Contour Tracing in Subdivision Schemes

H.C. Duifhuis
Advisor: G. Vegter

Informatica

RuG

Masters Thesis

Contour Tracing in Subdivision Schemes

H.C. Duifhuis
Advisor: G. Vegter

Rijksuniversiteit Groningen
Informatica
Postbus 800
9700 AV Groningen

6th March 2000

Contents

List of Figures

List of Tables

Chapter 1

Introduction

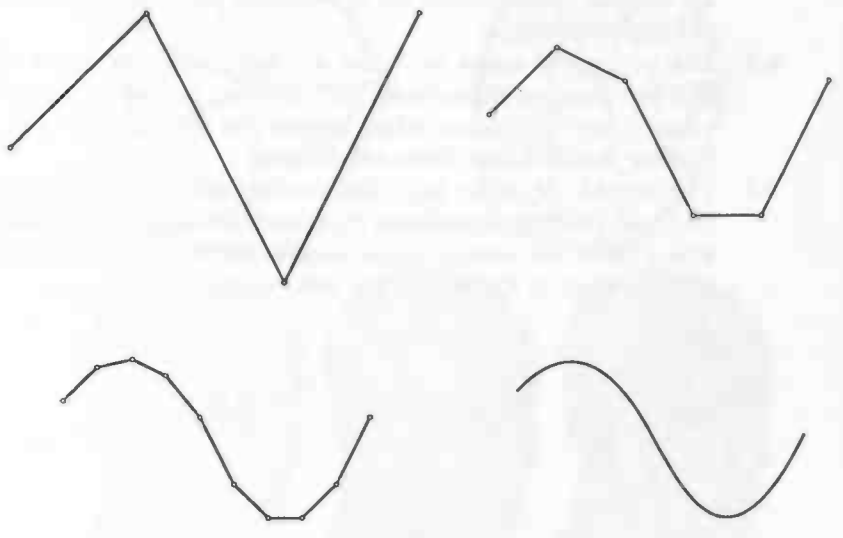


Figure 1.1: Chaikin's algorithm for a curve. The top left shows the initial function. The top right shows the function after one step. The bottom left shows the function after two steps and the bottom right shows the limit curve.

Surfaces play an important role in three-dimensional computer aided geometric design (CAGD). Flat faces are always represented by polyhedral meshes. A cube, for example, can be represented by 6 quadrilateral polygons, squares. But it is impossible to represent a smooth sphere using these polygons. There are several different ways to represent smoothly curved surfaces.

The classical way to do this is to use piecewise polynomial surfaces, like *tensor product Bézier surfaces* and *tensor product B-spline surfaces*. Tensor product non-uniform rational B-splines (NURBS) have become the most standard representation. A major disadvantage of these tensor product surfaces is the requirement that control nets, defining the surface, must consist of a regular rectangular grid of control points. Therefore these surfaces can only represent limited surface topologies (planes, cylinders and tori).

Chaikin's use of subdivision to create curves inspired Catmull and Clark and

simultaneously Doo and Sabin to use subdivision to create surfaces, called *subdivision surfaces.* This work provided the first method of constructing smooth surfaces of arbitrary topological type. Subdivision surfaces received little attention from the computer graphics community for many years, but regained interest because of the intimate connection between subdivision and multiresolution analysis.

George M. Chaikin [2] was the first to use *recursive subdivision* in 1974. *Chaikin's algorithm* can be thought of as a 'corner cutting' procedure to successively smooth down an initial piecewise-linear function to represent a smooth curve, as shown in Figure 1.1. At each step new vertices are placed $\frac{1}{4}$ and $\frac{3}{4}$ of the way between the old vertices. The new curve has twice as many segments, and repeatedly applied this process yields a piecewise-linear curve with a great number of segments that closely approximates a smooth curve.

Similarly the subdivision surfaces introduced by Catmull and Clark and also by Doo and Sabin are defined as the limit of an infinite refinement process of a 3D control mesh or control polyhedron, using a specific *subdivision algorithm.* These algorithms are based on the binary subdivision of uniform B-spline surfaces.

The main advantage of using subdivision surfaces is that surfaces of arbitrary topologies can be represented. Because of its recursive structure arbitrary detail can still be reached. Subdivision surfaces are a good compromise between polygonal meshes and spline patches. They can be treated as large collections of small polygons, and behave like composite patches. The simplicity behind the idea of subdivision makes it easy to understand and implement. In Chapter 8 some examples are illustrated.

In this work, three different subdivision surfaces will be discussed:

1. Loop surfaces, which are based on three-directional quartic box spline surfaces (see Figure 1.2).

2. Doo-Sabin surfaces, which are generalizations of biquadratic uniform B-spline surfaces.

3. Catmull-Clark surfaces, which are generalizations of bicubic uniform B-spline surfaces.

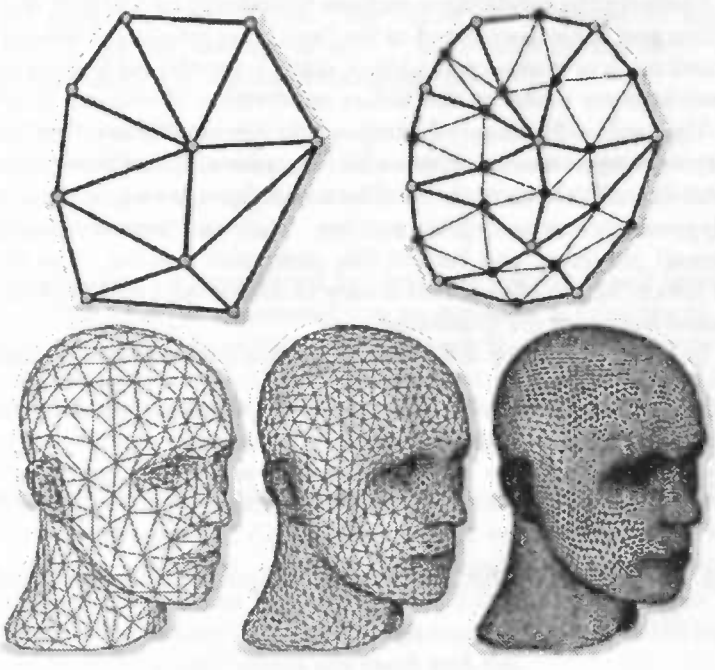


Figure 1.2: Loop subdivision

Chapter 2

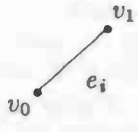
Background

Abstract. In this chapter some topological definitions that are important to our subdivision algorithms are recalled. B-spline curves and surfaces are introduced and some basic properties are mentioned.

2.1 Topology

Definition 1 *A vertex $v_i = (x_i, y_i, z_i)$ is a point in $\mathbb{R}^3$. Vertices will be referred to in some cases as* points *or* control points.

Definition 2 *An edge e_i is defined as a straight line segment between two vertices v_0, v_1.*

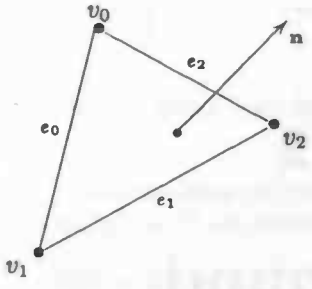


Vertices v_0 and v_1 are called the *endpoints* of the edge. In most data structures an edge is represented by its two endpoints. Its *midpoint* is defined as the average of the two endpoints.

Definition 3 *A face f_i is represented by a list of three or more coplanar*[1] *vertices $\{v_0, \ldots, v_{n-1}\}$ defining the enclosed (convex) region which is constructed by connecting each two successive vertices (and in addition v_{n-1} and v_0) in the*

[1]Three vertices are always coplanar, but more vertices are not necessarily. In Chapter 6 we will redefine a face for non-coplanar vertices. Until then are faces discussed are planar.

face representation.



The face normal **n** is defined as the normalized vector perpendicular to the plane defined by f_i. The order of the vertices defining f_i determines the direction of **n** using the right-hand-thread rule. We define the *facepoint* or *midpoint* or *centroid* of f_i to be the average of $v_0, \ldots, v_{n-1}$ defining the face. A face is called **quadrilateral** if $n = 4$ and **triangular** if $n = 3$.

Definition 4 *A polyhedron P is a pair (K, V) where $V = \{v_1, v_2, \ldots, v_m\}$ is a set of vertices defining its shape in $\mathbb{R}^3$ and K is a complex representing the connectivity of the vertices by edges and faces, thus determining the topological type of the polyhedron.*

We recall that edges are represented by their two endpoints and faces by their defining vertices. In both representations we use indices referring to elements in the vertex set V. A polyhedron is called **quadrilateral/triangular** if all of its faces are quadrilateral/triangular. The number of faces (or edges) joining in a vertex is called the **valence** of that vertex. Normally two faces join at each edge. If only one face joins at an edge, we call that edge a **boundary edge**. The **boundary** of P can now be defined as the set of all boundary edges of P. The endpoints of a boundary edge have a 'valence irregularity' in the sense that the number of edge and face joining in that vertex are not the same. To avoid boundary irregularities our main interest goes to **closed** polyhedra, i.e. polyhedra without a boundary. We will allow polyhedra with holes 'through' it, like a torus. The number of holes through a polyhedron is called the **genus**. Other terms used for a polyhedron are *mesh* and *net*.

For example, the polyhedron $P = (K, V)$ with vertex list V:

$$V = \{(-1,0,0), (0,-1,0), (1,0,0), (0,1,0), (0,0,1)\},$$

and complex K:

$$\begin{aligned} \textbf{vertices:} \quad & \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \\ \textbf{edges:} \quad & \{1,2\}, \{1,4\}, \{1,5\}, \{2,3\}, \{2,5\}, \{3,4\}, \{3,5\}, \{4,5\}, \\ \textbf{faces:} \quad & \{1,4,3,2\}, \{1,2,5\}, \{2,3,5\}, \{3,4,5\}, \{4,1,5\} \end{aligned}$$

represents a pyramid in $\mathbb{R}^3$ as shown in Figure 2.1.

Another important concept we are going to use in further chapters is the *neighborhood* of a vertex.

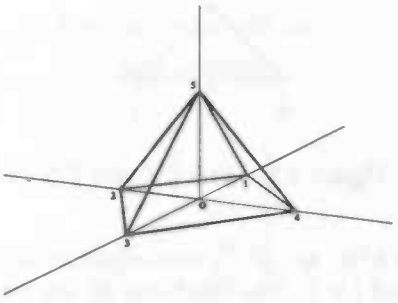


Figure 2.1: Polyhedron representation of a pyramid

Definition 5 *The k-neighborhood* $(k = 2, 3, \dots)$ *of a vertex* v_0 *is the union of the 1-neighborhoods of all vertices in the* $(k-1)$*-neighborhood of* v_0*, where the 1-neighborhood of a vertex* v_0 *is the set of vertices that can be connected to* v_0 *with no more than 1 edge, including* v_0 *itself.*

In the example of the pyramid, the 1-neighborhood of the fifth vertex includes all five vertices, while the 1-neighborhood of one of the other vertices include only four vertices.

2.2 B-spline curves

I will now summarize the most important features of B-spline curves and surfaces and B-spline refinement which is the basis for understanding and analyzing subdivision curves and surfaces. For proofs and background, see eg. [4, 12, 8].

A B-spline curve of degree l can be written as

$$S(t) = \sum_i \mathbf{p}_i B_i^l(t) \tag{2.1}$$

where the $\mathbf{p}_i$ form a set of control points which define the position in $\mathbb{R}^2$ and the B_i^l are order l B-spline basic functions which define the curvature of the spline. Each basis function is defined over a partition of the real axis called a *knot vector*. A zero-degree basis function is defined as follows:

$$B^0(t) = \begin{cases} 1 & \text{if } \ 0 \le t < 1 \\ 0 & \text{otherwise,} \end{cases}$$

The function $B^0(t)$ can be written in terms of its dilates, giving a *subdivision formula* for basis B-spline functions:

$$B^0(t) = B^0(2t) + B^0(2t-1).$$

A basis function of degree l is defined as the convolution of a basis function of degree $l-1$ with a zero-degree basis function:

$$\begin{aligned} B^l(t) &= \left(B^{l-1} \otimes B^0\right)(t) \\ &= \int_s B^{l-1}(s) B^0(t-s) ds, \end{aligned}$$

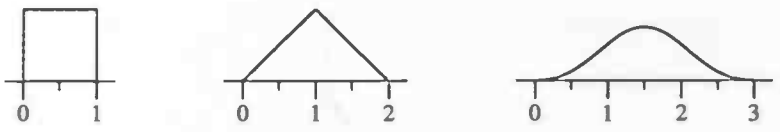


Figure 2.2: Level 0,1 and 2 basis functions

The functions $B^l(t)$ are C^{l-1}, non-negative polynomials, with compact support between 0 and $l+1$. The functions B_i^l are translates of B^l:

$$B_i^l(t) = B^l(t-i).$$

The sum of the translates is the function 1. By linearity of convolutions a subdivision formula for the functions $B^l(t)$ can be found:

$$B^l(t) = \sum_{k=0}^{l+1} s_k B^l(2t-k), \tag{2.2}$$

where s_k are constants, found using the following theorem ([12, 16]):

Theorem 1 *Let $h(t)$ denote the continuous convolution of two functions, $f(t)$ and $g(t)$, with subdivision formulas*

$$\begin{aligned} f(t) &= \sum_i a_i f(2t-i), \\ g(t) &= \sum_j b_j g(2t-j). \end{aligned}$$

Then, $h(t)$ has the subdivision formula

$$h(t) = \frac{1}{2}\sum_k c_k h(2t-k),$$

where the set c_k denotes the discrete convolution of the two sets a_i and b_j:

$$c_k = \sum_{i+j=k} a_i b_j.$$

Using the binomial theorem we now find

$$s_k = \frac{1}{2^l}\binom{l+1}{k}.$$

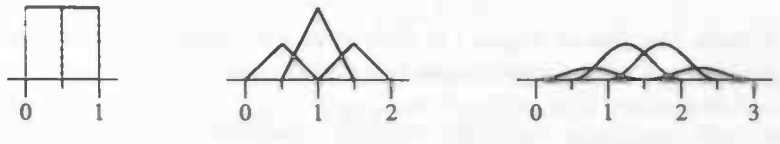


Figure 2.3: Subdivision of level 0,1 and 2 basis functions

In matrix notation, the B-spline curve will be

$$S(t) = \mathbf{B}^l(t)\mathbf{p}$$

where

$$\mathbf{p} = \begin{bmatrix} \vdots \\ \mathbf{p}_{-1} \\ \mathbf{p}_0 \\ \mathbf{p}_1 \\ \vdots \end{bmatrix}$$

and

$$\mathbf{B}^l(t) = \begin{bmatrix} \cdots & B^l(t+1) & B^l(t) & B^l(t-1) & \cdots \end{bmatrix}.$$

The subdivision formula can be written as

$$\mathbf{B}^l(t) = \mathbf{B}^l(2t)S,$$

where S is the *subdivision matrix*. The elements of S are related to the coefficients s_k in (2.2) by

$$S_{2i+k,i} = s_k.$$

We can rewrite $S(t)$ as

$$S(t) = \mathbf{B}^l(2t)S\mathbf{p},$$

where we can think of $\mathbf{B}^l(2t)$ as the new basis functions and $S\mathbf{p}$ as the new control points. This process is called a *subdivision step* and can be repeated indefinitely and the piecewise linear curve $\mathbf{B}^1(2^j t)S^j\mathbf{p}$ through the control points converges uniformly to $S(t)$.

Conclusively, we have found that we can recursively add additional control points to a B-spline curve of any degree, such that a piecewise-linear function through these control points would converge to the curve itself. This is the basis idea for subdivision curves and subsequently for subdivision surfaces, as we will see in the next section.

2.3 Tensor product B-spline surfaces

A parametric tensor product B-spline surface is written as

$$S(u,v) = \sum_i \sum_j \mathbf{p}_{ij} B_i^r(u) B_j^s(v), \tag{2.3}$$

where the $\mathbf{p}_{ij}$ form a rectangular control mesh.

A B-spline surface $\mathbf{S}(u,v)$ is C^r continuous with respect to u and C^s continuous with respect to v. If $r = s = 2$ then $\mathbf{S}(u,v)$ is called a *biquadratic* B-spline surface, and if $r = s = 3$ then $\mathbf{S}(u,v)$ is called a *bicubic* B-spline surface.

This refinement property of B-spline surfaces is the basis for the idea of subdivision surfaces. In particular, the three subdivision surfaces described in this work (Loop surfaces, Doo-Sabin surfaces and Catmull-Clark surfaces) are generalizations of existing spline surfaces, respectively three-directional box spline surfaces, biquadratic B-spline surfaces and bicubic B-spline surfaces. More detailed accounts will be given in the next chapter.

Chapter 3

Subdivision Curves

Abstract. The concept of general subdivision is introduced for curves, before it is applied to surfaces in the next chapter. In this chapter the main terminology such as masks and classifications are introduced. I will begin with discussing the first important subdivision method introduced by Chaikin.

3.1 Introduction

Chaikin's work [2] provided the first method of constructing curves through a simple process known as recursive subdivision, or just subdivision. This was the inspiration for creating surfaces using a similar process, called subdivision surfaces.

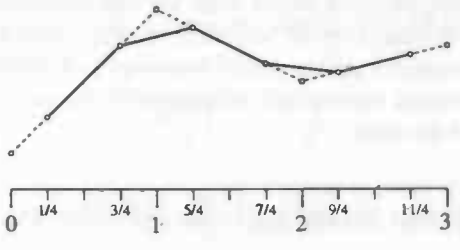


Figure 3.1: Chaikin's algorithm

The basic idea behind recursive subdivision is to create a function by repeatedly refining an initial piecewise linear function $f^0(x)$ to produce a sequence of increasingly detailed functions $f^1(x), f^2(x), \ldots$ that converge to a limit function

$$f(x) := \lim_{j \to \infty} f^j(x).$$

Let $f^0(x)$ be a piecewise linear function with vertices at the integers. In general, the function $f^1(x)$ will be a piecewise linear function with vertices at the *dyadic points* $i/2^j$. For most schemes, including Chaikin's algorithm, the values of

$f^j(x)$ at its vertices are computed very simply as follows:

$$f^j\left(\frac{i}{2^j}\right) = \sum_k r_k f^{j-1}\left(\frac{i+k}{2^j}\right).$$

One such a computation is called a *subdivision step*. The sequence $r = (\ldots, r_{-1}, r_0, r_1, \ldots)$ is called the *averaging mask* of the scheme. For Chaikin's algorithm, the averaging mask is $r = (r_0, r_1) = \frac{1}{2}(1,1)$.

3.2 Classification

Such a scheme, as Chaikin's algorithm, is called a *uniform subdivision scheme* because the same mask is used everywhere along the curve (i.e. r does not depend on i), and it is *stationary* because the same mask is used on each iteration of subdivision (i.e. r does not depend on j). If a curve is bounded by one or two endpoints extra masks should be introduced for the boundary vertices. The averaging mask for those vertices will be $r = (r_0, r_1, \ldots)$ or $r = (\ldots, r_{-1}, r_0)$.

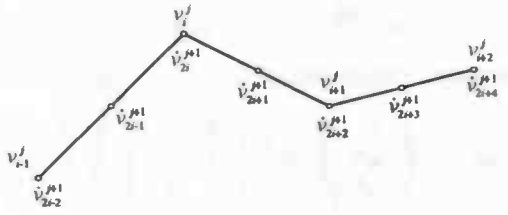


Figure 3.2: The splitting step.

A subdivision step can be divided into two steps: the *splitting step*, which explicitly introduces midpoints, and the *averaging step*, which computes the weighted averages indicated by the equation. All subdivision schemes for curves share the splitting step, but differ in the averaging step. In Figure 3.2 the general splitting step is shown. Each vertex v_i^j generates the vertex v_{2i}^{j+1} at the same position, and at the midpoint between each two vertices v_i^j and v_{i+1}^j the new vertex v_{2i+1}^{j+1} is inserted. Lane and Riesenfeld showed that uniform B-spline curves of degrees $l+1$ can be generated by the averaging mask

$$r = \frac{1}{2^l}\left(\binom{l}{0}, \binom{l}{1}, \ldots, \binom{l}{l}\right),$$

and that Chaikin's algorithm generates uniform quadratic ($l = 1$) B-spline curves.

An averaging step can be either *approximating*, like Chaikin's algorithm, or *interpolating*. An interpolating scheme doesn't change the positions of vertices by local averaging, once computed, while an approximating scheme generally does.

3.3 Evaluation masks

It turns out that functions defined through subdivision can be exactly evaluated at an arbitrarily dense set of points. This is accomplished by taking weighted av-

erages according to an *evaluation mask* specific to the subdivision procedure. If the subdivision scheme is smooth (i.e. the functions $f^j(x)$ converge to a smooth limit function $f(x)$), derivatives of the function can also be computed exactly using a *derivative mask*.

