Surface Segmentation and Shape Modification For 3D Geometric Objects

by

Mohsen Madi

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada

© Mohsen Madi
January 2000

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-51653-9

Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

Surface Segmentation and Shape Modification for 3D Geometric Objects

BY

Mohsen Madi

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree

of

Doctor of Philosophy

MOHSEN MADI ©2000

Permission has been granted to the Library of The University of Manitoba to lend or sell copies of this thesis/practicum, to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film, and to Dissertations Abstracts International to publish an abstract of this thesis/practicum.

The author reserves other publication rights, and neither this thesis/practicum nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

Abstract

The design and display of 3D models on a computer is usually interleaved by a series of data manipulations. Data are usually sets of 3D points that make up polygonal patches of the polyhedral models. Manipulations are the transformation operations that are applied to the 3D points to facilitate design and visual understanding of the graphical models. To further automate and facilitate design and visual understanding of the displayed models for applications such as the field of anatomy, or speed up the design phase for applications such as rapid prototyping, methods are proposed to interactively query data points on the surface for shape control, segment portions of displayed surfaces for independant modification, modify and re-integrate the modified segments to the main structure. This has the advantage of limiting the number of polygons that need to be manipulated for computationally cheaper and faster results.

This dissertation illustrates how polyhedral surfaces can be organized into special data structures to facilitate rapid selection of groups of data points, and provides novel algorithms to facilitate the applications of interactive operations on such selected data.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 3

1.3 Modeling & Visualization Techniques 4

1.4 Shape Refinement Control 4

1.5 Polyhedra Segmentation 5

1.6 Polyhedra Shape Modification 6

1.7 Integration of Sub-polyhedra 6

1.8 Thesis Particulars & Special Terms 7

2 Modeling & Visualization of Layered Objects **11**

2.1 Related Work 13

2.2 Background and Terminology 17

2.3 Algorithms and Data Structures 19

2.3.1 Forming the Layer 19

2.3.2 Organization of the Cuboids 27

2.3.3 Rendition of Cuboids 30

2.4 Implementation Issues 32

2.5 Sample Applications 35

2.5.1 Example 1 36

2.5.2 Example 2 39

2.6 Summary 41

3 Storage Organization of Polyhedral Surfaces **45**

3.1 Storage of Polyhedral Models 46

3.2 Triangular-Loop Data Structure 48

4 Interactive Manipulation of Polyhedral Surfaces **57**

4.1 Storage Organization of Vertices 58

4.2 Interactive Access of Vertices 59

5 Interactive 3D Point Location & Applications 61

5.1 A Modified TLDS 63

5.2 Interactive Flag-Status Update 65

5.3 Highlight & Selection of Data 66

5.4 Visual Example 68

5.4.1 Example 1 69

5.4.2 Example 2 71

5.4.3 Implementation Issues 73

5.5 Summary 74

6 Interactive Segmentation & Shape Modification of 3D Objects 75

6.1 Related Work in Comparison 76

6.1.1 Storage of Polygonal Models 77

6.1.2 Surface Segmentation 77

6.2 Interactive Shape Modification 78

6.3 Interactive Surface Segmentation 78

6.4 Integration of Modified Segments 84

6.5 Visual Examples 87
6.5.1 Example 1 87
6.5.2 Example 2 92
6.6 Summary 96

7 Conclusions & Future Directions 97

7.1 Practical Extensions 99
7.1.1 Modeling & Visualization of Layered Objects 99
7.1.2 Segmentation & Shape Modification of 3D Objects 100
7.2 Potential Applications 102
7.2.1 Computer-aided anatomy browsers 102
7.2.2 Construction of 3D cross sections 103
7.2.3 Interfacing with existing CAD packages 103
7.3 Future Work 105
7.3.1 Dealing with Surface Collisions 105
7.3.2 Vertex Addition in TLDS 106
7.3.3 Integration of Two Polyhedra 106
7.4 Final Remarks 107

List of Figures

2.1 A cuboid. 19

2.2 Approximation of surface normals. 21

2.3 Cuboid neighbors. 24

2.4 A graphical view of the data structure holding the cuboids. 28

2.5 faces that become visible 31

2.6 Re-rendering diagonal faces 34

2.7 Successive virtual holes on thigh 38

2.8 Successive virtual holes on arm 40

3.1 A simple three polygons model 49

3.2 A visual view of TLDS 54

5.1 Rendering in different modes 68

5.2 Preserving special features on an artificial object. 70

5.3 Preserving special features on an actual object. 72

6.1 Mapping from canvas to array 80

6.2 Inclusive & exclusive bounding 81

6.3 Segmentation, modification, & re-introduction example 85

6.4 Donkey-head shape modification. 89

6.5 Donkey-head and donkey-tail shape modification. 91

6.6 Vase-base shape modification. 93

6.7 Vase-opening shape modification. 95

List of Tables

1.1 Main thesis variables & symbols. 9

1.2 Thesis abbreviations. 9

1.3 Thesis algorithms. 10

1.4 Special thesis notation. 10

3.1 Amount of storage required by PEL, WEDS, and TLDS. 55

List of Algorithms

2.3.1 The FORM-LAYER algorithm. 26

2.3.2 The INSERT-CUBOID-IN-CPTABLE algorithm. 29

2.4.1 The DISPLAY-OBJECTS algorithm. 32

2.4.2 The HIDE-CUBOIDS algorithm. 35

3.2.1 The CREATE-TLDS-REPRESENTATION algorithm. 52

6.3.1 The CREATE-INCLUSIVE-SUB-POLYHEDRA algorithm. 82

6.3.2 The CREATE-EXCLUSIVE-SUB-POLYHEDRA algorithm. 83

6.4.1 The INTEGRATING-SUB-POLYHEDRA algorithm. 86

Chapter 1

Introduction

1.1 Motivation

In the course of building 3D models for purposes of design and production, geometric polyhedral surfaces composed of multi-size patches undergo many position transformations for editing and modification purposes to enhance user-control over the shape of the final product. This dissertation focuses on the development of various algorithms and data structures suitable for direct and rapid interactive manipulation of 3D objects.

The design of actual 3D objects is facilitated by means of virtual modeling on the computer screen, on which, a series of *data manipulations* take place throughout the design of the model. In computer graphics, these *data* are typically 3D points that constitute geometric polyhedral patches made up of triangular, quadrilateral, or other patches, from which 3D object models are defined. The *manipulation* of such data

facilitates object design and understanding; in the simplest form, it involves a series of data transformations such as rotation, scaling, and translation of the displayed objects.

A less researched, and a more computationally involved form of data *manipulation* considers operations such as the construction of *virtual holes* on layered objects, where part of the object is *virtually removed* to allow a see-through to hidden parts. Other operations include *shape modification*, where the spatial location of some of the points is modified independently of other points to perform operations known as *tweaking*, and *surface segmentation*, where a selected portion of the surface is *segmented off* for visualization and design purposes.

In the more involved form of data manipulation, all the operations have to directly identify a set of patches that are to be virtually removed, modified in shape, or segmented. Since these patches are defined by points in space, it is then a matter of locating these 3D points, and then applying the manipulation algorithms accordingly. A straight forward approach for locating these points is to apply linear or logarithmic searches within the data, in which the $\{x, y, z\}$ values of selected points on the displayed model are compared to the $\{x, y, z\}$ values of a subset of points of the input data. But if immediate and interactive responses are sought, then even logarithmic time searches may be slow with large data sets. A purpose of this thesis is to design algorithms and data structures that facilitate the interactive ability to find the exact location of selected 3D points in spontaneously, and to illustrate how to apply the solutions proposed to the involved form of data manipulation operations mentioned above.

1.2 Objectives

To simplify and speed up the interactivity process during the design phase of a model on the screen, existing techniques such as the *scan-line* and the *z-buffer* algorithms are modified by extending their functionality to include a third dimension, which incorporates depth of model data such as vertices, edges, and polygonal patches to facilitate immediate data access. Such modifications will allow for operations such as *interactive 3D point location*, *interactive construction of virtual holes* on the displayed objects for viewing hidden layers, and *interactive 3D surface segmentation and modification*. However, this requires that data for the polyhedra to be displayed, be in a representation that is convenient to support direct interaction with the user—interaction that allows access to the spatial data making up the polygons in constant running time. For the model representation, several generic and particular data structures are developed based on the geometry of the model.

Another purpose of this dissertation is to review current progress in this area; provide the necessary motivation behind the construction of such functionalities for CAD, CAM, and more relaxed applications in the context of computer graphics; present the necessary algorithms and data structures to facilitate the involved form of data manipulation; and provide visual examples which are a direct result of the implementation of the developed underlying structures and techniques.

The rest of this chapter is divided into several sections. Each section is introduced with a synopsis of a current research problem, and the solution this thesis proposes. These sections are subsequently, individually or in groups, expanded into chapters that

discuss related work on the problem in detail, provide algorithms and data structures that are suggested to solve the problems, and give sufficient visual examples that are a direct result of implementing the proposed concepts and methods.

1.3 Modeling & Visualization Techniques

Many natural and manufactured objects are layered. For example, a human thigh has layers of skin and bone, among others; architectural buildings have walls of exterior sheeting, insulation, and interior finishing. By modeling each layer with a sheet of cuboids, the thickness property that is inherent in the layers is preserved. In addition, modeling layers with cuboids simplifies the making of virtual holes in external layers to view portions of internal layers, which aids in geometric volume visualization. For this work, algorithms, data structures, and a prototype implementation that allows interactive construction of virtual holes on 3D layered objects are presented in Chapter 2.

1.4 Shape Refinement Control

The development of a smooth geometric model from a crudely approximated object is often sought in CAD/CAM, computer-aided geometric design and similar applications. For example, many surface fitting techniques have as their input data crudely sampled points from free form objects. It is then a problem to digitally reproduce such objects to look smooth again. However, regardless of the surface fitting technique used

(the choice is usually application dependent), there are often particular features on the approximated models that should not undergo the smoothing procedures. A technique for interactively *marking* such sampled points as special, for preservation of features such as cusps or geometric C^0 continuity across edges is presented in Chapter 5. The technique is based upon data structures and algorithms presented in Chapters 3 and 4.

1.5 Polyhedra Segmentation

During the design phase of the model on the computer screen, particular regions in the polyhedron may need intensive user-attention to cast the model into the desired shape. This may entail many transformation operations to put the model in a suitable position for the user to interact with for local editing. Instead of applying the transformation operations uniformally on all the patches (which may well be in the order of many thousands) of the polyhedron, where only a minute subset of the patches need direct user attention for manipulation, simple interactive techniques are provided to segment the main polyhedron into sub-polyhedra by using bounding volumes. These bounding volumes are applied on geometric 3D polyhedra to segment off a subset of patches to be displayed and worked on in isolation. This provides more direct interaction with the sub-polyhedra with very fast results and low memory requirements. For example, a large model may be segmented into n sub-polyhedra, each segment may be transformed, edited, and focused on independently. This is discussed in Chapters 3, 4 and 6.

1.6 Polyhedra Shape Modification

Real-time access of 3D model vertices on the 2D screen is not straightforward, for not only are the spatial coordinate positions of such data of importance, but also a vertex's associated edges and surrounding polygons need to be accessed, in real time, to edit the shape of polyhedron effortlessly (by modifying the spatial positions) and without data search of any kind. This requires that input data of a polyhedron, which are usually given in the form of a list of polygons defined in terms of vertices, be pre-processed and stored in a form that allows such data access to surrounding polygons and out-going edges in $O(1)$ processing time. Mechanisms and algorithms for such an application are discussed in Chapters 3, 4 and 6.

As an example of the need of direct shape modification techniques, a CAD engineer may start the design of a 3D model by a simple profile of several vertices. This profile is rotated to obtain a surface of revolution that would resemble the target part fairly closely. The engineer can then make small changes to achieve the target shape. However, without methods for direct interaction with vertices to modify their positions, it may be a time-consuming process to alter the *un-editable* surface of revolution into the desired target shape. A detailed example is illustrated in Chapter 6.

1.7 Integration of Sub-polyhedra

The main purpose of polyhedron segmentation into smaller sub-polyhedra is to apply data manipulation operations in all of its forms independently on the various segments.

Once any of these segments are modified in position (e.g., scaled, rotated, shifted, etc...) and/or changed to a shape desired by the user, these segments need to be re-introduced back to the main structure they were extracted from. This requires the application of a series of inverse operations with modifications in effect, and fitting back the segments both visually on the display to the original polyhedra, and in the numerical stored data where the $\{x, y, z\}$ triplet value of some of the vertices has been modified. This work is discussed in Chapters 3, 4, 6.

1.8 Thesis Particulars & Special Terms

The algorithms and data structures developed are implemented on UNIX/Linux platforms. OSF/Motif and the X11 libraries were used for implementing the required interfaces and data structures presented in this thesis, and for generating most of the figures. The xfig graphical package is used for generating the illustrative figures. The algorithms are implemented using the C language. LaTeX typesetting is used for writing this text.

Following are tables of symbols, abbreviations, algorithms, and other conventions used in this thesis. The lists are arranged in alphabetical order, and serve as a quick index for notations particular to this thesis. Table 1.1 shows the important symbols/variables used (mostly inside algorithms) . Table 1.2 presents the more important abbreviations used throughout the thesis. Table 1.3 lists all the algorithms developed in the thesis, with a summary of what each algorithm does. In Table 1.4, special notations that are used within and outside of the algorithms are listed. The C lan-

guage notation is used for presenting data structures and algorithms. For example, the character "*" may precede some variables in some data structures to indicate that the variable is a pointer. Pointers hold the address of a *variable* rather than the value. Pointer variables have the advantage that they do not have to be updated as the value of the pointed-to variable changes.

Symbol	**Description**
$\mathbb{E}$	Number of edges in a polyhedron
$E_j \in E$	The jth edge in array E of edges
$\mathbb{P}$	Number of polygons in a polyhedron
$P_k \in P$	The kth polygon in array P of polygons
$\mathbb{V}$	Number of vertices in a polyhedron
$V_i \in V$	The ith vertex in array V of vertices
$\infty / -\infty$	Largest/smallest machine representable value

Table 1.1: Main thesis variables & symbols.

Abbreviations	**Description**
BB	Bounding Box/Volume
CAD	Computer-Aided Design
CAM	Computer-Aided Manufacturing
CP	Cuboid Pointer table (or array)
EP	Edge Pointer table (or array)
PEL	Pointers-to-an-Edge-List
TLDS	Triangular-Loop Data Structure
VHR	Virtual Hole Radii
VP	Vertex Pointer table (or array)
WEDS	Winged-Edge Data Structure

Table 1.2: Thesis abbreviations.

Algorithms	Short Description
CREATE-EXCLUSIVE-SUB-POLYHEDRA	Forms sub-polyhedra of polygons that are completely inside BB
CREATE-INCLUSIVE-SUB-POLYHEDRA	Forms a sub-polyhedra of polygons that are completely or partially inside BB
CREATE-TLDS-REPRESENTATION	Converts polygons data input in raw-format into one of TLDS
DISPLAY-OBJECTS	Display visible faces in non-marked model cuboids
FORM-LAYER	Construct a sheet of cuboids
GET-EID	Returns the reference number of the edge connecting two vertices
GET-VID	Returns a vertex's reference number
HIDE-CUBOIDS	Display surface layers immediately hidden by obscuring cuboids
INSERT-CUBOID-IN-CPTABLE	Uses cuboid-centroid to insert it into CP table
INTEGRATE-SUB-POLYHEDRA	Introduces vertices with modified spatial positions into original structure

Table 1.3: Thesis algorithms.

Font Shape	Usage
`Underlined typewriter` text	Standard key words
Bold-faced sans serif text	User defined types
SMALL CAPS text	Algorithm & procedure names
// *Emphasized* text	Comments used in algorithms
Emphasized sans serif text	Variable names, used in & out of algorithms

Table 1.4: Special thesis notation.

Chapter 2

Modeling & Visualization of Layered Objects

Surface segmentation has important applications in computer graphics. One of the objectives of such a technique is the removal of segmented portions from a scene, so that hidden back layers become visible [10, 34]. This has diverse applications in fields such as anatomy, where professionals involved in the study of the human body may strip away selected portions of any layer(s), to visualize the geometric relationship among the different body organs for making medical decisions [4, 6, 35, 48, 56]. Motivation for such work is also discussed in [29], where data for a thorax is first obtained by attaching hundreds of electrodes to a human thorax-shaped electrolytic tank. The electrode locations are chosen so as to simplify the construction of a polyhedral model (consisting of triangles, tetrahedra, and cuboids) viewable on the 2D screen, which makes it possible to perform surface segmentation.

Surface segmentation also plays an important role in other disciplines such as CAD

and CAM[9, 11]. For example, in [11], hollowing out a CAD model to see its internal structures contributes to the reduction of material used, and gives control over shaping the internal structures of the model.

In this work, algorithms and data structures are introduced to interactively create any number of virtual holes at any area on an object or in the scene, to view portions of hidden geometric surfaces.

Architectural and CAD models composed of surface patches may be represented by cuboids* that are aligned to form of a sheet with thickness, by simply generating corresponding patches on a complementary surface (inner or outer surface, depending on what the original surface is designated to be). These patches are generated at specified distances along surface normals, which may be computed in certain ways to avoid visual anomalies at high curvature regions, see [40]. Corresponding points on corresponding patches are then connected together to form cuboids.

Representing thickness of irregular physical models (objects with no simple mathematical definition) may be achieved as described in [21], where all points on both exterior and interior are selected according to their physical locations, as is shown later in this chapter. By restricting object modeling to cuboids, uniform application of the methods introduced in this chapter is possible. More general layer topologies (such as triangular patches) are discussed from Chapter 3 onwards..

The user may control the size and shape of the virtual hole since the techniques to be

*A cuboid has the characteristics of a cube, but adjacent faces are not constrained to form a right angle in-between.

presented deal with the objects' components (i.e., the cuboids) directly. In addition, the cost involved for making virtual holes is chiefly the time it takes to render the portions behind the virtual hole, not the time it takes to locate where the surface portions to be processed are stored; they are located in $O(l)$ time, where l is the number of layers that may be penetrated by a ray cast from the viewer's eye onto the object(s). To construct virtual holes, the user interactively moves a pointing device to the region of interest and makes a selection. This causes the selected cuboids (e.g., the subset of cuboids surrounding the selected target region) not to be rendered, and causes the portions of the layer(s) immediately hidden by the removed subset to be spontaneously displayed.

The rest of the chapter is organized as follows: Section 2.1 takes a brief look at related work. Section 2.2 presents some background and terminology. Algorithms and data structures are presented in Section 2.3, followed by a section that discusses practical issues for implementation. Example applications of the techniques introduced are shown in Section 2.5. Concluding remarks are presented in Section 2.6.

2.1 Related Work

Among the techniques that may be tailored to view object interiors is that of *octrees* [5, 19, 23]. Octrees are designed mainly to give an efficient use of storage. The idea is to recursively subdivide a 3D space that contains the object(s) to be rendered, into octants (cubes) of similar sizes until the contents of octants are homogeneous. A scene rendering may then start by displaying the octants from back to front. One problem

that may be encountered if octrees are employed for making virtual holes is that a patch in 3D space may have its parts stored in more than one octant. Therefore, it may not be possible to control what is to be displayed. Furthermore, the size of octants in one cross-section may be different, so the size and shape of the virtual hole is also not under the control of the user.

Another technique for viewing hidden layers is the method of *ray tracing* [17, 28]. Ray tracing has been successfully used as an algorithm for photo-realistic rendering of objects. In typical ray tracing, the intensity of pixels on the rendered object is determined by the intersection points encountered when casting rays from the eye to the object, among other factors.

Two possible ways to view portions of hidden layers using basic ray tracing methods are:

- Remove the front object(s) and re-render the scene, or
- Make the front object(s) transparent and re-render the scene.

The first option does not allow the construction of *virtual holes* to view inner sections, or to see how the interior fits with the overall exterior of the rendered object. In the second option, it is possible to make the exterior layers transparent to view the interiors, but this means significant additional computation, and the result may not be satisfactory because refraction and other properties of the external layers have to be taken into consideration [23, 24], particularly as the number of overlapping virtual holes increases. So, although ray tracing can easily be adapted for objects in which certain

surface patches or segments are left out, the actual process of leaving out the patches or segments is a characteristic of how the object is modeled. This work proposes a modeling method for efficiently leaving out patches or segments of the rendering of displayed object.

