DESIGNING AND IMPLEMENTING A SURFACE MODELING SYSTEM

By

LE-JENG SHIUE

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2001

Copyright 2001

by

Le-Jeng Shiue

ACKNOWLEDGEMENTS

I owe my success in the research work and my career in computer science to my research advisor, Dr.Jörg Peters . I benefited from his vision, his broad and deep knowledge and his rich experiences in computer graphics.

I would also like to express my sincere gratitude to Dr. Paul A. Fishwick and Dr. Baba C. Vemuri for serving on my supervisory committee.

I am especially grateful to my parents for their emotional support and encouragement throughout my academic career.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vi

ABSTRACT viii

CHAPTERS

1 INTRODUCTION 1

1.1 Overview of Major Parametric Surface Representations 1
1.2 Overview of The Surface Modeling System 2
1.3 Contributions of The Thesis 4

2 BACKGROUND 5

2.1 Overview of CGAL 5
2.1.1 Halfedge Data Structure in CGAL 6
2.1.2 Representation of CGAL Polyhedron 8
2.2 Catmull-Clark Subdivision 9
2.3 Patching Catmull-Clark Meshes 13
2.3.1 Overview of PCCM 13
2.3.2 Patch Transformation 14

3 DESIGN OF THE REFINEMENT ALGORITHM 18

3.1 Design Issues 18
3.1.1 User Extended CGAL Polyhedron 18
3.1.2 Geometry Rules 18
3.1.3 Hierarchical Subdivision Polyhedron 19
3.2 Catmull-Clark Subdivision with CC scheme 23

4 DESIGN OF INTERACTIVE SURFACE MODELING 25

4.1 Design Issues 25
4.2 Quad Polyhedron—Halfedge Data Structure + Quad 26
4.2.1 Quad Map 26
4.2.2 Quad Polyhedron 27
4.3 Interactive Modeling with PCCM 32
4.3.1 Effect of Knot Insertion 35
4.3.2 Effect of Corner Smoothing 40
4.3.3 Algorithm for Local Updating 40

5 CONCLUSION AND FUTURE WORK 42

5.1 Conclusion 42
5.2 Future Work 46

REFERENCES 47

BIOGRAPHICAL SKETCH 48

LIST OF FIGURES

1.1 System diagram 3

2.1 Halfedge data structure 7

2.2 Structure of CGAL halfedge data structure. 7

2.3 Incidental operations of halfedges in the halfedge data structure. . . . 9

2.4 Structure of CGAL polyhedron. 10

2.5 Euler operations 10

2.6 Catmull-Clark subdivision 12

2.7 O-mesh 14

2.8 PCCM mesh 15

3.1 Mipmap polyhedron 20

3.2 Storage State 21

4.1 Quad polyhedron 29

4.2 Initialization of the quad polyhedron 31

4.3 Knot insertion 32

4.4 Relations between I-mesh, O-mesh and quad polyhderon 34

4.5 Interactive Modeler 36

4.6 Influence region 37

4.7 Influence region of knot insertion 39

5.1 Examples of the different steps of the surface modeling. 43

5.2 O-mesh and interactive modeling surfaces. 44

5.3 An example of modeling surfaces. 45

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master Of Science

DESIGNING AND IMPLEMENTING A SURFACE MODELING SYSTEM

By

Le-Jeng Shiue

May 2001

Chairman: Dr. Jörg Peters
Major Department: Computer and Information Science and Engineering

Subdivision surfaces and spline surfaces are widely used surface modeling techniques. Both techniques have some drawbacks when used in interactive modeling systems. Reevaluation of subdivision surfaces from start is prohibitively slow when updating small areas. For spline surfaces, the challenge is to maintain the smoothness across patches during the manipulation of control points of the individual patch.

In this thesis, I designed and implemented a new modeling system that supports interactive smooth free-form 3D modeling. The system uses the PCCM algorithm (Patching Catmull-Clark Meshes) to transform an intermediate mesh of Catmull-Clark subdivision to a parametric surface consisting of NURBS (Non-Uniform rational B-Spline) patches. The PCCM algorithm is extended to support interactive surface modeling and to update the surface instantly while still enforcing the smoothness of the surface.

This thesis also introduces a simple scheme for supporting the hierarchically subdivided polyhedron that serves as input to the PCCM algorithm and a new data

structure that makes implementation of PCCM more natural. In addition, the new data structure supports the local updates of PCCM surface and makes the implementation easy and efficient.

CHAPTER 1
INTRODUCTION

Applications such as special effects and animations require creation and manipulation of complex geometric models of arbitrary topology. These models can be created by digitization or complex modeling packages. To represent different scales of detail, these models are often composed of high resolution meshes. These high resolution meshes cause the difficulty of interactive modeling and user interface of manipulating of the models.

The contribution of this thesis is to create a new, simple and intuitive tool for a interactive modeling system. The focuses of the design of the system are the *creation of simple mesh*, *intuitive modeling operations* and *instant surface updating*.

1.1 Overview of Major Parametric Surface Representations

The popular techniques for modeling smooth parametric surface can be broadly classified into two distinct categories: *spline surfaces* and *subdivision surfaces*.

Spline surfaces usually consist of a set of smoothly joining individual parametric patches. When treated individually and manipulated via separate control meshes, enforcing the continuity across the patches becomes very complicated.

Subdivision surfaces are generated by recursively applying a set of mesh refinement rules. It can easily produce a tangent continuous limit surface. Users have to carefully select the initial mesh and manipulate the control points at different levels of the subdivision hierarchy since there is no direct and intuitive way to manipulate the limit surface.

PCCM(Patching Catmull-Clark Meshes)[5] provides another way to generate and represent the smooth and complex surface. The PCCM algorithm transforms intermediate meshes of subdivision surfaces to parametric surfaces that consist of smoothly joining NURBS (Non-Uniform rational B-Spline) patches. Chapter 2 provides the details of the algorithm and the representation.

1.2 Overview of The Surface Modeling System

Figure 1.1 illustrates our surface modeling system. The system is subdivided into five sub-systems.

1. **Mesh editor**: For an *initial mesh* of arbitrary shape and topology, users can modify it by using geometry and connectivity operations. This stage will output the *input mesh*.

2. **PCCM preprocessor**: To build the mesh with required connectivity for PCCM algorithm, the preprocessor first checks if the *input mesh* only consists of quadrilateral facets. If not, it refines the *input mesh* to become all rectangles by calling a general quadralization function or one step of the Catmull-Clark subdivision. The result mesh of this step is called *I-mesh*. The preprocessor then executes two steps of the Catmull-Clark subdivision generating the *O-mesh* as the input mesh for the PCCM algorithm.

3. **PCCM transformer**: Transform the *I-mesh* into the *PCCM mesh* consisting of the control nets of the parametric surfaces.

4. **Surface generator**: The surface generator evaluates the *PCCM mesh* as a set of NURBS patches and displays the resulting *PCCM surface*.

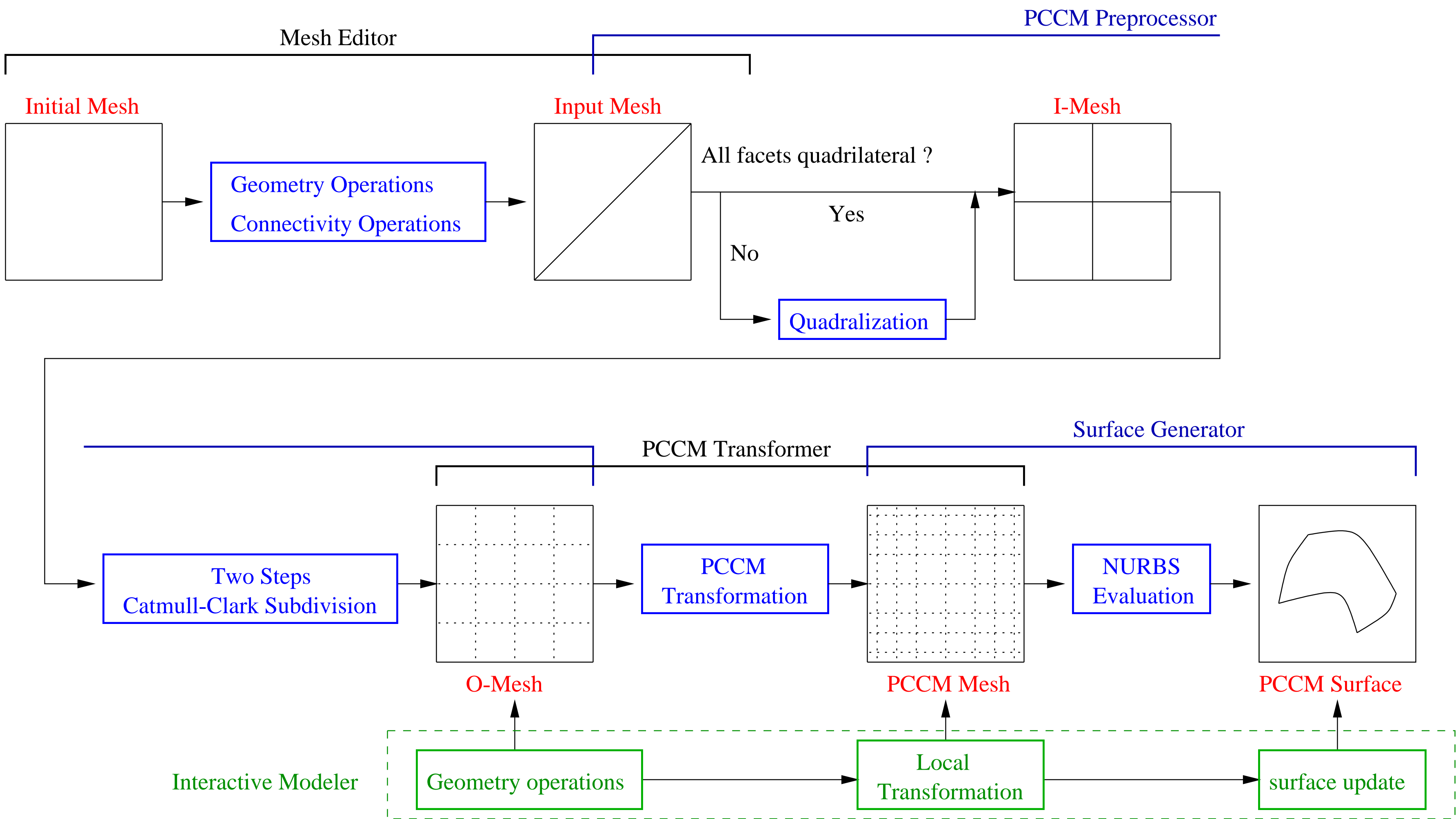


Figure 1.1: System diagram of the modeling system(see Figure 5.1 for instances).

5. **Interactive modeler**: While users manipulate the *O-mesh*, the interactive modeler defines the influence regions of the manipulated vertices, then it "re-PCCM"s these regions and locally updates the *PCCM mesh* and the *PCCM surface*.

1.3 Contributions of The Thesis

The objectives of this thesis are the study of the PCCM algorithm, the extension to the interactive modeling using PCCM and the extension of the CGAL library(see Chapter 2) to support the Catmull-Clark subdivision and the PCCM algorithm.

This thesis describes an encoding/decoding scheme for supporting the hierarchically subdivided polyhedron that can apply to any generic CGAL polyhderon. The hierarchical polyhderon also serves as the input to the PCCM algorithm and separates the algorithm and the data structure of refinement and transformation steps in PCCM. This thesis also introduces a new data structure, called quad polyhedron, and an interactive modeling scheme that make the local updating of the PCCM surface easy and efficient.

CHAPTER 2
BACKGROUND

In this thesis, the CGAL(Computational Geometry Algorithm Library) provides core data structures. Modeling algorithms include Catmull-Clark subdivision[1] and Patching Catmull-Clark Meshes(PCCM) algorithm [5]. This chapter gives a brief overview of the CGAL and details of the modeling algorithms.

2.1 Overview of CGAL

CGAL is a Computational Geometry Algorithm Library[6] that contains primitives, data structures, and algorithms for computational geometry. It was started six years ago in Europe in order to provide correct, efficient and reusable implementations.

CGAL consists of three layers[6, 4] and each layer is divided into smaller modular units. The **core library** consists of basic functionalities such as configuration, assertions, iterators and enumerations. The second layer, the **kernel library**, consists of constant size non-modifiable geometric primitives such as points and lines and operations on these primitives. The third layer, the **basic library**, consists of basic geometric data structures and algorithms. CGAL also has a support library which contains functionality with non-geometric aspects such as visualization, circulator, number types, etc.

The design goals for CGAL include robustness, generality, efficiency and ease-of-use. To fulfill these goals, developers of the CGAL opted for the *generic programming paradigm*[3] in most of the design of CGAL. Hence, *C++ templates*[7]—parameterized data types and functions—are heavily used and users have to specialize

(specify template parameters of the data types or functions) the CGAL data structures or algorithms with specific template arguments. After being specialized, CGAL classes can be used as normal C++ classes.

By replacing the template parameters, users can *horizontally* extend the classes of the CGAL library. Users can also *vertically* extend CGAL classes by inheriting the classes to build an object oriented classes hierarchy. There are two different ways of extending CGAL by inheritance. First, a new class can be inherited from a specialized class by treating it as a normal C++ class. Second, a new inheritated *template class* passes the template parameter(s) to the base CGAL template class.

In the implementation of our system, CGAL classes are extended both *horizontally* and *vertically* to support the modeling algorithms.

2.1.1 Halfedge Data Structure in CGAL

Halfedge data structure[8] is an *edge-centered* data structure which maintains the connectivity of the primitives—vertices, edges and facets (shown in Figure 2.1). Each edge is decomposed into two opposite oriented halfedges which store the pointers to the incident vertices and facets. Each facet is surrounded by a set of halfedges which are oriented counterclockwise as seen from the outside.