This based on the observation that vertices can be tracked throughout the subdivision process. A vertex v^0 of f^0 can be associated with a sequence of vertices v^j of f^j converging to a limit position v^∞ of f^∞. Let's consider the situation for uniform cubic B-spline subdivision. The averaging mask in this case is $r = \frac{1}{4}(1,2,1)$. The key observation in determining v^∞ is that in each step of subdivision, the position of v^j and immediate neighbors can be determined from v^{j-1} and its immediate neighbors. Let v_-^j and v_+^j denote the left and right neighbors of v^j. Then the splitting and averaging steps combined would give

$$\begin{aligned} v_-^j &= \frac{v_-^{j-1} + v^{j-1}}{2} \\ v^j &= \frac{v_-^{j-1} + 6v^{j-1} + v_+^{j-1}}{8} \\ v_+^j &= \frac{v^{j-1} + v_+^{j-1}}{2} \end{aligned}$$

In matrix notation, this becomes

$$\begin{pmatrix} v_-^j \\ v^j \\ v_+^j \end{pmatrix} = \underbrace{\frac{1}{8}\begin{pmatrix} 4 & 4 & 0 \\ 1 & 6 & 1 \\ 0 & 4 & 4 \end{pmatrix}}_{S} \begin{pmatrix} v_-^{j-1} \\ v^{j-1} \\ v_+^{j-1} \end{pmatrix},$$

where S is called the *local subdivision matrix* for the subdivision scheme. All schemes can be represented by an $n \times n$ local subdivision matrix, where n the the number of elements in the averaging mask. For Chaikin's algorithm the local subdivision matrix is

$$S = \frac{1}{4}\begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}.$$

For stationary schemes the (local) subdivision matrix is used at each subdivision step. Therefore we have the following formula to find the limit positions v^∞, v_-^∞ and v_+^∞

$$\begin{pmatrix} v_-^\infty \\ v^\infty \\ v_+^\infty \end{pmatrix} = \lim_{j\to\infty} \begin{pmatrix} v_-^j \\ v^j \\ v_+^j \end{pmatrix} = \lim_{j\to\infty} S^j \begin{pmatrix} v_-^0 \\ v^0 \\ v_+^0 \end{pmatrix}.$$

Studying the eigenstructure of S shows us what happens to v^∞, v_-^∞ and v_+^∞ and therefore determines the what the limit curve looks like (note that the scheme must be uniform and stationary). Let a local subdivision matrix S has eigenvalues $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$ with associated right eigenvectors $\mathbf{R_0}, \mathbf{R_1}, \ldots, \mathbf{R}_{n-1}$. Let M^0 denote the array $(v_0, v_1, \ldots, v_{n-1})^{\mathrm{T}}$ of length n. We can write M^0 as a linear combination of these eigenvectors, because they form a basis:

$$M = \sum_{i=0}^{n-1} a_i \mathbf{R}_i.$$

Applying S gives

$$\begin{aligned} SM &= S\sum_{i=0}^{n-1} a_i \mathbf{R}_i \\ &= \sum_{i=0}^{n-1} a_i S\mathbf{R}_i \\ &= \sum_{i=0}^{n-1} a_i \lambda_i \mathbf{R}_i \end{aligned}$$

Applying S j times gives

$$M^j = S^j M^0 = \sum_{i=0}^{n-1} a_i \lambda_i^j \mathbf{R}_i. \tag{3.1}$$

As j approaches infinity, M^j would grow without bound if any $\lambda_i > 1$. It possible to show that only a single eigenvalue should be 1 [12]. Let

$$1 = \lambda_0 > |\lambda_i|, \qquad i = 1, \ldots, n-1.$$

The limit vector M^∞ is then computed as follows:

$$M^\infty = \lim_{j\to\infty} S^j M^0 = \lim_{j\to\infty} \sum_{i=0}^{n-1} a_i \lambda_i^j \mathbf{R}_i = a_0.$$

The coefficient a_0 can be computed using the dominant left eigenvector (associated with λ_0) $\mathbf{L}_0$ as follows:

$$a_0 = \mathbf{L}_0 M^0.$$

Conclusively, the dominant left eigenvector serves as an evaluation mask. For Chaikin's algorithm, the evaluation mask is $\frac{1}{2}(1,1)$. And for the cubic B-spline subdivision, the evaluation mask is $\frac{1}{6}(1,4,1)$.

It is shown in [12] that a_1 represents the tangent vector at center point of M^∞ to the limit curve if all eigenvalues of S except λ_0 are less than λ_1. Therefore the left eigenvector $\mathbf{L}_1$ corresponding to the eigenvalue λ_1 serves as a derivative mask. For Chaikin's algorithm, the derivative mask is $\frac{1}{2}(1,-1)$. And for the cubic B-spline subdivision, the derivative mask is $(-1,0,1)$.

Also, the subdivision matrix should be *affine-invariant*, meaning that any distance-preserving transformation to the initial vertices shouldn't change the shape of the limit curve. Let $\vec{1}$ be an n-vector of 1's and $a \in \mathbb{R}^2$ a displacement in $\mathbb{R}^2$. We get

$$\begin{aligned} S(M^0 + \vec{1} \cdot a) &= SM^0 + S(\vec{1} \cdot a) \\ &= M^1 + S(\vec{1} \cdot a), \end{aligned}$$

where we would want

$$S(\vec{1} \cdot a) = \vec{1} \cdot a.$$

Therefore, $\vec{1}$ should be the right eigenvector of S corresponding to the eigenvalue λ_0.

Chapter 4

Subdivision Surfaces

Abstract. I will now introduce the main concept: subdivision surfaces. I will explain what they are, how can they be classified, and some convergence properties. In the last section I will compare some important features of subdivision with other representation methods (classic splines, implicit surfaces and variational surfaces) to show why it is such a powerful tool in CAGD.

4.1 Introduction

Recursively refining an initial polyhedron, increasing the number of faces and vertices, leads to an approximation of a smooth surface called a **subdivision surface**. This means that we create a sequence of polyhedra $P^0, P^1, \ldots, P^n$ by repeatedly applying one or a set of subdivision algorithms to an initial polyhedron P^0. Each polyhedron in the sequence should resemble the limit surface more closely than the previous one. In general, closed form expressions for limit surfaces are not known. However, various properties of subdivision surfaces are known:

- The topology is determined by the topology of the control polyhedron.
- Exact points on the surface can be computed.
- Exact tangent planes on the surface can be computed.

I will further explain these properties in Section 4.3. It turns out in practice that we only need a few subdivision steps ($n = 3$ or 4) to reach a desirable approximation of the limit surface, which means that by successfully rendering the approximation a smooth surface can be simulated.

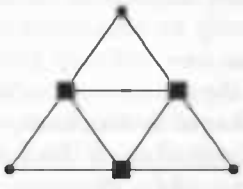
4.2 Classification

Similar to subdivision curves, each subdivision step (to construct P^{j+1} from P^j) consists of two substeps: a *splitting step* and an *averaging step*. The splitting step introduces new points for the new mesh after the subdivision and the averaging step computes the weighted averages of all points in the new mesh using a specific *averaging mask*.

Definition 6 *An **averaging mask** of a 3D subdivision algorithm is a 2D representation of the vertices and their weights in P^j needed to compute a new vertex in P^{j+1}.*

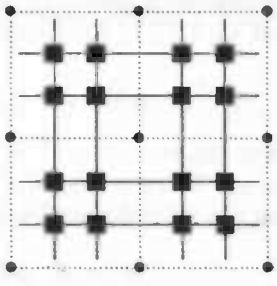
We can distinguish two types of splitting steps: *face schemes*, which replace each face with a number of subfaces, and *vertex schemes*, which replace each vertex and edge with a new face. Face schemes are also referred to as *vertex insertion* schemes, while vertex schemes are also called *corner-cutting* schemes. Two types of averaging steps for face schemes can be distinguished: *interpolating* or *approximating* schemes. An interpolating face scheme doesn't change the position of the old vertices, while an approximating face scheme does. Therefore the limit surface of an interpolating scheme passes through the vertices of P^0.

A subdivision schemes can also be classified as being either *triangular*, where triangular meshes are generated, or *quadrilateral*, where quadrilateral meshes are generated. When using a face scheme, our polyhedron is completely triangular/quadrilateral after only one subdivision step, whatever the original polyhedron looked like. If we're using a vertex scheme however, only new triangular/quadrilateral faces will be created but irregular vertices and faces will always generated new irregular faces. I will discuss these irregularities more rigorously in the next section. The following figures show standard quadrilateral and triangular splitting rules for face schemes.



The old vertices are represented by circles and the new (inserted) vertices by squares. In both cases four new faces are created. Each edge generates a new vertex (edge point) and in the quadrilateral case each face also generates a new vertex (face point).

The following figure shows a standard splitting step for a quadrilateral vertex scheme.



The old vertices are represented by circles and the new (inserted) vertices by squares. Each vertex, edge and face creates a new (quadrilateral) face.

As with curves, subdivision schemes can also be either *uniform* or *non-uniform*, and *stationary* or *non-stationary*. A uniform scheme uses the same

averaging mask for every vertex, and a stationary scheme uses the same averaging mask for every level of subdivision.

In the next chapter I will introduce three uniform and stationary subdivision schemes, i.e. Loop, Doo-Sabin and Catmull-Clark. Table 4.1 shows how they are classified:

	approximating face scheme	*vertex scheme*
quadrilateral	Catmull-Clark	Doo-Sabin
triangular	Loop	

Table 4.1: Classification of three subdivision schemes.

4.3 Analyzing the limit surface

The schemes that will be discussed are generalizations of spline refinement procedures and differ from them in the fact that they handle *extraordinary points*. Splines are defined over a strict triangular or quadrilateral mesh, where all vertices have valence 6 resp. 4 (called *ordinary* or *regular* points), while subdivision introduces irregular vertices, where a different number of edges and faces coincide. These vertices are called extraordinary points. Extraordinary points can only be introduced in the original mesh. The subdivision scheme will only create new ordinary points and will isolate the extraordinary points. Therefore the subdivision surface will be a spline surface except for a fixed and finite number of extraordinary points, which can be 'linked' to the ones introduced in the original mesh. Because spline surfaces are perfectly smooth we only have to analyze the surface near these extraordinary points.

To study local properties of a limit surface, such as differentiability, in the vicinity of a particular control point, we define a local subdivision matrix. This matrix holds the rules for the generation of one limit point and it's immediate neighbors, similar to a local subdivision matrix for curves. We can only define such a matrix if these rules specify that a finite number of vertices are used to compute any new point. The size of the matrix depends on the *invariant neighborhood*, i.e. the k-neighborhood of the original point which influences its new position after a subdivision step. From now on S will denote the local subdivision matrix, and $\lambda_0, \lambda_1, \ldots, \lambda_n$, its eigenvalues which satisfy

$$|\lambda_0| \geq |\lambda_1| \geq \cdots \geq |\lambda_n|.$$

Let M^0 denote the array $(v_0, v_1, \ldots, v_n)^{\mathrm{T}}$ of vertices constituting the invariant neighborhood of vertex v_0. The $n+1$ by $n+1$ subdivision matrix S represents the rules for computing the refined neighborhood M^1 of v_0^1:

$$M^1 = S \cdot M^0.$$

The subsequent neighborhoods M^k can be derived by

$$M^j = S \cdot M^{j-1} = S^j \cdot M^0, \qquad j = 1, 2, \ldots .$$

4.3.1 Affine Invariance

As with curves, we expect the subdivision matrix to be affine-invariant which implies 1 to be an eigenvalue of S which corresponding right eigenvector $\vec{1} = (1, \ldots, 1)^T$.

4.3.2 Convergence

The following theorem is proven in Reif [11] performing an eigenanalysis of S:

Theorem 2 *The polyhedron converges uniformly to a continuous surface if*

$$1 = \lambda_0 > |\lambda_1|$$

and $(1, 1, \ldots, 1)^T$ *is the right eigenvector corresponding to the eigenvalue 1.*

The limit point v_0^∞ of a vertex v_0^j of P^j at any subdivision level j is

$$v_0^\infty = \sum_{i=0}^{n} l_i v_i, \tag{4.1}$$

where $L_0 = (l_0, l_1, \ldots, l_n)$ is the normalized left eigenvector of λ_0. This vector formula forms an evaluation mask for limit points. An example of an evaluation mask is given in the next chapter in case of the Loop subdivision scheme.

4.3.3 Smoothness

There are no rigorous proofs for sufficient and necessary conditions for convergence of general subdivision schemes to smooth (i.e. C^1-continuous) surfaces. I will discuss a sufficient condition for smoothness presented by Reif [11]. It is based on the concept of the *characteristic map*. The characteristic map is a smooth map from some compact domain U to $\mathbb{R}^2$ which can be assigned to stationary linear subdivision schemes. It depends only on the structure of the algorithm and not on the data.

Let $\Omega = N_2 \backslash N_1$ the ring of faces around a vertex defined as the difference between the 2-neighborhood (N_2) and 1-neighborhood (N_1) of that vertex.

Definition 7 *For a local subdivision matrix S for a vertex with valence n and $1 > |\lambda_1| = |\lambda_2| > |\lambda_3|$, the* ***characteristic map*** $\Psi : U \times \mathbb{Z}_n \to \Omega \subset \mathbb{R}^3$ *is defined by*

$$\Psi : (u, v, j) \mapsto b(u, v, j)[L_1, L_2],$$

where U is either

$$U^\triangle = \{(u, v) | u, v \geq 0 \text{ and } 1 \leq u + v \leq 2\}$$

for triangular nets or

$$U^\square = \{(u, v) | u, v \geq 0 \text{ and } 1 \leq \max\{u, v\} \leq 2\}$$

for quadrilateral nets, and L_1, L_2 are the left eigenvectors of eigenvalues λ_1 and λ_2. The function $b(u, v, j)$ is a row vector of a certain set of basis functions *$b_k(u, v, j)$*

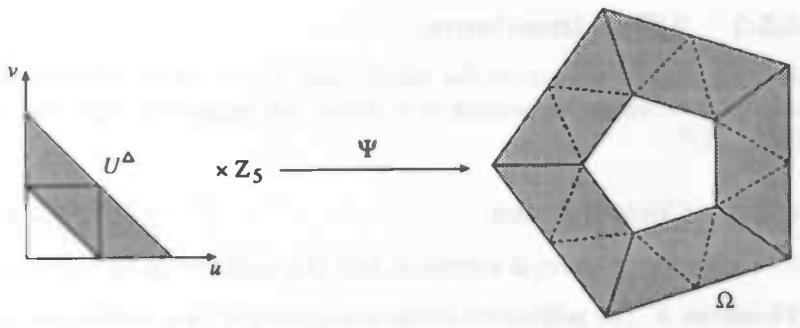


Figure 4.1: The characteristic map for a triangular scheme with a vertex with valence 5.

Theorem 3 (Reif's sufficient condition for smoothness) *Suppose the eigenvectors of a subdivision matrix form a basis, the largest three eigenvalues are real and satisfy*

$$\begin{aligned} 1 = \lambda_0 &> \lambda_1 = \lambda_2, \\ |\lambda_2| &> |\lambda_3|, \end{aligned}$$

and if the characteristic map is regular *(i.e. continuously differentiable and Jacobi matrix of maximum rank) then the corresponding algorithm generates tangent plane continuous limit surfaces. If the characteristic map is also* injective *these surfaces will also be C^1-continuous.*

Peters and Reif [10] introduced sufficient conditions for regularity and injectivity for the characteristic map for quadrilateral subdivision schemes. Let the restriction of Ψ to one (of k) segment be denoted Ψ^i (see Figure 4.2). Because of its symmetry properties analysis of Ψ can be reduced to analysis of one of these segments, say Ψ^0. Let $\Psi^0_{1,v}$ and $\Psi^0_{2,v}$ denote the partial derivatives of Ψ^0 with respect to v.

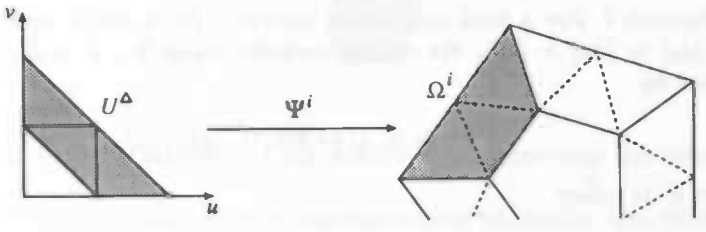


Figure 4.2: One segment of a triangular characteristic map.

Theorem 4 *If $\Psi^0_{1,v}(u,v), \Psi^0_{2,v}(u,v) > 0$ for all $(u,v) \in U$, then the characteristic map Ψ is regular and injective.*

The formula for the tangent plane at a limit position v_0^∞ is given by the left eigenvector of the eigenvalues λ_1 and λ_2 as follows. Let $L_1 = (a_0, a_1, \ldots, a_n)$

and $L_2 = (b_0, b_1, \ldots, b_n)$ be the left eigenvectors of λ_1 and λ_2, then the tangent plane is the span of the following vectors:

$$\begin{aligned} \vec{a} &= a_0 v_0 + a_1 v_1 + \cdots + a_n v_n, \quad \text{and} \\ \vec{b} &= b_0 v_0 + b_1 v_1 + \cdots + b_n v_n. \end{aligned}$$

These vector formulas for the tangents to the limit surface form two derivative masks or *tangent masks* for a limit point. An example of tangent masks is given in the next chapter in case of the Loop subdivision scheme.

4.4 Subdivision Surfaces vs. other Representations

In this section I will compare some important properties of subdivision surfaces with three other (smooth) surface representations: traditional *splines*, *implicit surfaces* (or iso-surfaces) and *variational surfaces*.

Efficiency: Subdivision algorithms are easy to implement and are computationally efficient, because only a small number of neighboring vertices are used to compute new ones and subdivision rules are few and simple. Splines are a little more efficient, because the subdivision rules are a little simpler. On the other hand, implicit surfaces are much more costly. An algorithm such as marching cubes is required to generate the polygonal approximation needed for rendering. Variational surfaces are even less efficient, because a global optimization problem has to be solved each time the surface is changed.

Arbitrary topology: Subdivision can handle arbitrary topology quite well without losing efficiency. Classic spline approaches lack this quality. Because all vertices should have the same valence, only closed of genus 1 can be represented. Implicit modeling methods handle arbitrary topology better than both above, but the genus, precise location and connectivity of a surface are difficult to control. Variational surfaces can handle arbitrary topology even better than any other method, but computational cost can be high.

Surface features: Subdivision allows flexible control over surface features. We can choose the locations of control points and manipulate the coefficients of the subdivision rules to create features such as creases control boundaries. Splines offer a more precise control, but it is computationally expensive to include features like arbitrary positioning. Implicit surfaces are very difficult to control, since all modeling is performed indirectly and there is much potential for undesirable interactions between different parts of the surface. Variational surfaces provide the most flexibility and exact control for creating features.

Complex geometry: For interactive applications such as the internet, efficiency is of the utmost importance. Therefore subdivision applies very well for these applications. Concepts such as level-of-detail rendering and compression can be handled sufficiently.

Chapter 5

Three Subdivision Algorithms

Abstract. In this chapter I will discuss the three most prominent subdivision algorithms creating smooth subdivision surfaces. I will explain how they are based on B-spline refinement processes and why they converge to C^1 limit surfaces.

5.1 Loop Algorithm

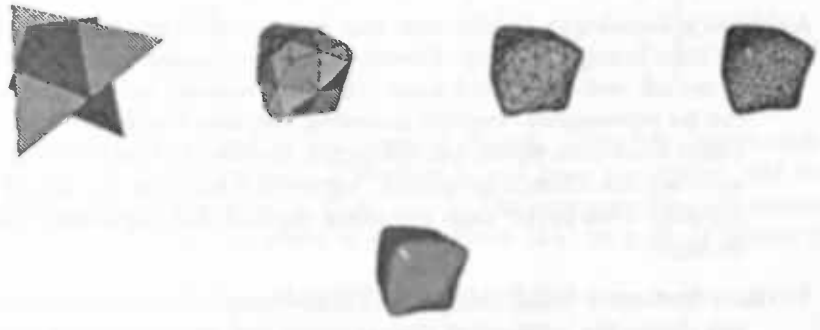


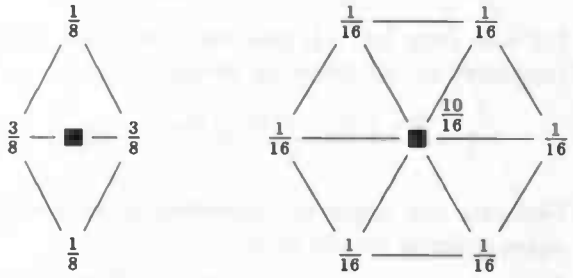
Figure 5.1: Three steps in a Loop subdivision scheme and the limit surface.

The Loop algorithm [9] is an approximating triangular face scheme. As all triangular face schemes the splitting step consists of adding vertices at each edge in the original mesh. Each triangle will be then subdivided into four sub-triangles by reconnecting the new vertices.