The algorithms and data structures provided here may be incorporated into ray tracing motors to enhance speed and interactivity for viewing hidden layers. In [44], for example, where all intersection points/segments of objects along a ray are sorted according to their distance from the casted ray before any rendering calculations are performed, the methods proposed here can be used to interactively remove surface portions to reveal hidden layers, indefinitely, since the notion of preprocessing is common to both works. Also, in [47], mechanisms for translucent surface rendering are developed to show internal components of the rendered object. However, this has its limits as the number of internal and overlapping components increases. By establishing a relationship between the voxels that belong to each subcomponent (e.g., bone, fat, or soft tissue), as shown in the work proposed here, the user can have control on not rendering portions of sub-components that may obscure the rendition of a deeper, hidden layer.

Other work that supports the notion of surface segmentation is found in [12]. Cignoni *et al* present the *MagicSphere* as a tool with many viewing filters. The 3D position of the MagicSphere on the object, and its radius determine the set of encircled polygons to be manipulated. The functionalities provided on the encircled polygons include visualization in isolation, or erasure from the scene to allow a see-through view. This

has its practical uses. For example, to visually simulate the path of a drilling tool, the MagicSphere can be moved along a prescribed path[12]. However, on layered models the MagicSphere as presented does not distinguish between polygons belonging to different layers or objects, since all polygons within the reach of the sphere radius are operated on similarly (e.g., clipped), which is not desirable in some applications. The work proposed here realizes the relationship between each object and its constituting polygons in a multi-object scene.

In most cases of the above cited work and other work [4, 9, 11, 28, 35, 38, 44, 46, 47, 56], the methods and algorithms were applied on volumetric data obtained from CT (computed tomography) scans, or object voxelization (by casting a grid of parallel rays inside the object's bounding block [11]). Although this has its advantages in the richness of detail representation, the number of data elements may be impractical for storage and interactive performance. This explains the need to resort to polyhedral representation [12], or to apply reductions in the number of representing polygons, of which there may be millions if, for example, *marching cubes* algorithms are used for 3D surface reconstruction [12, 26].

Representing objects by polygons has advantages regarding storage, ease of manipulation, and the increasing requirements of interactiveness with modeled objects. In [52], for example, the human body is represented by B-splines of four-sided patches, but some individual patches are triangulated to stitch the different parts of the body. However, representing objects with polygons also has its shortcomings when modeling complex (without a simple mathematical definition) objects: thickness is usually

not represented. Image processing techniques can be employed to simplify the task of 3D graphical reconstruction of digitized complex objects. For example, in [21], a part of the heart muscle is first digitized by an MR (magnetic resonance) scanner. Two contours (one circling the exterior, the other circling the interior of the heart muscle) on each of the sample cross sections are outlined. Corresponding points are then connected such that four-sided patches are formed. By connecting all points on the sample cross sections correspondingly, a cuboid representation of the object is formed. Such a representation has its advantages in animating the dynamic movements of the heart muscle in real time.

2.2 Background and Terminology

Objects in the methods presented here are modeled with sheets (layers) of cuboids – two four-sided patches, on the interior and exterior of the modeled object, connected together.

In architectural and CAD/CAM applications, where many models are based on four-sided patches that are generated from B-spline, Bézier, or any tensor-product surfaces, a complementary surface may be generated from the original one. Corresponding points from both surfaces are connected together to form a thick surface (sheet of cuboids). Results that lead to self-intersections from the complementing surface (i.e., in the case of the complementing surface being an *offset* surface) should be avoided, since the modeling technique presented here is for visualization purposes. This may be accomplished by visual inspection of the prototype model. Work discussed in [51,

52] offers means of addressing problems that may arise, should the objects modeled exhibit regions of high curvature such that self-intersections are very likely.

For models that correspond to data measured and sampled from physical objects, techniques similar to the work of [21] discussed above are more suitable and realistic. Similar techniques are used for the prototype implementation of the methods proposed here.

To form a cuboid, two corresponding patches, one from the interior of the object and the other from the exterior of the object are connected together by using them as back (*bk*, from the inside) and front (*fr*, from the outside) faces (patches). The corners of these two patches are connected together by adding four edges, each edge connecting two corresponding corners from the two patches, thereby generating four more faces (designated to represent the sides): top (*tp*), bottom (*bm*), right (*rt*), and left (*lt*) faces, as shown in Figure 2.1. A cuboid is the smallest object element that may be displayed as part of the scene, or else, *hidden* from the scene by not displaying it.

To see through the external layer of an object to view its inside at a specific region, a virtual hole is made by selecting and marking a set of cuboids for *hiding*. This has the effect of interactively displaying the set of cuboids behind the constructed virtual hole, whereas the rest of the scene is left intact. A right-handed coordinate system is assumed.

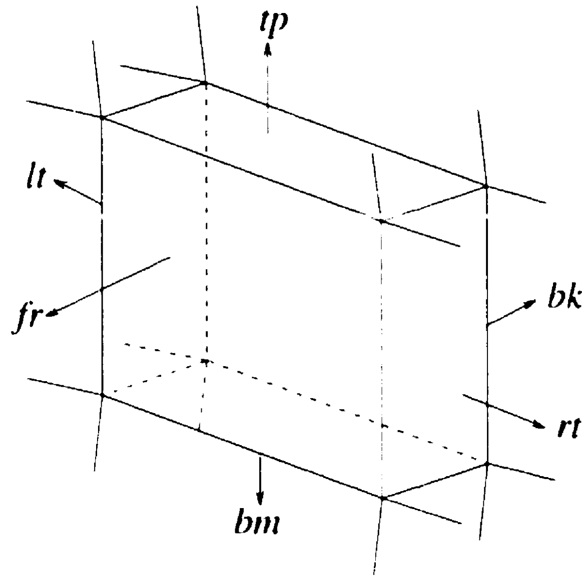

Figure 2.1: A cuboid.

2.3 Algorithms and Data Structures

Although the work described in this chapter is applicable to infinitely thin surfaces that are made up of patches, cuboids are used because of the realism they add to modeled objects.

2.3.1 Forming the Layer

For irregular objects, all points defining each cuboid are input according to their actual space position. For CAD/CAM models, the defining patch surface may be transformed into a sheet of cuboids by generating a complementary surface (e.g., exterior if original surface is taken to be the interior, or vise versa), whose patches have a one-to-one

correspondence to those on the original surface.

The complementary surface for regular models (objects with simple mathematical definition) may be produced in three ways:

(a) Independent of the original surface.

(b) Completely dependent on original surface, such as an offset surface, for example, or

(c) Partially dependent on original surface. For example, distance between corresponding patches increases/decreases along a parametric direction(s), or, each point in the complementing surface is generated as a function of the normals belonging to patches surrounding the corresponding point on the original surface.

In all these methods, however, a one-to-one correspondence between every two patches (one from the original, the other from the complementary surfaces) must be maintained. The method described in (c) is used to generate the complementing surfaces for the human arm model described later in Subsection 6.2. This method is elaborated on as an example of forming cuboids next.

Let w be the number of rows, l be the number of columns in the array of patches, and k indicate depth, such that $k = 1$ if the patch is in the original surface, and $k = 2$ if the patch is in the complementary surface. Further, let the patches that make up the original surface be labeled s_{ij1}, where $i = 1, 2, \ldots, w$, $j = 1, 2, \ldots, l$, and k is the depth described above. Each patch s_{ijk} has corners $\mathbf{p}_{ijk}, \mathbf{p}_{i+1,j,k}, \mathbf{p}_{i+1,j+1,k}, \mathbf{p}_{i,j+1,k}$

(counter-clockwise), as shown in Figure 2.2 by the patch in the top right corner. To transform the original surface into a sheet (surface with thickness property), a complementary surface is generated as a function of it. Each point in the complementary surface is generated as a function of the average normal vectors of the surrounding one, two, or four triangles. Depending on the point's location on the surface as shown in Figure 2.2, the unit normals $\mathbf{N}_a$ for a *corner* point, $\mathbf{N}_b$ for a *boundary* point, or $\mathbf{N}_c$ for an *interior* point, are calculated as follows:

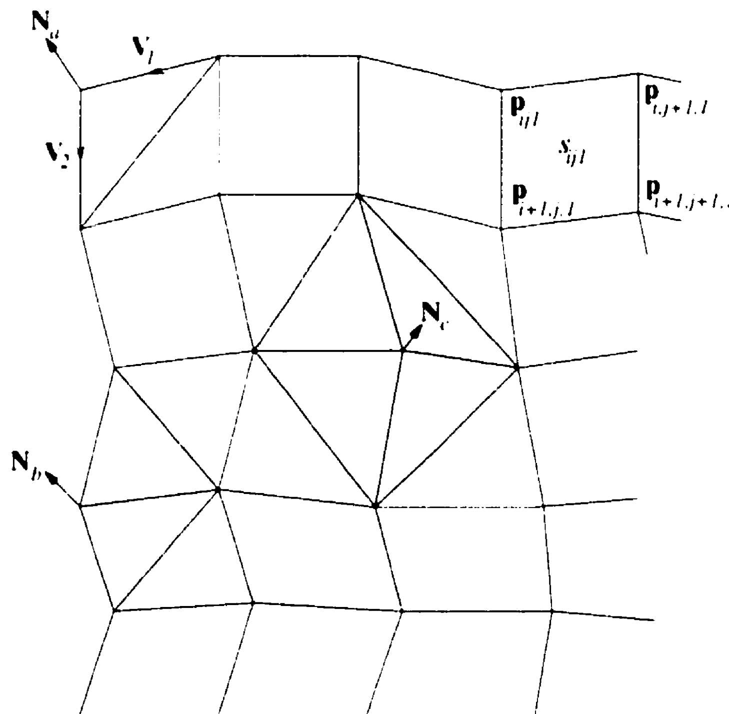

Figure 2.2: Approximation of surface normals.

1. **Point on corner of surface**, such as the point with unit normal $\mathbf{N}_a$ in Figure 2.2. Assuming that α is an offset distance scaler, $\mathbf{p}_{ij1}$ is the point on the corner of

the original surface, and that

$$\mathbf{N}_a = \frac{\mathbf{v}_1 \times \mathbf{v}_2}{||\mathbf{v}_1 \times \mathbf{v}_2||},$$

where

$$\mathbf{v}_1 = \mathbf{p}_{ij1} - \mathbf{p}_{i,j+1,1},$$

$$\mathbf{v}_2 = \mathbf{p}_{i+1,j,1} - \mathbf{p}_{ij1},$$

the corresponding corner point $\mathbf{p}_{ij2}$ on the complementing surface is:

$$\mathbf{p}_{ij2} = \alpha \mathbf{N}_a + \mathbf{p}_{ij1}.$$

2. **Point on boundary of surface**, such as the point with unit normal $\mathbf{N}_b$ in Figure 2.2. Here, the unit normal is calculated for each of the two shaded polygons, it is then averaged (divided by two), multiplied by α, and added to the corresponding $\mathbf{p}_{ij1}$.

3. **Point in surface interior**, such as that with unit normal $\mathbf{N}_c$ in the same figure. $\mathbf{N}_c$, the average of the unit normals of the four shaded polygons is multiplied by α, and then added to the corresponding $\mathbf{p}_{ij1}$ to generate $\mathbf{p}_{ij2}$.

In this regard, the complementing surface is not just a translation of the original surface, it is an approximation that depends on the curvature of the original surface, as an attempt to generate a realistic layer view of the object being modeled. There may be other ways to compute the normal vectors. For example, dividing all normals by a

standardized length (e.g., the length of the smallest normal of the processed patches) results in a thickness that is more dependent on the size of the patch in relation to others. Other techniques for computing vertex normals are found in [40].

To form the layer, which is a collection of all the cuboids belonging to one object in the scene, each one of the six faces of a cuboid is represented by four points $\mathbf{p}_i = \{x_i, y_i, z_i\}, i = 1, 2, 3, 4$, and a binary flag named ***visible*** to indicate whether the face is visible or not. The value of the flag ***visible*** is set to "TRUE" if it is not obscured by other faces and has a normal with positive z-component, "FALSE" otherwise.

A cuboid's data structure is defined by several fields, some of which are references to neighbor cuboids as shown in Figure 2.3; this facilitates immediate access to the neighbors of selected cuboids. The fields in the **cuboid** structure are as follows:

- Six faces: back (***bk***), front (***fr***), left (***lt***), right (***rt***), top (***tp***), and bottom (***bm***) faces.
- A ***centroid*** point that is calculated as the average point of the eight corner points of the cuboid. It is used to determine where the cuboid should be mapped to in a cuboid pointer (CP) table. The method of mapping a cuboid to a CP table cell is described in the following subsection.
- A binary flag to indicate whether a cuboid is to be ***hidden*** or displayed.

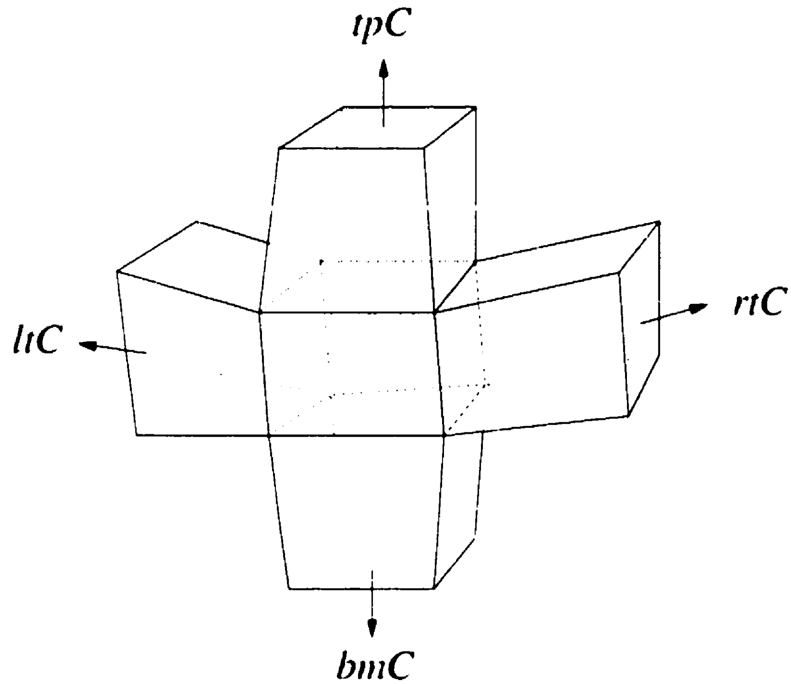

Figure 2.3: Cuboid neighbors.

- A color code to determine the three colors for the exterior (front faces), interior (back faces), and sides (left, bottom, right, and top faces) of each cuboid; each layer should be assigned a different ***clr-code*** for distinction.

- A pointer to the ***next*** cuboid whose ***centroid*** maps to the same CP table cell, but has a smaller z-component (i.e., it is further from the viewer).

- Four additional references, each of which points to a neighbor cuboid that is in the same layer: the cuboid on top (***tpC***), below (***bmC***), to the left (***ltC***), and to the right of it (***rtC***), as shown in Figure 2.3.

The following data structures for registering points in 3D (**xyzPoint**), a cuboid face (**cFace**), and a cuboid structure (**cuboid**) are written as an example of the data structure.[†]

```
struct xyzPoint {
 real x, y, z;
}
```

```
struct cFace {
 xyzPoint p1, p2, p3, p4;
 int visible;
}
```

```
struct cuboid {
 cFace fr, bk, tp, bm, lt, rt;
 xyzPoint centroid;
 int hidden;
 int clr-code;
 cuboid *next;
 cuboid *tpC, *bmC, *ltC, *rtC;
}
```

The side faces of a cuboid are *potentially visible* (that is, they are considered for rendering) if they are not obscured by neighbor cuboids. Further, each face of a cuboid is *visible*, if and only if, it is potentially *visible*, and has a normal with a positive z component, in which case it will be rendered, otherwise, it is not *visible* (i.e., *visible* set to "FALSE") and is not considered for rendering. It is important to distinguish between the effects of what is meant by *visible* and *hidden*. A cuboid whose *hidden*

[†]See Section 1.8 for font-usage.

flag is set to "TRUE" will not be rendered as part of the scene. However, if a cuboid's *hidden* flag is set to "FALSE", then only *visible* faces of it are rendered.

The following algorithm (FORM-LAYER) shows the major operations involved in forming each layer. The objective here is to make the faces of each cuboid identified by their names, and to make each cuboid aware of who its four neighbors are. In the following algorithm, it is assumed that the *clr-code* of each layer is given in the data file, the *hidden* flag for each cuboid is set to "FALSE", *visible* flags of side faces on the sheet boundary are set to "TRUE", and that some neighbors of cuboids on sheet boundary may be non-existent (e.g., the upper neighbors of cuboids on top row). For simplicity, the algorithm only addresses open layers (that is, first and last column do not meet); extra "book-keeping" is required to address closed objects (e.g., a closed cylinder, by making first and last columns become immediate neighbors).

Algorithm 2.3.1 The FORM-LAYER algorithm.

<u>for</u> $i \leftarrow 1$ <u>to</u> w
  <u>for</u> $j \leftarrow 1$ <u>to</u> l
- ▶ Identify each of the six faces of the (i, j)'th cuboid as ***fr***, ***bk***, ***tp***, ***bm***, ***lt***, or ***rt*** faces.
- ▶ *visible* ← TRUE for ***fr*** and ***bk*** faces, FALSE for others.
- ▶ *centroid* ← average $\{x, y, z\}$ value of the eight corner points.
- ▶ Identify the (i, j)'th cuboid neighbors such that cuboids in the $(i, j-1)$, $(i, j+1)$, $(i-1, j)$, and $(i+1, j)$ positions are the ***ltC***, ***rtC***, ***tpC***, and ***btC*** cuboids, respectively.
- ▶ Insert (i, j)'th cuboid in the CP table as shown in algorithm INSERT-CUBOID-IN-CPTABLE (*the next algorithm*).

2.3.2 Organization of the Cuboids

The cuboids are organized according to where their centroids fall on the view plane. An orthographic projection onto the x-y plane is used. In order to determine which cuboid(s) are being selected for being *hidden* promptly, they are stored in the two-dimensional CP table. Storage in this table depends on two entities:

1. The (x, y) value of its *centroid* to determine its location in the CP table, and
2. The z-component of its *centroid* to determine its depth within the obtained location.

The *centroid* is the average of a cuboid's eight vertices; it is used to determine where each cuboid should reside in the CP table. Figure 2.4 gives a graphical view of how cuboids are stored in the CP array.

The cuboids that form the different layers in a scene are displayed on an $r \times c$ pixel canvas. The canvas is subdivided into a grid of $R \times C$ squares, where $R = r/L$, $C = c/L$, and the side-length of a square, L, is a positive integer. References to these cuboids are stored in a CP table, which is an $R \times C$ array of cells. Each cell corresponds to a square in the canvas grid.

The idea behind the $R \times C$ CP array is the following: Since objects are displayed on a designated $r \times c$ pixel canvas on the screen, the canvas is subdivided into an $R \times C$ grid of squares by choosing $R = r/L$ and $C = c/L$, L being the length of a square-side in pixels.

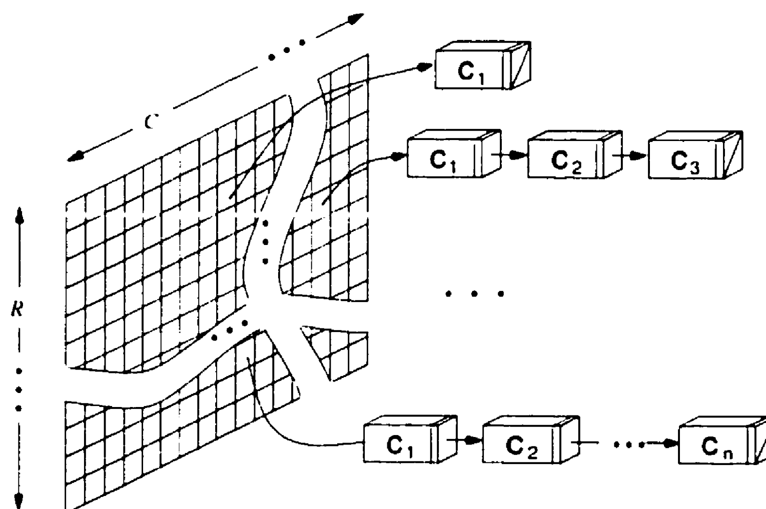

Figure 2.4: A graphical view of the data structure holding the cuboids.