CGAL separates the connectivity and geometry in the halfedge data structure as well as the primitive classes which specialize the halfedge data structure. CGAL halfedge data structure acts as a container which stores and maintains the connectivity information and the primitives store both the connectivity and/or geometry information. Figure 2.2 illustrates the design. The CGAL halfedge data structure, acting as a black box, stores three *lists* or *vectors*—depending on the type of the *storage organization* users have chosen—of the primitives and maintains the incident relation of the halfedge. On the other hand, the primitive bases are *replaceable* template parameters and can carry the geometry data, user defined data or even

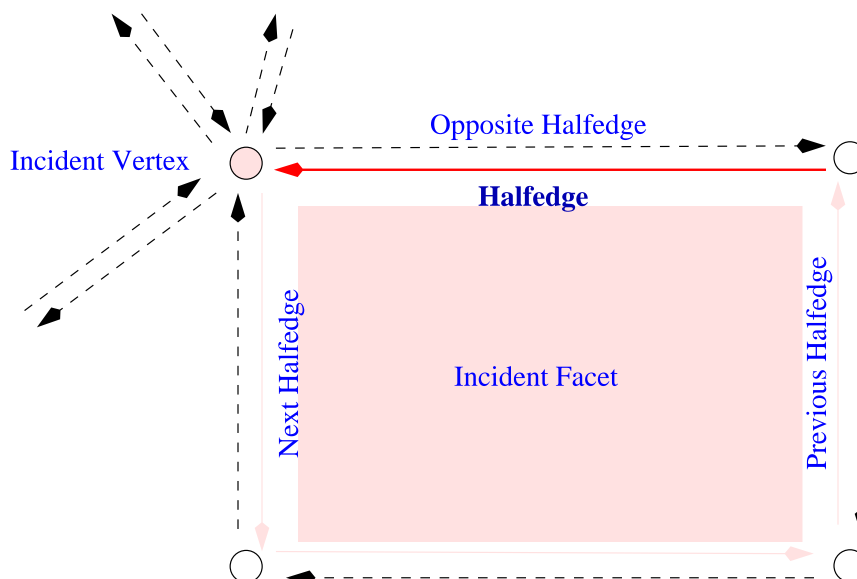


Figure 2.1: In a halfedge data structure, each halfedge has five incident primitives. Users can traverse the mesh by using these incident relations.

void pointer. Users can define their own primitive bases either by inheriting primitive bases from CGAL or by building new classes that comply with the functional requirements of the halfedge data structure.

CGAL currently supports two types of halfedge data structure with different internal storage organizations. One is a *doubly-linked list* and the other one is an *STL std::vector*[3]. A halfedge data structure using *list* has dynamic capacity and hence can easily add or remove the primitives. On the other side, a halfedge data structure

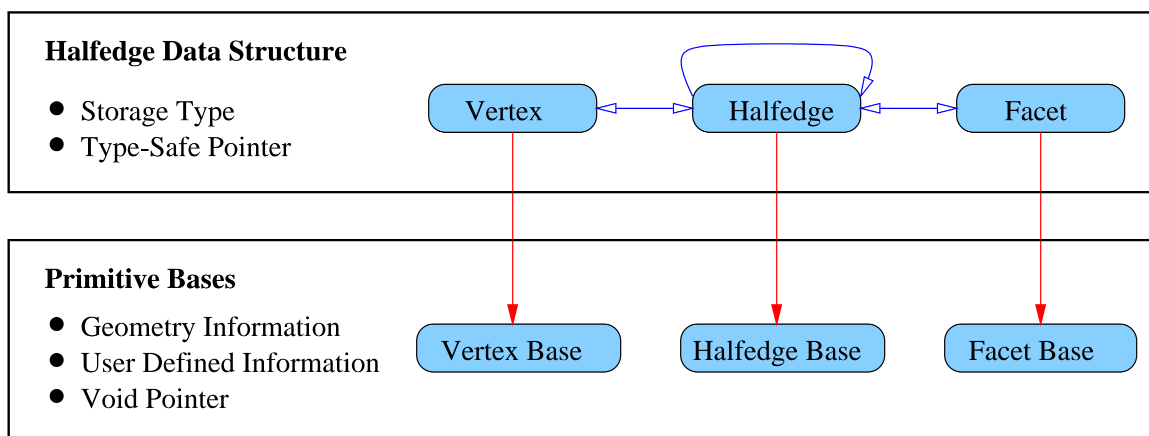


Figure 2.2: Structure of CGAL halfedge data structure.

using *vector* is efficient for pre-defined size of the primitives yet is very slow at adding or removing any primitives after the structure is created. Since vector structure does not use extra space to store pointer to the next element and can randomly access the elements in constant time, using vector is usually more efficient than using list on aspect of time and space. Both types of the halfedge data structures always add new primitives in the end of the internal storage structure.

CGAL provides a set of primitive base classes for different level of functional requirements. Users can directly use these bases or adapt them to fulfill some special demands and functionalities. For example, users can instantiate a halfedge data structure that has only connectivity information by using CGAL::Vertex_min_base, CGAL::Halfedge_min_base and CGAL::Facet_min_base in which no geometry data are stored. Or, users can create a new vertex class, say with color, by inheriting CGAL::Vertex_min_base and adding a new member variable as the color tag.

In CGAL halfedge data structure, operations are provided to access the primitives and to traverse the data structure. Although iterator and circulator are the primal interfaces of these operations, users can directly use the incidental operations of the halfedge to access and traverse the primitives of the data structure (shown in Figure 2.3).

2.1.2 Representation of CGAL Polyhedron

A polyhedron is a 3-dimensional solid bounded by not necessarily planar facets. A polyhedron surface is a set of connected facets that bound a polyhedron. For simplicity, we will use polyhedron instead of polyhedron surface throughout the thesis.

A polyhedron in CGAL is based on the halfedge data structure plus the geometric constraints, internal data protections and high level operations (see Figure 2.4). CGAL polyhedron is also a template class of the halfedge data structure with specific storage model—*vector* or *linked list*. To maintain combinatorial consistency,

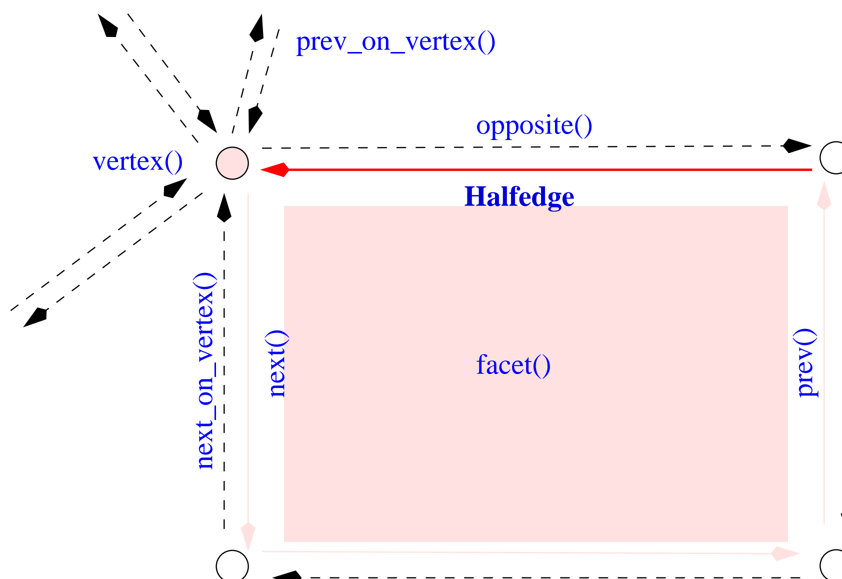


Figure 2.3: Incidental operations of halfedges in the halfedge data structure.

a CGAL polyhedron only provides Euler operations as the modification interface (a subset is shown in Figure 2.5).

2.2 Catmull-Clark Subdivision

Catmull-Clark subdivision[1] is an algorithm for *recursively* generating limit surfaces. In each pass, the algorithm generates more regular vertices, which are valence 4, and consequently approaches the limit surface. For regular meshes that have only regular vertices, the algorithm generates a standard B-spline surface with C^2 continuity. For irregular meshes, it generates a sequence of standard B-spline surface rings with C^2 continuity except at the extraordinary vertices.

After each pass of the Catmull-Clark subdivision, the result mesh is formed by a new set of vertices. Classified by the subdivision rules and the connectivity relation with the previous mesh, there are three types of the vertices in the refined mesh: facet vertices, edge vertices, and vertex vertices.

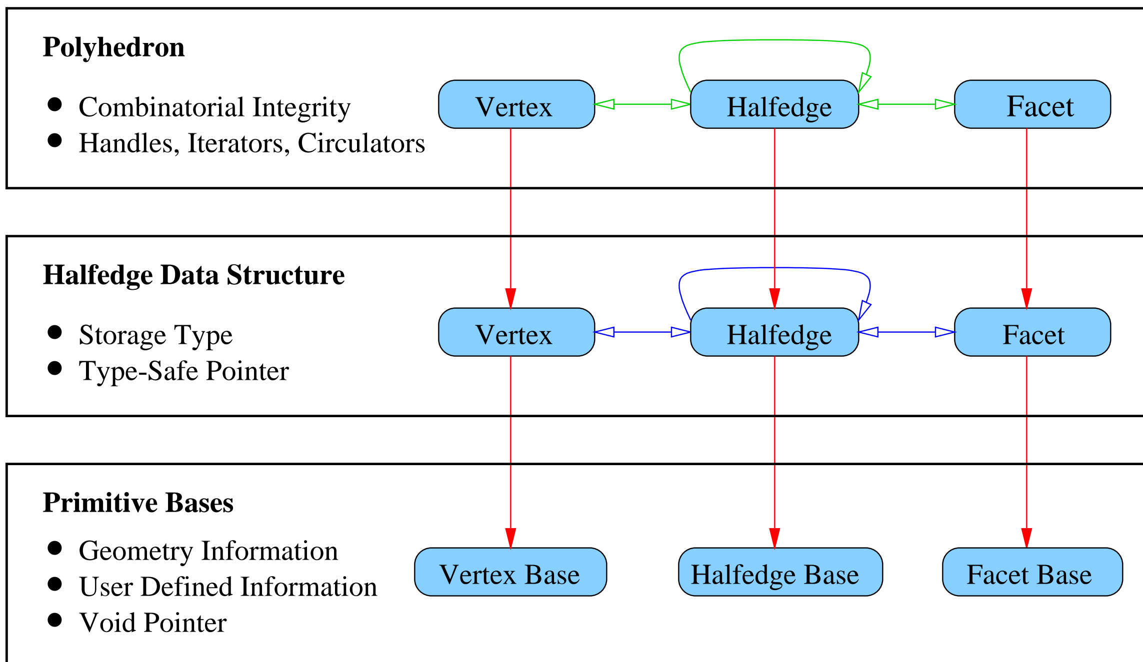


Figure 2.4: Structure of CGAL polyhedron.

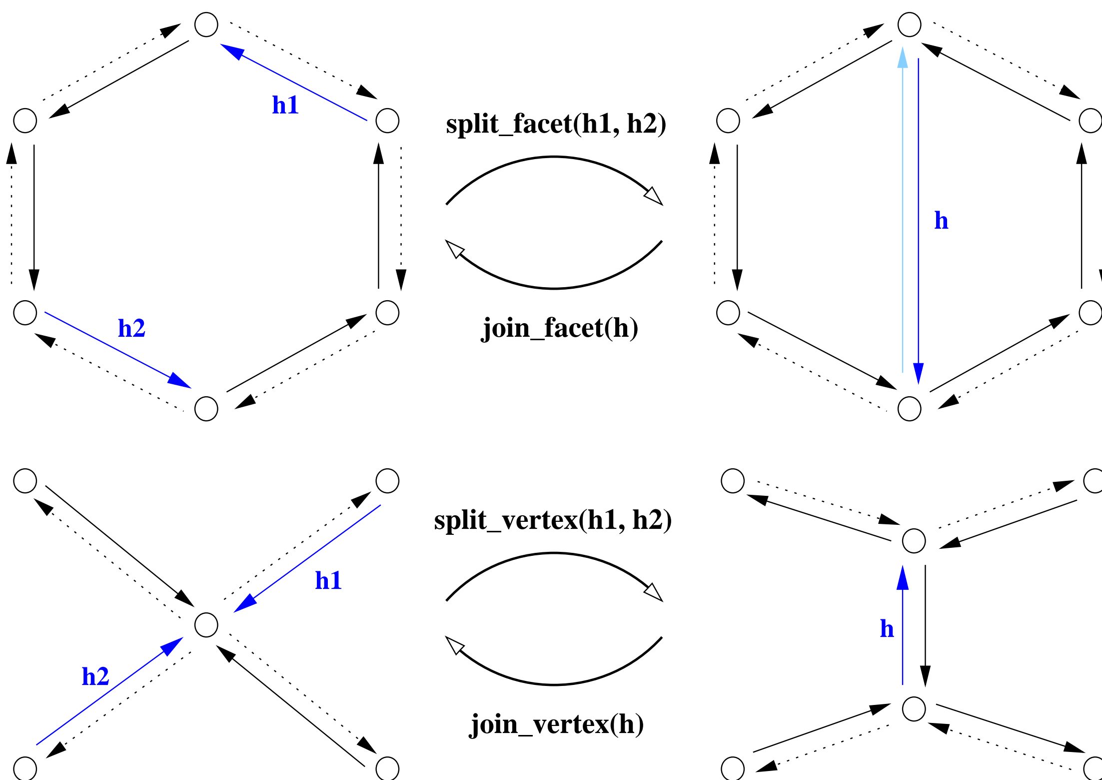


Figure 2.5: Only four Euler operations are shown here

Denote the input mesh as *0-mesh* and the mesh after the *d-th* pass be *d-mesh*. To generate the *d-mesh*, the *0-mesh* needs to be refined d times generating the new geometry data and build the refined connectivity.