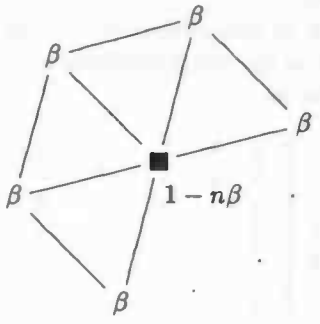
5.1.1 Three-directional Box Spline

The Loop algorithm is a generalization of three-directional quartic box spline refinement (see [16]). Unlike more conventional splines, the three-directional

box spline is defined on a regular triangular grid. Refinement rules for the averaging step can be expressed by the following two averaging masks for an edge rule and a vertex rule:



The vertex rule is only applicable to vertices with valence 6. The Loop algorithm generalizes this rule for vertices with valence $n \neq 6$ (extraordinary vertices). The averaging mask becomes



where the coefficient β should be chosen carefully to guarantee C^1 limit surfaces. Loop proposed the following coefficient

$$\beta = \frac{5}{8n} - \frac{(3 + 2\cos\frac{2\pi}{n})^2}{64n}$$

which, as we will later see, guarantees smooth limit surfaces.

5.1.2 The Algorithm

The Loop algorithm can be divided in the following substeps. These steps are shown in Figure 5.2. Let v_0 denote a vertex having neighbors $v_1, \ldots, v_n$.

1. For each vertex v_0, generate a new *vertex point* (v_0') which is calculated by the following formula:

$$v_0' = \frac{av_0 + bv_1 + \cdots + bv_n}{8},$$

where

$$a = 8 - nb$$

and

$$b = \frac{5}{n} - \frac{(3 + 2\cos\frac{2\pi}{n})^2}{8n}.$$

2. For each edge $\{v_0, v_i\}$, generate a new *edge point* (v_i') which is calculated by the following formula:

$$v_i' = \frac{3v_0 + v_{i-1} + 3v_i + v_{i+1}}{8}, \quad \text{for} \quad i = 1, \ldots, n.$$

3. Generate new edges by connecting all new edge points of the edges defining the old face.
4. Generate new edges by connecting each new vertex point to the new edge points of all old edges incident on the old vertex.

New faces are then defined as those enclosed by new edges.

5.1.3 Convergence and Smoothness

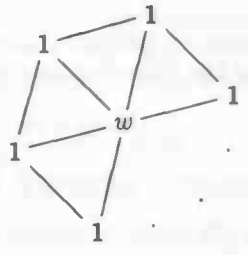
To prove convergence at extraordinary vertices in the Loop scheme we have to perform some checks on the subdivision matrix. The $(n+1)\times(n+1)$ subdivision matrix can be written as

$$S = \frac{1}{8}\begin{pmatrix} a & b & b & b & \cdots & \cdots & b & b \\ 3 & 3 & 1 & 0 & \cdots & \cdots & 0 & 1 \\ 3 & 1 & 3 & 1 & 0 & \cdots & 0 & 0 \\ 3 & 0 & 1 & 3 & 1 & 0 & \cdots & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 3 & 0 & \cdots & \cdots & 0 & 1 & 3 & 1 \\ 3 & 1 & 0 & \cdots & \cdots & 0 & 1 & 3 \end{pmatrix},$$

where a and b are defined in the previous section. They are chosen such that all the rows of S sum up to 1. Therefore $\lambda_0 = 1$ is an eigenvalue, and its corresponding right eigenvector is $(1, 1, \ldots, 1)$. The corresponding left eigenvector is $(w, 1, 1, \ldots, 1)/(w + n)$ where $w = 3/b$. From (4.1) on page 21 follows that

$$v_0^\infty = \frac{wv_0 + v_1 + \cdots + v_n}{w + n}.$$

This can be represented in the following evaluation mask:



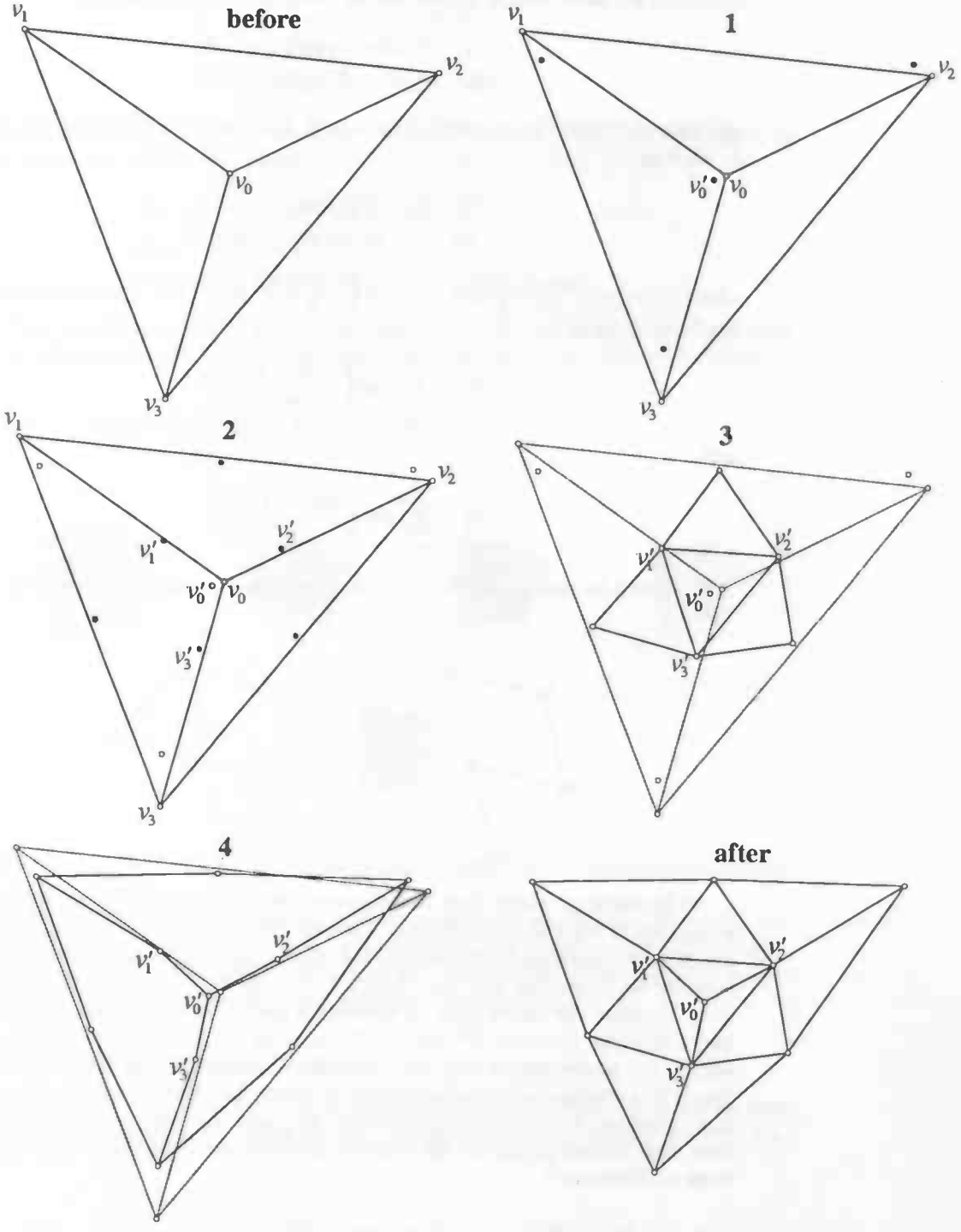


Figure 5.2: The Loop algorithm

To prove that Loop's subdivision matrix satisfies Reif's sufficient smoothness condition we have to look at the second three eigenvalues and find:

$$\begin{aligned} \lambda_{1,2} &= (3 + 2\cos(2\pi/n))/8, \\ \lambda_3 &= (3 + 2\cos(4\pi/n))/8 \end{aligned}$$

and thus the condition is satisfied for $n > 3$. For the left eigenvector for λ_1 and λ_2 we find

$$\begin{aligned} L_1 &= (0, c_1, c_2, \ldots, c_n) \\ L_2 &= (0, s_1, s_2, \ldots, s_n) \end{aligned}$$

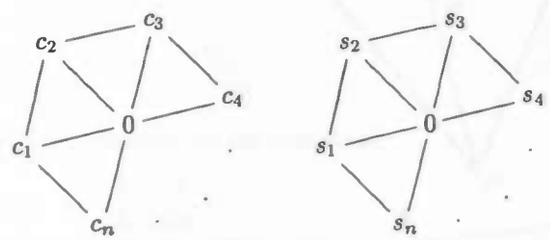
where $c_i = \cos\left(\frac{2\pi(i-1)}{n}\right)$ and $s_i = \sin\left(\frac{2\pi(i-1)}{n}\right)$. And thus, the tangent vectors can be written as

$$\vec{v_1} = \sum_{i=1}^{n} \cos\left(\frac{2\pi(i-1)}{n}\right) v_i$$

and

$$\vec{v_2} = \sum_{i=1}^{n} \sin\left(\frac{2\pi(i-1)}{n}\right) v_i.$$

This can be represented in the following tangent masks (see page 23):



It remains to prove that the characteristic map of the Loop subdivision algorithm is regular and injective. Umlauf [15] shows that with some adjustments, the conditions for regularity and injectivity can be applied to subdivision schemes for triangular nets. Let Φ be the characteristic map defined for a triangular domain. He shows that by covering a quadrilateral domain $U^{\square}$ with the two triangular domains $U^{\triangle}$ and $\lambda_1 U^{\triangle}$ the map for quadrilateral domain (Ψ^i) adopts the properties of map for triangular domains (Φ^i and $\lambda\Phi^i$). And therefore if Φ^0 satisfies conditions presented by Peters and Reif, Φ will also be regular and injective. Umlauf then proves that $\Phi^0_{1,v}$ and $\Phi^0_{2,v}$ are positive by proving that their Bézier points are all positive. Smoothness is therefore guaranteed in view of Theorem 4.

5.1.4 New Faces and Vertices Generation

Before step n, let F_n be the number of faces, E_n the number of edges and V_n the number of vertices. After step n, let F_{n+1}, E_{n+1} and V_{n+1} be the number

of faces, edges and vertices respectively, we have:

$$\begin{aligned} V_{n+1} &= V_n + E_n, \\ F_{n+1} &= 4F_n, \\ E_{n+1} &= 2E_n + 3F_n. \end{aligned}$$

Through recursion we can calculate the number of vertices, faces and edges as a formula of the initial number of vertices, faces and edges, giving:

$$\begin{aligned} V_n &= V_0 + (2^n - 1)E_0 + 21 * 5^{n-3}F_0, \qquad n > 2, \\ F_n &= 4^n F_0, \\ E_n &= 2^n E_0 + 3(2^{2n-1} - 2^{n-1})F_0. \end{aligned}$$

These formulas are quite useful, because we can use them to predict the size of our data structures and the computational costs for levels of approximation.

5.2 Doo-Sabin Algorithm

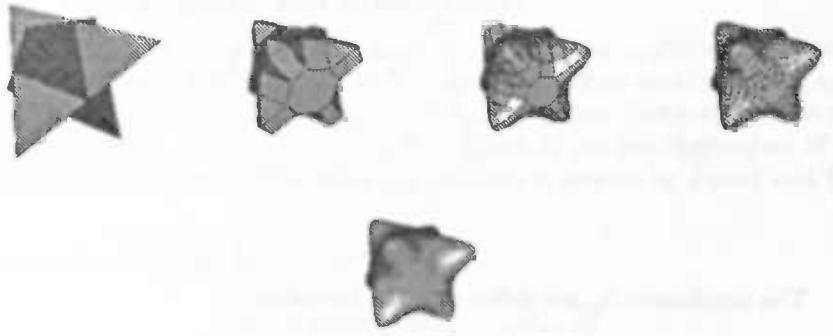


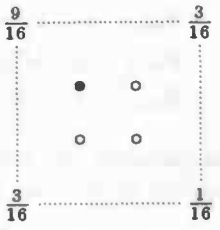
Figure 5.3: Three steps in a Doo-Sabin subdivision scheme and the limit surface.

Donald Doo and Malcolm Sabin [3] developed a quadrilateral vertex scheme. As well as all quadrilateral vertex schemes the splitting step consists of replacing each face, vertex and edge with a new face. Intuitively it can be seen as first 'cutting' off the vertices and then 'cutting' off the edges.

5.2.1 Biquadratic B-spline

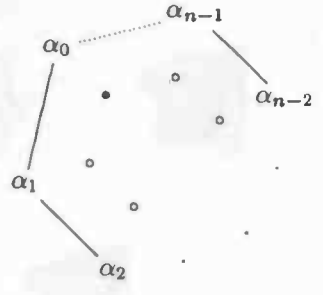
The Doo-Sabin algorithm is a generalization of biquadratic B-spline surface refinement. Refinement rules for the averaging step can be expressed by the

following mask for each of the four vertices in a quadrilateral:



The new vertex is represented by the solid circle. The other three points (open circles) are determined in the same way, by rotating the entries in the mask. A new face is constructed, connecting these four new vertices. Each vertex belongs to a new face corresponding to an old vertex and two new faces corresponding to the two old edges incident at that old vertex. This is shown in the previous chapter.

The Doo-Sabin algorithm generalizes this rule to faces with a different number of vertices, say n ($n \neq 4$). The averaging mask becomes



The coefficients α_i are defined by the formulas:

$$\begin{aligned} \alpha_0 &= \frac{1}{4} + \frac{5}{4n} \\ \alpha_i &= \frac{3 + 2\cos\frac{2i\pi}{n}}{4n}, \qquad \text{for} \quad i = 1, \ldots, n-1. \end{aligned}$$

5.2.2 The Algorithm

The algorithm can be divided in the following substeps. These steps are shown in Figure 5.4.

1. For each vertex of each face of the polyhedron, generate a new point which is the average of the vertex, the two *edge points* (the midpoints of the edges that are adjacent to this vertex in the polygon) and the face point of the face.
2. For each face, connect the new points that have been generated for each vertex of the face similar to the way their originals where connected in the old face.
3. For each edge, connect the new points that have been generated for the faces that are adjacent to this edge.

4. For each vertex, connect the new points that have been generated for the faces that are adjacent to this vertex.

5.2.3 Face Classification

Because for each face, vertex and edge a new face is formed, they can be easily classified. And characteristics can be generalized for each class.

Type F: an n-sided face gives a new and smaller n-sided face within itself. This type of new face is termed type F (formed by a face).

Type V: a vertex with valence n produces an n-sided face. This is termed type V (formed by a vertex).

Type E: each edge produces a 4-sided face. This is termed type E (formed by an edge).

These three types of faces are linked together the same way the elements they where formed by where connected in the original polyhedron. They share common edges and vertices and form the new polyhedron.

5.2.4 Convergence and Smoothness

The eigenvalues of the subdivision matrix are found by applying the discrete Fourier transform to $\alpha_i = \alpha_{n-i}, j \in \mathbb{Z}_n$ (see [10]). Affine-invariance and symmetry, i.e. $\sum_i \alpha_i = 1$ and $\alpha_i = \alpha_{n-i}, j \in \mathbb{Z}_n$ imply that this transform is real and of the form $\hat{\alpha} = [1, \hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_{n-1}]$. These $\hat{\alpha}_i$ are the eigenvalues of the local subdivision matrix. The following theorem is proven by Peters and Reif [10]:

Theorem 5 *If* $\lambda := \hat{\alpha}_1 = \hat{\alpha}_{n-1}$ *satisfies*

$$1 > \lambda > \max\{\frac{1}{4}, |\hat{\alpha}_2|, \ldots, |\hat{\alpha}_{n-2}|\},$$

$$128\lambda^2(1-\lambda) - 7\lambda - 2 + 9\lambda\cos\frac{2\pi}{n} > 0,$$

then the generated limit surface is smooth for almost every initial polyhedron.

In particular, the coefficients presented by D. Doo and M. Sabin comply with this condition, and hence the algorithm generates smooth limit surfaces.

5.2.5 New Faces and Vertices Generation

Before step n of the subdivision process, let F_n be the number of faces, E_n the number of edges and V_n the number of vertices. After step n, let F_{n+1}, E_{n+1} and V_{n+1} be the number of faces, edges and vertices respectively, we have:

$$F_{n+1} = F_n + E_n + V_n$$

Each old edge gives a quadrilateral with 4 new vertices; however any 2 adjacent edges in a face share 1 new vertex; hence,

$$V_{n+1} = \frac{4E_n}{2} = 2E_n$$

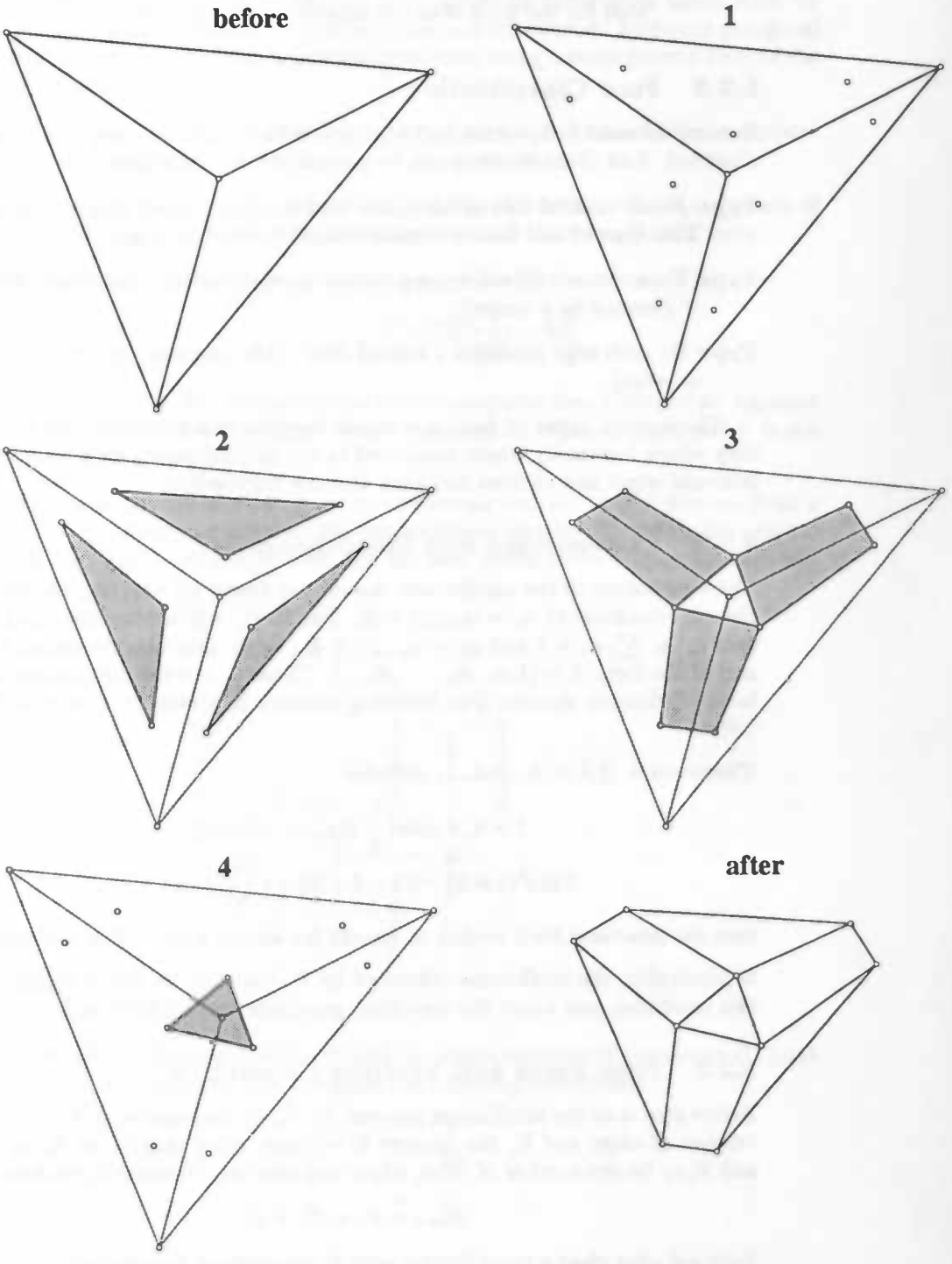


Figure 5.4: The Doo-Sabin algorithm

Each old edge gives a quadrilateral with 4 new edges; hence,

$$E_{n+1} = 4E_n.$$

Through recursion we can calculate the number of vertices, faces and edges as a formula of the initial number of vertices, faces and edges, giving:

$$\begin{aligned} V_n &= 2^{2n-1}E_0, \\ F_n &= F_0 + (2^{2n-1} - 1)E_0 + V_0, \\ E_n &= 4^n E_0. \end{aligned}$$

5.3 Catmull-Clark Algorithm

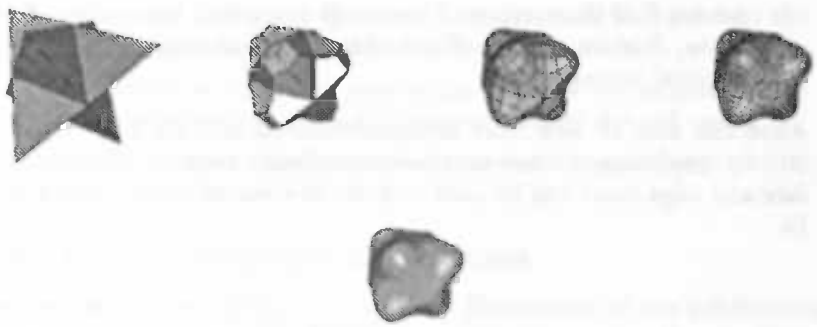
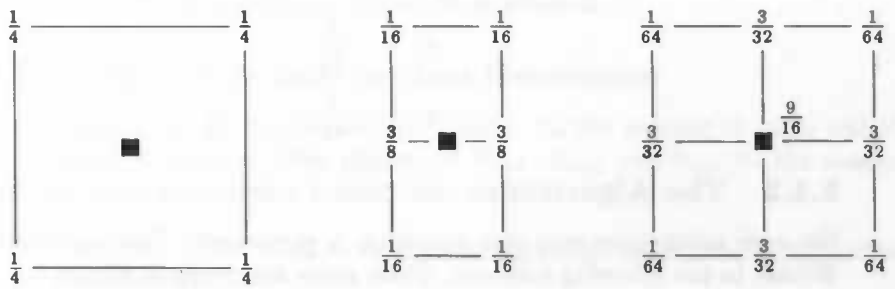


Figure 5.5: Three steps in a Catmull-Clark subdivision scheme and the limit surface.