After computing the $\{x, y, z\}$ value of the ***centroid***, the integer parts of (x, y), denoted by $(\bar{x}, \bar{y})$ are extracted. To determine the exact CP table cell the cuboid is inserted at, the pair $(\dot{x}, \dot{y}) = (\bar{x} \mod L, \bar{y} \mod L)$ is calculated such that the cuboid is stored in the $\dot{x}'th$ row, $\dot{y}'th$ column in the CP table cell.

Next, the z-component in ***centroid*** is used to determine where a cuboid should be stored within a cell in the CP array. If two or more cuboids are mapped to the same location in the CP table, the one that is closest to the viewer (i.e., has a larger z value) is stored first, and is made to reference the other cuboid through the ***next*** link, introduced earlier in the cuboid data structure. Based on this, once a cuboid becomes ***hidden***, side faces of the neighbor cuboids, and other hidden cuboids (that may belong to different layers) are immediately accessed through the earlier initialized neighbor links, or through the next link, respectively.

In Figure 2.4, a cell pointing to two or more cuboids in the form $C_i \rightarrow C_j \rightarrow \cdots$ implies that C_i is closer to the viewer than C_j because it has a greater z-component. The *insertion* sort algorithm is sufficient for maintaining references to *cuboids* in the CP table cells in sorted order [37]. The following algorithm, INSERT-CUBOID-IN-CPTABLE, summarizes the main steps taken in inserting cuboids in the CP table.

Algorithm 2.3.2 The INSERT-CUBOID-IN-CPTABLE algorithm.

- ▶ $(\dot{x}, \dot{y}) \leftarrow (x \mod L, y \mod L)$, where (x, y) are the integer parts of (x, y) of the *centroid*.
- ▶ `if` cell $(\dot{x}, \dot{y})$ in the CP table is empty, `then`
  insert cuboid
  `else`
  insert in the proper place in the linked list according to cuboid *centroid*'s depth.

The user makes virtual holes simply by selecting the desired region on the object(s) in the canvas by a pointing device. The location indicated by the user is recorded as an (i, j) pair of values on the 2D pixel canvas, from which, $(\dot{i}, \dot{j})$, where $\dot{i} = (i \mod L)$ and $\dot{j} = (j \mod L)$, are derived. The pair $(\dot{i}, \dot{j})$ is used to access the corresponding CP table cell and causes the *hidden* flag of cuboid(s) stored there to be set to "TRUE" (the complete algorithm, HIDE-CUBOIDS, is given later in Section 2.4). Once cuboids are set *hidden*, cuboid(s) that were obscured by them and also share the same cell(s) in the CP array are traversed and displayed as part of the scene.

2.3.3 Rendition of Cuboids

Initially, the *hidden* flag of each cuboid is set to "FALSE", so all cuboids are rendered. The visibility of each face of a cuboid, however, depends on a few factors. In Figure 2.3 for example, the *visible* flag of the *rt* face of the shaded cuboid will be set to "FALSE", because even though the face normal has a positive z-component, it is obscured by the *lt* face of the *rtC* cuboid to the right of it.

When the cuboids are first formed, front (*fr*) and back (*bk*) faces have potential for being visible on any cuboid, so their visibility flag (*visible*) is set to "TRUE". In addition, the following particular faces also have potential for being visible: top faces (*tp*) of cuboids on top row, bottom faces (*bm*) of cuboids on bottom row, left faces (*lt*) of cuboids on first column, and right faces (*rt*) of cuboids on last column, so their *visible* flag is also set to "TRUE". All other faces are initially not visible because they are obscured by their neighbor cuboids; their *visible* flag is thus set to "FALSE".

Before rendering, the normals of all faces with *potential* for being visible are calculated. Faces whose normals have a negative z-component are not visible to the viewer, so their *visible* flag to "FALSE". At rendering time, only *visible* faces of non-*hidden* cuboids are rendered (using the z-buffer algorithm), thereby saving time.

When a cuboid's *hidden* flag is set, there is potential for four or fewer non-*visible* faces (depending on the cuboid's location in the layer) of the surrounding neighbors to become visible and exhibit layer thickness. These faces are the bottom face (*bm*) of the top cuboid (*tpC*), *rt* face of *ltC*, *tp* face of *bmC*, and *lt* face of *rtC* cuboid.

Figure 2.5 shows examples of faces becoming visible after hiding some cuboids. The arrows on some cuboids indicate that they are selected to be not displayed.

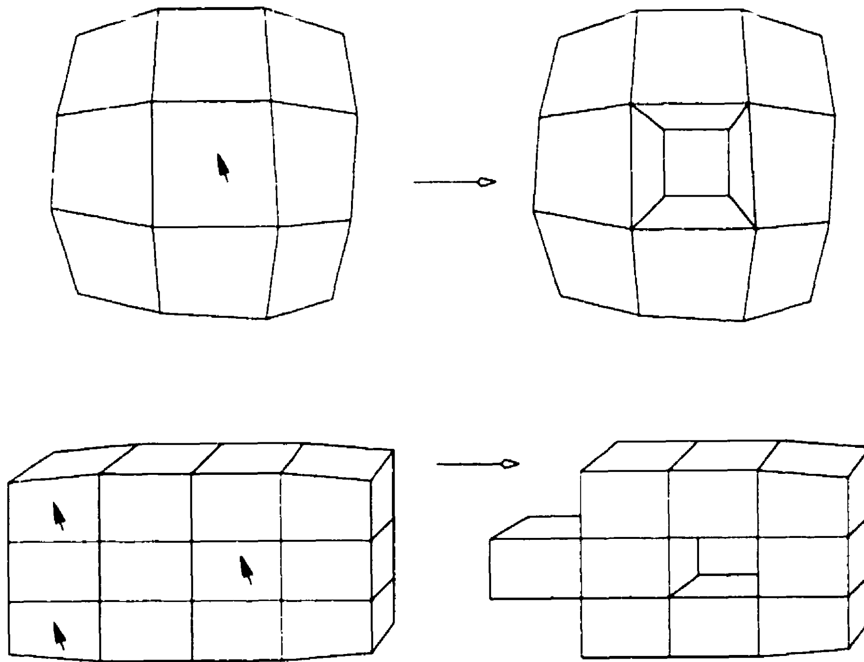

Figure 2.5: Four or fewer faces become visible when a cuboid is removed.

Because each cuboid already knows its four neighbors, examining the four or fewer faces that are candidates for becoming visible is only a matter of checking the following two conditions:

1. the face belongs to a non-*hidden*, existing neighbor cuboid, and

2. the face has a normal with $z > 0$.

Access to these neighbor cuboids (and hence, their *visible* faces) is directly obtained through the *tpC*, *ltC*, *bmC*, *rtC* references, initialized in the FORM-LAYER algorithm. Once a cuboid is *hidden*, it is disregarded at subsequent renderings or printings.

2.4 Implementation Issues

To display the scene whose objects are stored in the CP array, the RC cells in the CP array are traversed in an orderly fashion to test the contents of each cell. A non-empty cell will have the *visible* faces of its non-*hidden* cuboids rendered, with each pixel tested against the contents of the z-buffer: an $r \times c$ array named *zBuffer*, whose elements are pre-set to $-\infty$. The following algorithm shows some of the main operations involved when displaying the objects. With simple adjustments to the initial and end values of the for loops, DISPLAY-OBJECTS may be used to render smaller rectangular areas (e.g., only at locations where cuboids are set *hidden*).

Algorithm 2.4.1 The DISPLAY-OBJECTS algorithm.

for $i \leftarrow 1$ to R
  for $j \leftarrow 1$ to C
    if the (i, j)'th cell in the CP table is not empty, render *visible* faces of all non-*hidden* cuboids referenced in the list headed by the (i, j)'th cell, according to their *clr-code*.

By using the cuboid data structure introduced in Section 4.1, each cuboid is stored only once, but may have four or fewer references from neighboring cuboids pointing to it. In addition, the mechanism of pointers allows updates in the properties of any

cuboid (e.g., a cuboid becomes *hidden*, or the visibility flag of some faces of a cuboid changes status) to be directly seen by referencing cuboids. This is possible because pointers contain the physical address of a *cuboid*, rather than to the value of the fields, which may be updated several times.

When rendering the *visible* faces of non-*hidden* cuboids, the z-buffer algorithm is used. When setting a cuboid to be *hidden*, there is a potential for four or fewer faces of the neighboring cuboids to become *visible* (*rt* face of *ltC* cuboid, *tp* of *bmC*, *lt* of *rtC*, and *bm* of *tpC*). However, not all of the area defined by a face may become visible, some parts of the face may still be obscured by neighboring cuboids that are still part of the scene (the lower object in Figure 2.5 is an example of faces that become partially visible). By using the z-buffer technique, only pixels corresponding to non-obscured (x, y) points on visible faces are rendered.

Aside from the aforementioned faces of the neighbor cuboids, eight additional faces should be checked for rendering. When selecting a cuboid to be not displayed, it-s visible face(s) may have been partially obscuring the area dominated by a face of a diagonal-neighbor cuboid (diagonal neighbors are top-right, top-left, bottom-right, or bottom-left cuboids) where the direct neighbor cuboid is *hidden*. The faces that should be examined for redisplay are: *bm* and *tp* faces of the top-left and bottom-left diagonal neighbor cuboids, respectively, if the *ltC* cuboid is *hidden* (if *ltC* is not *hidden*, *bm* and *tp* faces of top-left and bottom-left cuboids, respectively, are not visible, and hence are disregarded). Figure 2.6 illustrates the above mentioned case, where the lightly-shaded faces represent the diagonal neighbors of the last removed cuboid.

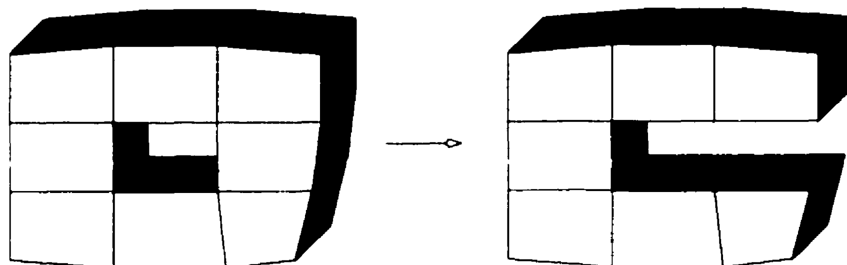

Figure 2.6: The removal of some cuboids causes some faces of diagonal neighbors to be re-rendered.

The other six faces that should be considered for redisplay are: *rt* and *lt* faces of bottom-left and bottom-right cuboids, respectively, if *bmC* is *hidden*; *tp* and *bm* faces of bottom-right and top-right cuboids, respectively, if *rtC* is *hidden*; and *lt* and *rt* faces of top-right and top-left cuboids, respectively, if *tpC* is *hidden*. From these faces, only faces whose normals have a positive z-component are rendered. The following algorithm, named HIDE-CUBOIDS describes the process involved when *hiding* cuboids, together with the aforementioned exceptions.

Occasionally, the position of the virtual hole may not be in the position intended by the user, because the accessed CP table cell points to different cuboids, or no visual effects take place because the accessed CP table cell is empty. This is due to the $(\dot{i}, \dot{j})$ pair which is derived from the user's input on the pixel canvas screen. In this case, the user may try to change the position of their input slightly to obtain the desired results.

If the position of a virtual hole is not satisfactory, the visual results shown in HIDE-CUBOIDS may be interactively "reversed", by "un-hiding" the last *hidden* cuboids on

Algorithm 2.4.2 The HIDE-CUBOIDS algorithm.

- Access the $(i, j)'th$ CP table cell, corresponding to the user's pointing device input as shown at end of Subsection 2.3.2.
- if cell is not empty, traverse list until first non-*hidden* cuboid is reached, set its *hidden* flag to "TRUE".
- Render all faces of cuboid with canvas background texture, updating *zBuffer* accordingly with $-\infty$.
- Render *visible bm, rt, tp, lt* faces of *tpC, ltC, bmC, rtC*, non-*hidden* cuboids, respectively.
- Render *visible*, side faces of non-*hidden*, diagonal neighbor cuboids if corresponding, neighbor cuboids are *hidden*.
- Render *all* cuboids that are pointed to through the *next* link. In addition, render non-*hidden* cuboids in immediate surrounding cells to cover possible visual holes.

the list in the current CP table cell, and rendering back the newly non-*hidden* cuboids.

2.5 Sample Applications

In the two examples shown below in figures 2.7(a) and 2.8(a), each shows an object initially without virtual holes, followed by a series of figures showing virtual holes in various positions.

Color coding was selected to easily distinguish among each object layers, and among the front, back and sides of each object. Surface smoothing is *intentionally* not applied here, so as to show the constituting cuboids clearly.

The pixel canvas used in these figures is 600 × 600 pixels. The square side L is 10

pixels, hence, both the cell canvas and the CP table (see Figure 2.4) are of dimension 60×60. The square side length L should be chosen to be less than the average edge length of the displayed cuboids, so that more cells are used to store the various cuboids. This has the advantage of fair distribution of cuboids over cells, thereby minimizing the length of the lists pointed to by the *next* link that have to be traversed to render the first non-*hidden* cuboid.

2.5.1 Example 1

This example illustrates the application of the proposed algorithms on data reconstructed from actual human cross section images (available at the World Wide Web address given in [13]). Twenty-seven odd-numbered slices from Slice 197 to Slice 249 were used to reconstruct the skin and the bone of the left thigh [‡]. For each cross section, 36 points were sampled at specific locations on the exterior and interior contour lines of the skin, and 24 points were sampled for the bone on each contour. Corresponding points on each cross section were connected to form patches, and corresponding patches from all pairs of consecutive cross sections (i.e., 197–199, 199–201, etc.) were connected to form cuboids. Figure 2.7(a) shows a top view of the reconstructed thigh. Figure 2.7(b) shows a side veiw of the thigh. Subsequent figures 2.7(c) to 2.7(f) represent a series of virtual holes constructed on the transformed thigh, using various user specified virtual hole radii (VHR).

In this prototype implementation, a virtual hole of "radius" m cuboids is created by

[‡] Individual slices are accessed as follows: http://www.mir.wustl.edu/visible_human/slice_*nnn*.GIF, where $nnn = 000, 001, \ldots, 374$.

the removal of a central cuboid (selected), and surrounding cuboids for a total of $(2m+1) \times (2m+1)$. Setting VHR=0 causes only the selected cuboid to be not displayed. Setting VHR=1 causes the selected cuboid and its eight neihbors to be not displayed, and so on.

It is worth noting here that accessing these $(2m+1)^2 - 1$ surrounding cuboids is direct (only one operation to access each cuboid), as a result of the cuboid neighbour links initialized earlier in the FORM-LAYER algorithm. Therefore, searching is not required.

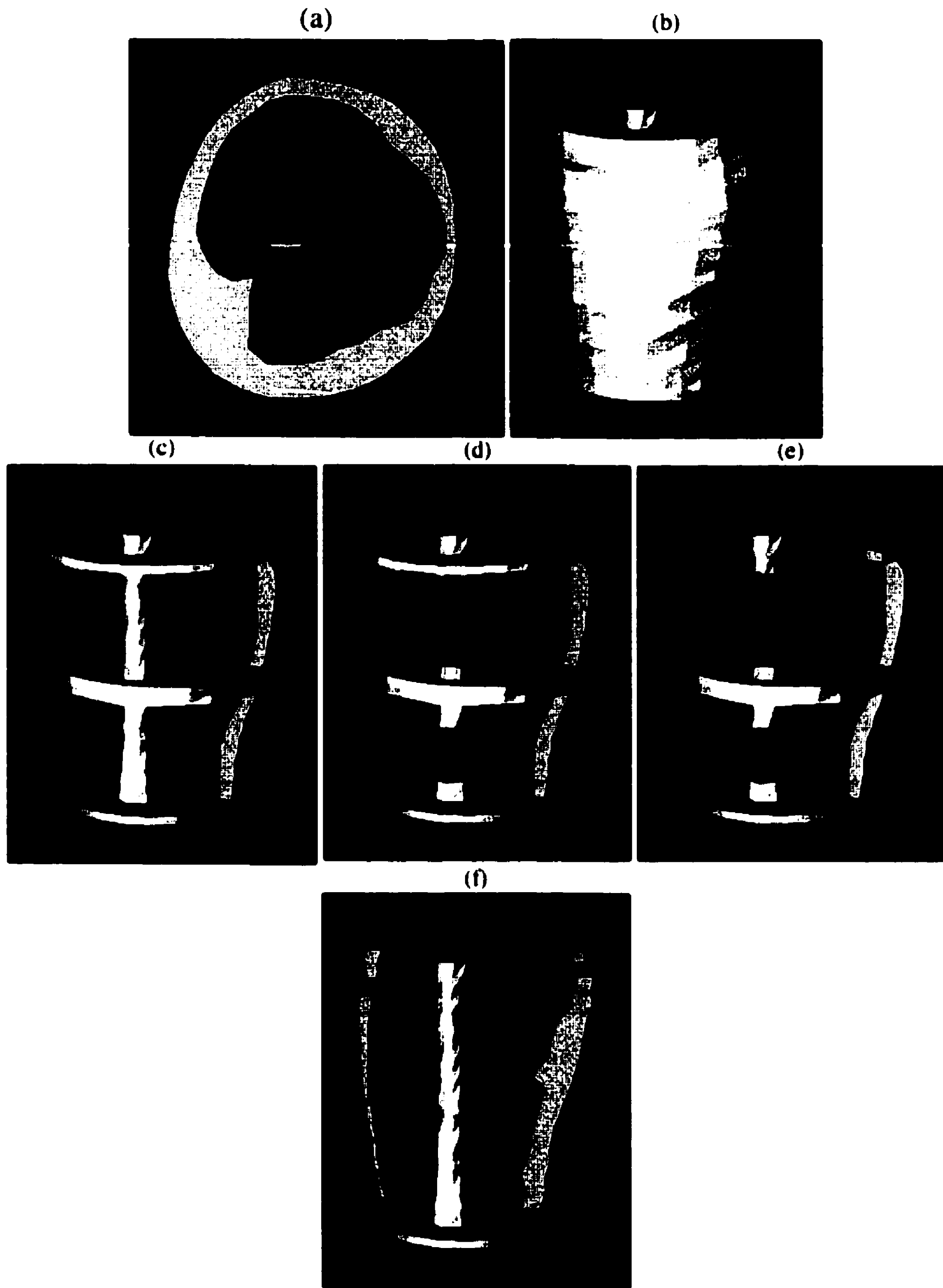

Figure 2.7: Successive virtual holes on reconstructed human thigh.

2.5.2 Example 2

In Figure 2.8(a), complementary surfaces are generated by the method described in Section 4.1 to form several sheets of human arm, elbow, and forearm, all of which are approximated with B-spline surfaces. The model is made of three distinct objects, the skin, the biceps (muscle), and the humerus (bone), as illustrated in [14]. This example consists of 6 figures (2.8(a) to 2.8(f)). Each image is a succession to the one preceding it revealing several constructed virtual holes. In Figure 2.8(f), only cuboids belonging to the skin layer are removed while leaving the rest of the objects in the scene intact; this is both possible and simple under the proposed data structure, since it is only a matter of following cuboid neighbour links and stopping when every cuboid becomes *hidden*.