To generate the geometry data of the vertices of d^+-*mesh*, which is refined from *d-mesh*, Catmull-Clark subdivision has the following rules shown in Figure 2.6:

- **Facet-vertex** FV_{d^+} = the average of all vertices surrounding the corresponding facets F_d of *d-mesh*.
- **Edge-vertex** EV_{d^+} = the average of the midpoints of the corresponding edges E_d of *d-mesh* with the average of the two facet vertices of d^+-*mesh* whose corresponding facets share the corresponding edges.
- **Vertex-vertex**

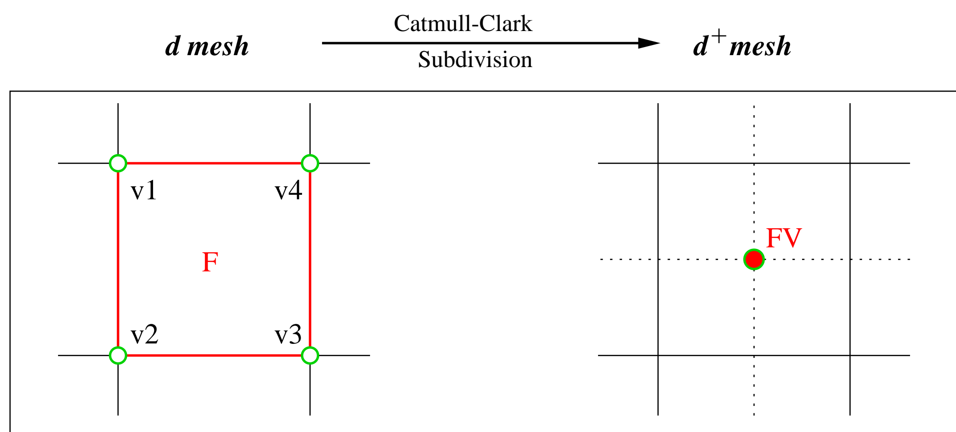
  $$VV_{d^+} = \frac{Q}{n} + \frac{2R}{n} + \frac{S(n-3)}{n}$$

  where

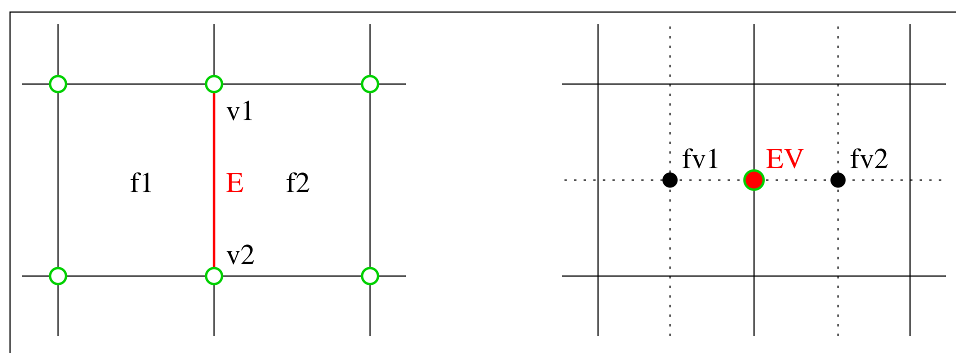
  - Q = the average of the facet-vertices of all facets adjacent to the vertex V.
  - R = the average of the midpoints of all edges incident on the vertex V.
  - S = the corresponding vertex V.
  - n = the valence of the corresponding vertex V.

To build the connectivity of d^+-*mesh*, Catmull-Clark subdivision has the following rules:

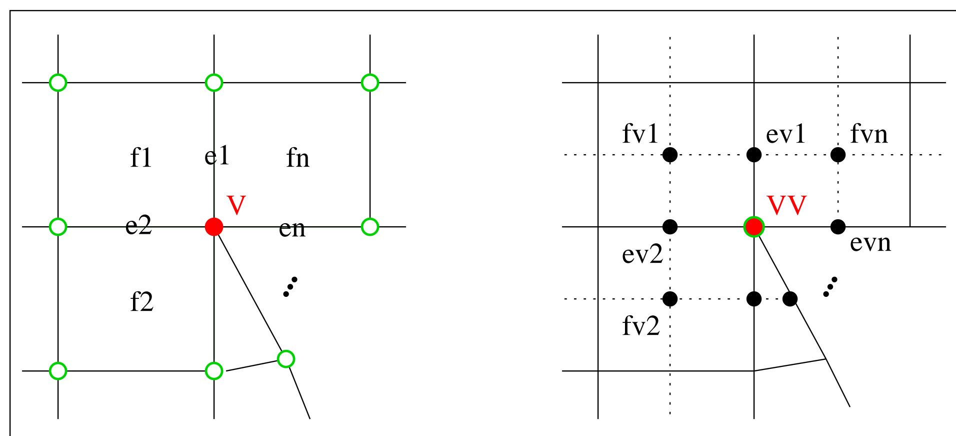
- Connect each facet-vertex to the edge-vertices of the edges surrounding the facet.
- Connect each vertex-vertex to the edge-vertices of the incident edges of the vertex.



(a) Facet-vertex rule



(b) Edge-vertex rule



(c) Vertex-vertex rule

Geometry data
Newly generated vertex
Newly generated edge

Figure 2.6: Catmull-Clark subdivision.

2.3 Patching Catmull-Clark Meshes

2.3.1 Overview of PCCM

Patching Catmull-Clark Meshes(PCCM) [5] creates a set of large, smoothly joining bicubic NURBS patches from an input mesh consisting of all four-sided facets. The PCCM algorithm transforms each facet of the input mesh to a bicubic NURBS patch. The patches forming the output surface join parametrically C^2 except at the extraordinary patch corners where they join with C^1 continuity.

The PCCM algorithm consists two procedures of generating the PCCM mesh from the input mesh. **Mesh refinement**, the first procedure, refines the input mesh, called **I-mesh** in the modeling system, to an **O-mesh**. An O-mesh consists a set of sub-meshes, called **qmesh**es, that one-to-one map to the facets of the I-mesh(see Figure 2.7). Each qmesh can be transformed to a quad—an array containing $(2i+1) \times (2i+1)$ entries, where $i \geq 2$. These array entries store the positions of the vertices of the corresponding qmesh. To generate the O-mesh, we use Catmull-Clark subdivision as the refinement algorithm. The number of the passes of the subdivision, denoted as i in previous equation, defines the size of the quad arrays. Since the **patch transformation**, the second procedure of the PCCM algorithm, accepts the quad arrays with arbitrary geometry data, the geometry rules of the Catmull-Clark subdivision are not mandatory and users can directly manipulate the vertices of the O-mesh.

Patch transformation uses a **quad structure** initialized from *O-mesh* as the input data to generate the control nets of the patches forming the PCCM surface and ensures the boundary continuity across the patches. The transformation consists of two steps—**knot insertion** and **corner smoothing**. The first step ensures C^2 continuity across the boundary of the patches except for extraordinary corners. The second step ensures the C^1 continuity around the extraordinary corners. After patch

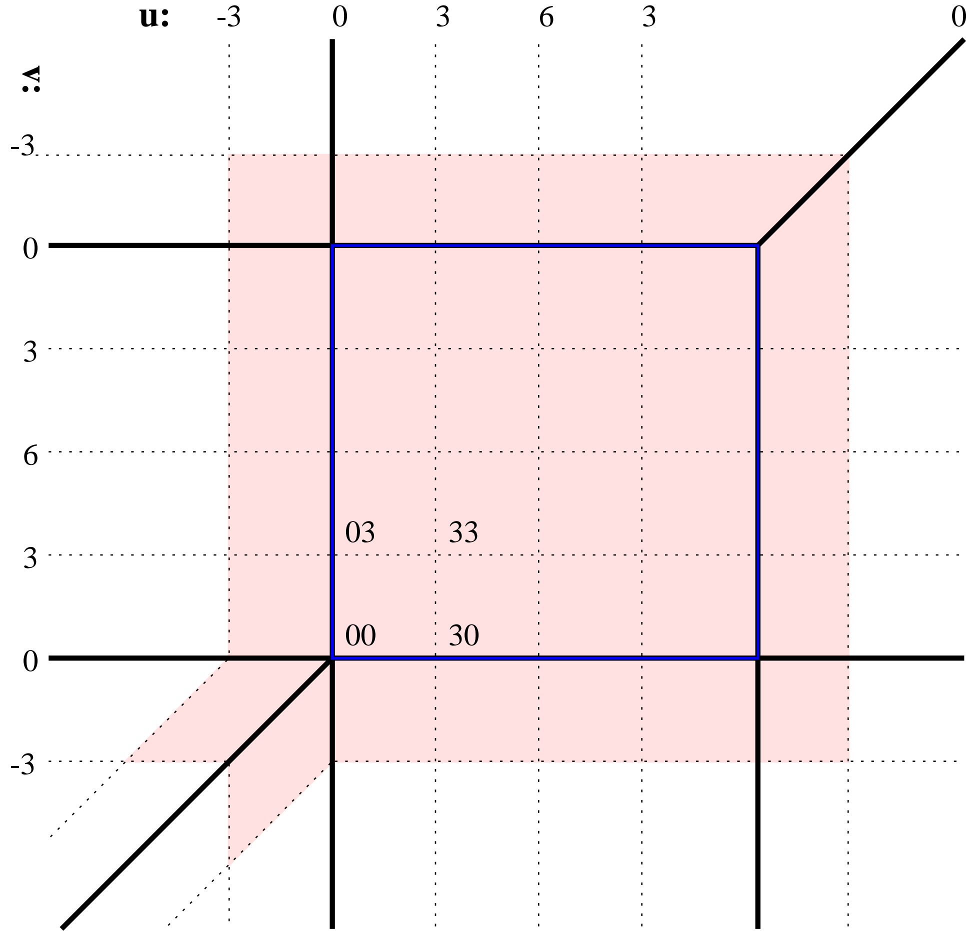


Figure 2.7: The O-mesh consists of a set of connected qmeshes(shaded area). Each qmesh subdivides an I-Mesh facet(blue rectangle). The I-mesh corresponds to heavy black lines. The initial data of the quad array corresponds to the positions in the qmesh. A quad array contains $(2i+1)\times(2i+1)$ control points indexed by the Greville abscissa which are scaled by 3.

transformation, the quad array contains $(2i + 5) \times (2i + 5)$ NURBS control points and would be input to `gluNurbsSurface`[9].

2.3.2 Patch Transformation

After refinement of the I-mesh, we will have the O-mesh structured as shown in Figure 2.7. Each quad, or **P-mesh**, corresponds to *a facet* in the I-mesh, and is a sub-mesh of the O-mesh. The quads surrounding a corner point, a vertex in the I-mesh, are arranged in *counterclockwise* order indexed by i, and the nodes $P_{\mathtt{uv}}(i)$ is the `uv` node of ith quad. The double subscripts uv are the Greville abscissa, scaled

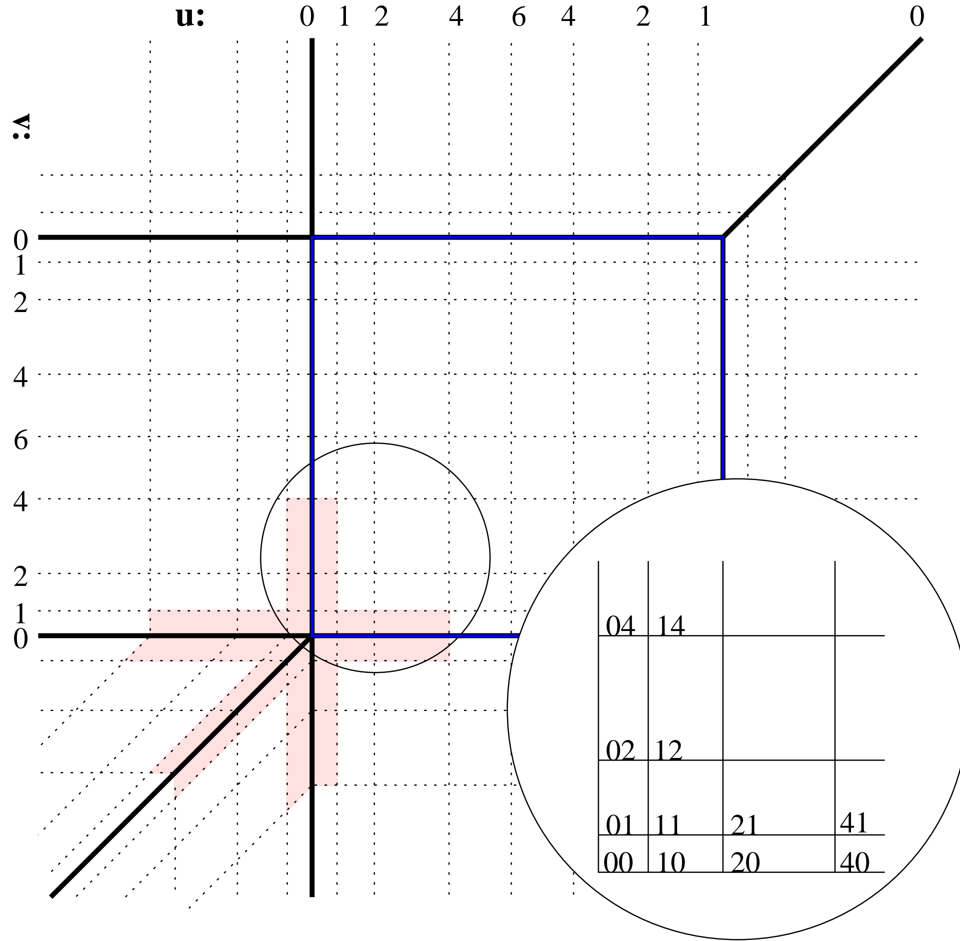


Figure 2.8: PCCM mesh: the enlargement shows the indices of the control points relevant for corner smoothing. Different corners have different orientation which is significant to the indices. The shaded area indicates the control points used and modified for corner smoothing.

by 3, of the PCCM surface. After the knot insertion of the PCCM transformation, the refined mesh, called **Q-mesh**, looks like in Figure 2.8. The same indexing of the P-mesh is used for the Q-mesh except it is denoted by $Q_{\mathtt{uv}}(i)$ and the double subscripts uv are no more scaled by 3. The mesh without extraordinary vertex will stay unchanged after the corner smoothing, which is mandatory only when the corners of the quad are extraordinary vertices.

After the transformation, the result mesh is called **PCCM mesh** It can be evaluated as the *control points* of a set of *NURBS patches*. These NURBS patches

form the PCCM surface and are C^2 continuity everywhere except at the extraordinary patch corners where are C^1 continuity.

Required O-mesh Connectivity

An O-mesh consists of a set of *chess board* sub-meshes of equal size that contain at least (5×5) vertices. These sub-meshes should only intersect to each other on the boundary and cover the whole O-mesh. Each sub-mesh plus one ring of the facets surround it corresponds to a quad array in the PCCM mesh. Figure 2.7 shows an example of an O-mesh.

Knot Insertion

For each quad, we initialize the array with the qmesh which is one ring larger than the sub-mesh inside the corresponding facet boundary: we borrow one layer of nodes from all direct and diagonal neighbor sub-mesh(shown in Figure 2.7).

To make each quad into standard interpolating form of an order 4 NURBS patch with uniform knots sequence, we insert three knots for Q_1, Q_2, and Q_4—first at u direction and then insert other three knots at v direction. The new points Q_{ul} are obtained from the old points P_{ul} via

$$\begin{aligned} Q_0 &= (P_{-3} + 4P_0 + P_3)/6 \\ Q_1 &= (2P_0 + P_3)/3 \\ Q_2 &= (P_0 + 2P_3)/3 \\ Q_4 &= (2P_3 + P_6)/3 \end{aligned}$$

All remaining $Q_{3i,3j} = P_{3i,3j}$, except for the corner point Q_{00} which we place directly on the Catmull-Clark limit surface:

$$Q_{00} = \frac{\sum nP_{00}(i) + 4P_{30}(i) + P_{33}(i)}{n(n+5)} \text{ where } n \text{ is the valence of the corner point}$$

Corner Smoothing

To obtain the tangent continuity instead of position continuity at the extraordinary corners, we need to modify the control points at the corners. The enlargement of Figure 2.8 shows the relevant double subscripts of Q-mesh after knot insertion.