E. Catmull and J. Clark [1] developed a quadrilateral face scheme. As well as all quadrilateral face schemes the splitting step consists of adding vertices at each edge and face in the original mesh. Each face will then be subdivided into a number of smaller faces.

5.3.1 Bicubic B-spline

The Catmull-Clark algorithm is a generalization of bicubic B-spline surface refinement. There are three averaging masks representing a face rule, an edge rule and a vertex rule:



These rules are only applicable to faces, edges and vertices in a strictly quadrilateral mesh. The Catmull-Clark algorithm generalizes these rules for arbitrary meshes, therefore three new rules are introduced. Arbitrary polygonal meshes cab be reduced to a quadrilateral mesh using a more general form of the algorithm:

Face rule: A new face point is computed as the average of all the vertices defining the face.

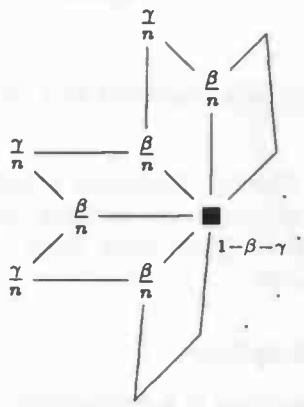
Edge rule: A new edge point is computed as the average of the endpoints of the edge and newly computed face points of the two adjacent faces.

Vertex rule: A new vertex point is computed with the following formula:

$$V_{\text{new}} = \frac{F + 2E + (n-3)V}{n},$$

where F is the average of the newly computed face points of all adjacent faces, E is the average of the midpoints of all adjacent edges and V is the original vertex.

After this step all new faces are quadrilaterals and the only difference from a strictly quadrilateral mesh are the extraordinary vertices. Therefore the original face and edge mask can be used and the new vertex (with valence n) mask will be



where the coefficients β and γ should be chosen carefully to guarantee C^1 limit surfaces. Catmull and Clark suggest the following coefficients:

$$\begin{aligned} \beta &= \frac{3}{2k} \\ \gamma &= \frac{1}{4k}. \end{aligned}$$

5.3.2 The Algorithm

For each subdivision step this algorithm is performed. The algorithm can be divided in the following substeps. These steps are shown in Figure 5.6.

1. For each face, generate a new *face point* which is the average of all the old points defining the face.
2. For each edge, generate a new *edge point* which is the average of the midpoint of the old edge with the average of the two new face points of the faces sharing the edge.
3. For each vertex, generate a new *vertex point* which is calculated by the following formula:

$$V_{\text{new}} = \frac{F + 2E + (n-3)V}{n},$$

where F is the average of the face points of the faces adjacent to the old vertex, E is the average of the midpoints of the edges adjacent to the old vertex, V is the corresponding vertex from the original polyhedron and n is the valence of the old vertex.

4. Generate new edges by connecting each new face point to the new edge points of the edges defining the old face.
5. Generate new edges by connecting each new vertex point to the new edge points of all old edges incident on the old vertex.

New faces are then defined as those enclosed by new edges.

5.3.3 Convergence and Smoothness

To prove convergence, let $\alpha := 1 - \beta - \gamma$. Eigenvalues of the subdivision matrix are given by Peters and Reif [10] and they suffice to the conditions for convergence, because

$$\begin{aligned} \lambda_0 &= 1 \\ \lambda_{1,2} &= \frac{4\alpha - 1 \pm \sqrt{(4\alpha-1)^2 + 8\beta - 4}}{8}. \end{aligned}$$

The following theorem is proven by Peters and Reif [10]:

Theorem 6 *Given the three weights α, β and γ. If*

$$2\left|4\alpha - 1 \pm \sqrt{(4\alpha-1)^2 + 8\beta - 4}\right| < c_n + 5 + \sqrt{(c_n+9)(c_n+1)},$$

where $c_n = \cos\frac{2\pi}{n}$, then the limit surface is smooth.

In particular, the coefficients presented by Catmull and Clark comply with this condition, and hence generate smooth limit surfaces.

5.3.4 New Faces and Vertices Generation

Before step n, let F_n be the number of faces, E_n the number of edges and V_n the number of vertices. After step n, let F_{n+1}, E_{n+1} and V_{n+1} be the number of faces, edges and vertices respectively, we have:

Vertices: $V_{n+1} = F_n + E_n + V_n$ (face points, edge points and vertex points respectively).

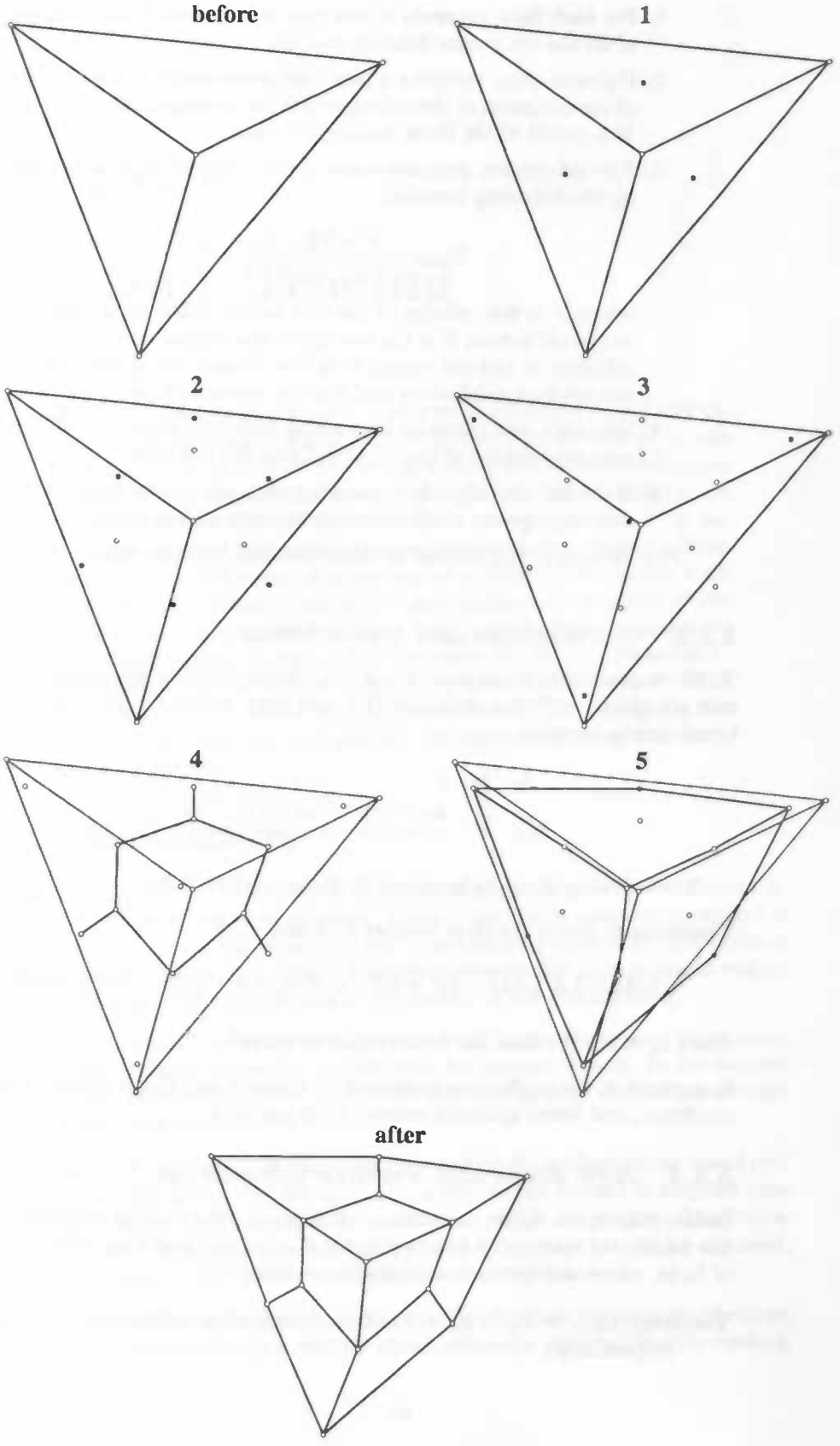


Figure 5.6: The Catmull-Clark algorithm

Faces: each old edge contributes to the construction of 4 new faces, but each two edges share one such face; hence,
$F_{n+1} = \frac{4E_n}{2} = 2E_n$.

Edges: each old edge contributes to the construction of 4 new edges; hence
$E_{n+1} = 4E_n$.

Through recursion we can calculate the number of vertices, faces and edges as a formula of the initial number of vertices, faces and edges, giving:

$$\begin{aligned} V_n &= V_0 + (2^{2n-1} - 1)E_0 + F_0, \\ F_n &= 2^{2n-1}F_0, \\ E_n &= 4^n E_0. \end{aligned}$$

Chapter 6

Contour Tracing

Abstract. In this chapter, I will address the concept of contours on subdivision curves and surfaces. Furthermore, I will present a rule for the position of contour vertices in a uniform B-spline subdivision process of any degree for curves. Subsequently I have studied the extension of this rule for the subdivision surfaces presented in the previous chapter. Moreover, I have considered the question: How well can the subdivision of a contour be predicted? It appears that for Loop and Catmull-Clark subdivision surfaces the position of the contour can be predicted in most cases, where little variation in curvature appears. At each subdivision step the contour lies in the 1-neighborhood of the contour in the previous subdivision level. This 1-neighborhood is called an *expectation band*. Unfortunately, the expectation band does not predict the contour very well for Doo-Sabin subdivision.

6.1 Introduction

Contours play an important role in 3D polyhedral graphics when data compression and rendering speed is an issue. More often highly complex geometry is needed for visualizations, requiring high computational costs to achieve optimal visual results, such as high detail. There have been a few approaches to reduce data and complexity without loss of the quality of the visualization.

- One approach is level-of-detail rendering. In this approach, more polygons are used for nearby objects than for distant objects. In the context of the previous chapters, one could say that nearby objects are further subdivided than distant objects.
- Another approach is texture mapping and its descendant image based rendering algorithm. In this approach, a high-detail texture is mapped onto an object with low complexity to simulate higher complexity. One limitation of this approach is that high quality silhouettes cannot be obtained, since this method uses a small number of polygons.
- Another approach, given by Gu et al. [5], addresses the problem of reducing complexity using a method called *silhouette clipping*. In this method,

in addition to the combination of the previous approaches, high level silhouettes are computed to clip parts of the low level projection to simulate high complexity on the silhouette.

In this last approach it is important to be able to compute a high level silhouette quickly. When using a subdivision scheme to create high level of detail, instead of computing the silhouette of the polyhedron at each subdivision level separately, an efficient way to trace the contour throughout the subdivision process could save a lot of unnecessary computations. In the following sections I will propose a hypothesis for predicting the position of a polyhedral silhouette given the position of the silhouette on the polyhedron at the previous subdivision level.

6.2 Curve Contours

Let us take a step back and start with the 2-dimensional case of subdivision curves. Let P be a piecewise linear curve or polygon in $\mathbb{R}^2$ with vertices $v_0, v_1, \ldots$ and edges $e_0, e_1, \ldots$. Figure 6.1 illustrates an example, where P is a closed polygon with 5 vertices and edges.

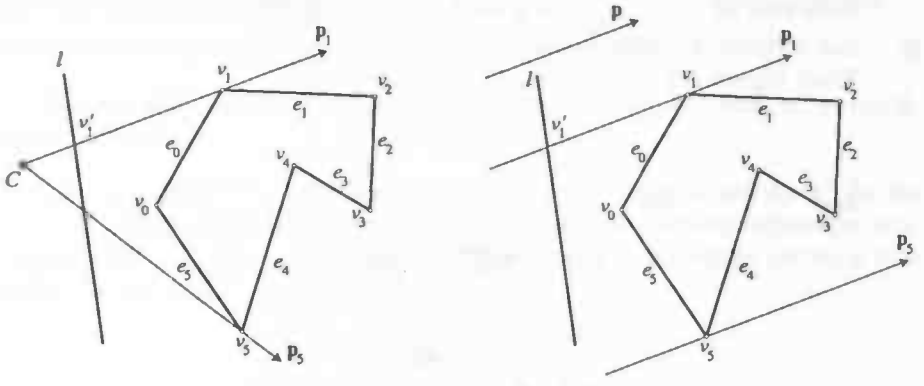


Figure 6.1: Viewing vector in a perspective and parallel projection of a polygon.

Define a parallel or perspective projection from $\mathbb{R}^2$ to a projection line l. For parallel projection, let the *viewing vector* $\mathbf{p}_i$ at a vertex v_i be the vector parallel to the projection vector $\mathbf{p}$ and intersecting v_i. For perspective projection, let the viewing vector be

$$\mathbf{p}_i = v_i - C,$$

where C is the *projection reference point* of the perspective projection.

Definition 8 *A vertex v_i of P is a* **contour vertex** *if both edges sharing v_i lie in the same halfplane defined by the supporting line of the viewing vector $\mathbf{p}_i$. The set of all contour vertices of P is called the* **contour** *of P.*

A normal of an edge determines whether the edge is either *front-facing* or *back-facing*. An edge is front-facing if the angle between its normal and the

viewing vector (at one of the edge's endpoints) is greater than $\frac{\pi}{2}$, and it is back-facing if that angle is smaller than or equal to $\frac{\pi}{2}$. It is not always possible to define a direction for a normal, that is, to define whether an edge is front- or back-facing. When P is a simple closed polygon, it is common to define the enclosed region to be the *inside* of the polygon and thus pointing each normal towards the *outside*. A contour vertex can then also be defined as a vertex incident to both a front-facing edge and a back-facing edge.

We shall look at what becomes of these contour vertices when subdivision is applied. Let us look again at Chaikin's algorithm. The averaging mask of this algorithm is $r = (r_0, r_1) = \frac{1}{2}(1,1)$. Together with the splitting step we have

$$\begin{aligned} v_{2i}^{j+1} &= \frac{3}{4}v_i^j + \frac{1}{4}v_{i+1}^j \\ v_{2i+1}^{j+1} &= \frac{1}{4}v_i^j + \frac{3}{4}v_{i+1}^j. \end{aligned}$$

Let $n_{\{a,b\}}$ be the well-defined (non necessarily unit) normal of the line through vertices $a = (a_x, a_y)$ and $b = (b_x, b_y)$ defined as follows:

$$n_{\{a,b\}} = (a_y - b_y, b_x - a_x).$$

The length of the normal is therefore equal to the length of the vector $b - a$. Note that $n_i^j := n_{\{v_i^j, v_{i+1}^j\}}$ is a normal of the edge $e_i^j = \{v_i^j, v_{i+1}^j\}$. So, if we order the vertices of a polygon clockwise, the edge normals have 'outward' direction. From Figure 6.2 it can be easily concluded that

$$n_{2i}^{j+1} = \frac{1}{2}n_i^j, \tag{6.1}$$

and

$$n_{2i+1}^{j+1} = \frac{1}{4}n_{\{v_i^j, v_{i+2}^j\}}. \tag{6.2}$$

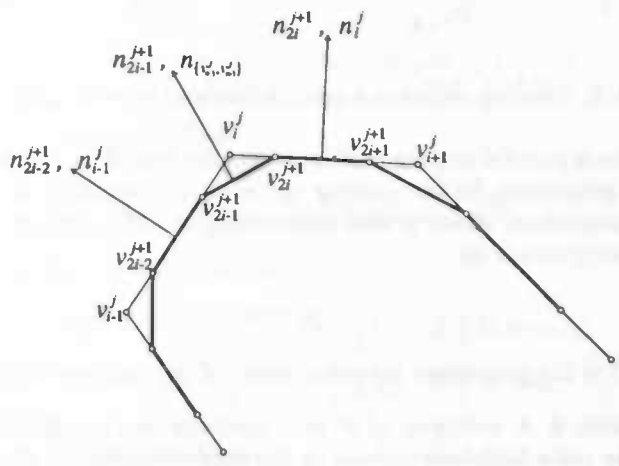


Figure 6.2: Equality of edge normal-directions in Chaikin's algorithm.

With this notion we can predict the position of a contour vertex of P in subdivision level $j+1$. This contour vertex is a direct 'descendant' of the

contour vertex we derive this prediction from. Let for example P be a closed polyhedron and let v_i^j be a contour vertex of P. If e_{i-1}^j is front-facing then e_i^j must be back-facing and vice versa. Due to (6.1), e_{2i-2}^{j+1} and e_{2i}^{j+1} are also front- and back-facing, respectively. Therefore either v_{2i-1}^{j+1} or v_{2i}^{j+1} is a contour vertex, depending of $n_{\{v_{i-1}^j, v_{i+1}^j\}}$. Following this line of reasoning, I have derived the following theorem:

Theorem 7 *Let P^0 be a piecewise linear curve or polygon and let P^j be the curve or polygon after j subdivision steps of Chaikin's algorithm. A vertex v_{2i-1}^{j+1} or v_{2i}^{j+1} of P^{j+1} can only be a contour vertex of P^{j+1} if v_i^j is a contour vertex of P^j.*

Proof To prove this theorem we have to prove that v_{2i-1}^{j+1} and v_{2i}^{j+1} can not be contour vertices of P^{j+1} if v_i^j is not a contour vertex of P^j. If v_i^j is not a contour vertex, edges e_{i-1}^j and e_i^j are both either front-facing or back-facing. Due to (6.1), both e_{2i-2}^{j+1} and e_{2i}^{j+1} are also front-facing (resp. back-facing). Furthermore, Figure 6.2 shows that $n_{\{v_{i-1}^j, v_{i+1}^j\}}$ lies somewhere 'between' n_{i-1}^j and n_i^j, which proves, due to (6.2), that also e_{2i-1}^{j+1} must be front-facing (resp. back-facing). This proves that neither v_{2i-1}^{j+1} or v_{2i}^{j+1} can be contour vertices.

⊠

The last theorem can be generalized for any uniform B-spline curve subdivision of degree l.