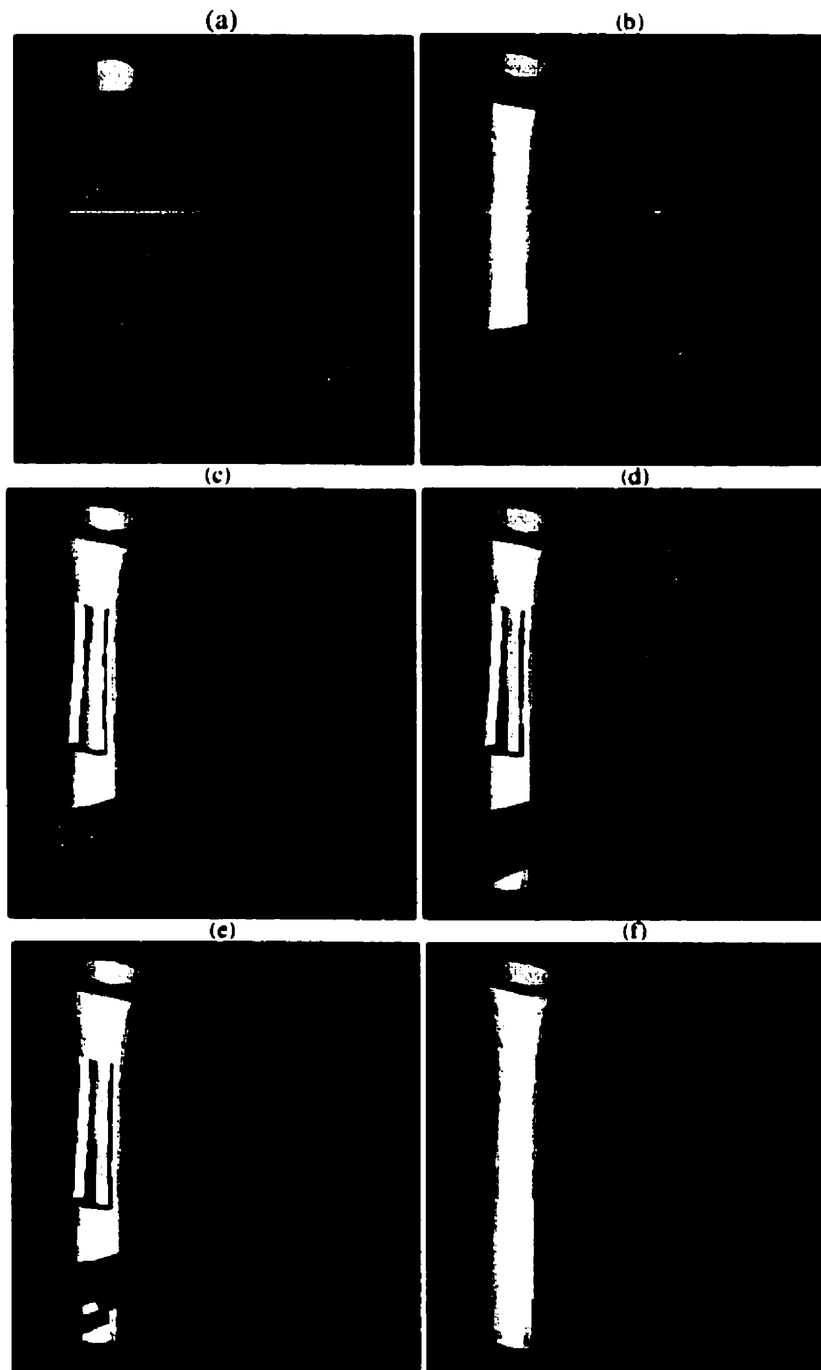

Figure 2.8: Successive virtual holes on artificial arm.

2.6 Summary

Algorithms and data structure for transforming a 3D surface of patches into a layer of cuboids, and for making virtual holes on layered objects to view interiors were introduced. The pixel canvas on which objects are displayed is subdivided into a rectangular grid that is directly mapped onto a 2D CP (cuboid pointer) array.

Determining the location of the cuboids belonging to the *first* virtual hole requires a single access (which involves computing $(\dot{i}, \dot{j})$, as described in Subsection 2.3.2); namely, to access the CP array cell that is mapped to, from the selected location of the pointing device on the pixel canvas. The cell has direct references to surrounding cuboids (which have references to their own surrounding cuboids, needed when the radius of the virtual hole is greater than 0) to be removed from the scene at successive renditions. Determining the location of cuboids that belong to an n'*th* interior virtual hole requires n accesses:

1. accessing the cell mapped to by the user's specified region on the pixel canvas, and
2. following $n-1$ ***next*** links from the *first* ***hidden*** cuboid to arrive at the cuboid(s) to be ***hidden***.

This implies that, regardless of the number of objects rendered in a scene (where each object may be composed of thousands of cuboids), finding the set of cuboids to be removed takes only m accesses in the worst case, where m is the number of layers that

have cuboids with the same $(\dot{x}, \dot{y})$ value (derived from (x, y) in a cuboid's *centroid* as shown in Section 4.2) where the user is making a virtual hole. As an example, consider looking at a room composed of six 50×50 cuboid objects: four walls, a floor and a ceiling, for a total of $15,000$ cuboids. Making the first virtual hole takes one step, to calculate the $(\dot{i}, \dot{j})$ to access the corresponding CP table cell, making a second virtual hole behind the first one takes two steps: one step to arrive at the cuboid(s) of the already constructed virtual hole, another to follow the *next* link to arrive at the cuboids of the first (and last in this example) hidden layer. So it takes a total of three steps to display the scene background behind the room where the two virtual holes are made.

The algorithms and data structures of this chapter focused on quadrilateral-patch surfaces, which allowed the exploitation of the regular structure inherent in surfaces representable by gridded data (i.e., data that is input in the form of an $n \times m$ matrix). But how can more general surface topologies be addressed in this case? In triangulated surfaces, for example, triangular patches (or *prismoids* for a thickened surface) have zero or one neighbour on each side of the triangle, but can have an unpredictable number of neighbour triangles on their vertices; any number ≥ 0 is typically acceptable. Therefore, the *up*, *down*, *left*, and *right* neighbour relation that is readily available in gridded data to each cuboid on data input cannot be applied to triangular surfaces. Nevertheless, each triangle (or prismoid) must still be aware of which are its surrounding triangles for interactive, real-time visualization requirements. This requires an involved process for pre-processing data for each object, in which the relationship between each vertex and its surrounding triangles is established, such that "searching"

is avoided. This is addressed in the following chapters.

Chapter 3

Storage Organization of Polyhedral Surfaces

In the previous chapter, the idea of constructing *virtual holes* on "thickened" quadrilateral-patch (cuboids) surfaces was studied. Virtual holes allowed a see-through of internal parts of the model by *segmenting off* a portion of the surface while leaving the rest of the surface intact and previously hidden parts visible. That work addressed only quadrilateral-patch surfaces, to exploit the regularity of the structure available given an $(n \times m)$ surface data points in matrix form.

In this chapter, a more general data structure to handle surfaces of patches with different number of sides are developed. The purpose of such a data structure is to allow various operations that may be interactively applied on the displayed surfaces such as:

- Interactive 3D point location (Chapter 4 & Chapter 5).
- Interactive surface segmentation (Chapter 6).

- Interactive shape modification (Chapter 6).

The rest of this chapter is organized as follows. Section 3.1 discusses related work regarding efficient storage of polyhedral models. Section 3.2 proposes a data structure that is designed to facilitate the application of the aforementioned operations on polyhedral surfaces in general. An algorithm for converting the representation of input data into a format compliant with the proposed data structure is also provided.

3.1 Storage of Polyhedral Models

The data structure chosen here for storing surface polygons is based on the *pointers-to-an-edge-list* (PEL) data structure, as described in [1, 17]. In PEL, given $\mathbb{V}$ vertices, $\mathbb{E}$ edges, and $\mathbb{P}$ polygons on a surface, then vertices V, edges E, and polygons P are defined such that each vertex $V_i \in V$ contains its space coordinates; each edge $E_j \in E$ references its two end vertices (by their index number), and the one or two polygons (by their index number) to which it belongs; and each polygon $P_k \in P$ references the set of its defining edges (by their index number) as follows:

$$
\begin{array}{rclll}
V_i & = & \mathbf{v}_i = \{x_i, y_i, z_i\}, & & i = 1, 2, \ldots, \mathbb{V}, \\
E_j & = & (V_{E_{j_{v1}}}, V_{E_{j_{v2}}}, P_{E_{j_{p1}}}, P_{E_{j_{p2}}}), & & j = 1, 2, \ldots, \mathbb{E}, \\
P_k & = & (E_{P_{k_1}}, E_{P_{k_2}}, \ldots, E_{P_{k_n}}), & & k = 1, 2, \ldots, \mathbb{P},
\end{array}
$$

where $n = 3, 4, \ldots$ is the number of sides in polygon P_k.

The PEL representation of polygonal models has been adopted in practice because every vertex is stored only once, thereby saving storage space. Also, since polygons

are defined in terms of edges, which are in turn defined in terms of vertices, changes in the fields of vertices such as the coordinates are immediate and global to all polygons and edges sharing such vertices. However, for purposes of shape modification, where information about a vertex's surrounding edges and/or polygons should be directly available, it is not easy under PEL representation, for example, to determine the set of edges that are incident to a vertex.

The *winged-edge* data structure (WEDS) [17, 55] uses PEL as a basis for representing polygons, but makes it possible to determine in constant time the set of vertices and faces (polygons) that are associated with an edge. This is so because pointers to vertices and faces are provided as cross-reference information—as data fields in the edges they are associated with. The vertices V, edges E, and polygons P in a surface represented by WEDS is thus defined as follows:

$$
\begin{aligned}
V_i &= \mathbf{v}_i = \{x_i, y_i, z_i\}, \\
E_j &= (V_{E_{j_{v1}}}, V_{E_{j_{v2}}}, P_{E_{j_{p1}}}, P_{E_{j_{p2}}}, E_{j1}, E_{j2}, E_{j3}, E_{j4}), \\
P_k &= (E_{P_{k_1}}, E_{P_{k_2}}, \ldots, E_{P_{k_n}}).
\end{aligned}
$$

where i, j, k, and n are as defined for the PEL structure. In the above structure, each edge E_j has eight fields: references to the two defining end vertices; references to the two polygons sharing the edge; and references to four other edges E_{j_m}, $m = 1, 2, 3, 4$, where each two of these edges emanate from one of its two referenced end vertices, and are associated with one of the two faces that share E_j. An algorithm for converting hierarchical geometric structure to WEDS representation is found in [8].

However, it is not a straightforward process to employ WEDS for applications such as performing tweaking operations on an object by interactively dragging its vertices for local editing of the object shape [17, 41], or for attempting to determine the normal at a vertex, e.g., for shading purposes, by using data of all surrounding polygons. This is because the set of surrounding polygons are not available from vertices, so some indirect maneuvering is needed through incident edges and that would adversely affect user-interaction performance time. In the following section, these shortcomings are remedied by a convenient data structure that is based on PEL, but that permits immediate access to surrounding polygons as one of several features is described; a slight variation of this data structure was first introduced in [30].

3.2 Triangular-Loop Data Structure

The triangle loop data structure (TLDS) is based on the PEL data structure described above, but is customized to support rapid object segmentation and shape modification by providing immediate access to neighbor-polygons of interactively selected vertices. These selected vertices may be freely re-located for tweaking operations or other design purposes [41], in which the surrounding polygons are to be determined to edit their shape. This is accomplished by providing suitable cross-reference information in the vertex fields during pre-processing of input polygons.

For each vertex in TLDS, the number of surrounding polygons S, and pointers to them are made available in the vertex structure. Therefore, given $\mathbb{V}$ vertices, $\mathbb{E}$ edges, and

$\mathbb{P}$ polygons on a surface, then vertices V, edges E, and polygons P are defined as follows:

$$
\begin{aligned}
V_i &= \{vID_i, \mathbf{v}_i, S_i, \{pID_{i_1}, pID_{i_2}, \ldots, pID_{i_{S_i}}\}\}, \\
E_j &= \{eID_j, (vID_{E_{j_1}}, vID_{E_{j_2}})\}, \\
P_k &= \{pID_k, (\pm eID_{P_{k_1}}, \pm eID_{P_{k_2}}, \ldots, \pm eID_{P_{k_n}})\},
\end{aligned}
$$

where i, j, k, and n are as defined in Section 3.1 for the PEL structure. A right-hand coordinate system is depicted: x-, y-, and z-axis values increase to the right of, top of, and towards the viewer, respectively. For rendering purposes, each edge identifier in polygon P_k is preceded by a "+" or a "−" sign to indicate the direction edges have to be read in (e.g., "+" implies counter-clockwise). Figure 3.1 illustrates an object example with some representative definitions.

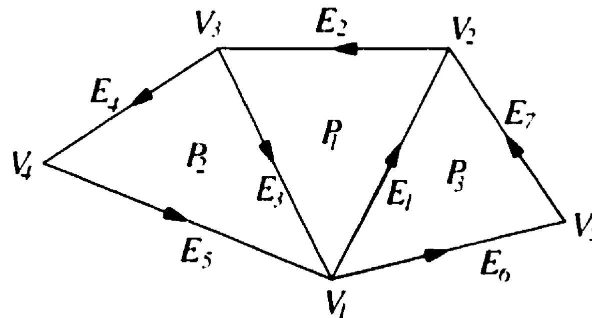

Figure 3.1: A simple model made of three polygons. In this TLDS representation, $V_1 = \{1, \{x_1, y_1, z_1\}, 3, \{1, 2, 3\}\}$, $E_2 = \{2, (2, 3)\}$, and $P_3 = \{3, (7, -1, 6)\}$.

Pre-processing is done only once for any given set of polygons. An algorithm to create a TLDS representation of the input polyhedron data follows [33]. Each polygon is

initially stored in an array $\mathcal{P}_{ij}$, where $i = 1, 2, \ldots, \mathbb{P}$, and $j = 1, 2, \ldots, n_i$, n being the number of vertices (which is the same as the number of edges) of the *ith* polygon, followed by n_i space coordinates $\mathbf{v}_j = \{x_j, y_j, z_j\}$ for each of these vertices.

In this algorithm (CREATE-TLDS-REPRESENTATION), the function GET-VID takes as input a vertex $\mathbf{v}_j$ and returns a value r—the *vID* number of the *jth* vertex $\mathbf{v}_j$ in polygon $\mathcal{P}_i$. The value r is either the next available sequential number that also indicates the number of processed vertices ($\mathbb{V} + 1$) thus far, in which case the current number of vertices $\mathbb{V}$ is updated by one; or any other number $\in [1, \mathbb{V}]$. The latter case implies that vertex $\mathbf{v}_j$ is already assigned a *vID* number, and is being re-visited as a result of processing a new polygon that also shares $\mathbf{v}_j$. One way of achieving this functionality of assigning *vID*s appropriately to processed vertices by the GET-VID function is by building an ordered *binary-tree* that uses the value x_j in $\mathbf{v}_j$ to determine the appropriate *node* for storing $\mathbf{v}_j$, but whose nodes are also ordered *linked-lists*, to handle the case of the existence of more than one $\mathbf{v}$ in the same y–z-plane.

Similarly, the function GET-EID takes as input the *vID*s of two consecutive vertices in a polygon from $\mathcal{P}_i$ and returns s—the *eID* number of the connecting edge. This value of s is in $[1, \mathbb{E}]$ if the edge has been seen before, or $s \leftarrow \mathbb{E} + 1$ if E is a new edge, in which case $\mathbb{E}$ is also updated by one. Note that GET-EID would always return the same value for any two consecutive *vID*s, irrespective of their order, so both GET-EID(vID_j,vID_{j-1}) and GET-EID(vID_{j-1},vID_j) would return the value s. However, when an edge is being re-visited (each edge is part of at most two polygons), the input order of the two *vID*s is reversed, an old s value is generated, and P_i's eID_j is assigned

the value $-s$, to maintain a consistent cyclic order in reading the edges, as illustrated above in Figure 3.1.

In an actual implementation, Algorithm 3.2.1 may be coded more efficiently than p-resented here, however, readability is chosen over efficiency for purposes of easy following.

Algorithm 3.2.1 The CREATE-TLDS-REPRESENTATION algorithm.

```
// Input: object polygons in hierarchical representation
// Output: a TLDS representation of the input object

𝕍 ← 0, 𝔼 ← 0, ℙ ← 0

while there are more polygons to process do
  ℙ ← ℙ + 1
  i ← ℙ                                          // A new polygon with pID = i

  for j ← 1 to (n_i + 1)                         // n_i = # edges/vertices in i'th polygon

    if (j ≤ n_i) then ........................................// Prepare V's
      v ← P_ij                                   // Get the j'th V from i'th polygon
      r_j ← GET-VID(v)

      if (r_j = 𝕍 + 1) then                      // V is a new vertex
        𝕍 ← 𝕍 + 1
        S_rj ← 1                 // Initialize # of surrounding polygons of V_rj to 1
        surrPolySet_rj ← i       // Initialize set of surrounding polygons of V_rj
      else                       // Update data of shared vertex V_rj
        S_rj ← S_rj + 1
        surrPolySet_rj ← surrPolySet_rj ∪ i

    if (j > 1) then ..........................................// Prepare E's
      j ← (j − 1) mod n_i + 1    // In case edge joins last & first vertices
      s ← GET-EID(r_{j−1}, r_j)

      if (s = 𝔼 + 1) then                        // A new edge
        𝔼 ← 𝔼 + 1
        E_{j−1}'s vID_{E_v1} ← r_{j−1}           // Set identifying vertices of new edge
        E_{j−1}'s vID_{E_v2} ← r_j

      if (s = 𝔼 + 1) then                        // Add new edge ID to polygon i
        P_i's eID_{P_{i,j−1}} ← s ..................// Identify edges of i'th P
      else                       // Edge already processed & assigned an eID
        P_i's eID_{P_{i,j−1}} ← −s   // Switch its traversal direction in polygon i
```

The identifier fields ***vID***, ***eID***, and ***pID*** are assigned sequentially for each newly created V_i, E_j, and P_k, respectively, as shown in the above algorithm; in addition to bookkeeping purposes during implementation time, their main use in TLDS or other similar structures will also become clear when integration of modified polyhedral segments is discussed in Chapter 6.

Note from the above definition of TLDS that once mechanisms that allow the user to select vertices, edges or polygons of displayed objects are available, it is a straightforward $O(1)$ process to access surrounding polygons, end vertices, or defining edges, respectively, from which the *triangular-loop* data structure gets its name. This feature is important when union operations that are needed in determining the set polygons within a bounding volume are performed, as will be discussed in Chapter 6. Although only vertices are discussed for selection here, work in [31] (also discussed in Chapter 5) exploits the same data structure to interactively select edges in addition to vertices. Figure 2 is an illustration of the structure.

Furthermore, observe that storage requirements for TLDS is comparable to that of WEDS. Given $\mathbb{V}$ vertices, $\mathbb{E}$ edges, and $\mathbb{P}$ polygons on a surface, and assuming that the identifier fields ***vID*** in V, ***eID*** in E, and ***pID*** in P are present in all structures for accessing purposes, the storage requirements for PEL, WEDS, and TLDS may be determined as follows.

Let ζ be the word-size needed for storing an **integer** number, and ξ be the word-size needed for storing a **real** number. For example, to calculate the storage requirement

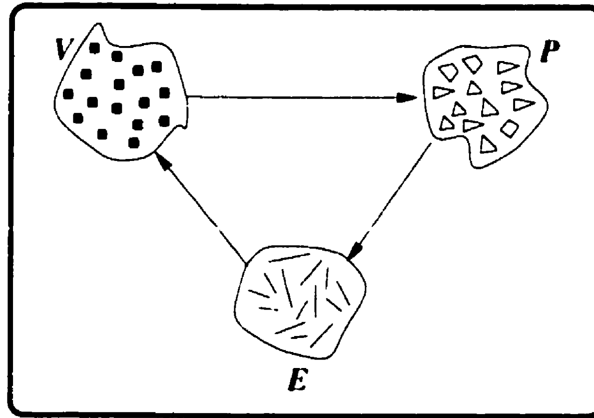

Figure 3.2: A visual view of TLDS: Selected vertices give access to surrounding polygons, which give access to the defining edges some of whose endpoints are the selected vertices.

of a vertex in TLDS, recall that a vertex is defined by

$$V_i = \{vID_i, \mathbf{v}_i, S_i, \{pID_{i_1}, pID_{i_2}, \ldots, pID_{i_{S_i}}\}\}.$$

So, each V_i requires one ς for storing vID_i, 3ξ for $\mathbf{v}_i$, and $(S + 1)\varsigma$ for the S field (the average number of a vertex's surrounding polygons) and the following polygon identifiers. Similarly, each edge would occupy 3ς of storage space; and each polygon would occupy $(1 + n)\varsigma$ of space. The following table illustrates total storage requirements for the three discussed structures:

	$\mathbb{V}$	$\mathbb{E}$	$\mathbb{P}$	Totals
PEL	$3\xi + \zeta$	3ζ	$(n+1)\zeta$	$3\xi + (n+5)\zeta$
WEDS	$3\xi + \zeta$	9ζ	$(n+1)\zeta$	$3\xi + (n+11)\zeta$
TLDS	$3\xi + (2+S)\zeta$	3ζ	$(n+1)\zeta$	$3\xi + (n+S+6)\zeta$

Table 3.1: Amount of storage required by PEL, WEDS, and TLDS.

Assuming a fixed average number of S surrounding polygons per vertex, it can be seen that the storage requirements of WEDS and TLDS are similar when $S = 5$.

The next chapter presents convenient access mechanisms of data of polyhedral surfaces that are stored in TLDS representation.