We define, for every n, two n by n matrices A_n and B_n with rows $i = 1, \ldots, n$ and columns $j = 1, \ldots, n$ and entries

$$A_n(i,j) = \frac{2a}{n}\cos(\frac{2\pi}{n}(i-j)) \qquad a = 1(\text{default}) \text{ and}$$
$$B_n(i,j) = \begin{cases} (-1)^{n_{i-j}} & \text{if } n \text{ is odd,} \\ (-1)^j - 2n_{i-j}(-1)^{j-i}/n & \text{if } n \text{ is even,} \end{cases}$$
$$n_{i-j} = \mod(n+i-j, n).$$

We collect the points $Q_{\mathtt{uv}}(i) \in \mathbf{R}^3$ generated by Knot Insertion for $i = 1, \ldots, n$ and $\mathtt{uv} \in \{00, 10, 20, 40\}$ into $\bar{Q}_{\mathtt{uv}} \in \mathbf{R}^{n\times 3}$. Only if n is even and greater than 4, do we compute $r = \sum_{i=1}^{n}(-1)^i\bar{Q}_{40}(i)/n$ and if $r \neq 0$ we add, for each i, $h_i = -(-1)^i r$ to $Q_{40}(i) = Q_{04}(i-1)$, $Q_{41}(i)$ and $Q_{14}(i-1)$ so that $\sum_{i=1}^{n}(-1)^i Q_{40}(i) = 0$ and $Q_{40}(i) = (Q_{41}(i) + Q_{14}(i-1))/2$. Otherwise all coefficients remain unchanged except

$$Q_{10} = Q_{00} + A_n\bar{Q}_{10},$$
$$Q_{20} = (Q_{40} + 6Q_{10} - 2Q_{00})/5,$$
$$Q_{11} = B_n\Big(Q_{10} + \frac{\cos(2\pi/n)}{6}(Q_{40} - Q_{20})\Big).$$

For $i = 1, \ldots, n$, we copy $Q_{\mathtt{v}0}(i+1) = Q_{0\mathtt{v}}(i)$ for $\mathtt{v} \in \{1, 2, 4\}$ and add $Q_{20}(i) - \bar{Q}_{20}(i)$ to $Q_{21}(i)$ and $Q_{12}(i-1)$.

CHAPTER 3
DESIGN OF THE REFINEMENT ALGORITHM

To build the required O-mesh connectivity, we use the Catmull-Clark subdivision as the refinement algorithm since it is widely implemented and used in current modeling systems. My design separates the connectivity and geometry rules in order to add more flexibility of changing the geometry rules dynamically.

3.1 Design Issues

There are three goals of the design. First, the subdivision function must be able to apply to any user extended CGAL polyhedron. Second, the user can change the geometry rules of the subdivision by simply replacing the rule object. Third, the subdivision function must support a hierarchical polyhedron that allows tracing the relation of the primitives in different layers of the subdivided polyhedron.

3.1.1 User Extended CGAL Polyhedron

To fulfill this goal, the subdivision is implemented as a *template class* in C++. Users can instantiate the subdivision class with their own specialized polyhedron class as the *template parameter*. The only requirement on the user extended polyhedron is that it must support a subset of *Euler operations* with the same interface(function names) as them in the CGAL(see Figure 2.5).

3.1.2 Geometry Rules

To give the users more control of the shape of the O-mesh and PCCM mesh, users are allowed to extend the subdivision with different geometry rules. By plugging new geometry rules classes when calling the subdivision function, users can easily

add more feature controls on the O-mesh. For example, blend ratio which controls the edge on the surface is one good feature that users can add to the subdivision implementation.

To achieve this goal, I template the subdivision functions of the subdivision class. A generic subdivision function—called `quadralize_polyhedron` in our implementation—in the subdivision class accepts a `RULE` class as the geometry rules. Two rule classes are currently supported. One is the *average rule*—every new generated vertex is the average of all its incident vertices in the previous layer. The other is the *standard Catmull-Clark rule* that I mentioned in the previous chapter. Both rule classes provide three operations—*vertex-vertex rule*, *edge-vertex rule* and *facet-vertex rule*. These three operations are mandatory for all user defined classes of the geometry rules.

3.1.3 Hierarchical Subdivision Polyhedron

For PCCM, users need to collect the geometry data of the vertices into their corresponding quad (see next chapter) before transforming the patches. To collect the data, the system need some functions to trace back the hierarchical relation of the primitives between different layers of the subdivision polyhedron. For instance, given a vertex of layer i, the tracing function should return the corresponding vertex of layer $i+2$. By building a *polyhedron container*, called **mipmap polyhedron**, which stores the subdivided polyhedron through the subdivision process, we can *implicitly* make the tractability available by storing the primitives in a specific order.

A mipmap polyhedron is a simple *vector* which stores the sequence of the subdivided polyhedrons(see Figure 3.1). Every time the mipmap polyhedron gets subdivided, it adds a new layer—the subdivided polyhedron from previous layer—at the end of the vector. Hence, for an *n-times* subdivided mipmap polyhedron, it has $n+1$ *layers* stored in the vector and the *initial polyhedron* is at *layer* 0. To support tracing operations, we can build the tracing table that *explicitly* registers the primitive

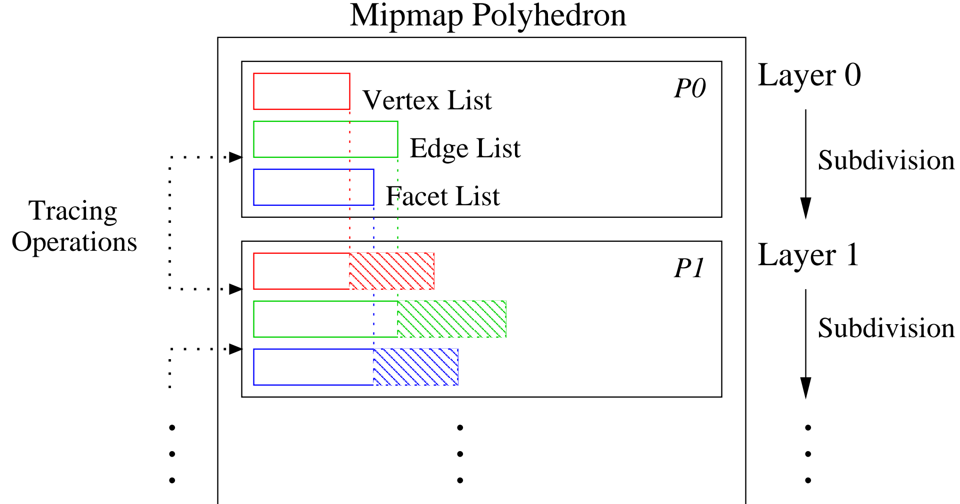


Figure 3.1: A mipmap polyhedron consists of a list of polyhedrons starting with the input polyhedron P_0. A polyhedron P_{i+1} is subdivided from the polyhedron P_i in the mipmap polyhedron. Each polyhedron contains three primitive lists. The shaded areas in the primitive lists indicate the newly generated primitives by the subdivision.

relation for every two layers in the mipmap polyhedron. Or we can *implicitly* decode the tracing information by encoding the newly generated primitives with a specific storage scheme. Considering the saving of memory space and slightly sacrificing the speed, I chose the *implicit scheme* as the solution to support the tracing operations.

The implicit scheme requires a *specific storage order* for initial and newly generated primitives. In any user extended CGAL polyhedron, these primitives are stored in three separate lists or vectors (vectors have speed advantage over lists)—vertex list, edge list (edge is represented by 2 consequently stored halfedges) and facet list(see Figure 3.2(a)). After one step of Catmull-Clark subdivision, all the newly generated primitives will be added at the *end* of each list and have the same storage order as shown in the Figure 3.2(b). We call this storage scheme **CC scheme** since it is specific to Catmull-Clark subdivision.

Definition 3.1 *A* ***primitive list*** *can be represented as a string of indices of the primitives. For the polyhedron of layer l in the mipmap polyhedron, VL_l, EL_l, and FL_l are the vertex, edge and facet list of the polyhedron and $\#v_l$, $\#e_l$ and $\#f_l$ are*

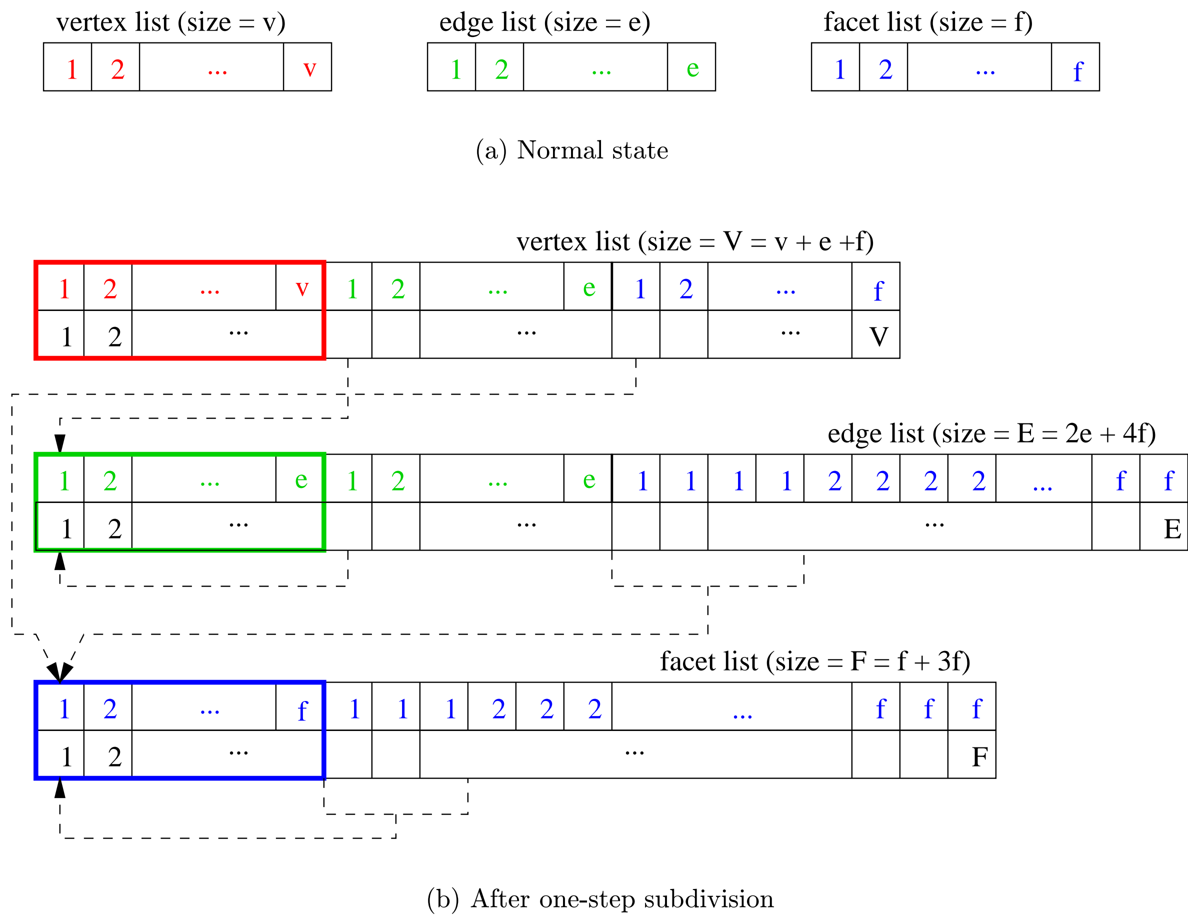


Figure 3.2: The storage state of the primitive lists.

Table 3.1: Tracing operations: the input of the tracing operations is the index of the primitive and the output is the index of the corresponding primitive or a set of the primitives.

Up-Tracing	Down-Tracing
$up_{vv}(v_l) \rightarrow v_{l+1}$	$down_{vv}(v_l) \rightarrow v_{l-1}$ $down_{ve}(v_l) \rightarrow e_{l-1}$ $down_{vf}(v_l) \rightarrow f_{l-1}$
$up_{ev}(e_l) \rightarrow v_{l+1}$ $up_{ee}(e_l) \rightarrow \{e_{l+1}, e'_{l+1}\}$	$down_{ee}(e_l) \rightarrow e_{l-1}$ $down_{ef}(e_l) \rightarrow f_{l-1}$
$up_{fv}(f_l) \rightarrow v_{l+1}$ $up_{fe}(f_l) \rightarrow \{e_{l+1}, e_{l+1}+1, e_{l+1}+2, e_{l+1}+3\}$ $up_{ff}(f_l) \rightarrow \{f_{l+1}, f'_{l+1}, f'_{l+1}+1, f'_{l+1}+2\}$	$down_{ff}(f_l) \rightarrow f_{l-1}$

*the size of each list. A primitive list with **CC scheme** has following storage state with the tracing operations defined in Table 3.1,*

$$AL_l = \{v_1 v_2 ... v_{\#v_l}\} \rightarrow^{subdivision} VL_{l+1} = VL_l \circ EL_l \circ FL_l$$

$$EL_l = \{e_1 e_2 ... e_{\#e_l}\} \rightarrow^{subdivision} EL_{l+1} = EL_l \circ EL_l \circ FL_l^4$$

$$FL_l = \{f_1 f_2 ... f_{\#f_l}\} \rightarrow^{subdivision} FL_{l+1} = FL_l \circ FL_l^3$$

$$\text{where } FL_l^n = \{\underbrace{f_0 f_0 ... f_0}_{\text{n}} f_1 ... f_{\#f_l}\}$$

$$\circ \text{ is the string concatenation.}$$

Definition 3.2 *If a polyhedron P_s subdivided from a polyhedron P stores the primitive lists complying CC scheme, P_s and P are **CC scheme compatible** and have complete tracing relations between them.*

For a mipmap polyhedron of depth L, we define the tracing operations for layer l as '$trac(p_l) \rightarrow p_{l+1}$' and '$trac(p_l) \rightarrow p_{l-1}$' where p_l is the primitive index of layer l and $0 \leq l \leq L$.

There are 2 categories of the tracing operation—**up-tracing** and **down-tracing**—and each category has six operations (see Table 3.1).