Theorem 8 *Let P^0 be a piecewise linear curve or polygon and let P^j be the curve or polygon after j subdivision steps of a uniform B-spline refinement process of degree $l+1 \geq 2$. A vertex of P^{j+1} can only be a contour vertex if it is within the set*

$$E_l(i) = \{v_{2i-\frac{l}{2}}^{j+1}, \ldots, v_{2i+\frac{l}{2}}^{j+1}\}$$

if l is even, or

$$E_l(i) = \{v_{2i-\frac{l+1}{2}}^{j+1}, \ldots, v_{2i+\frac{l-1}{2}}^{j+1}\}$$

if l is odd. The set E_l is associated with a contour vertex v_i^j of P^j and is called the expectation band *of v_i^j.*

Proof A uniform B-spline subdivision scheme has an averaging mask $r = \frac{1}{2^l}\left(\binom{l}{0}, \binom{l}{1}, \ldots, \binom{l}{l}\right)$. The indices differ in symmetry if l is odd or even:

$$r = \begin{cases} (r_{-\frac{l}{2}}, \ldots, r_{\frac{l}{2}}) & \text{if } l \text{ is even} \\ (r_{-\frac{l-1}{2}}, \ldots, r_{\frac{l+1}{2}}) & \text{if } l \text{ is odd} \end{cases}$$

When l is even, we can compute two successive normals as follows

$$\begin{aligned}
n_{2i}^{j+1} &= n_{\{v_{2i}^{j+1}, v_{2i+1}^{j+1}\}} \\
&= n_{\{\frac{1}{2^l}\sum_{k=-\frac{l}{2}}^{\frac{l}{2}} \binom{l}{\frac{l}{2}+k} \dot{v}_{2i+k}^{j+1}, \frac{1}{2^l}\sum_{k=-\frac{l}{2}}^{\frac{l}{2}} \binom{l}{\frac{l}{2}+k} \dot{v}_{2i+k+1}^{j+1}\}} \\
&= (\frac{1}{2^l}\sum_{k=-\frac{l}{2}}^{\frac{l}{2}} \binom{l}{\frac{l}{2}+k} \dot{v}_{2i+k,y}^{j+1} - \frac{1}{2^l}\sum_{k=-\frac{l}{2}}^{\frac{l}{2}} \binom{l}{\frac{l}{2}+k} \dot{v}_{2i+k+1,y}^{j+1}, \\
&\qquad \frac{1}{2^l}\sum_{k=-\frac{l}{2}}^{\frac{l}{2}} \binom{l}{\frac{l}{2}+k} \dot{v}_{2i+k+1,x}^{j+1} - \frac{1}{2^l}\sum_{k=-\frac{l}{2}}^{\frac{l}{2}} \binom{l}{\frac{l}{2}+k} \dot{v}_{2i+k,x}^{j+1}) \\
&= \frac{1}{2^l}\sum_{k=-\frac{l}{2}}^{\frac{l}{2}} \binom{l}{\frac{l}{2}+k} (\dot{v}_{2i+k,y}^{j+1} - \dot{v}_{2i+k+1,y}^{j+1}, \dot{v}_{2i+k+1,x}^{j+1} - \dot{v}_{2i+k,x}^{j+1}) \\
&= \frac{1}{2^l}\sum_{k=-\frac{l}{2}}^{\frac{l}{2}} \binom{l}{\frac{l}{2}+k} n_{\{\dot{v}_{2i+k}^{j+1}, \dot{v}_{2i+k+1}^{j+1}\}}
\end{aligned}$$

where the $\dot{v}_i^{j+1}$ are introduced in Section 3.2. Similarly

$$n_{2i+1}^{j+1} = \frac{1}{2^l}\sum_{k=-\frac{l}{2}}^{\frac{l}{2}} \binom{l}{\frac{l}{2}+k} n_{\{\dot{v}_{2i+k+1}^{j+1}, \dot{v}_{2i+k+2}^{j+1}\}}$$

We define $C_{l,i}^j$ to be the set of edge normals defining an edge normal (I) of the next subdivision level $(j+1)$. If all these normals are either front-facing or back-facing, the resulting normal will face the same way, because it is a weighted average of those normals, and all the weights (r_k) are positive. For n_{2i}^{j+1} this yields

$$C_{l,i}^j = \{n_{\{\dot{v}_{2i-\frac{l}{2}}^{j+1}, \dot{v}_{2i-\frac{l}{2}+1}^{j+1}\}}, \ldots, n_{\{\dot{v}_{2i+\frac{l}{2}}^{j+1}, \dot{v}_{2i+\frac{l}{2}+1}^{j+1}\}}\}$$

We can see from Figure 3.2 that

$$n_{\{\dot{v}_{2i}^{j+1}, \dot{v}_{2i+1}^{j+1}\}} = n_{\{\dot{v}_{2i+1}^{j+1}, \dot{v}_{2i+2}^{j+1}\}} = \frac{1}{2} n_i^j$$

for any i and j. And therefore we have

$$\begin{aligned}
C_{l,i}^j &= \{n_{i-\frac{l}{4}}^j, \ldots, n_{i+\frac{l}{4}}^j\} \quad \text{if } \tfrac{l}{2} \text{ is even} \\
C_{l,i}^j &= \{n_{i-\frac{l+2}{4}}^j, \ldots, n_{i+\frac{l-2}{4}}^j\} \quad \text{if } \tfrac{l}{2} \text{ is odd}
\end{aligned}$$

For n_{2i+1}^{j+1} similar sets can be found:

$$\begin{aligned}
C_{l,i}^j &= \{n_{i-\frac{l}{4}}^j, \ldots, n_{i+\frac{l}{4}}^j\} \quad \text{if } \tfrac{l}{2} \text{ is even} \\
C_{l,i}^j &= \{n_{i-\frac{l-2}{4}}^j, \ldots, n_{i+\frac{l+2}{4}}^j\} \quad \text{if } \tfrac{l}{2} \text{ is odd}
\end{aligned}$$

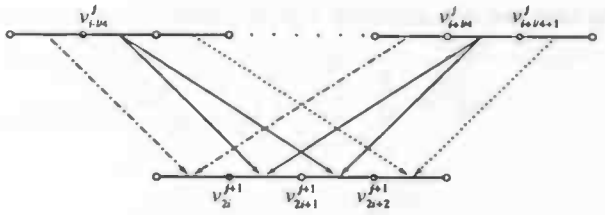


Figure 6.3: Dependency of normal in B-spline subdivision step of degree $l+1$, where $l/2$ is even.

We distinguish the two cases. First the situation where $l/2$ is even. In this case both n_{2i}^{j+1} and n_{2i+1}^{j+1} have the same set C_l. Now, if $v_{i-\frac{l}{4}}^{j}$ is a contour vertex, the set of possible contour vertices generated by $v_{i-\frac{l}{4}}^{j}$ of P^{j+1} is bounded to the right by v_{2i}^{j+1}. And if $v_{i+\frac{l}{4}+1}^{j}$ is a contour vertex, the set of possible contour vertices generated by $v_{i+\frac{l}{4}+1}^{j}$ of P^{j+1} is bounded to the left by v_{2i+2}^{j+1}. In other words, if v_i^j is a contour vertex of P^j the set of possible contour vertices of P^{j+1} generated by v_i^j is the set

$$E_l(i) = \{v_{2i-\frac{l}{2}}^{j+1}, \ldots, v_{2i+\frac{l}{2}}^{j+1}\}.$$

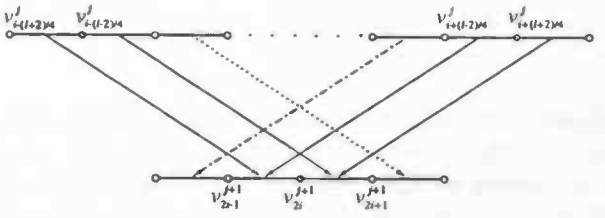


Figure 6.4: Dependency of normals in B-spline subdivision step of degree $l+1$, where $l/2$ is odd.

In the situation where $l/2$ is odd, we shall look at the normals n_{2i-1}^{j+1} and n_{2i}^{j+1}. These two normals have the same set C_l, bounded by the normals $n_{i-\frac{l+2}{4}}^{j}$ and $n_{i+\frac{l-2}{4}}^{j}$. Therefore, if $v_{i-\frac{l+2}{4}}^{j}$ is a contour vertex, the set of possible contour vertices generated by $v_{i-\frac{l+2}{4}}^{j}$ of P^{j+1} is bounded to the right by v_{2i-1}^{j+1}. And if $v_{i+\frac{l-2}{4}+1}^{j}$ is a contour vertex, the set of possible contour vertices generated by $v_{i+\frac{l-2}{4}+1}^{j}$ of P^{j+1} is bounded to the left by v_{2i+1}^{j+1}. In other words, if v_i^j is a contour vertex of P^j the set of possible contour vertices of P^{j+1} generated by v_i^j is the set

$$E_l(i) = \{v_{2i-\frac{l}{2}}^{j+1}, \ldots, v_{2i+\frac{l}{2}}^{j+1}\}.$$

This concludes the proof for the situation, where l is even. The other situation, where l is odd, is proven in the same way with only a difference in indices.

When l is odd, we can compute two successive normals as follows

$$n_{2i}^{j+1} = \frac{1}{2^l} \sum_{k=-\frac{l-1}{2}}^{\frac{l+1}{2}} \binom{l}{\frac{l-1}{2}+k} n_{\{v_{2i+k}^{j+1}, v_{2i+k+1}^{j+1}\}}$$

and

$$n_{2i+1}^{j+1} = \frac{1}{2^l} \sum_{k=-\frac{l-1}{2}}^{\frac{l+1}{2}} \binom{l}{\frac{l-1}{2}+k} n_{\{v_{2i+k+1}^{j+1}, v_{2i+k+2}^{j+1}\}}$$

For n_{2i}^{j+1} this yields

$$C_{l,i}^{j} = \{n_{\{v_{2i-\frac{l-1}{2}}^{j+1}, v_{2i-\frac{l-1}{2}+1}^{j+1}\}}, \ldots, n_{\{v_{2i+\frac{l+1}{2}}^{j+1}, v_{2i+\frac{l+1}{2}+1}^{j+1}\}}\}$$

Here two cases can be distinguished as follows:

$$C_{l,i}^{j} = \{n_{i-\frac{l-1}{4}}^{j}, \ldots, n_{i+\frac{l-1}{4}}^{j}\} \quad \text{if } \tfrac{l-1}{2} \text{ is even } (\tfrac{l+1}{2} \text{ is odd})$$

$$C_{l,i}^{j} = \{n_{i-\frac{l+1}{4}}^{j}, \ldots, n_{i+\frac{l+1}{4}}^{j}\} \quad \text{if } \tfrac{l-1}{2} \text{ is odd } (\tfrac{l+1}{2} \text{ is even})$$

For n_{2i+1}^{j+1} similar sets can be found:

$$C_{l,i}^{j} = \{n_{i-\frac{l-1}{4}}^{j}, \ldots, n_{i+\frac{l+3}{4}}^{j}\} \quad \text{if } \tfrac{l-1}{2} \text{ is even}$$

$$C_{l,i}^{j} = \{n_{i-\frac{l-3}{4}}^{j}, \ldots, n_{i+\frac{l+1}{4}}^{j}\} \quad \text{if } \tfrac{l-1}{2} \text{ is odd}$$

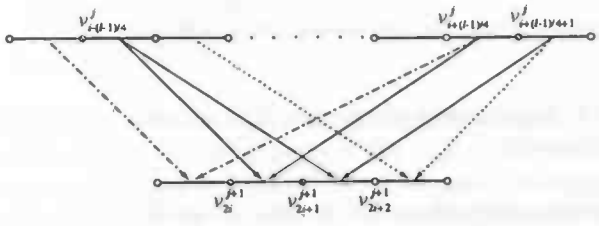


Figure 6.5: Dependency of normals in B-spline subdivision step of degree $l+1$, where $(l-1)/2$ is even.

Again, we distinguish the two cases. First we consider the situation where $(l-1)/2$ is even for n_{2i}^{j+1} and n_{2i+1}^{j+1}. It can be concluded from Figure 6.2 that if $v_{i-\frac{l-1}{4}}^{j}$ is a contour vertex, the set of possible contour vertices generated by $v_{i-\frac{l-1}{4}}^{j}$ of P^{j+1} is bounded to the right by v_{2i}^{j+1}. And if $v_{i+\frac{l-1}{4}+1}^{j} = v_{i+\frac{l+3}{4}}^{j}$ is a contour vertex, the set of possible contour vertices generated by $v_{i+\frac{l-1}{4}+1}^{j}$ of P^{j+1} is bounded to the left by v_{2i+1}^{j+1}. In other words, if v_i^j is a contour vertex of P^j the set of possible contour vertices of P^{j+1} generated by v_i^j is the set

$$E_l(i) = \{v_{2i-\frac{l+1}{2}}^{j+1}, \ldots, v_{2i+\frac{l-1}{2}}^{j+1}\}.$$

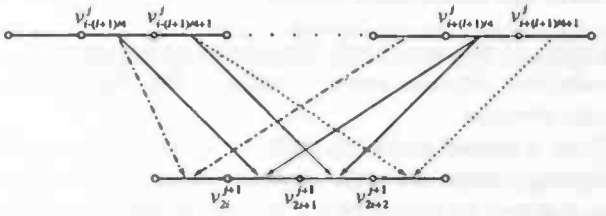


Figure 6.6: Dependency of normals in B-spline subdivision step of degree $l+1$, where $(l-1)/2$ is odd.

For the situation where $(l-1)/2$ is odd, it can be concluded from Figure 6.2 that if $v^j_{i-\frac{l+1}{4}+1} = v^j_{i-\frac{l-3}{4}}$ is a contour vertex, the set of possible contour vertices generated by $v^j_{i-\frac{l+1}{4}+1}$ of P^{j+1} is bounded to the right by v^{j+1}_{2i+1}. And if $v^j_{i+\frac{l+1}{4}+1}$ is a contour vertex, the set of possible contour vertices generated by $v^j_{i+\frac{l+1}{4}+1}$ of P^{j+1} is bounded to the left by v^{j+1}_{2i+2}. In other words, if v^j_i is a contour vertex of P^j the set of possible contour vertices of P^{j+1} generated by v^j_i is the set

$$E_l(i) = \{v^{j+1}_{2i-\frac{l+1}{2}}, \ldots, v^{j+1}_{2i+\frac{l-1}{2}}\}.$$

This concludes the proof.

⊠

With this theorem a significant computational reduction can be made when tracing a contour throughout a uniform B-spline subdivision process. This proves an inexpensive way to find contour vertices of high-level subdivision curves. Instead of first performing a number of subdivision algorithms and second checking for all vertices to be in the contour, the following steps can be performed to reduce the number of vertices that have to be checked:

1. Find the contour of the initial curve by checking all vertices.
2. Perform the subdivision steps to reach the desired curve. At each level, find the contour for the curve at that level using Theorem 8. This will finally result in the desired contour.

The number of vertices to be checked for each contour vertex is $l+1$. Let the initial curve P^0 have n vertices and n_c contour vertices, where $n_c \leq n$ (n_c is usually a lot smaller than n). For the subdivided curve P^1, instead of checking all $2n$ vertices, only $(l+1)n_c$ vertices have to be checked. Assuming that each contour vertex generates exactly one contour vertex in the next subdivision level, we can conclude that for a subdivided curve P^j, instead of checking all $2^j n$ vertices, only $j(l+1)n_c$ have to be checked during the whole subdivision process.

6.3 Surface Contours

The extension of the approach described in the previous section from two to three dimensions requires some altering of the definitions, but they are intuitively quite obvious.

Let P be a piecewise linear surface, for example a polyhedron, and let $\mathbf{p}_i$ be the viewing vector through a vertex v_i of P. Furthermore, define $\mathbf{p}_{ij}$ as the plane spanned by two viewing vectors $\mathbf{p}_i$ and $\mathbf{p}_j$. Note that the following definition is only applicable when working with planar faces only. The next section deals with situations where faces are not necessary planar.

Definition 9 *An edge e_{v_i,v_j} of P is called a contour edge if either it is a boundary edge or if both faces sharing e_{v_i,v_j} lie in the same halfspace defined by $\mathbf{p}_{ij}$.*

The endpoints of a contour edge can be denoted as *contour vertices*. It proves that all contour vertices are incident with two or more contour edges. On the other hand, not each edge with two contour endpoints is a contour edge. This leads to the conclusion that there are no contour vertices without any neighboring contour vertices.

Definition 10 *The set of all contour edges of P is called the contour of P. We will denote the contour of P by C_P.*

The contour consists of one or more closed sets of edges, i.e. each edge of the contour adjoins to one or more other contour edges. As a rule of thumb (see [5]), if a high resolution polyhedron has n_f faces, and it is tessellated evenly, the contour is usually made up of $O(n_f)$ edges. Let C'_P be the projection of C_P on a certain projection plane (or view plane) p.

Definition 11 *The silhouette of P on p is defined as the outline of the union of all regions enclosed by C'_P.*

The contour of an object specifies the boundary between the visible and invisible part of the object. Invisible parts of the object may then be culled out, resulting in a reasonable data reduction. The silhouette distinguishes the parts of the projection plane which are occupied by the projection of the object and the parts that are not.

6.4 Quadrilateral schemes

One has to be careful, when dealing with quadrilateral faces schemes (e.g. Catmull-Clark and Doo-Sabin), because non-planar faces can be created. Normally this does not cause fundamental problems, because the further an object is subdivided, the more each non-planar face will approximate a planar face. A renderer will usually triangulate all faces before the rendering starts. But where contours are concerned, they might cause some problems. Because there are different normals on one face some contour edges might not be detected. A solution for this problems is close at hand. But first, we need a new definition for a face.

Definition 12 *A face is a smooth surface, lying within the convex hull of a set of three or more vertices. The surface interpolates the boundary edges, which are line segments connecting each two successive vertices (including the last vertex with the first) in the set.*

The face has to be closed within the hull and not self-intersecting. In this case, each pair of points on a face can be connected by a smooth curve lying completely on that face. Normals on the surface are interpolated vectors between the *vertex-normals*, which are the vectors perpendicular two both edges sharing the vertex.

A contour edge is now defined as follows.

Definition 13 *An edge is a contour edge if it is either a boundary edge, or if it is shared by two faces f_i and f_j, the vertex-normals of f_i in both endpoints are front-facing*[1] *and the vertex-normals of f_j in both endpoints are back-facing, or vice versa.*

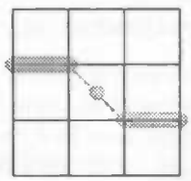


Figure 6.7: A 'hole' in the contour generated by a non-planar face. Such a face is called a *contour face*.

In this definition not all contour vertices are shared by two or more contour edges. In this case 'holes' seem to appear in the contour as in Figure 6.7, which was previously defined as a (number of) closed set of edges. Here two contour vertices do not share an edge, but they do share face. Now, we have to introduce two new definitions:

Definition 14 *A vertex is a contour vertex if it is either a boundary vertex, or if it is shared by a number of faces for which the vertex-normals in that vertex are not all front-facing or all back-facing.*

Definition 15 *A face is a contour face if its vertex-normals in its defining vertices are not all front-facing or all back-facing.*

If we triangulate each contour face new contour edges arise (and sometimes old ones disappear), but no new contour faces arise. If we do this, the problem of 'holes' in the contour is solved. This is a smart step before rendering a projected polyhedron, but we might not want to triangulate any face until the object is sufficiently subdivided to preserve uniformity. Only after the final subdivision step the necessary faces should then be triangulated to create the proper contour.

[1] Labeling normals to be either front- or back-facing (i.e. determining their direction) is usually a choice of the person who created the object. Commonly when referring to a closed polyhedron, normals are pointing away from it's 'inside'.

6.5 Analyzing Contours in Surface Subdivision

The next logical step would be to extend Theorem 8 to three dimensions in a similar manner as with curves. The expectation band is then defined as a set of edges in a subdivided polyhedron P^{j+1}, determined by the contour edges of P^j, which are the most likely edges to be contour edges of P^{j+1}. The extension to three-dimensional tensor product B-spline subdivision surfaces seems obvious: The expectation band $E_l(i)$ would be defined by the tensor product of the two two-dimensional bands of one specific vertex generated by the two B-spline curves passing through that vertex.

Based on uniform B-spline subdivision for curves, I have defined the expectation band for contours in the three schemes discussed in the previous chapter. My research is about how well this band predicts the contour in subdivision schemes for surfaces as it does for curves. And if we can specify the situations where the prediction fails.

6.5.1 Testing the expectation band

The expectation band in three dimensions is visualized as a collection (a band) of connecting faces. The edges of all these faces are considered the actual expectation band, that is; the edges most likely to be contour edges after a subdivision step. For the following three subdivision schemes I have determined the expectation band as

Loop the set of edges connecting the vertices in the 1-neighborhood of vertex points generated by contour vertices and edge points generated by contour edges (see Figure 6.8).

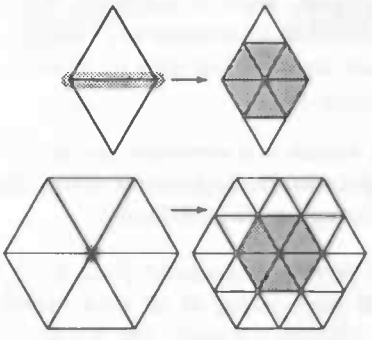


Figure 6.8: Faces of the expectation band generated by a contour edge and a regular contour vertex in Loop subdivision.

Doo-Sabin the set of edges of the vertex faces (type V) generated by contour vertices, edge faces (type E) generated by contour edges and face faces (type F) generated by contour faces (see Figure 6.9).

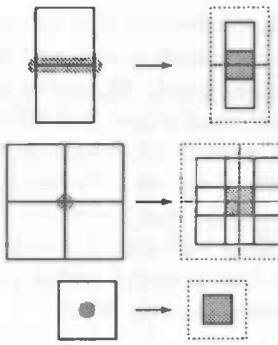


Figure 6.9: Faces of the expectation band generated by a contour edge and a regular contour vertex and a regular contour face in Doo-Sabin subdivision.

Catmull-Clark the set of edges connecting the vertices in the 1-neighborhood of vertex points generated by contour vertices and face points generated by contour faces (see Figure 6.10).