Chapter 4

Interactive Manipulation of Polyhedral Surfaces

Assuming that data points are in TLDS representation, the process of interactively locating 3D vertices for purposes of shape control and modification is a direct two-fold process:

1. Storing the points in a two-dimensional array whose dimensions are determined by the display resolution, and

2. Accessing those points as required, directly and without searching, in constant-time processing performance.

These two issues are as follows.

4.1 Storage Organization of Vertices

In order to interactively locate a vertex in 3D space to modify the status of some of its associated fields, perform tweaking operations on it, or to successfully segment a set of vertices that form a sub-polyhedron for local editing, each vertex in the V array is stored in a two-dimensional ($R \times C$) vertex pointer (VP) table, where R and C are the numbers of rows and columns of pixels in the display area, respectively. The array is at the resolution of the display area.

Initially, each cell in the VP array is pre-set to point to a null linked-list. As the objects are being prepared for display through a transformation specified by the user, each TLDS data point V_i is stored in the VP table depending on two factors:

1. **The (x_i, y_i) pair of $\mathbf{v}_i$ in V_i.** This pair specifies a corresponding (r_i, c_i) location in the VP table, where $r = 1, 2, \ldots, R$, and $c = 1, 2, \ldots, C$, are the integer parts of x_i and y_i, respectively. Note that each of the ($R \times C$) cells in the VP table has a *one-to-one* mapping to one of the pixels on the ($R \times C$) display area through the (r_i, c_i) pair.

2. **The z_i value of $\mathbf{v}_i$ in V_i.** If the VP array cell (r_i, c_i) is empty, then it is initialized to reference the corresponding V_i; otherwise, an appropriate sorting algorithm (such as the *insertion sort* algorithm [37]) uses the value z_i to appropriately position V_i in the linked-list of vertices referenced by the cell (r_i, c_i) in descending order. That is, closest vertices to the viewer (vertices with greater z values) are stored first for immediate access.

In this regard, the VP array is used in the same fashion as the z-buffer array in the *depth buffer* algorithm that is used in conjunction with *hidden surface removal* algorithms [24], but on a vertex rather than pixel level. Furthermore, it includes the functionality of storing all vertices that share the same (r, c) coordinates, which is provided by the linked-lists, as opposed to just a scalar value of the closest point on the surface displayed at (r, c).

These features, as will be apparent in the following chapters, have significant contributions to ease and simplicity of performing operations such as surface segmentation and shape modification.

4.2 Interactive Access of Vertices

As objects are displayed on the display area in edit mode (e.g., by having all vertices highlighted for ease of selection by the user), the user first selects the vertex required for shape modification (i.e., for purposes of performing tweaking operations [17]). This may be accomplished interactively by placing the cursor on the highlighted vertex and selecting it. Selection makes an (r_i, c_i) pair available. Such a pair is used to access the (r_i, c_i) cell of the VP table, which is pre-loaded with a linked-list of at least one element; the first element refers to the selected V_i vertex.

Now, recall that $V_i = \{vID_i, \mathbf{v}_i, S_i, \{pID_{i_1}, pID_{i_2}, \ldots, pID_{i_{S_i}}\}\}$, implying that once a vertex V_i is selected through the (r_i, c_i) integer coordinates (specified indirectly by the user through a pointing device), constant time access is granted to the following information:

- The reference number, vID_i, of V_i,
- the spatial fields of $\mathbf{v_i} = \{x_i, y_i, z_i\}$,
- the number of surrounding polygons S_i, together with references to all the S_i surrounding polygons in the P array,
- all the edges that share V_i as an end point, and those that encircle V_i, available indirectly from the E array through the obtained S_i polygon references (recall that polygons are defined in terms of edges in the E array).

It is now possible to perform shape manipulation operations as illustrated in the next two chapters.

Chapter 5

Interactive 3D Point Location & Applications

In this chapter, a method for rapid, interactive location of 3D points is introduced. The methods are applicable to data that are TLDS (triangular-loop data structure) compliant, with accessing mechanisms detailed in the previous chapter. As an example, it is shown how such an approach is useful in applications such as *surface smoothing* [30].

As indicated in Chapter 1, the display of 3D *modeled* objects on a computer screen is usually the final phase of a long process of data manipulation. A scenario that is relevant to the application discussed below involves obtaining sampled data points from a physical object by laser scanning or a coordinate measuring machine [42]. A crude wire-frame triangulated model is then produced from the obtained sampled data using techniques such as those shown in [15]. Next, refinement algorithms are used to fit triangular patches to the produced triangulated model to generate a refined triangular model consisting of many smaller triangles. A common method of refinement is to

first fit surface patches to the crude triangular patches [22, 36]. Smaller triangles are then generated from the fitted surface. These refinement algorithms are applied uniformly on all sampled points with the intention of preserving positional and tangent plane continuity across the triangular patches to obtain visual smoothness.

Surface smoothing is important in applications such as rapid prototyping, e.g., the technology of layered manufacturing, which uses stereo-lithography apparatus (SLA) for producing physical models of products or parts in a short time. The generated refined triangles are input into the SLA to build a plastic model of the triangulated object by curing liquid polymer into a solid with an ultra violet (UV) laser.

Because particular features such as cusps or tangent discontinuity are sometimes inherent in the object being modeled, it is not always desirable that the smoothing process be applied uniformly to all vertices and edges composing the model. Smoothing can have the effect of producing artificial results that are not intended on the produced physical model. It would be useful to have an interactive and immediate way of locating vertices and edges that define such particular features in order to exclude them from the smoothing process. A manual search of vertices and edges that define the special features of an object in data files is not practical, for each of the displayed models may be composed of several hundred vertices.

Once the location of special features data are determined in the data file, they are *marked* by having their associated flags set accordingly. Marked data (vertices and edges of special feature regions) are excluded from the smoothing process so as to preserve special features on the refined triangulated objects. The technique for mark-

ing vertices and edges in a crude triangulated model for special feature preservation is described here. It is devised to complement smoothing algorithms in general, but with reference to the refinement algorithm for smoothing as described in [53, 54].

Generalization of the 3D point location algorithm to tensor-product quadrilateral-patch surfaces, to direct the refinement process as in [18], is possible. One way of achieving this is to first triangulate the parametric surface by adding a bisecting diagonal through each of the quadrilateral patches, and then re-arranging the representation of the triangles into a special data structure that is described below.

The remainder of the chapter is organized as follows. A convenient modification of the TLDS data structure for storing sampled data points to allow for interactive 3D point location is described in Section 5.1. The actual process of locating and marking 3D points and edges for marking is described in Section 5.2. Section 5.3 presents mechanisms that facilitate user interaction with the displayed objects to simplify the selection process. Some implementation issues are discussed in Section 5.4. The chapter is concluded with some visual examples and concluding remarks.

5.1 A Modified TLDS

A wire-frame triangulated model of an object is defined by a number of triangles in space whose vertices are traversed in a consistent order (e.g., counter-clockwise). The display of such a model involves reading from a file all the triangles, three vertices at a time, and then calculating their normals for shading purposes before rendering. The

triangles as input, i.e., sets of three vertices, are not in a form that is suitable for processing. For example, if a vertex $v = \{x, y, z\}$ is a common vertex in S surrounding triangles, then an update in the status of v requires an update in all the S surrounding triangles that share v. To avoid the need to perform multiple updates, and to access the selected (*marked*) vertices and edges directly, the CREATE-TLDS-REPRESENTATION algorithm (introduced in Chapter 3) is applied on the input triangles to create a TLD-S representation of the data. The TLDS representation in this case is convenient for interactively locating and modifying the statuses (e.g., the associated flags) of 3D vertices and edges without searching. See Figure 3.1 for a simple model that is in TLDS representation

Recall from Chapter 3 the TLDS representation of vertices, edges, and polygons:

$$
\begin{aligned}
V_i &= \{vID_i, v_i, S_i, \{pID_{i_1}, pID_{i_2}, \ldots, pID_{i_{S_i}}\}\}, \\
E_j &= \{eID_j, (vID_{E_{j_1}}, vID_{E_{j_2}})\}, \\
P_k &= \{pID_k, (\pm eID_{P_{k_1}}, \pm eID_{P_{k_2}}, \ldots, \pm eID_{P_{k_n}})\}.
\end{aligned}
$$

To preserve special features in this surface-smoothing application, some of the vertices and/or edges should not undergo the smoothing process. These vertices and/or edges are interactively *marked* to preserve the object shape after smoothing. When data are marked, the smoothing algorithm will not smooth their surrounding regions.

To implement this functionality of *marking*, a binary flag ($vSmooth_i$) is added to the V_i structure to indicate whether or not the vertex belongs to a special feature region. The added flag, $vSmooth_i$, is initially pre-set to "TRUE", to indicate that all vertices

should undergo the smoothing process. The value of $\mathit{vSmooth}_i$ may be changed to "FALSE" by *marking* the associated vertex interactively.

Similarly for edges, a flag ($\mathit{eSmooth}_j$) is added to the E_j structure in TLDS. An edge is shared between at most two polygons. The default setting of the $\mathit{eSmooth}_j$ flag is to "TRUE"; Setting the flag to "FALSE" indicates that the boundary between the two polygons sharing E_j should not be smoothed, e.g., to preserve only C^0 continuity [16].

Because all the model polygons are triangles, the definition of $P_k, k = 1, 2, \ldots, \mathbb{P}$ is also efficiently modified such that the number of defining edges is limited to three. A modified TLDS for this surface-smoothing application may therefore be described as follows:

$$
\begin{aligned}
V_i &= \{\mathit{vID}_i, \mathbf{v}_i, \mathit{vSmooth}_i, S_i, \{\mathit{pID}_{i_1}, \mathit{pID}_{i_2}, \ldots, \mathit{pID}_{i_{S_i}}\}\}, \\
E_j &= \{\mathit{eID}_j, \mathit{eSmooth}_j, (\mathit{vID}_{E_{j_1}}, \mathit{vID}_{E_{j_2}})\}, \\
P_k &= \{\mathit{pID}_k, (\pm\mathit{eID}_{P_{k_1}}, \pm\mathit{eID}_{P_{k_2}}, \pm\mathit{eID}_{P_{k_3}})\}.
\end{aligned}
$$

5.2 Interactive Flag-Status Update

Recall in the previous chapter that a data structure to facilitate the interactive access of the object data (polygon vertices) is illustrated by the 2D VP (vertex-pointer) table. For this application, where edges are also selected for marking, an extra table to facilitate rapid access of them is required.

The edge-pointer (EP) table is similar in both aspects of *storing* and *accessing* of points to the organization of the VP table (see Chapter 4). To facilitate the selection of edges however, each edge is identified with a point that lies on the edge center (i.e., the average of its two end points). This center point, $\mathbf{c}_j$, is what is stored in the EP table to allow access to the rest of the fields of E_j, among which is the $eSmooth_j$ flag whose value is to be toggled between "TRUE" and "FALSE".

The definition of edges E_j defined above may therefore be modified to include the center point $\mathbf{c}_j$ in their definition as follows:

$$E_j = \{eID_j, \mathbf{c}_j, eSmooth_j, (vID_{E_{j1}}, vID_{E_{j2}})\}.$$

5.3 Highlight & Selection of Data

Coplanar patches that share the same vertices and/or edges may not be distinguishable from each other when shaded, therefore, it is necessary to use special highlighting to indicate the locations of all vertices and edges on the modeled object. On the displayed model, vertices are highlighted by small squares. In order to select a highlighted vertex for marking/un-marking, the user moves a pointing device (e.g., a mouse) on that square and then selects it (e.g., by clicking a designated mouse button) to toggle the value of $vSmooth_i$ from "FALSE" to "TRUE" and vice versa, interactively.

When the user selects the desired vertex for marking/un-marking, the coordinates of the location selected are recorded as an (r_i, c_i) pair of values on the two-dimensional pixel canvas. The (r_i, c_i) pair is then used to access the corresponding cell in the VP

table, which points to the corresponding vertex V_i. As a result, the value of *vSmooth*$_i$ is toggled immediately.

Since objects may consist of thousands of vertices, some of which may be clustered in the area of special features (i.e., where marking is to be done), functionalities such as scaling (to zoom in) and different color highlighting that changes interactively to reflect the status of *vSmooth*$_i$ for vertices are needed. The user may select between two display modes, depending on whether vertices or edges are to be selected for marking/un-marking. An illustration of a crude triangulated object with visible vertices highlighted differently to reflect their marking statuses through the *vSmooth*$_i$ flags is shown in Figure 5.1(a).

Similarly, when displaying in edge mode, each edge E_j is highlighted by a small circle that lies on its center defined by $\mathbf{c}_j$. The user may modify the *eSmooth*$_j$ value of the selected edge by selecting the edge-representing circle. The coordinates of the circle are recorded as an (r_j, c_j) pair that are used to access the corresponding cell in the EP table to retrieve the fields of edge E_j. As a result, the value of *eSmooth*$_j$ is modified accordingly. Figure 5.1(b) shows the same figure shown in Figure 5.1(a), but rendered in edge-display mode.

Observe that because of the structure of TLDS (based on PEL), each vertex V_i and edge E_j are stored only once, but may be pointed to several times when identifying edges and triangles, respectively. A change in the value of any flag *vSmooth*$_i$ in the VP array, or any flag *eSmooth*$_j$ in the EP array is global and is reflected in the

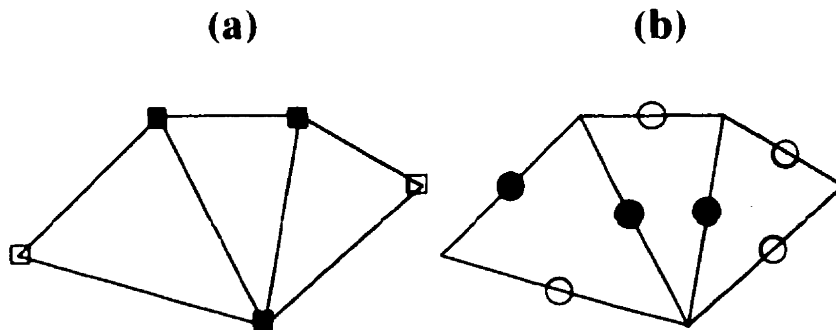

Figure 5.1: The same figure rendered in different modes. Highlighted vertices/edges reflect their different marking statuses.

definition of all the associated edges and triangles, respectively. In addition, locating vertices in the VP array, or edges in the EP array is only a matter of accessing the corresponding cell, whose locations are simply determined by the pointing device used by the user.

When inputting these data with the flags set as desired by the user, the surface smoothing algorithm will not attempt to smooth out regions around vertices or edges of particular features. This is because the associated flags ($vSmooth_i$ and $eSmooth_j$) of these selected vertices and edges are set to "FALSE", respectively.

5.4 Visual Example

In the two examples shown below in Figure 5.2 and Figure 5.3, each shows the object in the following order:

(a) A crude triangulated form of the object,

(b) The object after the application of uniform surface refinement as described in [54], and

(c) The object after the application of surface refinement with preservation of special features, by the aid of marking techniques developed thus far (i.e., using the TLDS representation and the interactive data selection and value-setting techniques).

An elaboration on the preservation of special features follow.

5.4.1 Example 1

This example illustrates a typical design problem, for example, creating a geometric model of an automobile's wheel-trim (or hubcap). In Figure 5.2(a), the object is approximated with $\mathbb{V} = 405$, $\mathbb{E} = 1125$, and $\mathbb{T} = 720$. After the application of the smoothing algorithm, each triangle in Figure 5.2(a)is refined into 36 triangles, resulting in $25,920$ triangles for the refined figure.

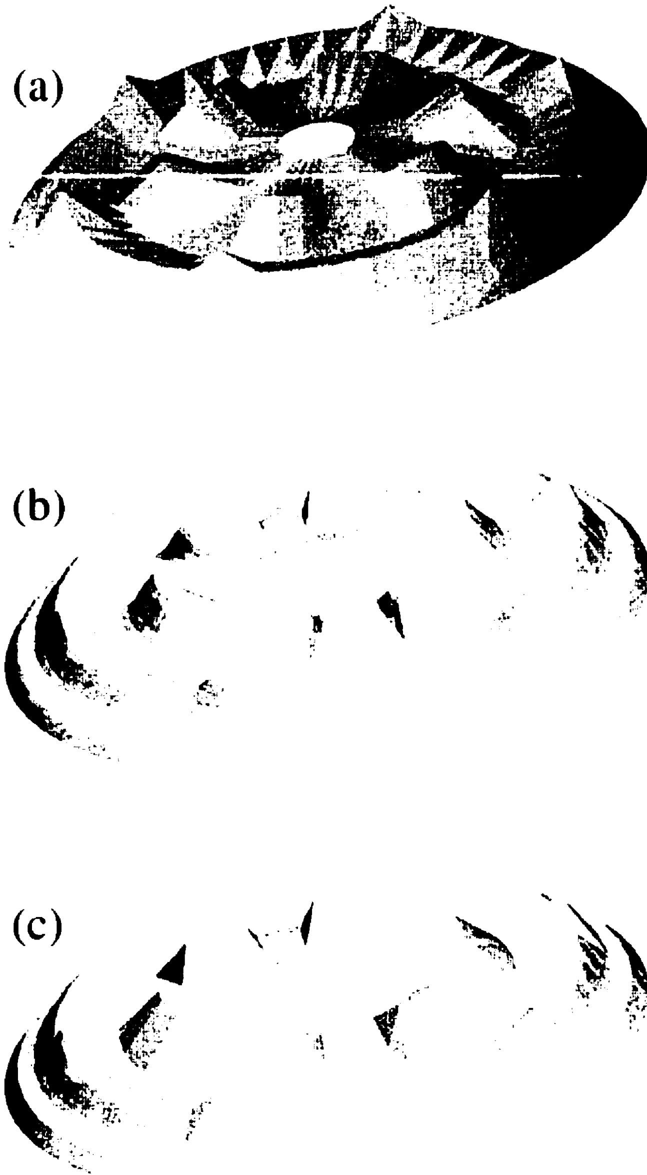

Figure 5.2: Preserving special features on an artificial object.

The uniform smoothing process results in the shape shown in Figure 5.2(b). Notice that all sharp points and edges were eliminated. To preserve the special feature where smoothing should not be applied, the peaks of three (each alternate one) of the humps on the outer rim were marked; the edges at the other humps on the outer rim were also marked. The vertices and edges at all humps on the inner rim were marked. Finally, the edges along the contour above the circumference were marked. The resulting model is shown in Figure 5.2(c).

5.4.2 Example 2

This example uses data from an actual object. A physical model of an ornamental donkey was laser scanned. The data were then used to produce a crudely triangulated model. The crude triangulated model is shown in Figure 5.3(a) with $\mathbb{V} = 439$, $\mathbb{E} = 1311$, and $\mathbb{T} = 874$ [54]. The refined models have 31,468 triangles each.

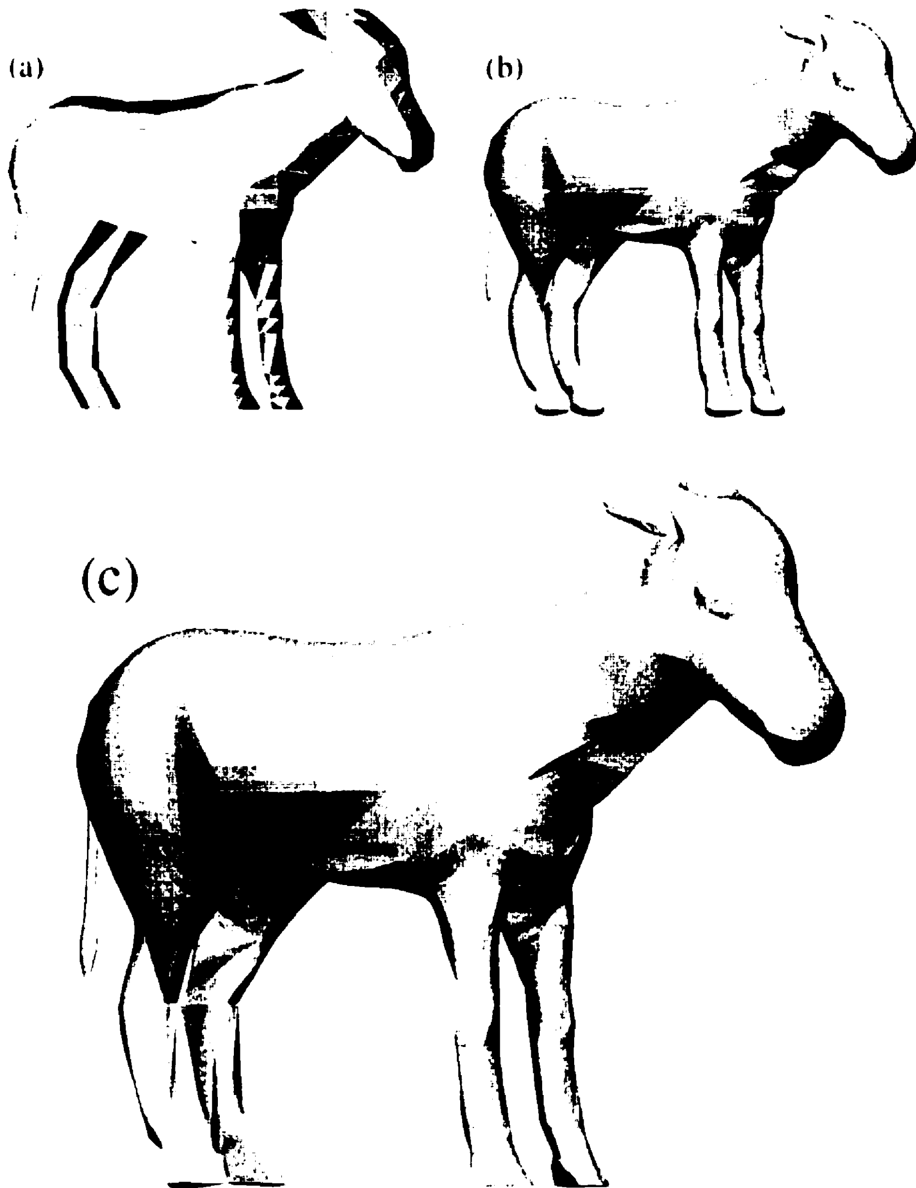

Figure 5.3: Preserving special features on an actual object.