1. Up-tracing operations

(a) $\mathbf{up_{vv}}(v_l) \to v_{l+1} : v_{l+1} = v_l$

(b) $\mathbf{up_{ev}}(e_l) \to v_{l+1} : v_{l+1} = e_l + \#v_l$

(c) $\mathbf{up_{ee}}(e_l) \to \{e_{l+1}, e'_{l+1}\} : e_{l+1} = e_l, e'_{l+1} = e_l + \#e_l$

(d) $\mathbf{up_{fv}}(f_l) \to v_{l+1} : v_{l+1} = f_l + \#v_l + \#e_l$

(e) $\mathbf{up_{fe}}(f_l) \to \{e_{l+1}, e_{l+1} + 1, e_{l+1} + 2, e_{l+1} + 3\} : e_{l+1} = e_l + \#e_l + \#e_l$

(f) $\mathbf{up_{ff}}(f_l) \to \{f_{l+1}, f'_{l+1}, f'_{l+1} + 1, f'_{l+1} + 2\} : f_{l+1} = f_l, f'_{l+1} = f_l + \#f_l$

2. Down-tracing operations

(a) $\mathbf{down_{vv}}(v_l) \to v_{l-1} : v_{l-1} = v_l$

(b) $\mathbf{down_{ve}}(v_l) \to e_{l-1} : e_{l-1} = v_l - \#v_{l-1}$

(c) $\mathbf{down_{vf}}(v_l) \to f_{l-1} : f_{l-1} = v_l - \#v_{l-1} - \#e_{l-1}$

(d) $\mathbf{down_{ee}}(e_l) \to e_{l-1} : e_{l-1} = \begin{cases} e_l, & \text{if } e_l \leq \#e_{l-1} \\ e_l - \#e_{l-1}, & \text{if } e_l > \#e_{l-1} \end{cases}$

(e) $\mathbf{down_{ef}}(e_l) \to f_{l-1} : f_{l-1} = (e_l - \#e_{l-1} - \#e_{l-1})/4 + 1$

(f) $\mathbf{down_{ff}}(f_l) \to f_{l-1} : f_{l-1} = \begin{cases} f_l, & \text{if } f_l \leq \#f_{l-1} \\ (f_l - \#f_{l-1})/3 + 1, & \text{if } f_l > \#f_{l-1} \end{cases}$

3.2 Catmull-Clark Subdivision with CC scheme

To make the subdivided polyhedron CC scheme compatible in the modeling system, the subdivision algorithm first *copys* the initial polyhedron P_l to the subdivided polyhedron P_{l+1} and preserves the index order of the primitives. This step ensures

$$\begin{cases} VL_{l+1} = VL_l \\ EL_{l+1} = EL_l \\ FL_{l+1} = FL_l \end{cases}$$

Then, following the sequence of the edges stored in the edge list, add an *edge vertex* on each edge. Adding the edge vertex on a edge is equivalent to splitting the edge into two edges. Hence, the new edge is one-to-one mapped to the split edge. So far

the lists become

$$\begin{cases} VL_{l+1} = VL_l \circ EL_l \\ EL_{l+1} = EL_l \circ EL_l \\ FL_{l+1} = FL_l \end{cases}$$

Next step, the algorithm *quadralize* the facets following the sequence of the facets stored in the list. For each facet, it randomly select two neighbor edge-vertices and connect them with a new edge. On this new edge, it add a new facet-vertex and then connect the new vertex to all remaining edge-vertices around this facet. After this step, the primitive lists have one new vertex, four new edges and three new facets stored with facets storing order. Now, the lists become

$$\begin{cases} VL_{l+1} = VL_l \circ EL_l \circ FL_l \\ EL_{l+1} = EL_l \circ EL_l \circ FL_l^4 \\ FL_{l+1} = FL_l \circ FL_l^3 \end{cases}$$

and all the CC scheme operations are well defined.

Algorithm 1 shows the pseudo code for this procedure. `addVertex()` and `addEdge()` can be implemented by using the Euler operations described in Figure 2.5.

Algorithm 1 Catmull-Clark subdivision with CC scheme

Require: input polyhedron ($\mathbf{P_{l-1}}$)
Require: output polyhedron ($\mathbf{P_l}$)
Ensure: CC scheme primitive lists ($\mathbf{VL_l}$, $\mathbf{EL_l}$ and $\mathbf{FL_l}$) in $\mathbf{P_l}$

$\mathbf{P_l} = \mathbf{P_{l-1}}$
for $i = 1$ to size(EL_l) **do**
  addVertex($EL_l[i]$)
end for
for $i = 1$ to size(FL_l) **do**
  vertex $v1$, $v2$ = randomly selected edge vertices of $FL_l[i]$
  edge e = addEdge($v1$, $v2$)
  vertex v = addVertex(e)
  addEdge(v, all remaining edge vertices)
end for

CHAPTER 4
DESIGN OF INTERACTIVE SURFACE MODELING

For an interactive modeling system, real time feedback is the basic requirement. To fulfill this requirement, the system needs a surface representation that not only provides a smooth surface but also can locally update a deformed region of the surface. In this chapter, I will introduce a new data structure—quad polyhedron—to be the surface representation and illustrate how to implement the PCCM algorithm and the local update of the surface using the new data structure.

4.1 Design Issues

PCCM surfacees consist of a set of smoothly joining NURBS patches; hence, users can manipulate the control points of the meshes that reprented the PCCM surfaces to sculpture the surface. But this direct manipulation on the control points might destroy the smoothness condition across the boundaries of the patches. Instead of manipulating the PCCM mesh, the system will only allow users to manipulate the *O-mesh* and "re-PCCM" the *O-mesh* in order to enforce the smoothness conditions. To acheive the real-time feedback for an interactive system, we cannot afford to re-PCCM the complete O-mesh. Hence, the update region decided by the manipulation region must be restricted as small as possible yet still maintains the smoothness condition across the boundaries.

The goals of my design will be as follows,

1. **Maintain the smoothness condition**: the patches should join parametrically C^2 everywhere except at the extraordinary patch corners where they should join with C^1 continuity.

2. **Minimize the update region**: only re-PCCM and update the smallest region that maintains the smoothness conditions.

4.2 Quad Polyhedron—Halfedge Data Structure + Quad

The PCCM algorithm can be naturally implemented with an *array-based* data structure—the **quad map**[5]. But considering the requirements of the interactive system and the deficiencies of quad map listed below, an extended data structure—the **quad polyhedron**— will be introduced.

4.2.1 Quad Map

A **quad map** contains a *quad list* storing the geometry data and a *vertex list* supporting the connectivity of the represented mesh. Each quad in the quad list stores a $(2^i + 5) \times (2^i + 5)$ array, where i is the number of passes of Catmull-Clark subdivision and $i \geq 2$, containing the geometry data of the vertices. The state of valid entries of the array will be varied along the PCCM transformation(see Figure 4.3). Each entry in the vertex list stores the indices of incident quads and the corner index of the incident quad corresponding to this entry.

A quad map can be used to implement the Catmull-Clark subdivisionand then as the input of the PCCM transformation. The initial state of each quad contains a 4×4 array centered in the quad array. The initial array maps to the sub-mesh centered to the facet of the input mesh which has only four-sided facets. After i-times subdivision, the size of array grows up to $(2^i + 3) \times (2^i + 3)$ and has the valid data to proceed PCCM transformation. By enlarging the array to $(2^i + 5) \times (2^i + 5)$ in order to reserve the space for newly inserted knots, the structure is ready for PCCM.

PCCM with this structure consists of two potentially parallel steps:

a. For each quad, apply *Knot Insertion*.

b. For each extraordinary mesh node, use the vertex list to

–*collect* $Q_{00}(1)$ and $Q_{\texttt{uv}}(i)$, $\texttt{uv} \in \{10, 20, 40\}$.

–*compute* $Q_{\texttt{uv}}(i)$, $\texttt{uv} \in \{10, 20, 11\}$, $Q_{20} - \bar{Q}_{20}$ and possibly $Q_{40} - \bar{Q}_{40}$.

–*distribute* $Q_{\texttt{uv}}(i)$, $\texttt{uv} \in \{10, 01, 20, 02, 11\}$ and add to $\{21, 12\}$ and possibly $\{04, 40, 14, 41\}$.

The quad arrays can be input directly to `gluNurbsSurface` or displayed as the quad mesh.

There are some disadvantages in using the structure for PCCM:

- Weak connectivity information.
- Different orientation of the indices for different corners.
- Need to collect and distribute the data.
- A quad array is the smallest updating unit.

The first two disadvantages make the code hard to implement and extend. The last two cause the inefficiency of the transformation and updating.

4.2.2 Quad Polyhedron

Considering the disadvantages of the quad map, I build a new data structure, named **quad polyhedron**, which combines the quad structure and the CGAL polyhedron(see Figure 4.1).

Each facet in the quad polyhedron is a quad array just as it in the quad map. The initial quad array contains the vertices in the sub-mesh corresponding to the facet of the *I-mesh*. The halfedges in the quad polyhedron not only maintain the connectivity of the *I-mesh* but also a corner view and a shadow array corresponding to the incident corner of the halfedge. The corner view is an index reference array that does not store any valid geometry information but redirects the access to the actual

position in the quad array. It works like an index mapping function for quad arrays. By using the corner view, accessing to the four corner regions of the quad array can be simplified to the same index orientation that is consistent to the incidental halfedge. The size of corner view is one quarter of the quad array and can be treated as an independent transformation unit. The shadow array has the same size as well as the same indexing scheme of the corresponding corner view, but it only exists when incident corner is an extraordinary vertex. The shadow array is an internal buffer which stores the geometry data right after applying knot insertion in quad array. It is used to speed up the update of the PCCM mesh and localize the influence region caused by corner smoothing.

The first step of the algorithm to build the quad polyhedron is to build the same connectivity of the I-mesh. Then it initializes the quad array associated with each facet in the quad polyhedron. The initialization can be done parallelly for all quad arrays.

Assume that we have an I-mesh IM generated form the input mesh I and a quad polyhedron QP having the same connectivity of IM. After locating facets F_i in I, F_{im} in IM and F_{qp} in QP by traversing the corresponding facet lists concurrently, we can initialize the quad array of F_{qp} by using the CC scheme operations and polyhedron traversing operations.

To initialize the quad array of F_{qp}, my algorithm first locates the *center vertex* and the *corner vertices* of F_{im}. Since the IM was i-times subdivided from I, all the center vertex and the corner vertices can be traced by CC scheme operations on F_i. After locating the center vertex and the corner vertices, the algorithm *synchronizes* the orientation of the facets (F_i and F_{im}) by matching the corner vertices in F_i and F_{im}. It then defines a pivot edge that always points to the first corner vertex in F_{im}. By traversing the IM starting with the pivot edge, the algorithm collects the

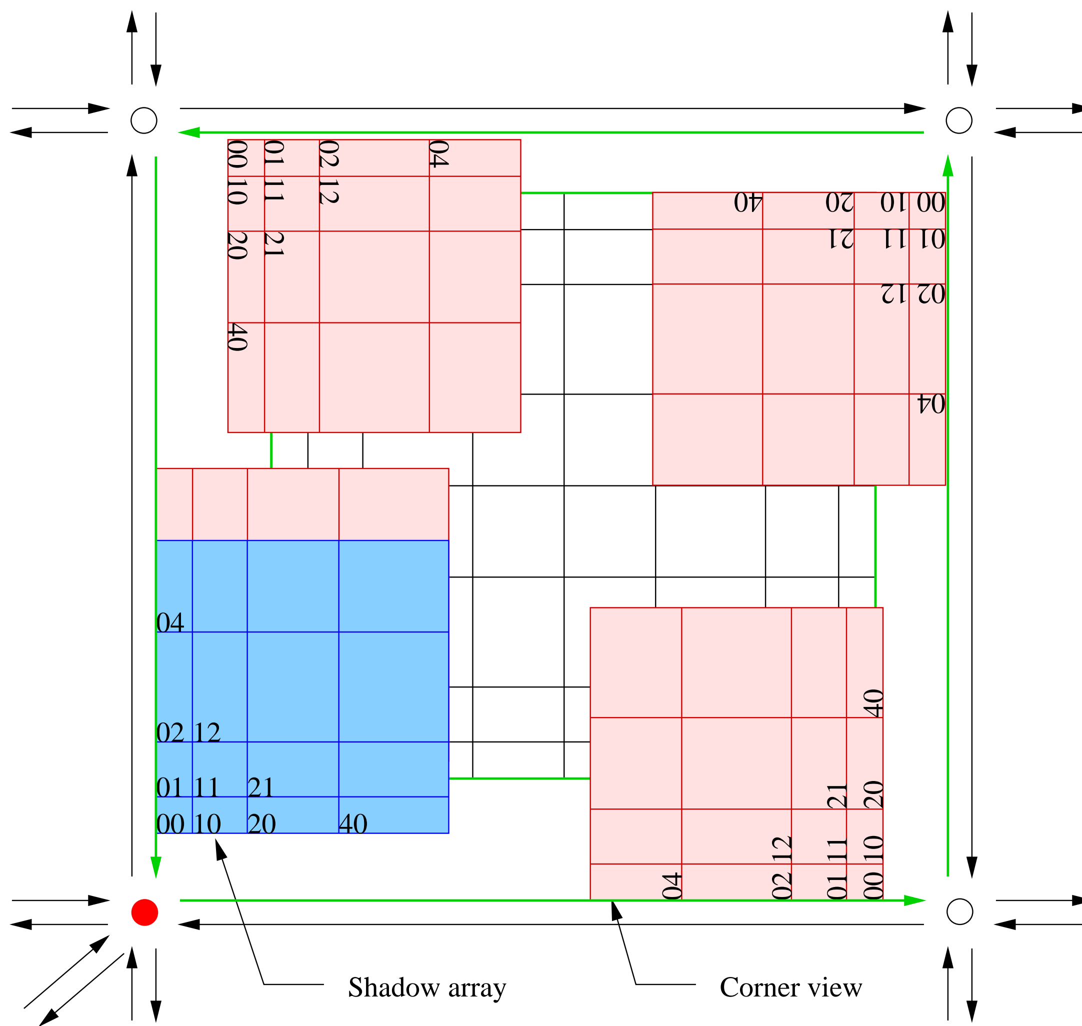


Figure 4.1: Each facet in the quad polyhedron contains a quad array that has four corner views associated with each halfedge. The orientation of the indices in the corner view is consistent with the halfedge. A shadow array is only associated with the extraordinary corner and has the same size and indices of the corresponding corner view.

geometry data into the center region of the quad array. This procedure is shown in Figure 4.2.