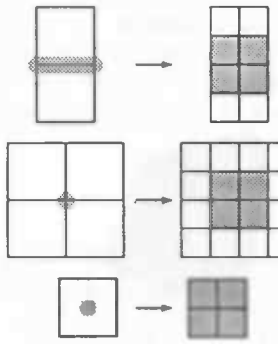


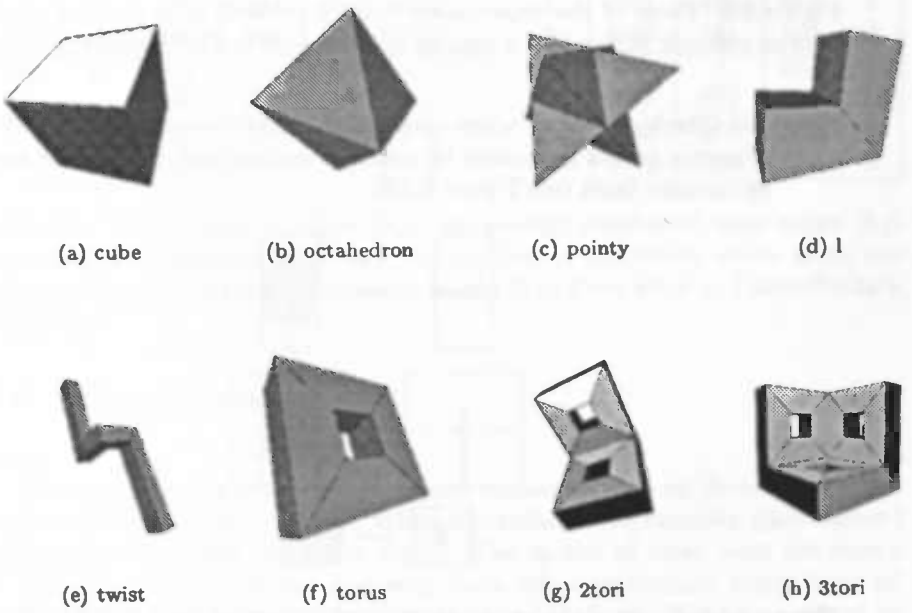
Figure 6.10: Faces of the expectation band generated by a contour edge and a regular contour vertex and a regular contour face in Catmull-Clark subdivision.

These sets enclose a region of faces in the polyhedron. The number of faces in the band generated by one contour vertex or contour edge depends on the invariant neighborhood of the subdivision scheme the same way it does for curves. For Loop, each contour vertex generates n faces (where n is the valence of that vertex) and each contour edge generates 6 faces, of which 2 faces are not generated by vertices as well. The overall *width* of the band is therefore 2 faces. Doo-Sabin and Catmull-Clark subdivision are generalized extensions of uniform B-spline curve subdivision of degree 2 and 3. For Doo-Sabin, each contour vertex, edge and face generate 1 face in the band. The overall width of the band is therefore 1 face. And for Catmull-Clark, each contour vertex generates n faces (where n is the valence of that vertex) and each contour face generates m faces (where m are the number of vertices defining the face). The overall width of the band is 2 faces.

I have generated data for testing the expectation band as follows; on a sphere surrounding a polyhedral object from 114 uniformly distributed directions a viewpoint is set determining the contour of that object, and three subdivision

steps are applied. At each subdivision step the expectation band and the contour edges (and contour faces) are determined, and it is checked whether these edges are edges of the expectation band. In case of contour faces, it is checked if all edges defining the face are band edges. For all three subdivision schemes I have performed this test with several different objects.

These tests are performed on the following polyhedral objects; a simple cube (`cube`), an octahedron (`octahedron`), a union of two tetrahedrons (`pointy`), an L-shaped object (`l`), a long bar with 3 bends of 90 degrees (`twist`), a torus (`torus`), a union of two tori (`2tori`) and a union of three tori (`3tori`). The first two objects are convex, the rest is not.



(a) cube (b) octahedron (c) pointy (d) l

(e) twist (f) torus (g) 2tori (h) 3tori

Figure 6.11: Eight polyhedral objects tested.

6.5.2 Test results

In the tables Table 6.1, Table 6.2 and Table 6.3 we refer to the objects in Figure 6.11. At each subdivision for each object, n_e represents the total number of edges, $\bar{n}_b$ represents the average number of edges in the expectation band over all 114 directions, $\bar{n}_c$ represents the average number of contour edges (plus contour faces) over all 114 directions and d represents the number of directions where there are contour edges (or faces) which are not part of the expectation band.

	level	n_e	$\bar{n}_b$	$\bar{n}_c$	d		level	n_e	$\bar{n}_b$	$\bar{n}_c$	d
(a)	0	18	-	5.9	-	(e)	0	54	-	15	-
	1	72	58	10.5	0		1	216	174	30.2	4
	2	288	131	21	0		2	864	389	61.6	0
	3	1152	278	41.7	0		3	3456	802	122	0
(b)	0	12	-	4.6	-	(f)	0	48	-	11.8	-
	1	48	41.5	8.35	0		1	192	152	26.6	8
	2	192	102	17	0		2	768	343	53.2	0
	3	768	223	33	0		3	3072	699	108	0
(c)	0	36	-	19	-	(g)	0	90	-	20.9	-
	1	144	126	21.8	0		1	360	288	50.6	13
	2	576	250	43.2	0		2	1440	661	102	1
	3	2304	547	83.4	0		3	5760	1330	203	0
(d)	0	30	-	8.7	-	(h)	0	126	-	28.4	-
	1	120	94.1	16.3	2		1	504	401	69	17
	2	480	208	31.5	0		2	2016	905	140	1
	3	1920	420	63.7	0		3	8064	1820	279	0

Table 6.1: The number of edges (n_e), the average number of band edges ($\bar{n}_b$) and contour edges ($\bar{n}_c$), and the number of directions where there are contour edges outside the expectation band (d) in three steps of Loop subdivision.

	level	n_e	$\bar{n}_b$	$\bar{n}_c$	d		level	n_e	$\bar{n}_b$	$\bar{n}_c$	d
(a)	0	12	-	5.9	-	(e)	0	36	-	15	-
	1	48	29.5	10.3	0		1	144	83.9	29.5	7
	2	192	61.9	20.8	0		2	576	178	59.9	10
	3	768	127	42.9	0		3	2304	365	121	27
(b)	0	12	-	4.6	-	(f)	0	32	-	11.8	-
	1	48	27.4	10.5	0		1	128	70.7	24.1	8
	2	192	62.7	21.6	0		2	512	145	48.8	16
	3	768	130	42.9	0		3	2048	293	97.1	0
(c)	0	36	-	19	-	(g)	0	60	-	21	-
	1	144	96.2	31.9	0		1	240	132	45.9	24
	2	576	191	65.7	0		2	960	283	92.9	94
	3	2304	399	132	32		3	3840	567	186	92
(d)	0	20	-	8.8	-	(h)	0	84	-	28.4	-
	1	80	46.9	16.5	6		1	336	182	64	28
	2	320	99.4	34.3	0		2	1344	394	130	93
	3	1280	207	69	0		3	5376	791	260	97

Table 6.2: The number of edges (n_e), the average number of band edges ($\bar{n}_b$) and contour edges/faces ($\bar{n}_c$), and the number of directions where there are contour edges outside the expectation band (d) in three steps of Doo-Sabin subdivision.

	level	n_e	$\bar{n}_b$	$\bar{n}_c$	d		level	n_e	$\bar{n}_b$	$\bar{n}_c$	d
(a)	0	12	-	5.9	-	(e)	0	36	-	15	-
	1	48	40.4	10.5	0		1	144	121	29.1	0
	2	192	104	20.6	0		2	576	291	57.2	0
	3	768	214	42.9	0		3	2304	580	114	0
(b)	0	12	-	4.6	-	(f)	0	32	-	11.8	-
	1	48	42	10.9	0		1	128	108	24.4	0
	2	192	101	21.8	0		2	512	242	48.7	0
	3	768	218	42.8	0		3	2048	487	99.9	0
(c)	0	36	-	19	-	(g)	0	60	-	21	-
	1	144	129	29.8	0		1	240	206	45.8	8
	2	576	284	55.9	0		2	960	470	91.3	2
	3	2304	562	111	0		3	3840	941	183	0
(d)	0	20	-	8.8	-	(h)	0	84	-	28.4	-
	1	80	65.7	16	0		1	336	287	61.8	14
	2	320	158	31.2	0		2	1344	641	124	5
	3	1280	314	63.4	0		3	5376	1290	252	0

Table 6.3: The number of edges (n_e), the average number of band edges ($\bar{n}_b$) and contour edges/faces ($\bar{n}_c$), and the number of directions where there are contour edges outside the expectation band (d) in three steps of Catmull-Clark subdivision.

6.6 Conclusion

Loop

In case of Loop subdivision in simple convex objects no flaws to the prediction of the band is detected. When somewhat more complex (non-convex) objects are tested this no longer holds. The results of these tests are shown in Table 6.1. Usually in the first step there are some contour edges 'near to' the band, i.e. within the 2-neighborhood of vertex or edge points generated by the contour. A broadening of the expectation band from a 1-neighborhood to a 2-neighborhood of vertex and edge points generated by the contour would solve this problem. Note that objects in Loop subdivision are completely triangular and therefore no contour faces have to be introduced.

The more complex an object, the more flaws are detected. This gives the impression, that in regions where the curvature varies a lot in within a smaller region (i.e. within a 1- or 2-neighborhood of a certain vertex) it is hard to predict the position of the contour after a subdivision. When `2tori` and `3tori` are tested, there is one direction in which the second subdivision gives a secondary problem. In this case there is a back-face created surrounded by front-faces 'away from' the band, i.e. not within the 2-neighborhood of vertex or edge points generated by the contour. These faces are situated on the border region of the union of 2 tori, where apparently the Gaussian curvature of the limit surface is 0. But in other case, this characteristic does not give the same result and therefore I can make no solid statement about that. Furthermore, these tests prove that the chance of this occurrence is about 1%. In general, the number of edges outside the expectation band is always very small (i.e. no more than 4 in the objects tested).

Conclusion: When finding the silhouette of a high-level Loop surface, one can confine in testing the expectation band for contour edges *after* the second subdivision step. As can been seen in Table 6.1, this gives a reasonable computational reduction, because the number of band edges are about 7 times the number of contour edges, which in turn is usually $O(\sqrt{n_f})$. Further research may follow to examine special surface features, like zero Gaussian curvature, and their influence to contour subdivision.

Doo-Sabin

The prediction of the band does not seem to hold at all when Doo-Sabin subdivision, the (generalized) 3-dimensional extension of Chaikin's algorithm, is applied. Only the two convex objects are tested flawlessly. The expectation band has a width of 1 only face. This seems to narrow for for a good prediction. All contour edges outside the band are 'near to' the band, such as in case of Loop subdivision. For all objects tested a significant number of steps generated problems, as can be seen in Table 6.2.

Conclusion: The extension of Theorem 8 is not applicable to Doo-Sabin subdivision. A broadening of the band probably only reduces the number of flaws. To do this, is to eliminate more flaws, but discarding the generality of theorem, as introduced with curves.

Catmull-Clark

Catmull and Clark's scheme is the (generalized) 3-dimensional extension of level 3 uniform B-spline subdivision for curves. It shows, given the test results in Table 6.3, for almost every subdivision step in every direction for each object the expectation band predicts the position of contour correctly. Only for the last two objects tested (`2tori` and `3tori`) a slight number of flaws to prediction occur. I cannot detect any specific surface characteristics, but is surprising that all flaws are generated by *contour surfaces* adjacent to the expectation band. The number of these faces is always small (1, 2 or 3). The second advantage is that after two subdivision steps, similar to Loop, Catmull-Clark subdivision does not create more flaws to the prediction of the band.

Conclusion: When finding the silhouette of a high-level Catmull-Clark surface, one can confine in testing the expectation band for contour edges *after* the second subdivision step. For convex polyhedra this can even be done from the first step. As can been seen in Table 6.3, this gives a reasonable computational reduction, because the number of band edges are about 5 times the number of contour edges. Further research may follow to eliminate the flaws created by contour faces.

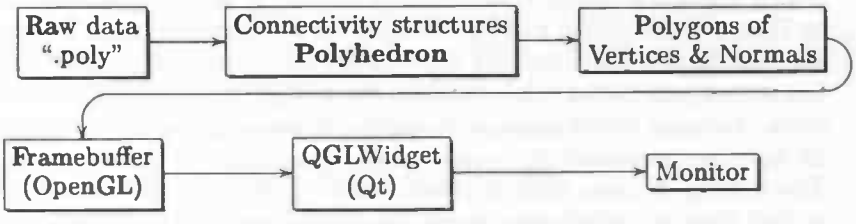
Chapter 7

Implementation

Abstract. In this chapter I will explain how I implemented the main data structures, the three subdivision algorithms and contour generation and the determination of the expectation band for these subdivision algorithms.

7.1 Introduction

Viewing pipeline for implementation:



The implementation was written in C++ (see [8] for a Maple implementation given by Ha Quang Le). It is based on the base class **Polyhedron**, which contains several data structures for and some information about a 3D polyhedron. The subdivision algorithms are implemented in classes derived from this base class. For example, an object of type **DooSabin** is a polyhedron which can be refined using the *Doo-Sabin* subdivision algorithm. A special class **Contour** holds all the information about a *parent* Polyhedron and creates the *expectation-band* when subdivided. The data for a polyhedra is read from a file in a specific *poly*-format; beginning with the number of vertices and faces, followed by a list of vertex coordinates and a list of faces, expressed by a list of vertex indices of the previous vertex list (N.B. the vertex list starts with index 1). The following example is a data representation of a cube:

```
vertices: 8
faces: 6
```

```
v 0 0 0
v 1 0 0
v 1 1 0
v 0 1 0
v 0 0 1
v 1 0 1
v 1 1 1
v 0 1 1

f 1 4 3 2
f 1 2 6 5
f 2 3 7 6
f 3 4 8 7
f 4 1 5 8
f 5 6 7 8
```

The graphical user interface (GUI) is written in Qt (a toolkit for C/C++). Qt has a special widget, for displaying OpenGL graphics. OpenGL renders the projection of a 3D polyhedron, taking care of hidden-face removal and lighting. The data of the polyhedron is presented to OpenGL by a collection of faces, each face (`GL_POLYGON`) being a set of coordinate positions (`glVertex3f()`) and normals (`glNormal3f()`) in these vertices. The surfaces can be represented by flat shading using face normals or smooth shading (Gouraud) using interpolation between vertex normals.

7.2 Basic Class Structure

A polyhedron is defined as a structure of vertices, edges and faces, but the faces implicitly define the edges. Therefore I created the three following basic classes: **Vertex**, **Face** and **Polyhedron**.

7.2.1 Vertex

A **Vertex** object defines a coordinate point in $\mathbb{R}^3$ with a normal vector (because it is point on an object surface). I have chosen for this structure, for easy interaction with OpenGL.

See Appendix A.1 for the class interface.

7.2.2 Face

This class contains the vertices defining that face. They are stored in counter-clockwise order when looking at the face from outside the object. This is also how OpenGL distinguishes the in- and outside of a face. This class is also able to give the midpoint of the face, points on its edges and normals in its vertices. Normals in the vertices, as discussed in the previous section, are calculated averages of these face normals of all faces adjacent to that vertex.

See Appendix A.2 for the class interface.

7.2.3 Polyhedron

This class contain seven structures. The first is the set of vertices defining the shape of the object. The other six different sets define the connectivity between vertices, faces and edges:

1. A numbered list of the **Vertex** objects.
2. A numbered list of the faces expressed in terms of the numbers of their defining vertices.
3. A numbered list of the edges expressed in terms of the numbers of their defining vertices.
4. The list of vertices, where each vertex is expressed in terms of the numbers of the edges coincident at this vertex.
5. The list of vertices, where each vertex is expressed in terms of the numbers of the faces coincident at this vertex.
6. The list of faces expressed in terms of the numbers of their defining edges.
7. The list of edges, where each edge is expressed in terms of the numbers of the faces coincident at this edge.

List no. 2 and 6, defining the faces, contain sorted lists of vertex and edge numbers in order to establish the inner and outer side of the object. They are sorted in the same way; in counterclockwise order, if viewed from outside the polyhedron. Note that this structure can only be applied if beforehand we denote an inside and outside of an object. Because we omit open polyhedra, this can easily be done. Lists described in 4 and 5 are not sorted.

Each structure can be accessed by a specific member function of the class **Polyhedron**, where i is a list index starting at 0:

1. **Vertex const &getVertex(uint** i **) const.**
2. **uint const *getFace(uint** i**, true) const.**
3. **uint const *getEdge(uint** i**, true) const.**
4. **uint const *getVertex(uint** i**, true) const.**
5. **uint const *getVertex(uint** i**, false) const.**
6. **uint const *getFace(uint** i**, false) const.**
7. **uint const *getEdge(uint** i**, false) const.**

Considering the functions **getFace**; each ith edge of a face is the edge between the ith and $(i + 1)$th vertex of that face (Note that the last edge is the edge between the last and first vertex). There is also an overloaded function **getFace()** that returns a **Face** object. The valence of a vertex can be obtained by the member function **getVertexSize()**, and the number of vertices and edges defining each face can be obtained by the member function **getFaceSize()**. Each edge is defined by *two* vertices and *two* faces, because there can not be any boundaries in a closed polyhedron.

The sizes of each structure can be obtained by the following member functions:

1. **uint const getNVertices() const.**
2. **uint const getNFaces() const.**
3. **uint const getNEdges() const.**
4. same as 1.
5. same as 1.
6. same as 2.
7. same as 3.

See Appendix A.3 for the class interface.

7.3 Loop Algorithm

For the implementation we look again at the Loop algorithm in Section 5.1.2 and go through it step by step.

1. For each vertex v_0, generate a new *vertex point* which is calculated by the following formula:

$$v_0' = \frac{av_0 + bv_1 + \cdots + bv_n}{8},$$

where

$$a = 8 - nb$$

and

$$b = \frac{5}{n} - \frac{(3 + 2\cos\frac{2\pi}{n})^2}{8n}.$$

To calculate this formula, we need all the neighboring vertices of each vertex. We can accomplish this by looking at all its adjacent edges (use structure 4) and the two vertices defining those edges (use structure 3). To sum all the neighboring vertices, first sum all the vertices defining all the adjacent edges and then subtract the original vertex n times. The formula is then easily calculated and the new vertices are stored in the new vertex list (of structure type 1).

2. For each edge $\{v_0, v_i\}$, generate a new *edge point* which is calculated by the following formula:

$$v_i' = \frac{3v_0 + v_{i-1} + 3v_i + v_{i+1}}{8}, \quad \text{for} \quad i = 1, \ldots, n.$$

To calculate this formula, we first add the two midpoints of the faces adjacent to this edge (use structure 7 and the **Face** class). Multiply these by 3, because they are averaged. Then add the two defining vertices of the edge (use structure 3) and divide the result by 8. New vertices are appended to the new vertex list.

3. Generate new edges by connecting all new edge points of the edges defining the old face.

4. Generate new edges by connecting each new vertex point to the new edge points of all old edges incident on the old vertex.

Step 3 and 4 are performed by looking at each old face and first form a new face by connecting each of its edge points, and second for each of its (three) vertices connect the corresponding new vertex points with the two edge points of the corresponding edges of this face adjacent to the vertex. So, for this we need structures 2 and 6.

7.4 Doo-Sabin Algorithm

For the implementation we look again at the Doo-Sabin algorithm in Section 5.2.2 and go through it step by step.

1. For each vertex of each face of the polyhedron, generate a new point which is the average of the vertex, the two *edge points* (the midpoints of the edges that are adjacent to this vertex in the polygon) and the face point of the face.

For this step we traverse the face list using the function **getFace(** i **)**. This function returns a **Face** object. For each vertex of that face we can easily generate a new vertex, using the functions **getVertex()**, **getEdgepoint()** and **getMidpoint()** of class **Face**.

The following three steps are immediately performed when generating each new vertex. This is possible, because we know each new vertex belongs to four new faces. These correspond to the face it is generated from (type F), the two edges of the that face coincident at that vertex (type E) and the old vertex (type V). In our new list of faces, these three types are distinguished and numbered as their corresponding face, edge or vertex is numbered. This way we can easily trace them.

2. For each face, connect the new points that have been generated for each vertex of the face.

3. For each edge, connect the new points that have been generated for the faces that are adjacent to this edge.

4. For each vertex, connect the new points that have been generated for the faces that are adjacent to this vertex.

As we are traversing through the faces, we calculate the vertex and edge numbers of that face using structures 2 and 6 of the class **Polyhedron**. So as we generate a new vertex for the ith vertex, we add its number to the vertex lists of the following four faces; the face of type F corresponding the face under consideration, the faces of type E corresponding the $(i-1)$th and ith edge of that face, and the face of type V corresponding that vertex. Like this, all vertices of all new faces are stored.

But the vertices should also be stored in the correct order to distinguish the outside of each face. Each new face of type F has the same number of vertices as the face is was generated from and they are stored in the same way, because the new face is a similar, smaller version of the old one. So they are already sorted correctly and we don't have to change them.

The vertices of the new faces of type E can be sorted by looking at the way they were stored. We know that the first and last two come from the same (old) faces. And because they where stored in a counterclockwise fashion they are stored in a clockwise fashion to an adjacent face. Therefore the order of the first two vertices is reversed as well as for the last two. Note that we have to consider that the last vertex and the first are succeeding vertices.