The application of uniform surface refinement caused special features such as cusps on the ankles or tail-end of the donkey to be smoothed, resulting in an artificial look of those parts as shown in Figure 5.3(b). By rendering the donkey in vertex mode and "zooming in" on those parts when necessary, some vertices were marked so that they are not included in the smoothing process. In addition, edges on the bottom of the donkey's hooves and neck are also marked for exclusion during the refinement process. Results are shown in Figure 5.3(c). The difference in the right hind leg is particularly noticeable.

5.4.3 Implementation Issues

In transparent objects, some hidden polygons are visible, implying that more edges and vertices are also visible. However, only visible, non-hidden vertices should be highlighted, so that the surface image is not densely cluttered with highlighted vertices and edges. This conveniently limits the user to selecting points that are directly visible to them.

When marking/un-marking vertices, a group of related edges may be automatically marked/un-marked instead of individual processing. For example, in Figure 5.2(c), by marking the top vertex of the closest hump on the outer rim, all the edges sharing that vertex may be automatically marked as a user-option. This is possible since each vertex points to its set of surrounding triangles, and triangles point to their constructing edges, from which a subset shares the common vertex that causes them to be marked. Other scenarios for automating the process of marking/un-marking are possible and

are application dependent.

5.5 Summary

A 3D point location algorithm was developed to aid in the preservation of special features before a crude triangulated model undergoes procedures of surface refinement. It is shown that the process of locating 3D points requires no searching to be done. It takes a single operation to determine the location of the required memory cell containing the selected points for marking/un-marking for special features preservation. Although the work shown here applies to crude triangulated models, the data structure may be generalized to work on geometric polyhedral models composed of more general multi-size polygonal patches, simultaneously; this feature is important in other computer graphics applications [29].

In the next chapter, interactive shape modification and surface segmentation are focused on.

Chapter 6

Interactive Segmentation & Shape Modification of 3D Objects

This chapter illustrates how polyhedral surfaces can be organized into special data structures to facilitate rapid selection of vertices for purposes of shape modification. In addition, the chapter illustrates how those same surfaces can be segmented into sub-polyhedra for zooming and vertex manipulation during design, then re-introduced as modified segments into the original structure, interactively. Algorithms for accomplishing such tasks are presented.

In the previous chapter and Chapter 2, several methods for obtaining data from actual objects to assist in the construction of 3D computer models were discussed. Regardless of the source of data, it is desirable to make available the ability to perform local editing of the position of any point on the surface whilst being displayed. This may entail the application of a number of object transformations so that the modeled object is in a suitable position for the user for editing. Since only a small portion of the object

may be focused on at a time for manipulation, it is more convenient to segment such a portion for modification in isolation, and then re-introduce the modified segment for integration with the rest of the surface. This significantly reduces the number of patches (hence data points) that have to be manipulated, thereby decreasing computation time and increasing design throughput.

The rest of the chapter is organized as follows. Section 6.1 discusses related work to data structures, segmentation and shape modification. Section 6.2 proposes techniques for shape modification. The process of surface segmentation is discussed in Section 6.3. Section 6.4 presents the mechanisms necessary for the integration of segmented sub-polyhedra into the original structure. Visual examples produced by the algorithms outlined in the previous sections are provided in Section 6.5. The chapter is concluded with a summary and some practical implementation issues.

6.1 Related Work in Comparison

Two stages to facilitate efficient local manipulation of surface features of a 3D object are required. One stage allows rapid interactive selection of 3D vertices and as a result, related edges and polygons on display are also accessed, as discussed in the previous chapter and [30]. The other stage allows interactive segmentation of a portion of the polyhedral model as means of zooming in on, and manipulating the desired regions of the surface before integration with the original structure. These two stages may be applied to the surface several times and in any order. Related work, on representative data structures and algorithms, is described next.

6.1.1 Storage of Polygonal Models

The data structure to facilitate surface segmentation and shape modification is the TLDS (triangular-loop data structure) presented earlier in Chapter 3. It is used directly to allow the aforementioned operations (segmentation and shape modification) on polyhedral surfaces discussed here.

6.1.2 Surface Segmentation

As discussed in Chapter 2, surface segmentation has gained considerable attention from researchers whose work is based on volume visualization techniques, the focus is mostly on volumetric surfaces whose data are usually obtained from input processes such as CT (computed tomography) scans [9, 21, 28, 34, 35, 38, 44, 47, 56], or object voxelization (by casting a grid of parallel rays inside the object's bounding block) as in [11]. Some of the applications that volumetric data (voxels) are suitable for and others for which geometric data (polygons) are more suitable, are discussed in [31].

Segmentation of geometric surfaces on the other hand is not as well studied. Current techniques on surface segmentation require spatial partitioning representation data structures such as quadtrees, octrees, k-D trees or binary space partitioning for organizing the model data (vertices, edges and polygons) [17]. In [10] for example, motivation for such spatial representation is considered *potentially* suitable for segmentation and visualization purposes, but is stated as an on-going research problem. A representative spatial representation known as *octrees* may be tailored for achieving surface segmentation [5, 17, 23, 27, 39, 49]. However, using octrees for the type of

applications discussed here would still have the shortcomings discussed previously in Chapter 2.

6.2 Interactive Shape Modification

Once a vertex V_i is selected by means indicated (see sections 4.2 and 5.3), it is a straightforward process to modify the shape of the object locally. This is accomplished by dragging the selected vertex freely in the x–y-plane to a new position.

Because polygon-vertices data of all surrounding polygons is readily available from the selected vertex (see Chapter 3), it is of very low cost to visually animate the changes in real-time in the shape of the surrounding polygons, and to compute their new plane normals for shading purposes while V_i is being interactively dragged. No searching of any kind for vertices, polygons or edges is necessary.

Once the position of the dragged vertex is satisfactory, the final coordinates of the cursor become the new (x_i, y_i) pair of $\mathbf{v}_i$ in V_i. No extra bookkeeping is needed for any of the other fields of V_i, polygons or edges, which is one of the advantages of TLDS, inherited from the PEL representation.

6.3 Interactive Surface Segmentation

The fundamental purpose of segmenting a polyhedral surface into sub-polyhedra is for zooming and vertex manipulation during design. For example, a displayed object

model may consist of thousands of polygons. To facilitate and speed up the design process, users usually take the displayed model(s) through a series of transformations into a convenient position that facilitates manipulation. Instead of transforming all displayed polygons however, it is more efficient to apply transformations to only a small percentage of the displayed polygons where the editing is to take place. Once the sub-polyhedron is modified, it is re-introduced into the original model structure with modifications in effect, as are detailed in Section 6.4.

To segment off a sub-collection of polygons, the user interactively draws a bounding box (BB) over the desired region to determine the set of vertices that reside inside it. Since the drawing area has the same resolution as the VP array, the BB which is a rectangular subset of the pixels in the drawing area will have a one-to-one correspondence to cells of similar coordinates in the VP array. These cells are "scan-lined" using the *scan-line* algorithm [24] for example, so that each cell is examined for the contents of its non-empty linked-list of vertices (see Section 4.1). Figure 6.1 provides a visual illustration of the relationship between the display area and the VP array.

The shaded VP array cells inside the BB are checked for loaded linked-lists in the same manner pixels are *scan-lined* using the *scan-line* algorithm, e.g., left-right, top-down traversal. The subset of vertices SV found in the linked-lists of loaded cells are considered to be part of the sub-polyhedra to be created. In this regard, the BB also acts as a bounding volume with a depth that includes and extends from z_c, the depth of the closest vertex V_c inside the BB, to z_f, the depth of the furthest vertex V_f in the

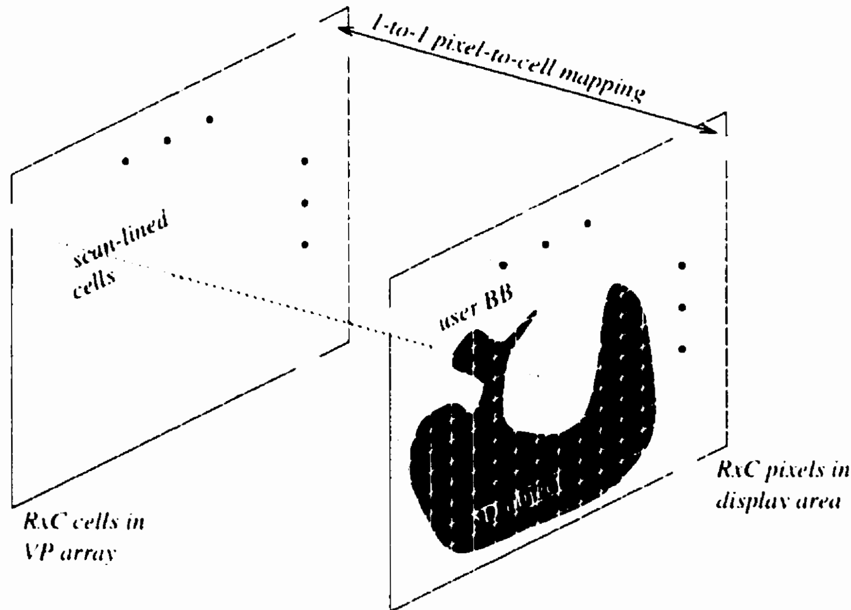

Figure 6.1: Relationship between canvas and VP array. Only those cells that are mapped to from pixels in BB are checked.

BB. Note that a BB may result in more than one sub-polyhedron, e.g., both ends of a U-shaped object can be captured with a single BB.

The complete process of formulating an independent sub-polyhedron that contains a subset of all the vertices SV, a subset of the polygons SP and their edges SE may easily be outlined by the following two algorithms. Without loss of generality, assume that the two opposite end points on the BB are the top-left point (r_1, c_1), and bottom-right point (r_n, c_n), where $1 \leq r_1 \leq r_n \leq R$, and $1 \leq c_1 \leq c_n \leq C$.

The first algorithm (CREATE-INCLUSIVE-SUB-POLYHEDRA) is straightforward. All polygons referenced by any vertex bounded inside the BB are considered part of the

new sub-structure to be segmented, even though some of the vertices of those polygons might not be bounded. Figure 6.2(a) illustrates an example of *inclusive bounding*.

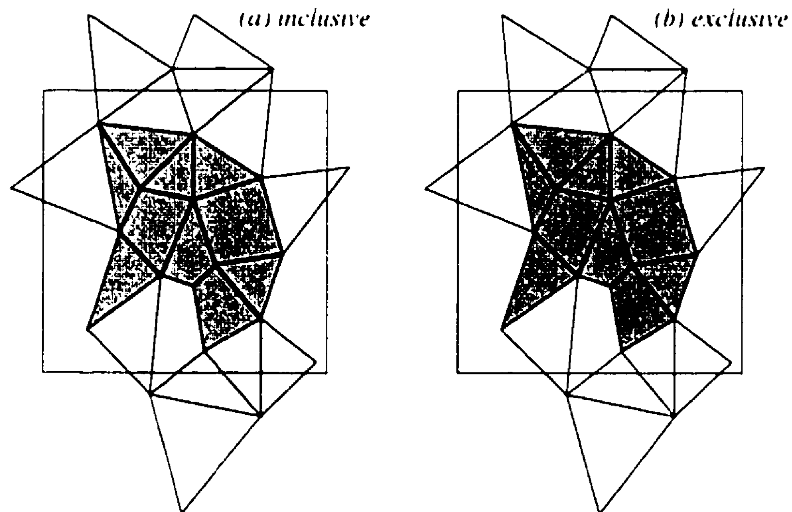

Figure 6.2: (a) *Inclusive bounding:* all (dark) or part of (light) polygon vertices are inside BB. (b) *Exclusive bounding:* only if all polygon vertices are inside BB.

Algorithm 6.3.1 The CREATE-INCLUSIVE-SUB-POLYHEDRA algorithm.

$SV, SE, SP \leftarrow \phi$
<u>for</u> $r \leftarrow r1$ <u>to</u> rn
  <u>for</u> $c \leftarrow c1$ <u>to</u> cn
    <u>if</u> linked-list$_{rc}$ of cell (r, c) is not null
    <u>then</u> $\forall V_i \in$ linked-list$_{rc}$
      $SV \leftarrow SV \cup V_i$
$\forall V_i \in SV$
  <u>for each</u> polygon $P_k \in S_i$ polygons referenced by V_i
    <u>for each</u> edge E_j referenced by polygon P_k
      $SE \leftarrow SE \cup E_j$
    $SP \leftarrow SP \cup P_k$

In the next algorithm (CREATE-EXCLUSIVE-SUB-POLYHEDRA), only referenced polygons whose vertices are *all* bounded, are considered part of the new sub-structure while others are ruled out. It is possible to determine the exclusively bounded polygon set by exploiting the circular structure of TLDS as shown in the algorithm. Figure 6.2(b) illustrates an example of *exclusive bounding*.

Note that in the case of this *exclusive* algorithm, the array SV may reference polygons that are not part of the new structure. To adjust for such a situation, a special temporary integer array PID of length k is initialized with a sentinel value throughout but in the position of included polygons P_k. In the event that an excluded polygon P_k is referenced, the sentinel value present in PID_k is used to indicate such an absence. As a result, no further processing (e.g., accessing the edges of P_k) is carried forth.

Algorithm 6.3.2 The CREATE-EXCLUSIVE-SUB-POLYHEDRA algorithm.

$SV, SE, SP \leftarrow \phi$
$\forall k \in [1, \mathbb{P}], PID_k \leftarrow 0$
  for $a \leftarrow a1$ to an
    for $b \leftarrow b1$ to bn
      if linked-list$_{ab}$ of cell (a, b) is not null
      then $\forall V_i \in$ linked-list$_{ab}$
        $\boxed{SV \leftarrow SV \cup V_i}$
$\forall V_i \in SV$
  for each polygon $P_k \in S_i$ polygons referenced by V_i
    $allPolygonVerticesIn \leftarrow$ TRUE
    $tempSE \leftarrow \phi$
    for each edge E_j referenced by P_k
      if either of end-points ($V_{E_{j1}}$ or $V_{E_{j1}}$) is out of BB then
        $allPolygonVerticesIn \leftarrow$ FALSE
        break out of this loop
      else
        $tempSE \leftarrow tempSE \cup E_j$
    if $allPolygonVerticesIn$ is TRUE then
      $\boxed{SE \leftarrow SE \cup tempSE}$
      $\boxed{SP \leftarrow SP \cup P_k}$
      $PID_k \leftarrow k$

It is now possible to treat and display the newly created sub-polyhedra as an independent substructure, based on data from the arrays SV, SE, and SP highlighted by the boxed statements in the above algorithms. Note also that the segmentation process may be applied on segmented sub-polyhedra as many times as desired for complete control over the entity to be segmented. Re-introduction of sub-polyhedra may be done to the parent subsurface or to the original surface; it is considered an implementation issue.

6.4 Integration of Modified Segments

Once a segmented sub-surface is manipulated (tweaked, rotated, translated, etc.) as desired by the user, it is re-introduced back in the original structure with some of the new modifications in effect (e.g., tweaking), while other changes are undone (such as uniform transformations).

An illustration of surface segmentation, modification and merging with a simple object surface consisting of seven triangles is shown in Figure 6.3. In Figure 6.3(a), a BB is outlined over the desired region to be segmented. Figure 6.3(b) shows the resulting sub-polyhedron. Note how only *exclusively bounded* triangles within the BB are considered. The space position of some of the vertices on the segment is modified as desired, as shown in Figure 6.3(c). Finally, the segment is re-introduced to the original structure. As a result, the new spatial positions of the dragged vertices overwrite the original values in the original V array. Results are shown in Figure 6.3(d).

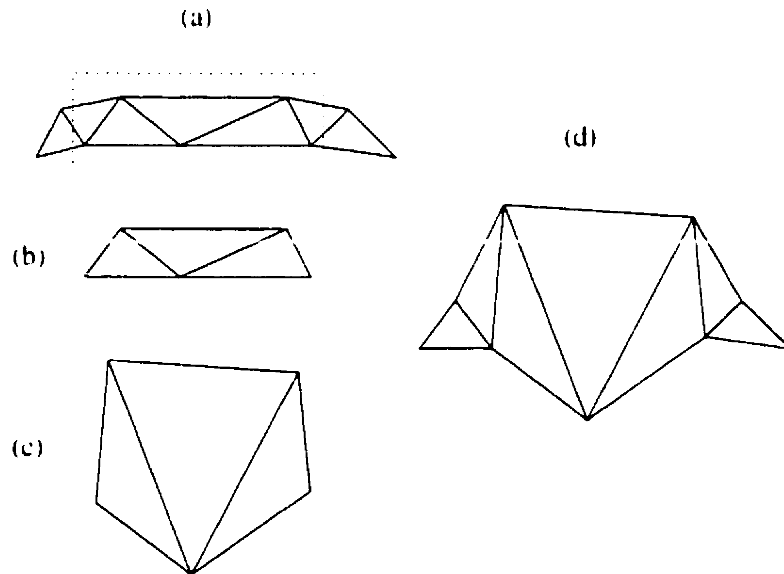

Figure 6.3: An illustration of segmentation, modification, and re-introduction of segmented sub-polyhedra into main structure.

The algorithm below summarizes the steps starting with a sub-polyhedron going through a homogeneous coordinates transformation matrix $M = \dot{T} \cdot \dot{R} \cdot \dot{S}$, where $\dot{T}$ is the *translation-matrix*, $\dot{R} = \dot{R}_x(\alpha) \cdot \dot{R}_y(\theta) \cdot \dot{R}_z(\gamma)$ is the compound *rotation-matrix*, and $\dot{S}$ is the *scaling-matrix*.

Algorithm 6.4.1 The INTEGRATING-SUB-POLYHEDRA algorithm.

$\forall V_i \in SV$
  $\tilde{\mathbf{v}}_i \leftarrow M \cdot \mathbf{v}_i$
while not done modifying do
  if a vertex $\tilde{\mathbf{v}}_i$ is selected and its position is modified to new location (r, c) in display area
  then
    $(\tilde{x}_i, \tilde{y}_i) \leftarrow (r, c)$
    $\mathbf{v}_i \leftarrow M^{-1} \cdot \tilde{\mathbf{v}}_i$

Whereas the above algorithm only considers modifications in the $(\tilde{x}_i, \tilde{y}_i)$ coordinates, the $\tilde{z}_i$ value is either:

(a) implicitly modified whenever the vertex $\tilde{V}_i$ is independently modified under a transformation other than that of the original structure, or

(b) explicitly modified by changing its value using tools provided by the interface for example.

Note that the only data that need to be re-introduced to the original structure(s) are that of the vertices V using vertex identifiers ***vID*** to overwrite the spatial positions of $\mathbf{v}$. The other identifiers (***eID*** and ***pID***) for edges and polygons are needed only for referencing purposes.