Algorithm 2 Building Quad Polyhedron

Require: facet list in input mesh **I**: $\mathbf{FL_i}$
Require: facet list in I-mesh **IM**: $\mathbf{FL_{im}}$
Ensure: facet list in quad polyhedron **QP**: $\mathbf{FL_{qp}}$

Build **QP** from **I** with the same connectivity
for $i = 1$ to size($\mathbf{FL_i}$) **do**
vertex $cv = down_{fv}(\mathbf{FL_i}[i])$
$\{v_i^1, v_i^2, v_i^3, v_i^4\} = \{$1st,2nd,3rd,4th$\}$ corner of $\mathbf{FL_i}[i]$
$\{v_i m^1, v_i m^2, v_i m^3, v_i m^4\} = down_{vv}(\{v_i^1, v_i^2, v_i^; v_i^4\})$
$v_i m =$ the corner vertex traversed from cv by up and then left.
synchronize the orientation between $\mathbf{FL_i}[i]$ and $\mathbf{FL_{im}}[i]$ by matching $v_i m$ and $\{v_i^1,\ v_i^2,\ v_i^3,\ v_i^4\}$
locate pivot edge pe that points to the corner vertex matching $v_i m$
travse the submesh started from pe and collect the data into quad array of $\mathbf{FL_{qp}}[i]$
end for

Each quad array can be processed by PCCM transformation in parallel. The first step of PCCM algorithm applys the knot insertion in each quad array. First, it vertically and then horizontally scans and inserts new knots in the quad array. It then circulates the edge around each vertex using traverse functions of CGAL polyhedron and inserts the new value of corner knots. Since the corner views associated with each edge give us the unified accessing to the knots, the corner insertion can be very simple without synchronizing the orientation of indices. Figure 4.3 shows the change of the region of the valid data in the quad array through different steps of the knot insertion. In the second step of PCCM algorithm, it applys the corner smoothing in the vertex list and skips the ordinary vertices. Similar to the corner insertion, it uses the vertex circulator around the extraordinary vertex and the corner views to directly access the necessary data—it does not need to collect and then distribute the data as the quad map does. Before applying the corner smoothing, for each corner view associating the extraordinary vertex, the algorithm preserves the state of the corner

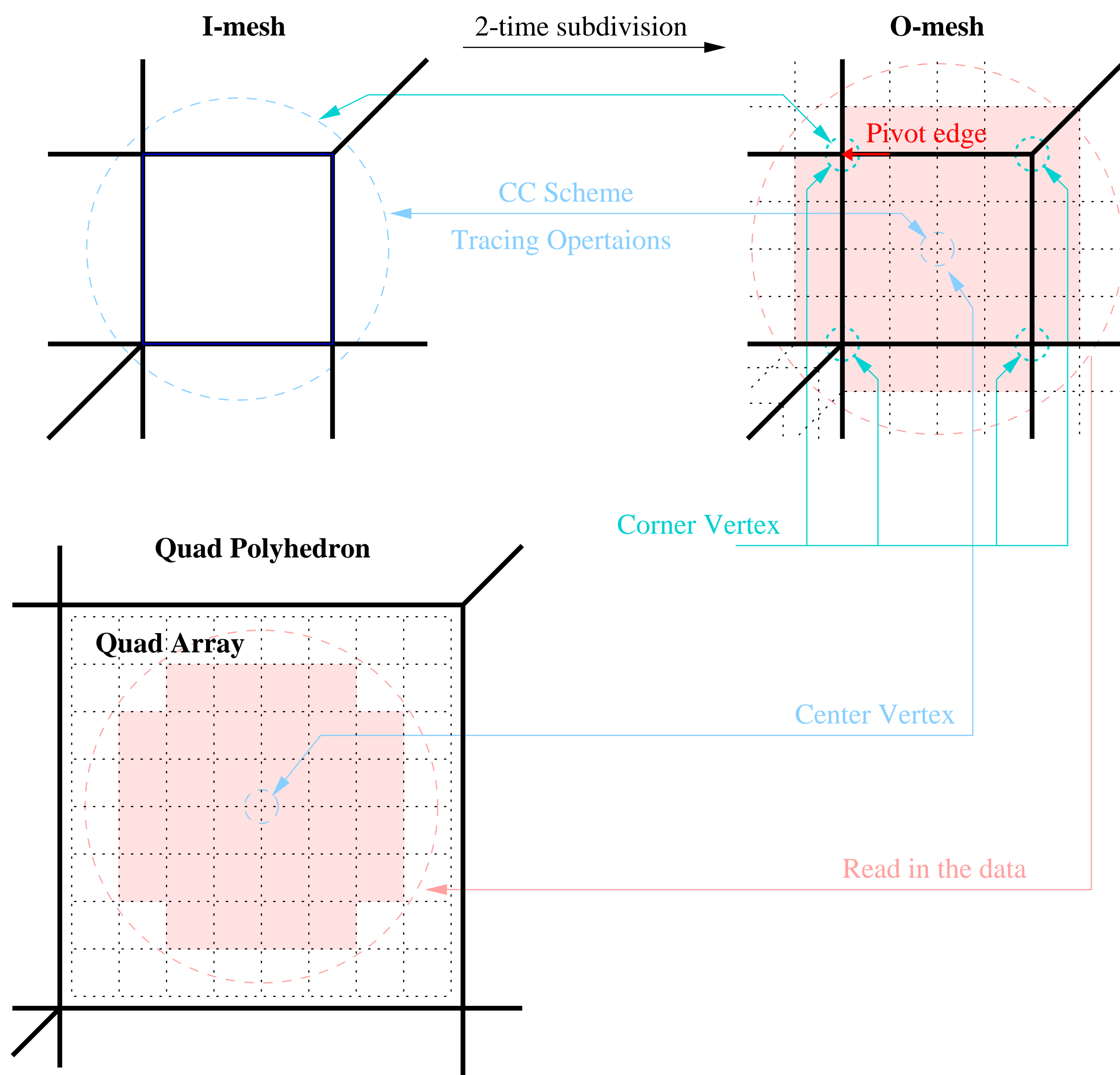


Figure 4.2: Initialization of the quad polyhedron: only the shaded region in the quad array has valid data after initialization and the valid region will grow up to whole array after the first step of the PCCM transformation.

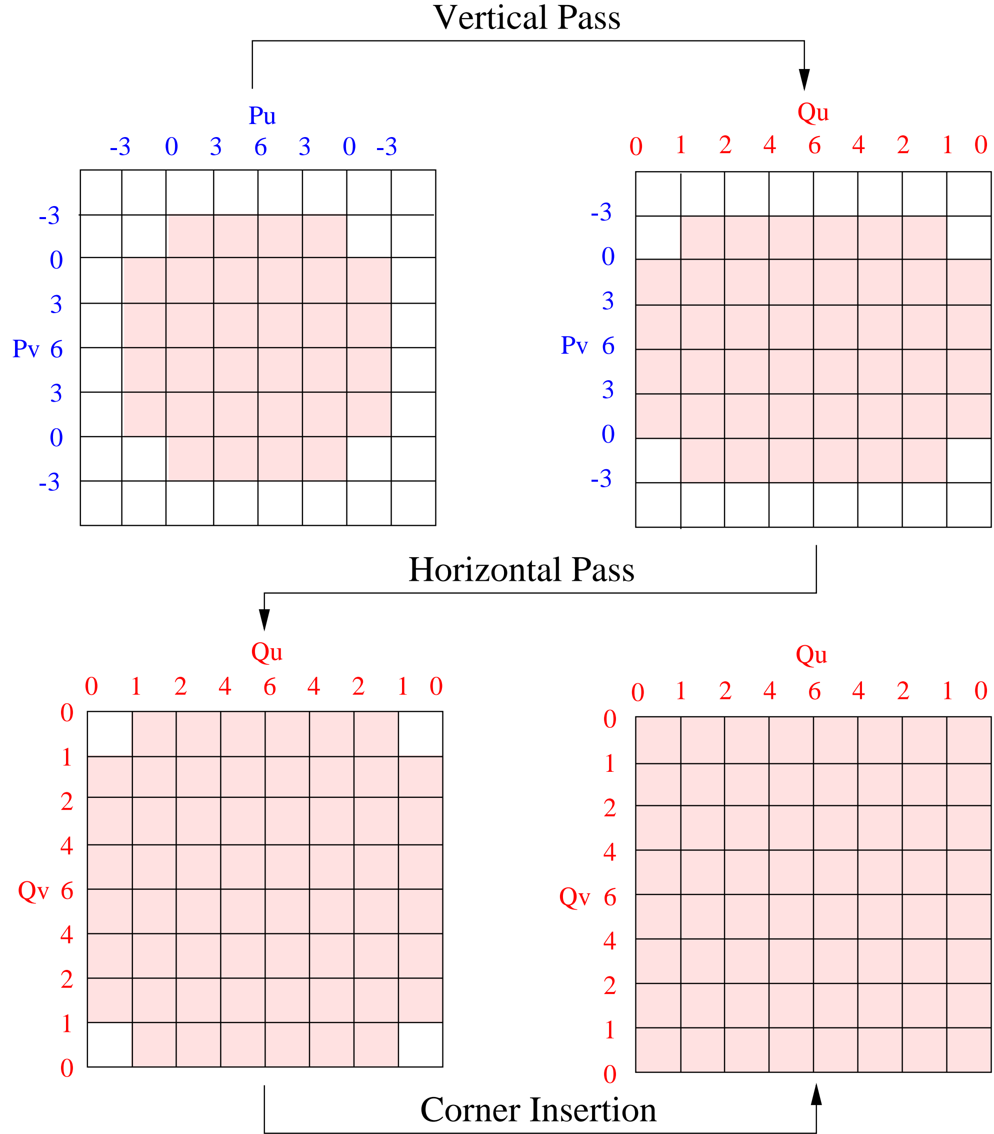


Figure 4.3: Knot insertion: the shaded area indicates the valid data.

view by storing it in the corresponding shadow array. Since the shadow array has the same size of the corner view, the preservation only needs to call the copy operation from the corner view to the shadow array.

4.3 Interactive Modeling with PCCM

For each facet f_i of the *input mesh*, exists a corresponding submesh fm_i in the *I-mesh*, called **fmesh**. fm_i is a $(2^d + 1) \times (2^d + 1)$ array centered by the *facet vertex* FL_i of f_i, where d indicates the number of passes of the Catmull-Clark subdivision.

Algorithm 3 PCCM using quad polyhedron

Require: initialized quad polyhedron: **QP**
Ensure: smoothly joining patches

```
for each quad array QA in QP do
   apply vertical pass of the knot insertion
   apply horizontal pass of the knot insertion
end for
for each vertex v in QP do
   apply corner insertion of the knot insertion
   update all corresponded corners in quad arrays
end for
for each vertex v in QP do
   if v is an extraordinary vertex then
      apply corner smoothing in v using operation of the corner view
   end if
end for
```

For fm_i, I define a **qmesh** qm_i as the submesh that is one layer of the facet larger than fm_i. Each qm_i has a corresponding quad array q_i of the quad polyhedron and the vertices of qm_i specify the initial data in q_i. Figure 4.4 illustrates these definitions.

Apparently, any modification of the vertices of qm_i will invalidate q_i and trigger the updating of the corresponding patch. However, since the intersection of all qmeshes of the I-mesh is not empty, any change of a single vertex might cause the update of several quad arrays.

To make the update region as small as possible, I equally divide the quad array into four sub-arrays adjacent to the center vertex and make these sub-arrays the update units for the update procedure. I call these sub-arrays *corner quads* since each of them associates with a corner vertex of the quads.

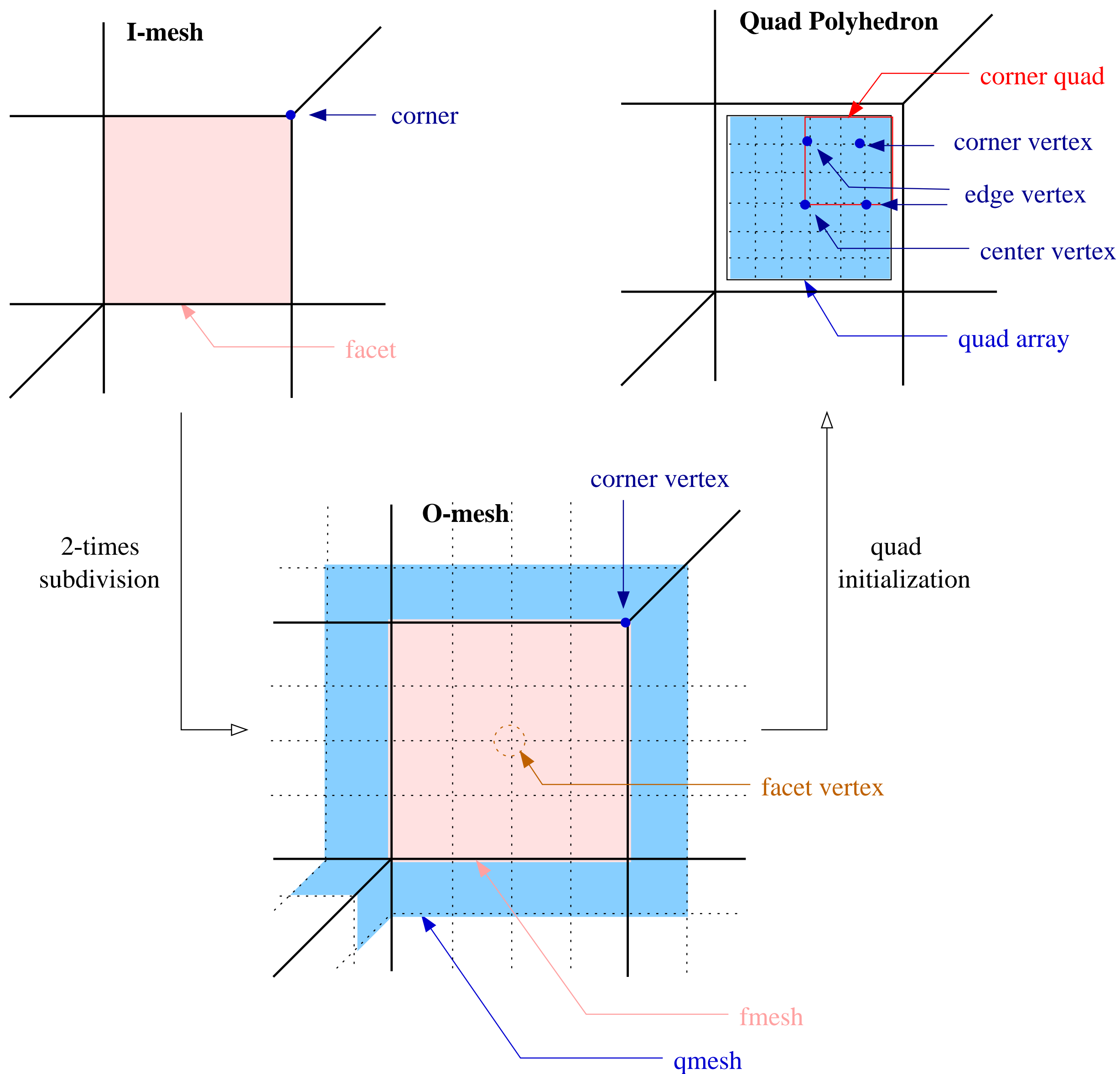


Figure 4.4: Relations between I-mesh, O-mesh and quad polyhderon.