We sort the vertices in the faces of type V by looking at the faces of type E correspondent to the old edges coincident at the old vertex. To do this first make a list of edges adjacent to that vertex: when traversing the faces store the $(i-1)$th edge for each ith vertex. This way each vertex of the new face of type V corresponds to a separate edge. Then the following is done: For each vertex, consider the face of type E stored for this vertex. The vertex list of this face also contains this vertex. And because we took the $(i-1)$th edge for each ith vertex we know that the vertex previous to this one, should be the next one in the vertex list of the face under consideration. Then take this next vertex and repeat the process until all vertices are sorted.

7.5 Catmull-Clark Algorithm

For the implementation we look again at the Catmull-Clark algorithm in Section 5.3.2 and go through it step by step.

1. For each face, generate a new *face point* which is the average of all the old points defining the face.

This is done by traversing through the faces, using the function **getMidpoint()** of the **Face** class and storing these in a new vertex list (structure type 1). Note that in this new list vertices are distinguished as face, edge or vertex points and numbered as their corresponding face, edge or vertex. This way we can easily trace them.

2. For each edge, generate a new *edge point* which is the average of the midpoint of the old edge with the average of the two new face points of the faces sharing the edge.

Using structures 3 and 7 of the **Polyhedron** class we find the vertices and faces adjacent to each edge. Then average the vertices with the two face points (use the first part of the – already formed – vertex list) of those faces. These new vertex are appended to the new vertex list. In this step we store this midpoints of each edge for the next step.

3. For each vertex, generate a new *vertex point* which is calculated by the following formula:

$$V_{\text{new}} = \frac{F + 2E + (n-3)V}{n},$$

where F is the average of the face points of the faces adjacent to the old vertex, E is the average of the midpoints of the edges adjacent to the old vertex, V is the corresponding vertex from the original polyhedron and n is the valence of the old vertex.

We rewrite the formula:

$$\begin{aligned} V_{\text{new}} &= \frac{F + 2E + (n-3)V}{n} \\ &= \frac{\frac{\sum F_i}{n} + 2\frac{\sum E_i}{n} + (n-3)V}{n} \\ &= \frac{\sum(F_i + 2E_i) + n(n-3)V}{n^2} \end{aligned}$$

where F_i and E_i are face points and midpoints of edges adjacent to the original vertex. Using structures 4 and 5 we find these faces and edges. We find the face points in the first part of the – already formed – vertex list and the midpoints of the edges were stored in the previous step. The formula is now easily calculated and the vertices are again appended to the new vertex list.

4. Generate new edges by connecting each new face point to the new edge points of the edges defining the old face.

5. Generate new edges by connecting each new vertex point to the new edge points of all old edges incident on the old vertex.

Step 4 and 5 are performed by looking at each old face and for each of its vertices form a new face connecting the following four points: the corresponding new vertex point, the two edge points of the corresponding edges of this face adjacent to the vertex and the new face point. So, for this we need structures 2 and 6.

7.6 Contour

A **Contour** object finds a contour of a given **Polyhedron** using a given *projection reference point* or *viewpoint*. The parent polyhedron can be obtained using the public member function

Polyhedron const *getParent() const;.

When an object is created, it checks all edges of the parent polyhedron if it is a contour edge or not. Each edge is shared by two faces and is a contour edge if

1. one of these faces is front-facing, and

2. the other face is back-facing.

A face is *front-facing* if the angle between its face normal and the viewing vector through its midpoint is smaller than $\pi/2$ and it is *back-facing* if that angle is $\pi/2$ or greater. Subsequently, the endpoints of a contour edge are determined as contour vertices.

The array indices of the contour edges, contour vertices and contour faces in the **Polyhedron** data structures are then stored in the **Contour** object. These indices can be obtained using the public member functions

uint getEdge(uint *i*) const;
uint getVertex(uint *i*) const;
uint getFace(uint *i*) const;.

A **Contour** can be subdivided assuming its parent **Polyhedron** is already subdivided. It finds the new contour and uses the old contour to create the *expectation band*. The expectation band is a collection of faces and each of those faces can be obtained using the function

uint getBandFace(uint *i*) const;

It returns the face index in the array structure of the parent **Polyhedron**. The edges of these faces are considered to have the highest expectation of becoming a next-level contour edge. The way the band is formed depends on the way the polyhedron is subdivided. First is checked if the **Polyhedron** is a **Loop**, **DooSabin** or a **CatmullClark** object. Next, the following distinctions are made:

Loop: In this case, the expectation band is formed by the faces incident with the vertex points generated by the contour vertices or the edge points generated by the contour edges.

DooSabin: In this case, the expectation band is formed by the edge-faces (type E), vertex-faces (type V) and face-faces (type F) generated by the contour edges, vertices and faces.

CatmullClark: In this case, the expectation band is formed by the faces incident with the vertex points generated by the contour vertices and face points generated by the contour faces.

Chapter 8

Gallery

In this chapter a few series of pictures are shown which are generated by a 'screen-grab' of the screen output of the implementing program. These series consist of an initial polyhedron, followed by three subdivided polyhedra. These are all flat shaded and the edges are black. The last object is a simulation of the limit surface, generated by smooth-shading the third subdivided polyhedron and omitting the black edges. As you can see, at this resolution they look like smooth surfaces.

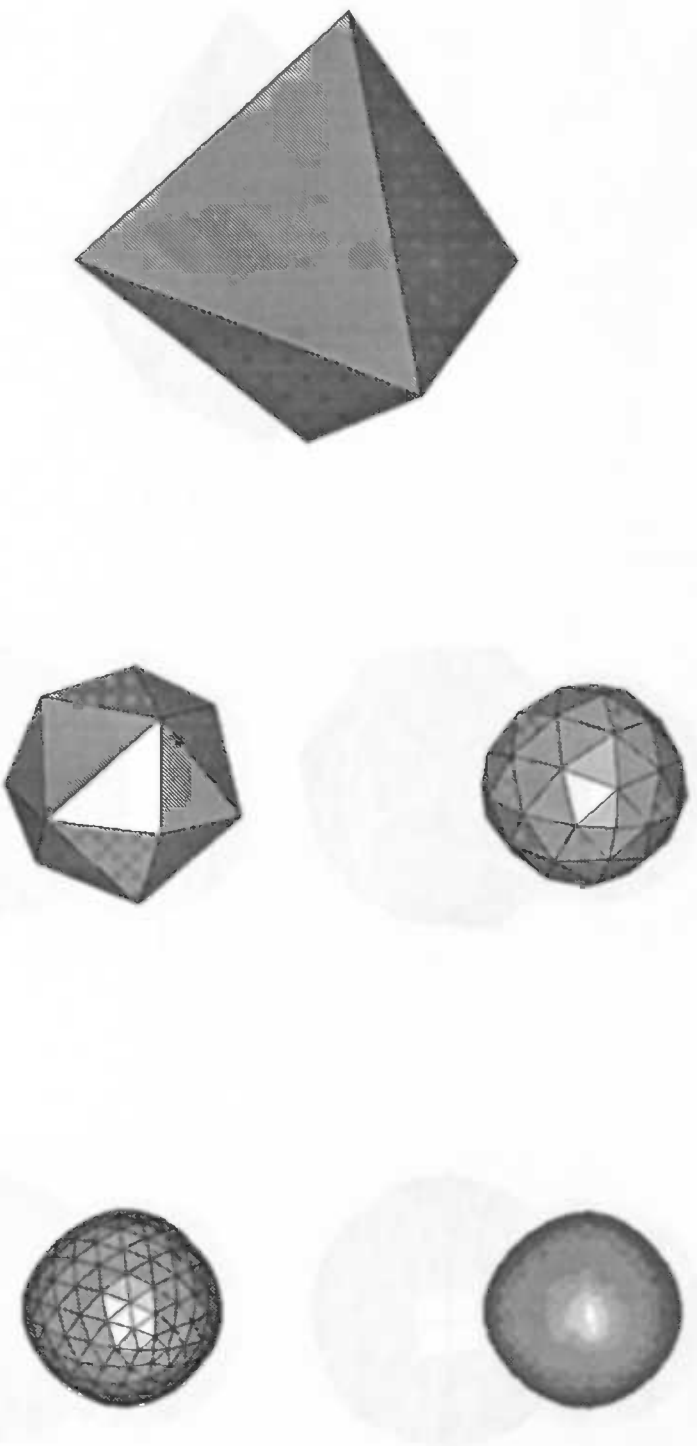


Figure 8.1: Three Loop subdivision steps of an octahedron and the limit surface.

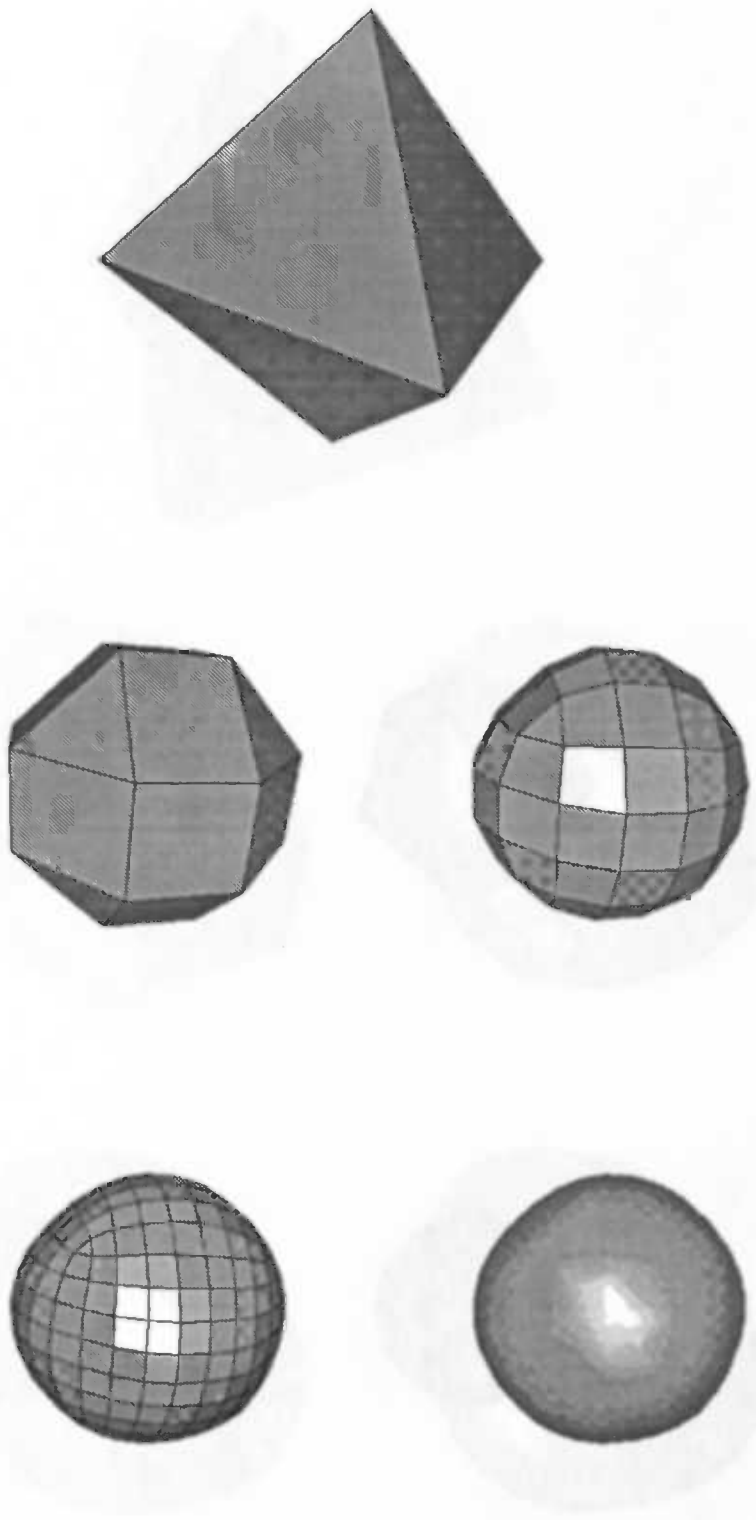


Figure 8.2: Three Doo-Sabin subdivision steps of an octahedron and the limit surface.

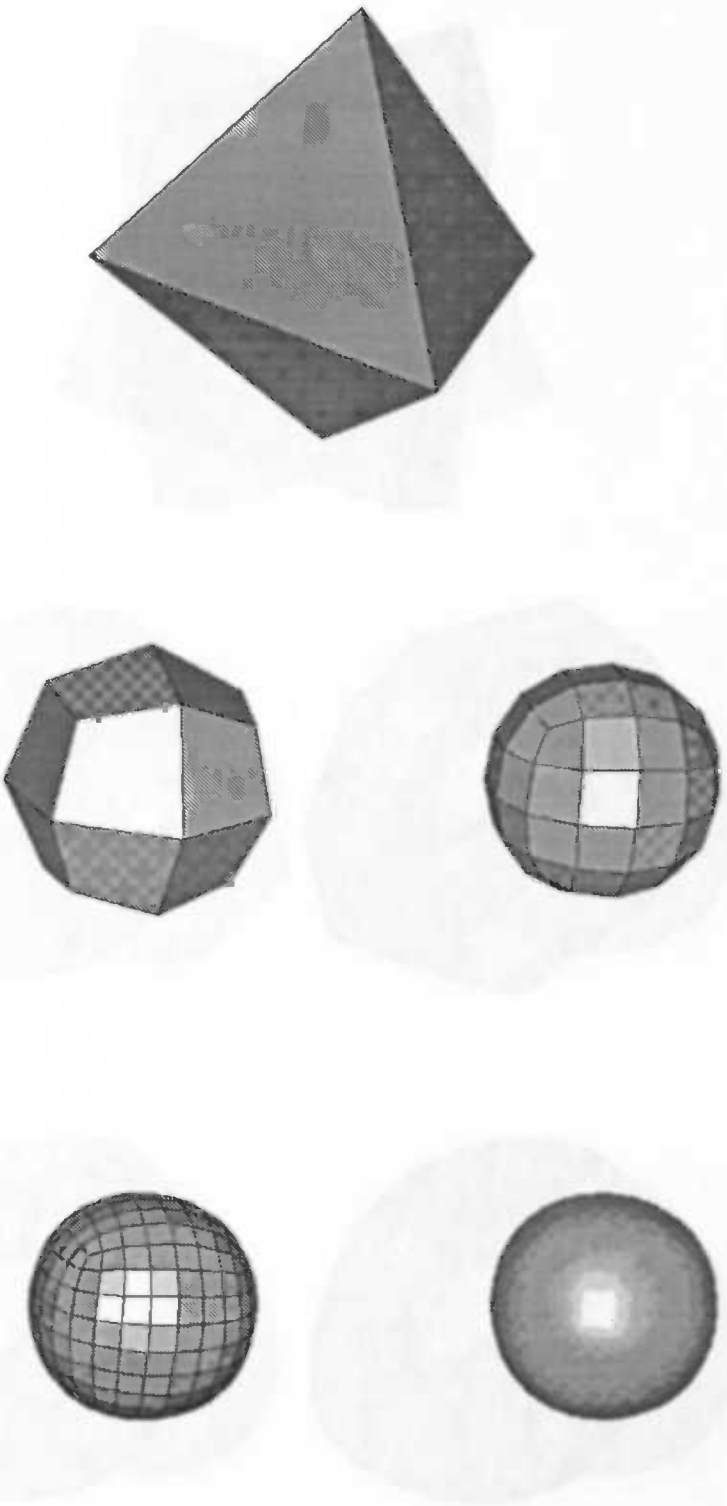


Figure 8.3: Three Catmull-Clark subdivision steps of an octahedron and the limit surface.

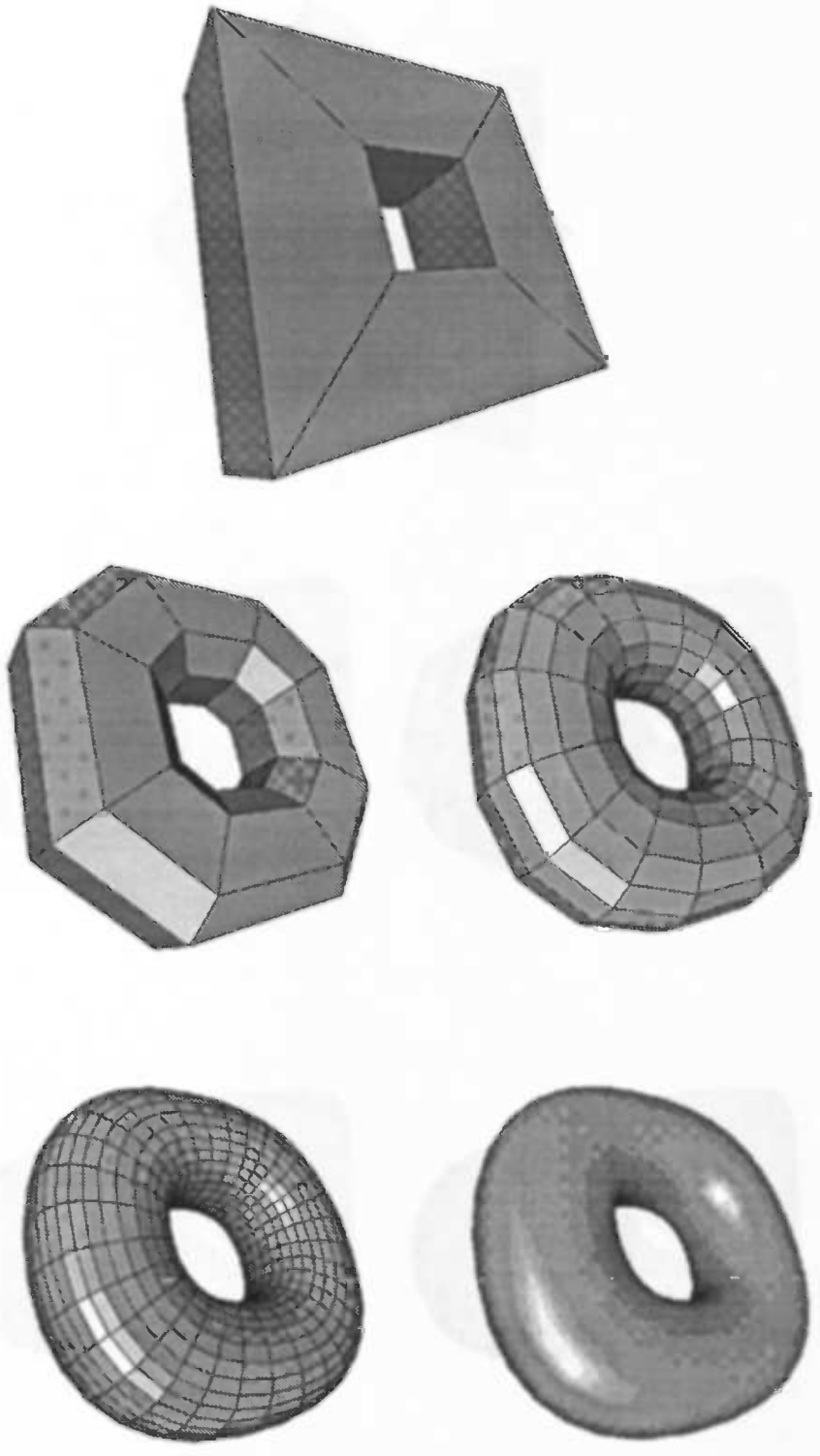


Figure 8.4: Three Doo-Sabin subdivision steps of a torus-like object and the limit surface.