6.5 Visual Examples

The following examples illustrate the algorithms and data structures developed to facilitate surface modification and segmentation. Each of the two sample objects is taken through a series of operations before being integrated with the original structure. Algorithms and techniques provided in [30, 54] are used to generate the final refined model in each case to give a smoother model appearance.

6.5.1 Example 1

This example uses data from an actual object—the ornamental donkey introduced previously in Section 5.4.2 with $\mathbb{V} = 439$, $\mathbb{E} = 1311$, and $\mathbb{T} = 874$.

Figure 6.4 illustrates a series of segmentations and editing of vertex location operations. The intention is to form a horn on the donkey's head, and subsequently to prolong and curve the tail.

To isolate the segment containing the head from the rest of the body, a BB is drawn over the head as shown in Figure 6.4(a). This results in the subsurface shown in Figure 6.4(b). The head is rotated to face the viewer as shown in Figure 6.4(c). Here, only 158 vertices and 300 triangles are manipulated. The vertex that would later be the top vertex in the horn is highlighted by a circle. To make the horn thin, the six surrounding vertices are pulled inwards towards the horn-vertex, as shown in Figure 6.4(d). The results of pulling the horn up and then rotating back the head for a profile look are shown in figures 6.5(e) and 6.5(f), respectively. The horn is tilted forward a small

distance, and the bottom-front vertex of the horn (highlighted by a circle) is lowered some distance for artistic appearance, as shown in Figure 6.5(g).

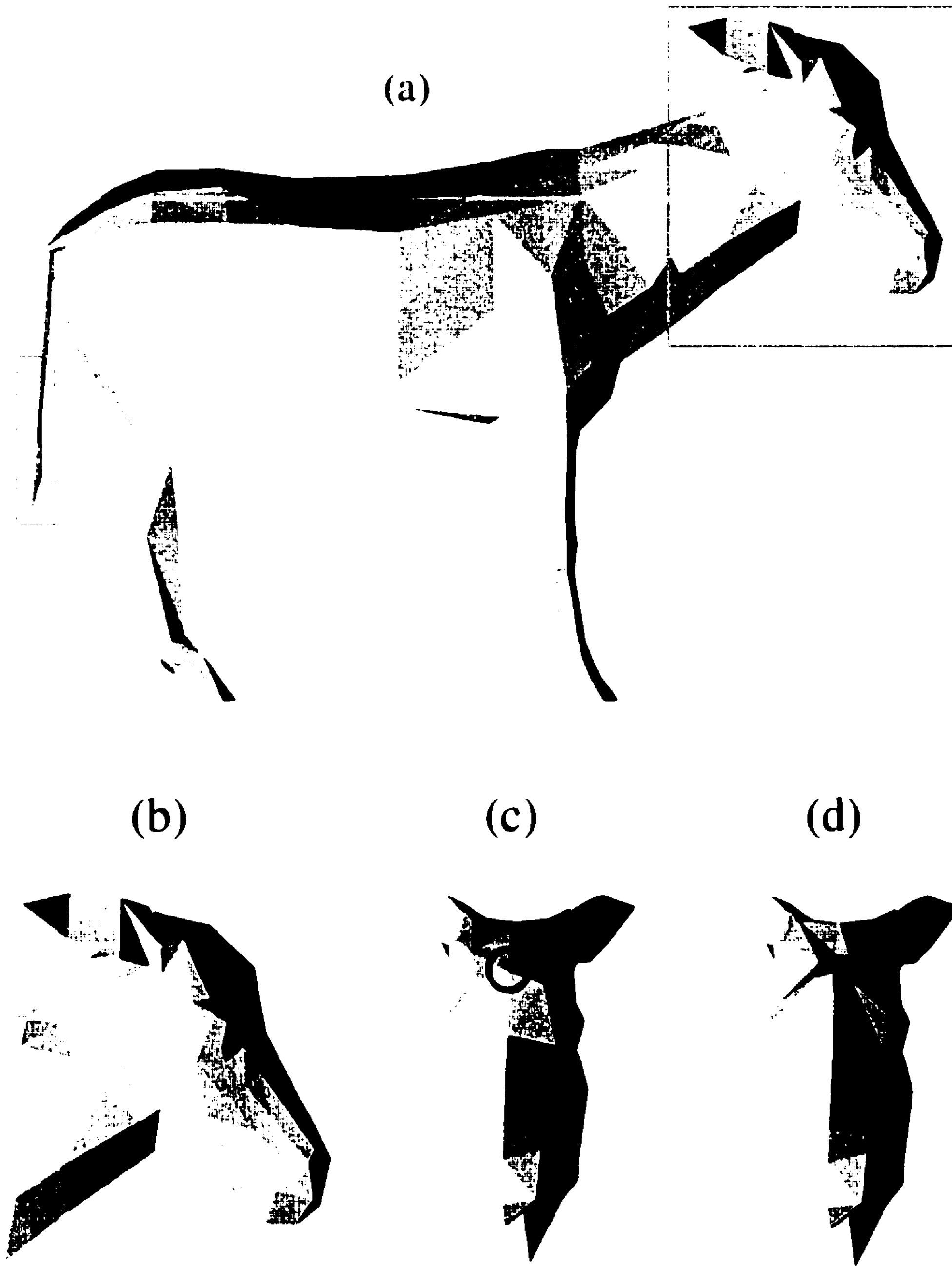

Figure 6.4: Donkey-head shape modification.

The attention is now on the tail-end of Figure 6.5(h), segmented by the BB shown on the tail-end in Figure 6.4(a). Here, only 15 vertices and 24 triangles are being manipulated. In addition, because the tail is very small in relation to the rest of the donkey, it is scaled by a factor of 250, followed by a series of other transformations, so that vertices are sparse enough for easy manipulation, as shown in Figure 6.5(i). To modify the tail-end shape without performing surface segmentation would be a tedious and a computationally intensive task, since any transformation operation would be applied uniformally to all other polygons unnecessarily. Figure 6.5(j) shows the end result after the head and tail are modified and re-introduced back to the rest of the body. A smoothed model of the modified donkey is shown in Figure 6.5(k).

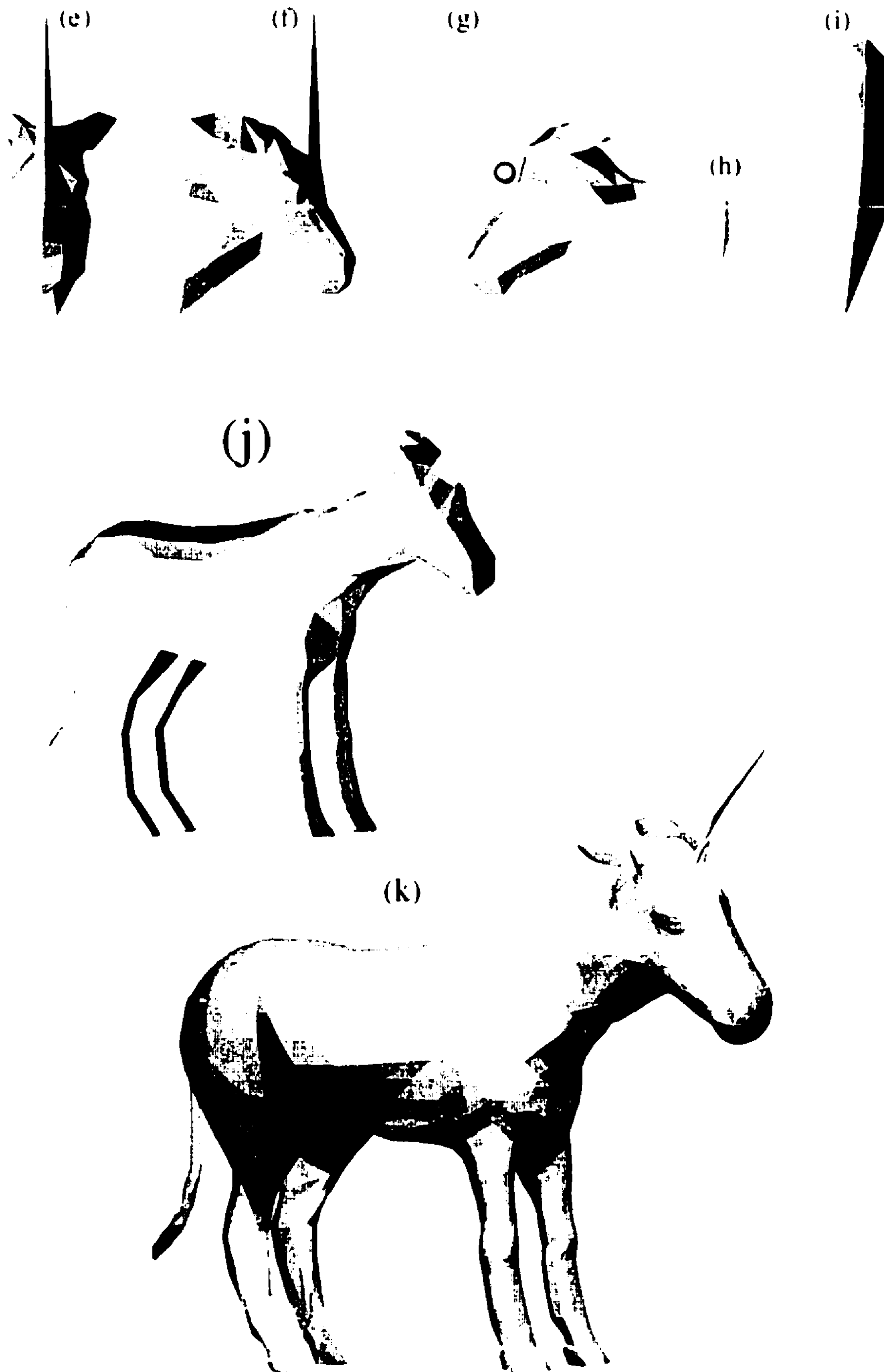

Figure 6.5: Donkey-head and donkey-tail shape modification.

6.5.2 Example 2

This example illustrates a typical design problem. For example, direct manipulation of a geometric model designed as a surface of revolution. The profile has 20 vertices and is rotated by angle increments of 22.5 degrees about the y-axis. Each quadrilateral is subdivided into two triangles for rendering purposes. The resulting polyhedron is then input into the CREATE-TLDS-REPRESENTATION algorithm to generate a TLDS representation of the model. It is shown in Figure 6.6(a) with $\mathbb{V} = 320$, $\mathbb{E} = 928$, and $\mathbb{T} = 608$. The intention is to modify the shape of the vase base, and then to construct a pointed-pouring region at its opening.

First, the vase is displayed in a convenient way to segment the bottom three cross-sections of the vase. A BB is drawn over the area to be segmented, as shown in Figure 6.6(b). A positive $90°$ rotation of the segment about the x-axis is shown in Figure 6.6(c). Modification intended in the direction of the arrows are now straight-forward by the aid of a gridded display area; results are shown in Figure 6.6(d). Note that the simplicity and intuition in dragging inner vertices to modify the shape is a result of the segmentation and isolated manipulation. This is especially convenient when the region to be edited is completely contained in the interior regions of the model. Figure 6.6(e) shows the end result of the first phase after integrating the base with the vase.

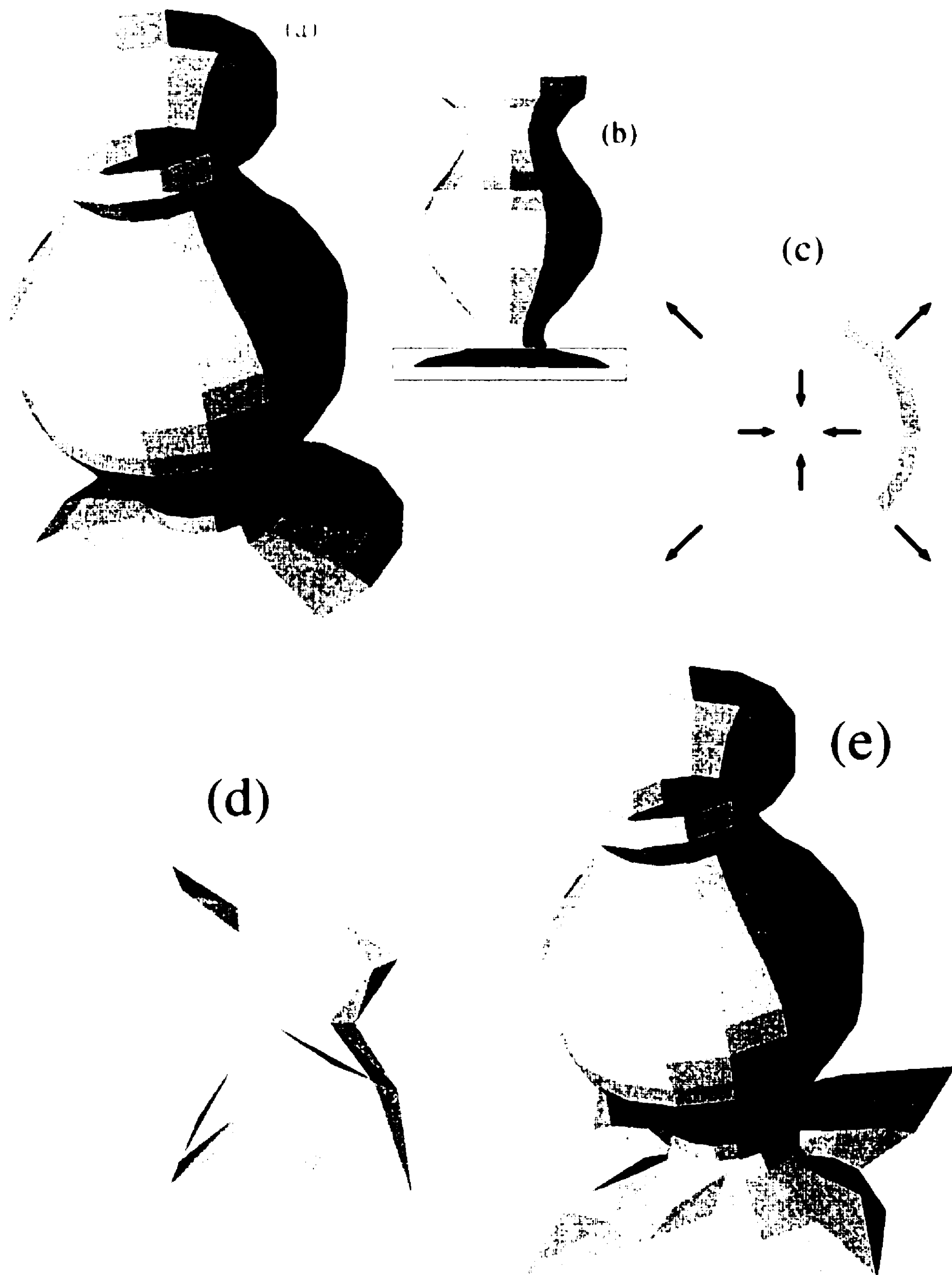

Figure 6.6: Vase-base shape modification.

In the second phase, the modified vase is transformed in a convenient position for performing surface segmentation again, as shown in Figure 6.7(f). The top three cross sections are segmented and are shown in Figure 6.7(g). The three vertices whose positions are to be modified are highlighted by the circle as shown. These vertices are pulled outwards a short distance. The segmented part is then rotated back (-90°) about the x-axis. The two external vertices are pulled some distance downwards. A 40° rotation about the x-axis shows the end result of the segmented part after modifications, shown in Figure 6.7(h). The end result of the second phase modification is shown in Figure 6.7(i). This modified object then undergoes some surface refinement for a smoother appearance, shown in Figure 6.7(j).

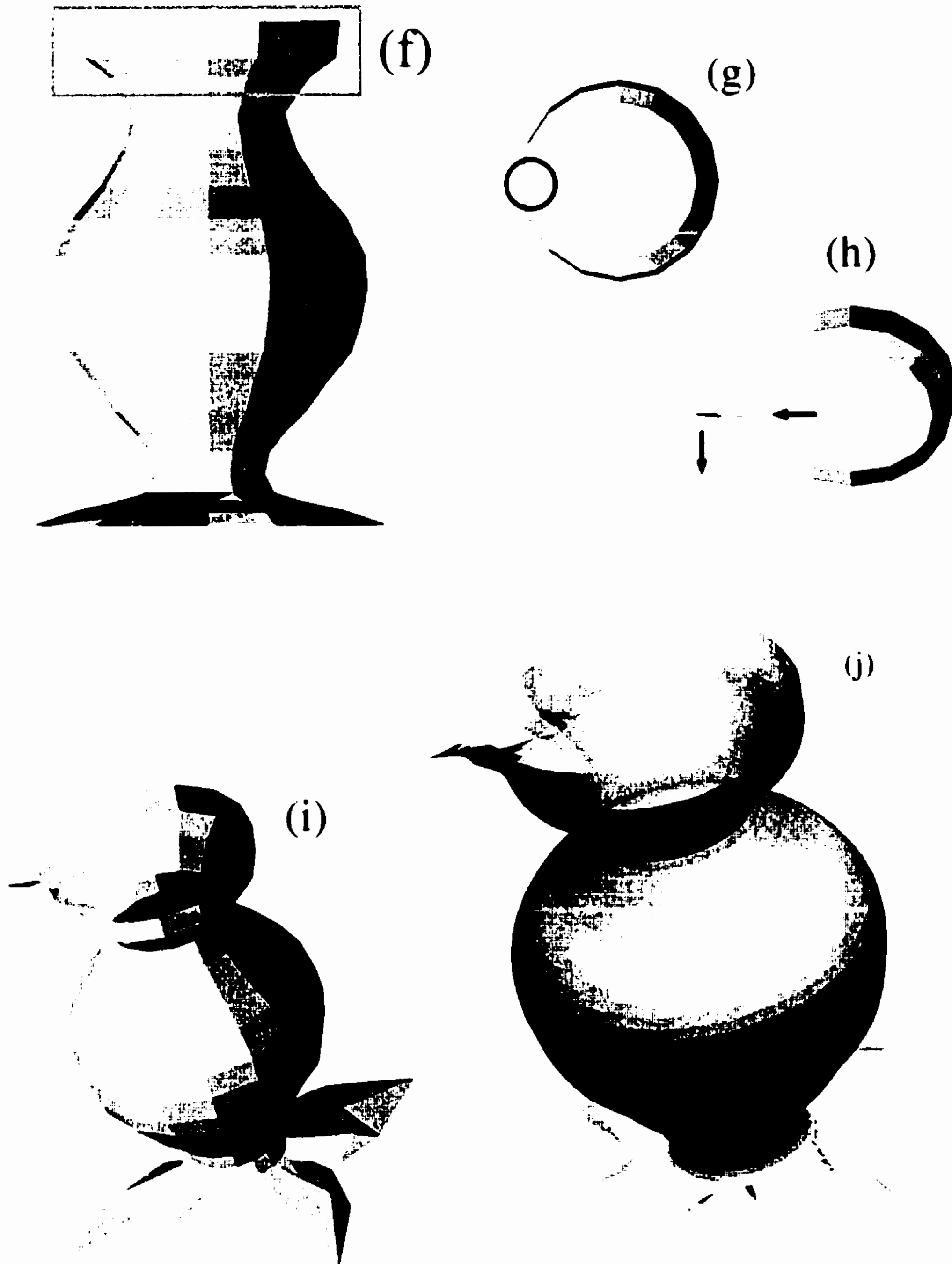

Figure 6.7: Vase-opening shape modification.

6.6 Summary

Data structures and algorithms to aid in geometric segmentation of polyhedra and shape editing were developed. The *triangular-loop-data-structure* (TLDS) was shown to be quite suitable for application such as the identification of surrounding vertices, edges, and polygons for rapid vertex-normal calculation or interactive shape modification. Because of the following reasons:

- Actual CAD/CAM models are getting larger with respect to number of constructing vertices.
- Users, specially in the CAD/CAM fields, increasingly expect improved visual performance, and rather instantaneous system responses during the design stage for rapid product development,

the algorithms proposed were developed to exploit TLDS to segment off a subset of/locate/or modify the position of vertices in real time, with much less memory requirements than would be needed by spatial data structures such as octrees.

The techniques described here may not be immediately suitable for general polyhedral models (whose faces may have more than three vertices). An exception may occur if the vertex position to be modified belongs to one of these faces, in which case, non-planar polygons may result. However, we can always obtain a triangulated model from any polyhedral model to remedy such an exception.

Chapter 7

Conclusions & Future Directions

In this dissertation, various modeling and visualization techniques designed for geometric 3D surfaces were established. The techniques were developed to handle all sorts of geometric surface topologies. It was shown that by moving from quadrilateral-patch surfaces to any other structures, such as triangular-patch or multi-size-patch surfaces, meant a change in the techniques of data organization and interaction with the displayed surfaces.

