEEE Computer Society, 1995.

[7] Danielsson, P.-E., and Lin., Q. Efficient detection of second-degree variations on 2D and 3D images. Journal of Visual Communication and Image Representation, 12 (2001): 255-305.

[8] Adamson, A. and Alexa, M. Point-sampled cell complexes. SIGGRAPH '06: ACM SIGGRAPH 2006 Papers, 671-680. New York: ACM Press, 2006.

[9] Charussuriyong N. and Kanongchaiyos P., 3D Object Modeling Method for Multimedia Using Cellular Structured Space. ICCIMA '05: Proceedings of the Sixth International Conference on Computational Intelligence and Multimedia Applications, 247-252. Washington: IEEE Computer Society, 2005.

[10] Matsumoto, K. and Kunii, T. A Cellular Design System for Soft- and Varied Sized- Objects. Proceedings of the First International Symposium on Cyber Worlds (CW '02), 386-393. IEEE Computer Society Press, 2002.

VITA

Varakorn Ungvichian was born on 22 May, 1983. Ungvichian graduated with a Bachelor of Engineering (B.Eng.), 2nd class honours, in Computer Engineering from the Faculty of Engineering, Chulalongkorn University in 2005, and entered the Master of Engineering curriculum at the Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University in 2005.