My update procedure is defined by two mapping functions:

$\mathbf{ki} : \mathbf{V} \rightarrow \mathbf{C}$

$\mathbf{cs} : \mathbf{C} \rightarrow \mathbf{EV}$

where **V** is a set of the vertices of the O-mesh,

C is a set of the corner quads of the quad polyhderon,

EV is a set of the extraordinary vertices of the quad polyhedron.

Figure 4.5 illustrates the functionality of *ki* and *cs* functions in the interactive modeler.

4.3.1 Effect of Knot Insertion

For each qmesh, I define nine exclusive equivalent sets of vertices, called influence region, which are classified into three types— V region, E region, and F region(see Figure 4.6). Each region contains a set of vertices and affects a set of corner quads that define the update region. All vertices belonged to the same influence region should define the same update region.

Definition 4.1 ***Primitive meshes and regions***

- **V mesh**: *a ring of incident facets to the corner vertex.*
- **E mesh**: *a* $3 \times (d^2 - 3)$ *array along the quad edge. The center of the region resides in the center of the quad edge.*
- **F mesh**: *a* $(d^2 - 3) \times (d^2 - 3)$ *array centered inside the fmesh.*
- **V region**: *the vertices in the V mesh.*
- **E region**: *the vertices in the E mesh.*
- **F region**: *the vertices in the F mesh.*

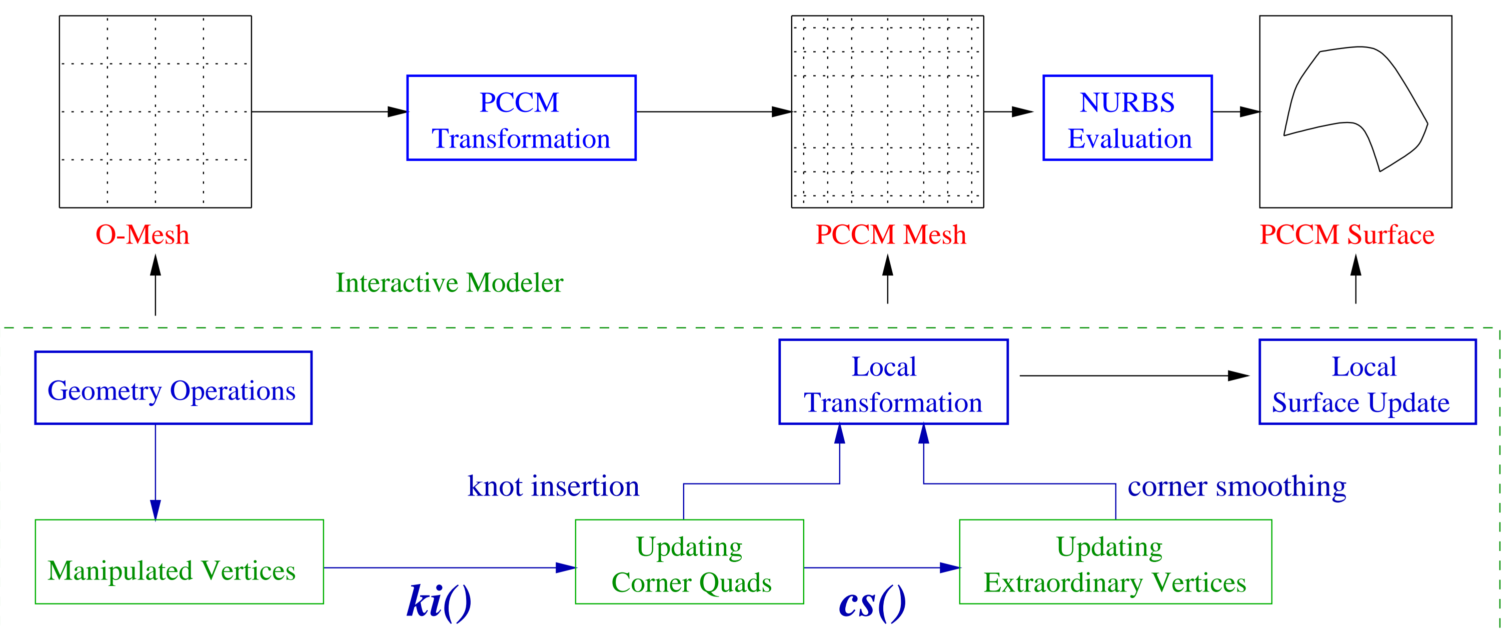


Figure 4.5: Interactive Modeler

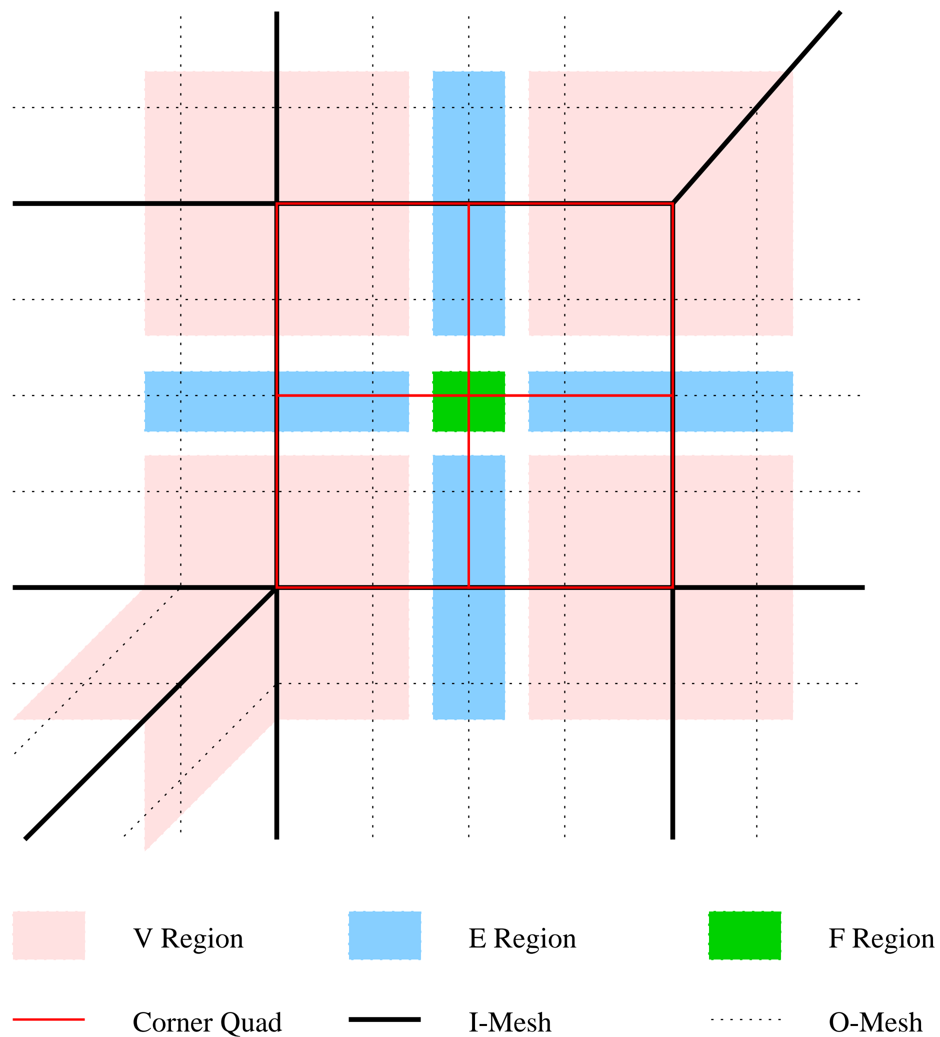


Figure 4.6: For each quad, there are nine exclusive influence regions—four V regions, four E regions and one F region.

For a set of manipulated vertices $V = \{v_1, v_2, ..., v_n\}$, we define

$$ki(V) = ki(v_1) \cup ki(v_2) \cup ...ki(v_n).$$

Let $RV = \{v_1, v_2, ..., v_n\}$ be one of these region vertex sets and a function

$$rv(v) = RV, \text{ if vertex } v \text{ in the O-mesh } \in RV.$$

Since $ki(RV) = ki(v_1) \cup ki(v_2) \cup ...ki(v_n)$ and $ki(v_1) = ki(v_2) = ... = ki(v_n)$, then

$$ki(RV) = ki(v), \text{ where } v \in RV.$$

Hence,

$$ki(v) = ki(rv(v)) = ki(RV).$$

Definition 4.2 ki *function*

- $ki(v) = ki(rv(v)) = ki(RV)$ *= the set of overlapped corner quads.*
- $ki(\{v_1, ..., v_n\}) = ki(v_1) \cup ki(v_2) \cup ...ki(v_n).$
- *ki(V region) = all corner quads that are adjacent to the corner vertex.*
- *ki(E region) = four corner quads that are adjacent to the quad edge.*
- *ki(F region) = four corner quads making up the corresponding quad.*

Figure 4.7 details each influence region. For the *two-pass knot insertion*, an **F region** sits inside the center of the sub-mesh and intersects four corner quads making up the quad. Four **E regions** sit along the four edges and each intersects a pair of corner quads of two incident quads. Four **V regions** sit around the four corners and each intersects all corner quads around each corners. For *corner insertion*, since we need all P_3 around the P_0 and P_0 itself, the influence region covers all these needed knots and has the same intersection as the V region in two-pass knot insertion.

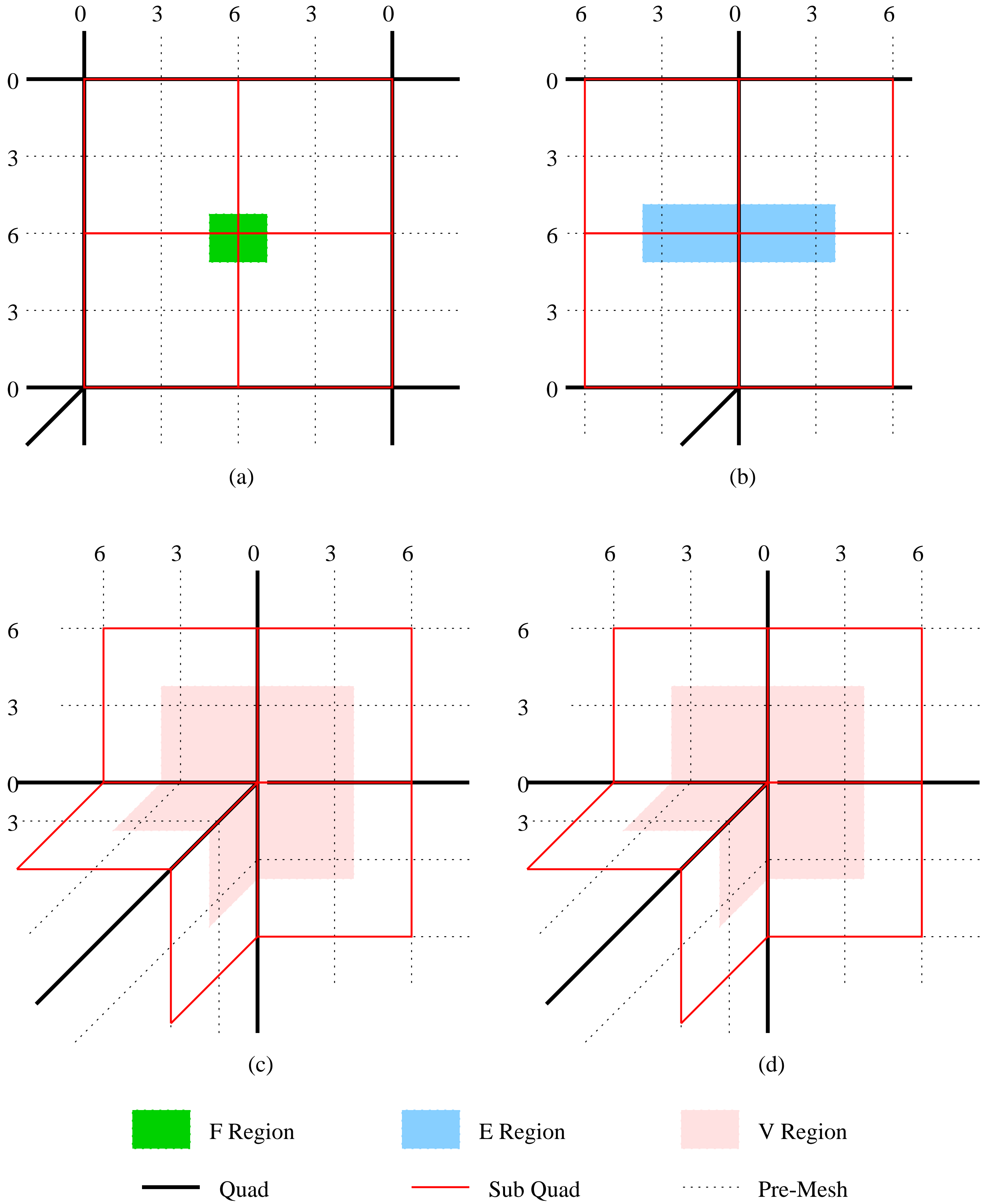


Figure 4.7: Influence region of knot insertion: (a),(b),(c) indicate the three different influence regions caused by the two-pass knot insertion. (d) indicates the influence region caused by the corner insertion.

4.3.2 Effect of Corner Smoothing

For an extraordinary vertex of valence n, there are n corner quads around the vertex. If any one of these corner quads gets updated by knot insertion, every one in these corner quads need to be updated by corner smoothing.

Definition 4.3 cs *function*

For a set of corner quads $\mathbf{CQ} = \{q_1, q_2, ..., q_m\}$,

- $cs(q) = \begin{cases} \textit{the corner vertex cv of q} & , \textit{if cv is an extraordinary vertex.} \\ \phi & , \textit{otherwise .} \end{cases}$
- $cs(\mathbf{CQ}) = cs(q_1) \cup cs(q_2) \cup ...cs(q_m)$.

Let $CQ = \{q_1, q_2, ..., q_m\}$ be the update region of knot insertion, $cs(CQ)$ returns a set of extraordinary vertices EV. Then we update the corner quads around the vertices in the EV by applying corner smoothing on EV. To speed up the updating, we use shadow arrays that preserve the *Q-mesh* data before corner smoothing instead of applying the quad initialization and knot insertion on the corner quads that are not in the CQ.

4.3.3 Algorithm for Local Updating

Algorithm.4 gives us the systematic look of how to use the functions we defined in previous sections to locally update the PCCM surface.