Many algorithms and data structures were developed to facilitate operations that include:

- Reconstruction of 3D surfaces and volumes by point-sampling from readily available images, such as CT scans.
- Generation of complementary surfaces from existing ones and then connecting them at corresponding locations. This leads to exhibiting thickness in modeled objects, which is naturally present in many objects.

- Construction of virtual holes to view hidden layer portions.
- 3D point location for selecting specific points for special treatment.
- Segmentation of surfaces into various smaller sub-surfaces for rapid design purposes.
- Modifying the shape of a surface by direct user-interaction.

To address applications of geometric modeling to areas such as rapid prototyping, most of the algorithms and data structures developed in this dissertation focused on these two fundamental issues:

- Efficiency in the storage organization of surface data (i.e., vertices, edges, and polygons and their associated fields) to allow immediate access to the required data.
- Facility in the interaction with the displayed surfaces, to segment off or isolate a portion of the surface by means of bounding volumes; tweak vertices to alter the shape of polygons on the surface as desired; or to locate a set of polygons to extract data associated with them for independent manipulation.

All the theoretical work of this dissertation is complemented by actual implementation and illustrative visual examples throughout. The algorithms and data structures are built on existing, well-studied methods and structures such as the *scan-line* algorithm, the *z-buffer* algorithm, bounding boxes and bounding volumes, *pointer-to-an-edge-list* structures, and so on. The practical issues of addressing memory and performance

speed are also presented and shown to be competitive with other similar data organization techniques.

7.1 Practical Extensions

The concepts developed in the previous chapters used limited representative examples to illustrate ease of implementation of presented data structures and algorithms, and practical applications to apply such concepts upon. Below are suggestions to make feasible the generic application of presented concepts to a wider set of modeling and visualization problems.

7.1.1 Modeling & Visualization of Layered Objects

In Chapter 2, new techniques for constructing virtual holes on quadrilateral-patch surfaces were studied [31]. The following is a list of related topics.

More virtual hole shapes

The algorithms in Chapter 2 allowed only a square-shaped area of size ($n \times n$) cuboids to be removed (see Subsection 2.5.1). Other virtual hole shapes such as circular, user-drawn polygons, or even free-hand drawn shapes may be implemented to suit applications accordingly.

Handling dome-shaped regions

Representing surfaces by quadrilateral patches has the robustness advantages mentioned at end of Section 2.6. Therefore, for dome-shaped objects, or spherical-shaped objects (such as the sun or the globe), where all the patches are four-sided except at the pole regions, simple mechanisms may be developed (such as covering the two poles by a special patch) while insisting on a quadrilateral-patch representation elsewhere.

Vertices versus centroids selection

In addition to *centroids*, virtual holes can be made by selecting patch-corner vertices to cause all surrounding polygons to be *hidden*. Patch-corner vertices may actually be more convenient for the user to construct virtual holes since they are easily located.

7.1.2 Segmentation & Shape Modification of 3D Objects

Chapter 6 presented mechanisms for manipulating geometric 3D objects by using segmentation and direct interactive shape modification techniques [32]. Other straightforward functionalities to supplement such work should include:

Flexible bounding volume shapes

In addition to the cubical bounding volume illustrated in Chapter 6, the user should be allowed to control the shape and size (e.g., by specifying depth) of bounding volumes

to suit different applications. Free-hand drawing over the region to be segmented is among the example shapes to be considered.

"Click-to-magnify" functionality

To better visualize and manipulate the surface mesh, the user selects the region of interest and the portion size to be selected. This causes the selected region to be magnified (probably in a different viewing window) to facilitate surface-structure understanding and manipulation.

"Polygon-set-eraser" functionality

This functionality should be implemented as a complement to the "polygon-set-isolator" (where the selected/bounded set is isolated for independent manipulation) presented in this work. This allows a see-through of the hidden surface portions in the original surface (similar to the functionality of virtual holes presented in Chapter 2).

"Click-to-erase/isolate" functionality

Here, the user selects a vertex and as a result, all the surrounding polygons of the selected vertex are "erased" or "isolated", to allow a see-through of hidden surface portions, or to view the set of surrounding triangles in isolation, respectively. In this case, vertices in multi-size-patch surfaces play the same role as do *centroids* in quadrilateral-patch surfaces.

7.2 Potential Applications

The applications discussed in the previous chapters were selected as representative examples to aid in understanding the motivation and objectives of this dissertation. This section suggests other potential applications that may be attained by the use of the techniques developed herein.

In Chapter 2, representative examples for modeling objects and constructing virtual holes focused on simple applications in anatomy. For example, after modeling a human arm using B-spline surfaces, or a human thigh from CT scans, it is shown how constructing virtual holes allowed a see-through of internal layers such as bones and muscles.

Chapter 6 allowed direct object segmentation and object editing mechanisms of any polyhedral geometric model. Such mechanisms may be integrated and exploited to serve in the following applications.

7.2.1 Computer-aided anatomy browsers

In the Anatomy Section in [50], almost all the human-body's skeletal, external, and internal parts are illustrated in an artistic layout using NURBS, i.e., using polygons. By re-organizing the data in one of the formats developed earlier by algorithms 2.3.1 or 3.2.1, professionals involved in the study of the human body may strip away selected portions of any layer(s), to visualize the geometric relationship among the different body organs for making medical decisions; anatomy students can visualize

the modeled body, and make virtual holes at regions of interest to better understand the internal body structure. Motivation for such work is abundant in the literature for purposes of understanding, teaching, surgical decision making, visualization and art, which are some of the examples [3, 4, 6, 13, 20, 25, 43].

In addition to anatomy, other disciplines that depend on the computerized design of 3D models such as in architecture, geography, air and ground transport, household appliances, etc...., may also be visualized and manipulated using the techniques developed in chapters two through six of this thesis, since models in these disciplines are represented by polyhedra [50].

7.2.2 Construction of 3D cross sections

Forming 3D cross-sections of spherical objects such as the sun or the earth, where the object is represented by many concentric layers that become smaller in diameter towards the center [45], this is practical since each layer of cuboids may be designated to represent one of the strata of the modeled object. More complex color-coding techniques (such as texture modeling) to complement the modeling techniques developed herein should prove useful.

7.2.3 Interfacing with existing CAD packages

Many known CAD software packages can manipulate the modeled-object surfaces in a variety of ways. One of which is to allow the model to be viewed and stored under many representations such as implicit, parametric, or polygonized representations, for

example. Once the object is represented by polygons (i.e., patches), it becomes a straightforward process to incorporate the algorithms developed herein in the form of 'extensions', to aid in the design process.

To illustrate, consider the **Auto-Z** package, an automated solid modeling extension package for **AutoCAD R14** [2]. **Auto-Z** adds to a feature already existing in **Auto-CAD** known as "`slice`" or "`slicing`" the functionality of constructing *complex* multi-plane slicing (as opposed to the traditional single-plane slicing, which is now referred to as *simple* slicing). Complex slicing allows the user to select multiple intersecting planes and edges with user-specified depths to control the size and shape of the cut. A part of a cylindrical-shaped object may be sliced-off for example, to reveal the interior for further object-insertion. Slicing involves performing clipping operation that does not differentiate between polygons belonging to one polyhedron from another.

However, by forcing a polyhedral representation of the object, data can be passed through FORM-LAYER (Algorithm 2.3.1) if the faces are four-sided, or through the CREATE-TLDS-REPRESENTATION (Algorithm 3.2.1) otherwise, to put the data in a format compliant with the visualization and editing algorithms developed here (see Page xi & Table 1.3 for information on all algorithms and their functionalities). For example, virtual holes may be made any where on the polyhedron to allow a see-through of hidden layers (as shown in Chapter 2), or the desired region may be segmented several times, under different transformation for manipulating the sub-polyhedron in isolation. In addition, it is very simple to control what is to be segmented, e.g., only the

set of polygons that belong to a certain polyhedron. This is possible since each polygon knows to which object it belongs (see Section 2.1). The costs of such operations were discussed in Chapter 3.

7.3 Future Work

Although the theoretical foundation that underlies the data structure and algorithms of this dissertation proposed solutions to polyhedral surfaces of various topologies, there are still many areas, both of theoretical and applicative nature, that should be explored further. Below are sample problems that may be of interest.

7.3.1 Dealing with Surface Collisions

Visual examples illustrated in Chapter 2, Section 2.5 devised mechanisms for designing multi-layered objects, in which some of the layers may be modeled inside others. However, to make such a modeling scheme robust and avoid possible visual anomalies, mechanisms to detect when the surface/layer of one object penetrates the surface/layer of another should be employed, to avoid getting "arbitrary" visual effects [34, 51, 52]. The data structures and algorithms introduced in Chapter 2 may be accordingly modified to alert the user when cuboids (or more generally, polygons) belonging to one object penetrate the space occupied by the cuboids of another object.

7.3.2 Vertex Addition in TLDS

Given a model in TLDS representation, how to modify the object shape by adding and manipulating extra vertices that should eventually be integrated with the rest of the structure? This issue may arise when the structure's original set of data (vertices, edges, and polygons) should be left intact, but shape modifications are also necessary at certain regions. In Subsection 6.5.1 for example, a horn was constructed by designating one of the vertices on the donkey head to be the tip of the horn. However, this could have been accomplished by adding such a vertex independently, and then indicate the set of vertices from the original structure that it should be connected to to form the horn shape. Further decisions have to be settled if the indicated set of vertices belonged to more than one polygon of course, particularly with the shared edges. The final operation is that such a vertex should be integrated with the rest of the TLDS representation of the edited polyhedron. Other examples may require the addition of many vertices that should be position-adjusted as they are inserted, e.g., to add a handle to a displayed tea-pot.

7.3.3 Integration of Two Polyhedra

Given two models, each in TLDS representation, what are possible mechanisms to attach them together into a single TLDS representation? As a representative example, consider the attachment process of two already-designed polyhedra models, one representing an fuselage of an airplane, the other representing one of its side wings. Although it is more practical and convenient to design each part independently, attach-

ing them to form a single TLDS polyhedron allows the rapid overall visual inspection of how they fit together, and facilitates the application of shape modification and segmentation techniques discussed in the previous chapters.

Nevertheless, there are many issues that have to be settled when integrating two polyhedra into one polyhedron, especially when the initial design did not consider any future integration operations. The following are two sample issues that should be considered at the region where the join is to occur:

- If the two polyhedra are transformation-incompatible (i.e., each designed with different rotation angles, different scale vector, etc...), mechanisms to co-register each, e.g., using point-matching methods should be applied before integration takes place.
- Will the number of vertices in each polyhedron have to be constrained such that they are equal at the place of join? If not, what steps should be taken to establish correspondence between the vertices from each polyhedron?

7.4 Final Remarks

In this dissertation, many data structures and algorithms are proposed to facilitate interactive immediate interaction between the user and 3D displayed objects. The data structures and algorithms are developed for performing operations such as *surface segmentation & integration, shape editing & modification, virtual hole construction* of layered objects, and *surface querying*.

Bibliography

[1] Anand, V., *Computer Graphics and Geometric Modeling for Engineers*, John Wiley & Sons, Inc., NY, 1993.

[2] Auto-Z, *Auto-Z: Automated Solid Modeling for AutoCAD*, EMT Software, Inc., Package details & user-manual are found on WWW at http://www.auto-z.com. See also http://www.auto-z.com/nfeatures.htm.

[3] Beaumont, I., "User Modelling and System Adaptation in the Interactive Anatomy Tutoring System Anatomy-Tutor", *Proc. of the Sixth Int. Conf. on Human-Computer Interaction*, II.9 Learning Environments 2, Vol. II. Human Centered System Design, 1995;901-906.

[4] Bouvier, M., Bushyhead, A.L., Benson, A.N., *The Visible Human Project: Cross-sectional Anatomy Tutor: An Interactive Course for Anatomy Education and Evaluation*, On Web at http://www.jbpub.com/visiblehuman/index.htm.

[5] Brunet, P., Navazo, I., "Solid representation and operation using extended octrees", *ACM Transactions on Graphics*, 1990;9(2):170–197.

[6] Caponetti, L., Fanelli, A.M., "Computer-aided simulation for bone surgery", *IEEE CG&A*, 1993;13(6):86–91.

[7] Carlbom, I., Chakravarty, I., vanDerSchel, D., "A hierarchical data structure for representing the spatial decomposition of 3D objects", *IEEE CG&A*, 1985;5(4):24-31.

[8] Chan, K and Tan, S.T., "Hierarchical structure to winged-edge structure: a conversion algorithm", *The Visual Computer*, September 1988;4(3):133–141.

[9] Chandru, v., Manohar, S., and Prakash, C.E., "Voxel-based modeling for layered manufacturing", *IEEE CG&A*, 1995;(15)6;42–47.

[10] Chazelle, B., Application challenges to computational geometry: CG impact task force report, *Princeton University*, Nov. 1996. On Web at http://www.cs.princeton.edu/ chazelle/taskforce/CGreport.ps.

[11] Chiu, W.K., Tan, S.T., "Using dexels to make hollow models for rapid prototyping", *Computer-Aided Design*, 1998;30(7):539–547.

[12] Cignoni, P., Montani, C., Scopigno, R., "MagicSphere: an insight tool for 3D data visualization", *Computer Graphics Forum* (Proc. Eurographics), Basil Blackwell Ltd., 1994;13(3):317–328.

[13] Cross section images from *Visible Human* Project. Available at http://www.mir.wustl.edu/visible_human/atlas.HTML. See also detailed descriptions of these images in *IEEE CG&A*, 1996;16(1):15.

[14] Evans, W.F., *Anatomy and Physiology: The Basic Principles*, Prentice-Hall, Englewood Cliffs, NJ, 1971:139.

[15] Fang, T., Piegl, L., "Delaunay triangulation in three dimensions", *IEEE CG&A*, 1995;15(5):62–69.

[16] Farin, G., *Curves and Surfaces for Computer-Aided Geometric Design: A Practical Guide*, Academic Press, Boston, 1993.

[17] Foley, J., van Dam, A., Feiner, S., Hughes, J., *Computer Graphics: Principles and Practices*, Second Ed., Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[18] Forsey, D., Bartels, R., "Hierarchical B-spline refinement", *Computer Graphics*, 1988;22(4):205–212.

[19] Gargantini, I., Atkinson, H., "Ray tracing an octree: numerical evaluation of the first intersection" *Computer Graphics Forum*, 1993;12(4):199-210.

[20] Grimson, W., "Image understanding, medical imaging, neurosurgery: Medical Application of Image Understanding", *IEEE Expert*, Oct. 1995:18-28.

[21] Guttman, M.A., Zerhouni, E.A., McVeigh, E.R., "Analysis of cardiac function from MR images", *IEEE CG&A*, 1997;17(1):30–38.

[22] Hagen, H., Nielson, G., Nakajima, Y., "Surface design using triangular patches", *Computer-Aided Geometric Design*, 1996;13:895–904.

[23] Hearn, B., Baker, M., *Computer Graphics*, Prentice-Hall, Englewood Cliffs, NJ, 1994.

[24] Hill, F.S., *Computer Graphics*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[25] Höhne, K, Pflesser, B., Pommert, A., Riemer, M., Schiemann, T., Schubert, R., Tiede, U., "A 'Virtual Body' Model for Surgical Education and Rehearsal", *Computer*, 1996;29(1):25-31.

[26] Kalvin, A.D., Taylor, R.H., "Superfaces: Polygonal mesh simplification with bounded error", *IEEE CG&A*, 1996;16(3):64–77.

[27] Kunii, L., Satoh, T., Yamaguchi, K., "Generation of Topological Boundary Representations From Octree Encoding", *IEEE CG&A*, 1985;5(3):29-38.

[28] Levoy, M., "Efficient ray tracing of volume data", *ACM Trans. on Graphics*, 1990;9(3):245–261.

[29] MacLeod, R.S., Johnson, C.R., Matheson, M.A., "Visualizing bioelectric fields", *IEEE CG&A*, 1993;13(4):10–12.

[30] Madi, M., Walton, D., "3D point location for special feature preservation during surface smoothing", *The 7-th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media'99–WSCG'99*, University of West Bohemia, Plzen, Czech Republic, Conf. Proc., Feb. 1999;1:173–179.

[31] Madi, M., Walton, D., "Modeling and visualization of layered objects", *Computers & Graphics*, 1999;23(3):331-342.

[32] Madi, M. and Walton D., "Interactive segmentation & shape modification of geometric 3D objects", *submitted*, May 1999.

[33] Madi, M. and Walton D., "From Hierarchical Structures to Triangular-Loop Structures: A Representation Transformation Algorithm", *To be submitted.*

[34] Mahony, D.B., "Visualizing volumes", *Computer Graphics World*, 1997;(20)7:42–48.

[35] Muraki, S., "Volume data and wavelet transforms", *IEEE CG&A*, 1993;13(4):50–56.

[36] Neamtu, M., Pfluger, P., "Degenerate polynomial patches of degree 4 and 5 used for geometrically smooth interpolation in $\mathbb{R}^3$", *Computer-Aided Geometric Design*, 1994;11:451–474.

[37] Neapolitan R., Naimipour, K., *Foundations of Algorithms*, D.C. Heath and Company, Lexington MA, 1996:260.

[38] Nelson, T.R. and Elvins, T.T., "Visualization of 3D ultrasound data", *IEEE CG&A*, 1993;15(6):50–57.

[39] Ng, W. and Tan, S., "An evaluation of recursive space subdivision scheme using different subdivision cells", *CSG'96, An Int. Conference on Set Theoretic Solid Modelling*, Winchester, UK, April 1996:125-139.

[40] Overveld C., Wyvill, B., "Phong normal interpolation revisited", *ACM Trans. on Graphics*, 1997;16(4):400–419.

[41] Pugh, D., "Designing solid objects using interactive sketch interpretation", *Computer Graphics*, 1992;25(2):117-126.

[42] Seiler, A., Balendran, V., Sivayoganathan, K., Sackfield, A., "Reverse engineering from unidirectional CMM scan data", *Int. J. Adv. Manuf. Technol.*, 1996;11:276–284.

[43] Scheepers, F., Parent, R., Carlson, W., May, S., "Anatomy-Based Modeling of the Human Musculature", *Computer Graphics*, 1997;31(3A):163-172..

[44] Sobierajski, L.M., Kaufman, A.E., "Volumetric ray tracing", *ACM Symp. on Volume Visualization*, Washington, Oct. 1994:11–18.

[45] Tarbuck, E., Lutgens, F., *Earth Science*, Seventh Ed., Macmillan College Publishing Company, NY, 1994.

[46] Treinish, L., Silver, D., "Visualization blackboard", *IEEE CG&A*, 1996;16(1):7–9.

[47] Udupa, J.K., Odhner, D., "Shell rendering", *IEEE CG&A*, 1993;13(6):58–67.

[48] Vannier, M.W., Commean, P.K., Brunsden, B.S., "Visualization of prosthesis fit in lower-limb amputees", *IEEE CG&A*, 1997;17(5):16–29.

[49] Vemuri, B., Cao, Y. Chen, L., "Fast collision detection algorithms with applications to particle flow", *Computer Graphics Forum*, 1998;17(2):121-134.

[50] Viewpoint Premier Catalog, Autumn 1998 Edition, more than 12,000 polygonized models, *Digital Viewpoint Inc*. On the WWW at http://www.viewpoint.com.

[51] Volino, P. and Thalmann, N., "Efficient self-collision detection on smoothly discretized surface animation using geometrical shape regularity", *Computer Graphics Forum* (Proc. Eurographics), Basil Blackwell Ltd., 1994;13(3):155–166.

[52] Volino, P., Thalmann, N.M., Jianhue, S., Thalmann, D., "An evolving system for simulating clothes on virtual actors" *IEEE CG&A*, 1996;16(5):42–51.

[53] Walton, D., Meek, S., "A triangular G^1 patch from boundary curves", *Computer-Aided Design*, 1996;28(2):113–123.

[54] Walton, D., Yeung, M., "G^1 refinement of data for rapid prototyping", *Int. Conf. on Manuf. Automation Proc.*, ICMA 97, Hong Kong, 1997;1:214–219.

[55] Weiler, K., "Edge-based data structures for solid modeling in curved-surface environments", *IEEE CG&A*, 1985;5(1):21–40.

[56] Wong, K., Siu, T., Heng, P., Sun, H., "Interactive volume cutting", *Proc. GI'98*, June 1998:99–106.