Algorithm 4 Local updating

Require: modified vertices $MV = \{mv_1, mv_2, ..., mv_n\}$
Ensure: smooth PCCM surface

corner quads $CQ = ki(MV) = ki(rv(mv_1)) \cup ki(rv(mv_2)) \cup ...ki(rv(mv_n))$
for each corner quad q in CQ **do**
  initialize q from the modified O-mesh
  apply knot insertion on q
  update corresponding shadow quad
end for
extraordinary vertices $EV = cs(CQ)$
for each vertex v in EV **do**
  apply corner smoothing on v using shadow quads around v
  update corner quads around v
end for
update the PCCM surface

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis I presented an extension of the PCCM algorithm supporting the interactive manipulation of the surface. To make this PCCM algorithm and its extension easy and efficient, I designed the *quad polyhedron* and applied it to the implementation.

This thesis introduced a simple encoding/decoding scheme, the *CC scheme*, that helps to trace the primitives relation of different level in a sequence of Catmull-Clark subdivided polyhedrons. I also presented a specific Catmull-Clark subdivision algorithm for generating the *CC scheme compatible* subdivided polyhedron. Using the tracing operations supported by the CC scheme, I introduced an initialization algorithm for the quad polyhedron. And then by applying PCCM transformation, we generate the PCCM surface represented by the quad polyhedron.

Finally, this thesis talked about the interactive modeling PCCM surfaces. I define an influence region as a set of manipulated vertices in the O-mesh. Influence regions localize the update region of the PCCM mesh and surface and hence make the interactive modeling possible.

Figure 1.1 illustrates the result mesh/surface of the different steps in my system. Figure 5.2 and Figure 5.3 show the resulting surfaces of interactive modeling.

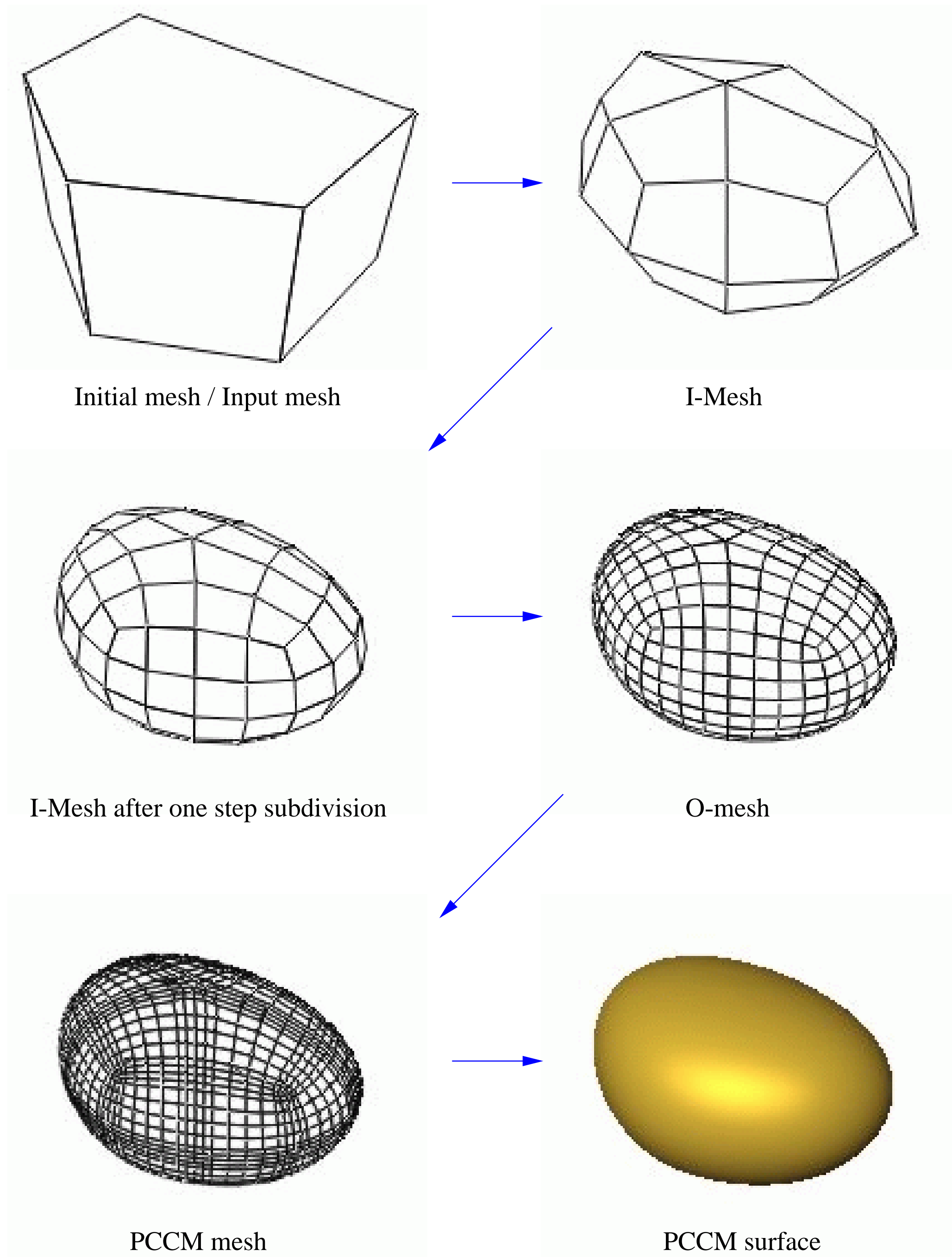


Figure 5.1: Examples of the different steps of the surface modeling.

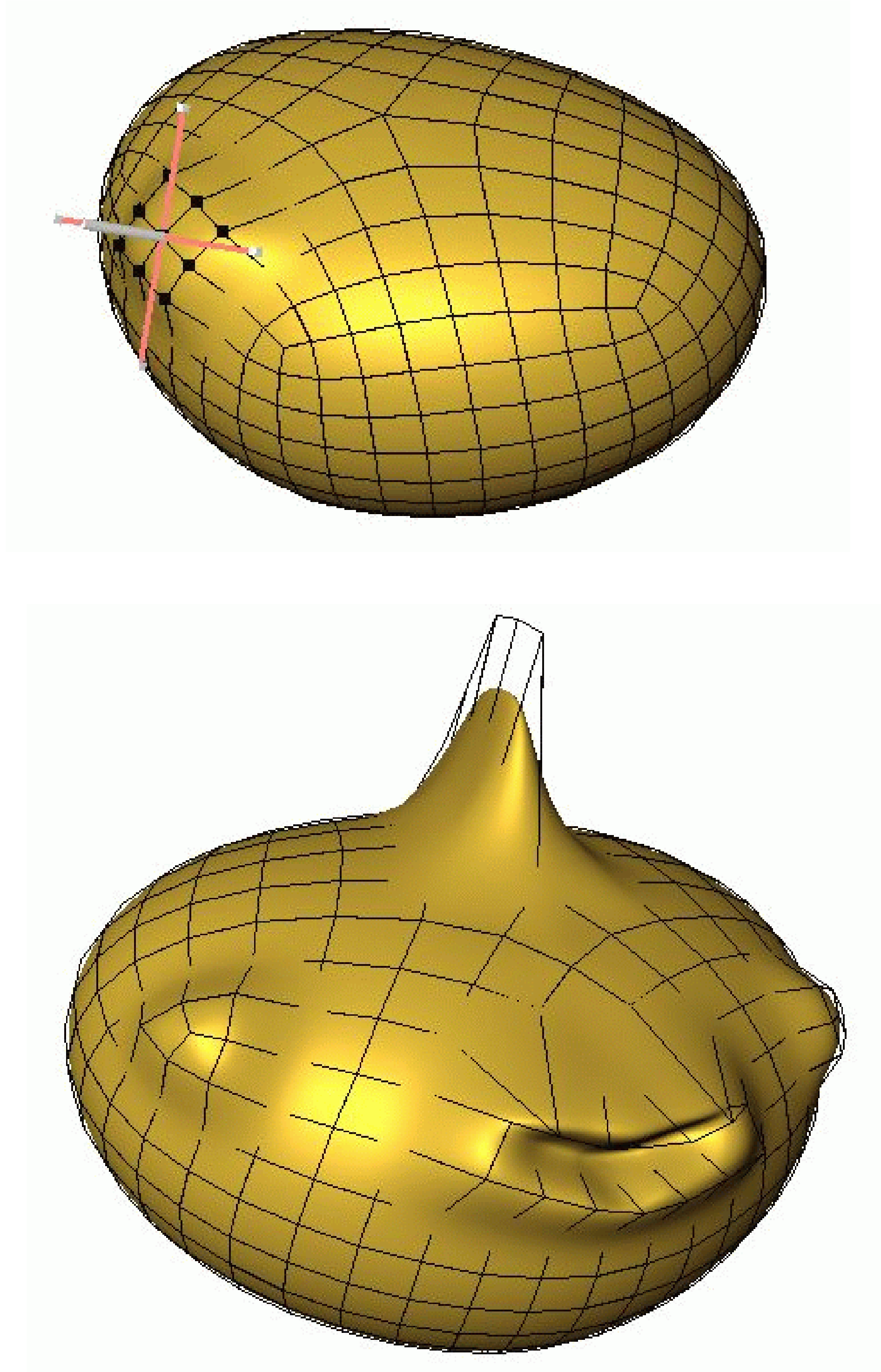


Figure 5.2: O-mesh and interactive modeling surfaces.

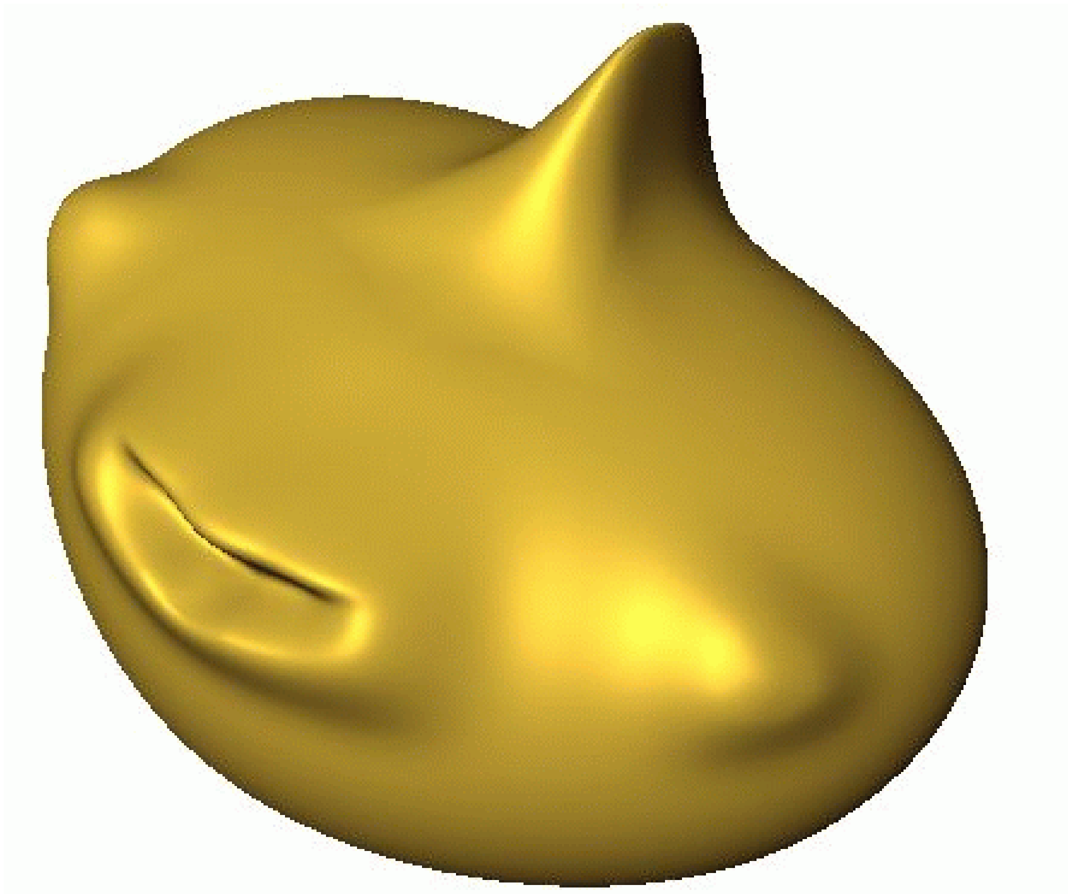


Figure 5.3: An example of modeling surfaces.

5.2 Future Work

Much more work remains to be done to build a user-friendly interactive modeling system. Future work might include using the extension of LeSS(localized-hierarchy surface splines)[2] to support finer control of the surface, designing the boundary rules for the PCCM transformation and extending the manipulation to change the genus of the object.

Another avenue of exploration, virtual tools are to be explored in my system. Currently, my system only supports a vertex displacement tool. Other virtual tools like edge sharpening, facet twisting and extrusion remain to be implemented. And the intuitive operations that do not require high precision are also good candidates to be supported in future system.

REFERENCES

[1] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10:350–355, Oct 1978.

[2] C. Gonzalez-Ochoa and J. Peters. Localized-hierarchy surface splines. In S.N. Spencer J. Rossignac, editor, *ACM Symposium on Interactive 3D Graphics*, pages 7–15, 1999. http://www.cise.ufl.edu/research/SurfLab/papers/.

[3] Nicolai M. Josuttis. *The C++ Standard Library—A Tutorial and Reference*. Addison-Wesley, Berkeley Califorina, 1999.

[4] Lutz Kettner. Algorithm library design. Class notes of comp 290-001, U. of North Carolina, Spring 2000. http://www.cs.unc.edu/~kettner/courses/.

[5] J. Peters. Patching Catmull-Clark meshes. *SIGGRAPH 2000 Conference Proceedings*, pages 255–258, 2000. http://www.cise.ufl.edu/research/SurfLab/papers/.

[6] Stefan Schirra, Remco Veltkamp, and Mariette Yvinec, editors. *Computational Geometry Algorithms Library online reference*. 1999. http://www.cgal.org/.

[7] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Berkeley Califorina, 3rd edition, 1997.

[8] K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, pages 21–40, January 1985.

[9] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*. Addison-Wesley, Berkeley Califorina, 2nd edition, 1997.

BIOGRAPHICAL SKETCH

Le-Jeng Shiue was born on December 1st, 1972, in Keelung, Taiwan. He received his B.S degree in electronic engineering from the National Taiwan Institute of Technology in 1994. He joined the University of Florida in August 1999 and started to pursue a master's degree in computer science. He is currently a research assistant of computer graphics and surface modeling for Dr. Jorg Peters and worked as a teaching assistant for the class of computer graphics in 2000. His research interests include data structure, algorithm, computer graphics and surface modeling.