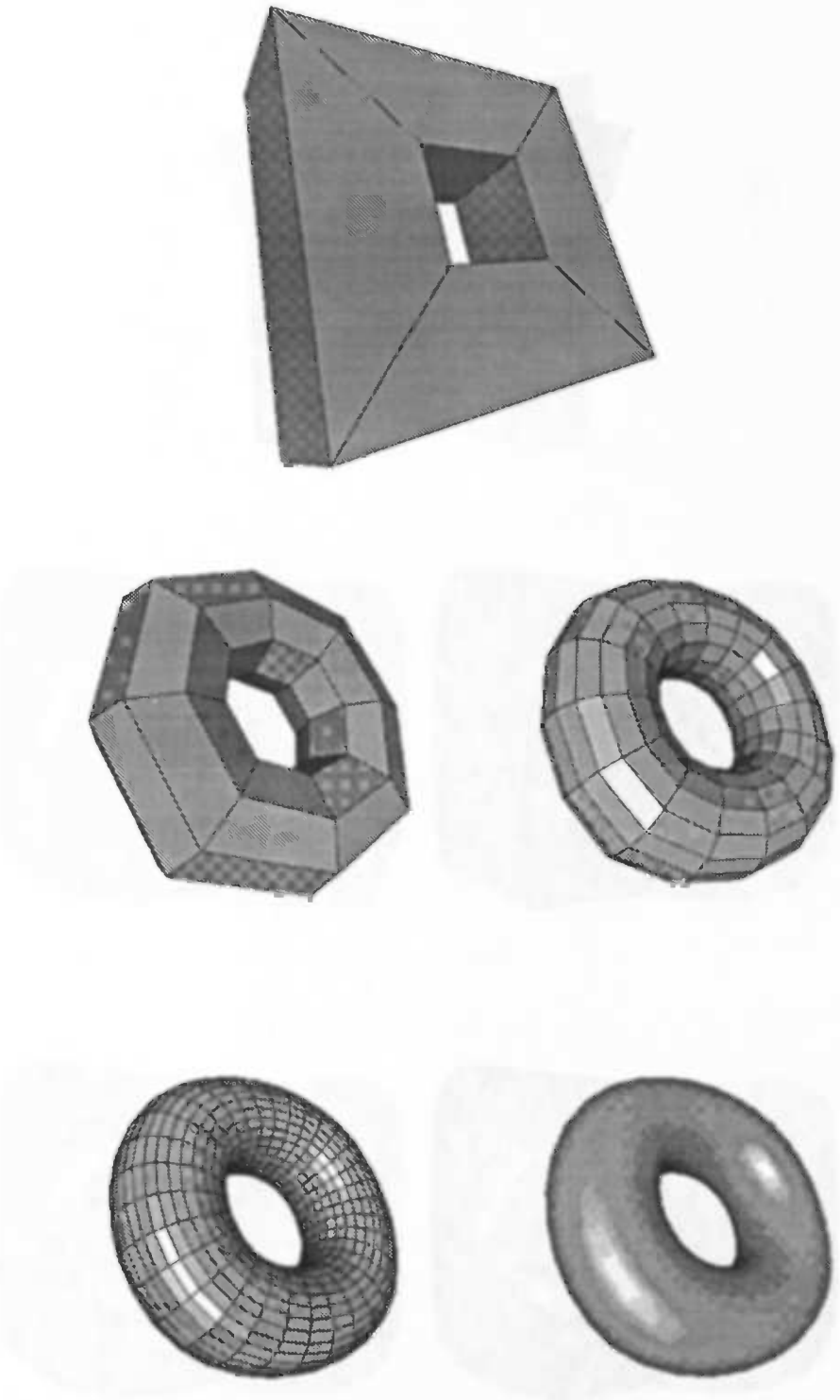


Figure 8.5: Three Catmull-Clark subdivision steps of a torus-like object and the limit surface.

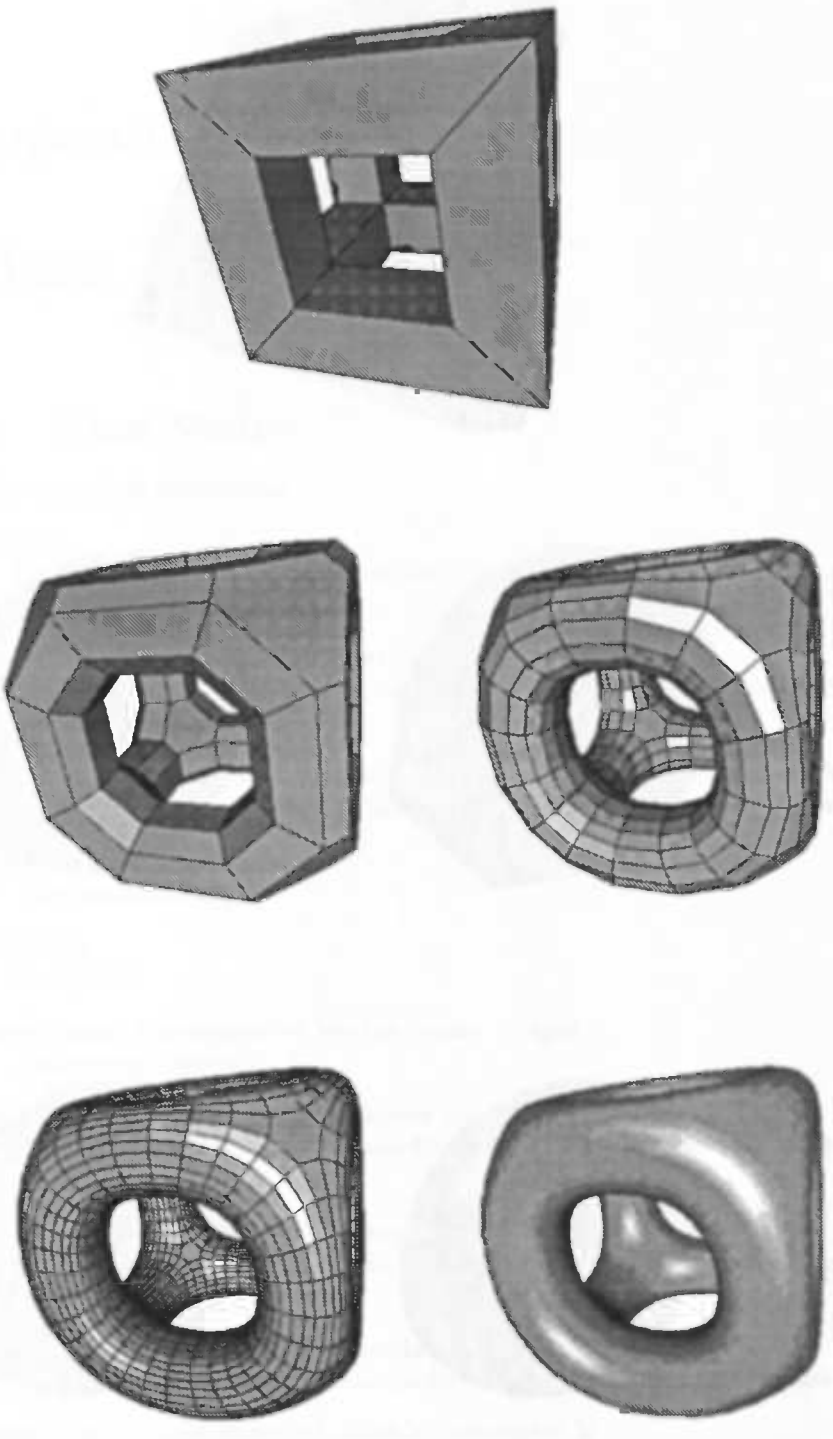


Figure 8.6: An initial polyhedron, subdivided three times with Doo-Sabin and the limit surface.

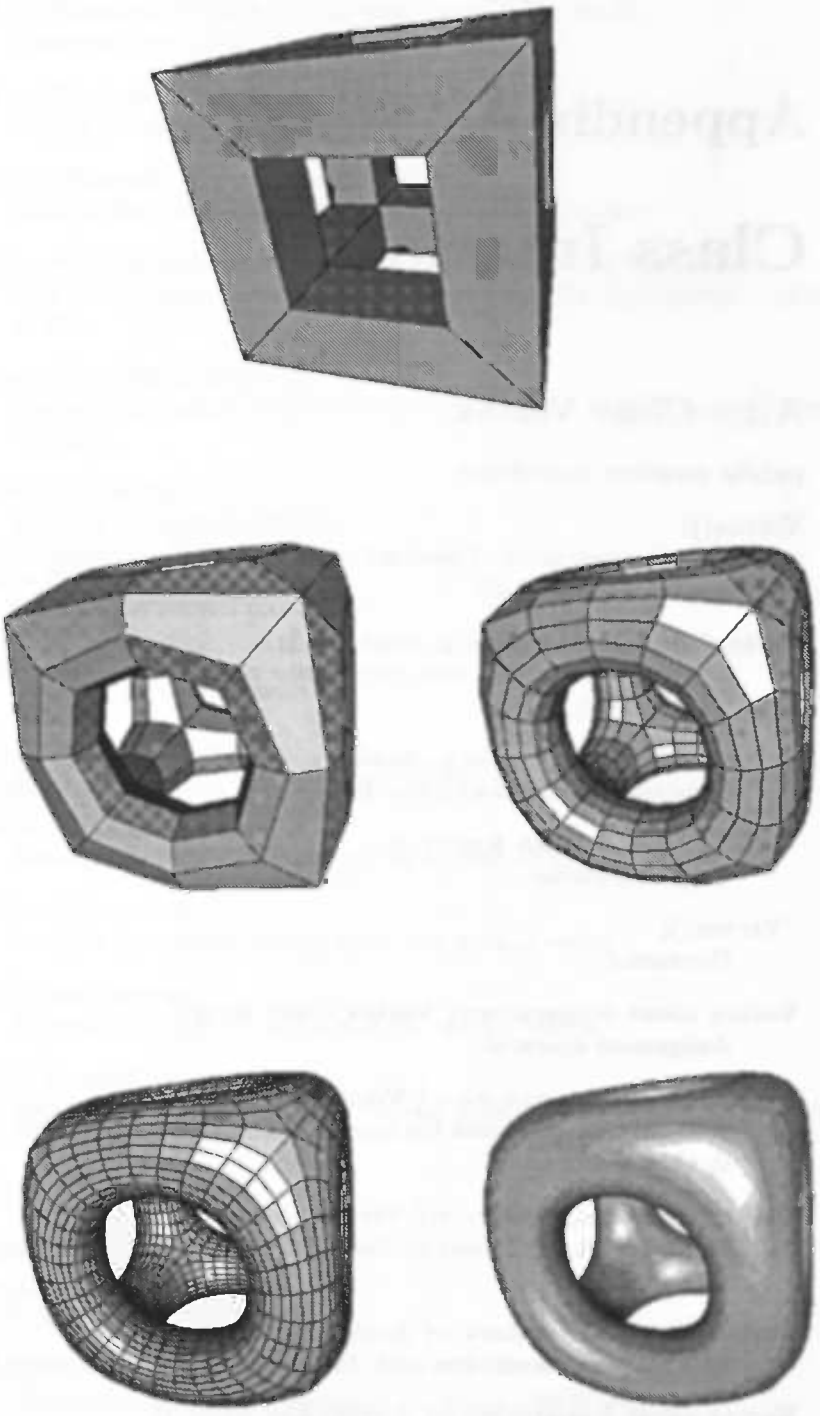


Figure 8.7: The same polyhedron, subdivided three times with Catmull-Clark and the limit surface.

Appendix A

Class Interfaces

A.1 Class Vertex

public member functions:

Vertex();
Default constructor. Constructs an object with coordinates $(0,0,0)$ and normal vector $(0,0,0)$.

Vertex(double x**, double** y**, double** z **);**
Constructs an object with coordinates x, y and z. The normal vector becomes $(0,0,0)$.

Vertex(double x**, double** y**, double** z**, double** nx**, double** ny**, double** nz **);**
Constructs an object with coordinates (x,y,z) and normal vector (nx,ny,nz).

Vertex(Vertex const &*other* **);**
Copy-constructor.

~Vertex();
Destructor.

Vertex const &operator=(Vertex const &*right* **);**
Assignment operator.

Vertex const &operator+ =(Vertex const &*right* **);**
Adds to all coordinates the coordinates of *right* and returns a reference to **this**.

Vertex const &operator– =(Vertex const &*right* **);**
Subtracts all coordinates by the coordinates of *right* and returns a reference to **this**.

Vertex const &operator* =(double *factor* **);**
Multiplies all coordinates with *factor* and returns a reference to **this**.

Vertex const &operator/ =(double *nominator* **);**
Divides all coordinates by *nominator* and returns a reference to **this**.

void setPosition(double x**, double** y**, double** z **);**
Changes the coordinates to (x, y, z).

void setNormal(double nx**, double** ny**, double** nz **);**
Changes the normal vector to (nx, ny, nz).

void setNormal(Vertex const &*other* **);**
Copies the normal vector from *other*.

void addNormal(Vertex const &*other* **);**
Adds to the normal vector the normal vector of *other*.

Vertex const &translate(double tx**, double** ty**, double** tz **);**
Add to the coordinates tx, ty and tz respectively and returns a reference to **this**.

Vertex const &normalize();
Changes the normal vector such that its length is 1 and returns a reference to **this**.

double x() const;
Returns the first coordinate.

double y() const;
Returns the second coordinate.

double z() const;
Returns the third coordinate.

double const *n() const;
Returns a pointer to the first coordinate normal vector;

double nx() const;
Returns the first coordinate of the normal vector.

double ny() const;
Returns the second coordinate of the normal vector.

double nz() const;
Returns the third coordinate of the normal vector.

double length() const;
Returns the distance between the coordinate position in $\mathbb{R}^3$ and the origin.

A.2 Class Face

public member variable:
typedef unsigned **uint;**
public member functions:

Face();
Default constructor. Constructs an object with 0 vertices.

Face(uint *nvertices* **);**
Constructs an object with *nvertices* default vertices.

Face(uint *nvertices*, **Vertex const ***vertices* **);**
Constructs an object with *nvertices* vertices listed the array *vertices*.

Face(Face const &*other* **);**
Copy-constructor.

~Face();
Destructor.

Face const &operator=(Face const &*right* **);**
Assignment operator.

void resize(uint *nvertices* **);**
Changes the number of vertices to *nvertices*. If *nvertices* is less than the previous number of vertices, the vertex list is truncated, otherwise default vertices are appended.

void newVertex(uint *i*, **Vertex const &***vertex* **);**
Changes the *i*th vertex to *vertex*.

uint getNVertices() const;
Returns the number of vertices of the face.

Vertex const &getVertex(uint *i* **) const;**
Returns a reference to the *i*th vertex of the face.

Vertex getEdgepoint(uint *i*, **double** *ratio* **) const;**
Returns a point on the edge between the *i*th and the $(i+1)$th vertex with ratio *ratio*.

Vertex getMidpoint() const;
Returns the average of all the vertices of the face.

Vertex getNormal(uint *i* **) const;**
Returns the vector perpendicular to two adjacent edges of the face. Its sign is determined by the order the vertices are stored.

A.3 Class Polyhedron

public member variable:
typedef unsigned **uint;**
public member functions:

Polyhedron();
Default constructor. Constructs an object with 0 vertices and 0 faces.

Polyhedron(uint *nvertices*, **uint** *nfaces* **);**
Constructs an object with *nvertices* default vertices and *nfaces* default faces.

Polyhedron(Polyhedron const &*other* **);**
Copy-constructor.

~Polyhedron();
Destructor.

Polyhedron const &operator=(Polyhedron const &*right*);
Assignment operator.

void resize(uint *nvertices*, **uint** *nfaces* **);**
Changes the number of vertices to *nvertices* and the number of faces to *nfaces*. If *nvertices* (*nfaces*) is less than the previous number of vertices (faces), the vertex (face) list is truncated, otherwise default vertices (faces) are appended.

void newVertex(uint *i*, **Vertex const** *&vertex* **);**
Changes the *i*th vertex to *vertex*.

void newNormal(uint *i*, **double** *nx*, **double** *ny*, **double** *nz* **);**
Changes the normal of the *i*th vertex to (nx, ny, nz).

void newFace(uint *i*, **uint const *face**, **uint** *size* **= 3);**
Changes the *i*th face to *face*, where *face* is a pointer array of *size* vertex numbers.

void complete(bool *newNormals* **= true);**
Calculates all connectivity arrays and a bounding box and if *newNormals* is true all normals in all vertices as well.

uint getNVertices() const;
Returns the number of vertices of the polyhedron.

uint getNFaces() const;
Returns the number of faces of the polyhedron.

uint getNEdges() const;
Returns the number of edges of the polyhedron.

uint getPrevNVertices() const;
Returns the number of vertices of the polyhedron before it was subdivided (0 if not subdivided).

uint getPrevNFaces() const;
Returns the number of faces of the polyhedron before it was subdivided (0 if not subdivided).

uint getPrevNEdges() const;
Returns the number of edges of the polyhedron before it was subdivided (0 if not subdivided).

Vertex const &getVertex(uint *i* **) const;**
Returns a reference to the *i*th vertex of the polyhedron.

Face getFace(uint *i* **) const;**
Returns a Face object of the *i*th face of the polyhedron.

uint const *getVertex(uint *i*, **bool** *edges_or_faces* **) const;**
Returns a pointer to the *i*th vertex represented by the numbers of edges coincident at that vertex if *edges_or_faces* is true or faces coincident at that vertex otherwise.

uint const *getFace(uint *i*, **bool** *vertices_or_edges* **) const;**
Returns a pointer to the *i*th face represented by the numbers of vertices defining that face if *vertices_or_edges* is true or edges defining that face otherwise.

uint const *getEdge(uint *i*, **bool** *vertices_or_faces* **) const;**
Returns a pointer to the *i*th edge represented by the numbers of vertices defining that edge if *vertices_or_faces* is true or faces defining that edge otherwise.

uint getVertexSize(uint *i* **) const;**
Returns the valence of the *i*th vertex.

uint getFaceSize(uint *i* **) const;**
Returns the number of vertices defining the *i*th face.

Vertex const *boundingBox() const;
Returns a pointer to an array of vertices defining the bounding box of the polyhedron.

virtual void subdivide();
Function to be used by derived classes for the subdivision routine. Increases the subdivision level by one stores the number of vertices, faces and edges.

uint getLevel() const;
Returns the subdivision level.

int getEulerCharacteristic() const;
Returns the Euler characteristic of the polyhedron.

protected member functions:

void resizeFaces(uint *nfaces* **);**
Changes the number of faces to *nfaces*. If *nfaces* is less than the previous number of faces, the face list is truncated, otherwise default faces are appended.

void computeArrays();
Computes the following connectivity arrays: vertices defined by faces, vertices defined by edges, faces defined by edges, edges defined by vertices, edges defined by faces, edges defined by vertices and faces defined by edges.

void computeNormals();
Computes the normals in the vertices.

void computeBounds();
Computes the bounding box of the polyhedron.

class MyUintArray;
A dynamic array-object used for the storage of all connectivity arrays.

A.4 Class DooSabin

Derived from **Polyhedron**.

public member functions:

DooSabin();
Default Constructor.

DooSabin(Polyhedron const *&other* **);**
Copy-constructor.

void subdivide();
Performs the **Doo-Sabin** subdivision on the polyhedron and replaces **this** with the subdivided polyhedron.

A.5 Class CatmullClark

Derived from **Polyhedron**.

public member functions:

CatmullClark();
Default Constructor.

CatmullClark(Polyhedron const *&other* **);**
Copy-constructor.

void subdivide();
Performs the **Catmull-Clark** subdivision on the polyhedron and replaces **this** with the subdivided polyhedron.

A.6 Class Loop

Derived from **Polyhedron**.

public member functions:

Loop();
Default Constructor.

Loop(Polyhedron const *&other* **);**
Copy-constructor. The Polyhedron copied is first triangulated.

void subdivide();
Performs the **Loop** subdivision on the polyhedron and replaces **this** with the subdivided polyhedron.

A.7 Class Contour

public member variable:
typedef unsigned **uint**;
public member functions:

Contour();
Default Constructor.

Contour(Polyhedron const ****parent*, double const** *eye*[3]);**
Constructs an object by searching all edges of *parent* for contour edges considering the projection reference point *eye*.

Contour(Contour const &*other* **);**
Copy-constructor.

Contour const &operator=(Contour const &*right* **);**
Assignment operator.

uint getNEdges() const;
Returns the number of edges in the contour.

uint getNVertices() const;
Returns the number of vertices in the contour.

uint getNFaces() const;
Returns the number of contour faces.

uint bandSize() const;
Returns the number of faces in the expectation-band (0 if not yet subdivided).

Polyhedron const *getParent() const;
Returns a pointer to the parent Polyhedron object.

uint getEdge(uint *i* **) const;**
Returns the (Polyhedron) index of the *i*th edge of the contour.

uint getVertex(uint *i* **) const;**
Returns the (Polyhedron) index of the *i*th vertex of the contour.

uint getFace(uint *i* **) const;**
Returns the (Polyhedron) index of the *i*th face of the contour.

uint getBandFace(uint *i* **) const;**
Returns the (Polyhedron) index of the *i*th face in the expectation-band.

void subdivide(); Finds new contour assuming the parent Polyhedron has subdivided and makes a expectation-band.

Bibliography

[1] E. Catmull & J. Clark. *Recursive generated B-spline surfaces on arbitrary topological meshes*, Computer Aided Design, 10 (1978), pp. 350–355.

[2] G. Chaikin. *An algorithm for high speed curve generation*, Computer Graphics and Image Processing, 3, 1974.

[3] D. Doo & M.A. Sabin. *Behaviour of recursive division surfaces near extraordinary points*, Computer Aided Design, 10 (1978), pp. 356–360.

[4] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design, A Practical Guide*, Academic press, 1990, Second Edition, ISBN 0-12-249051-7.

[5] X. Gu, et al. *Silhouette Mapping*, Harvard University, 1999, Computer Science Technical Report: TR-1-99.

[6] M. Henle. *A Combinatorial Introduction to Topology*, W.H. Freeman and Company, San Fransisco, 1979, ISBN 0-7167-0083-2

[7] L. Kettner and E. Welzl. *Contour Edge Analysis for Polyhedron Projections*, Institut für Theoretische Informatik, ETH Zürich, 1997, CH-8092 Zürich, Switzerland.

[8] H.Q. Le. *Subdivision Surfaces*, Unpublished manuscript, University of Waterloo, 1997.

[9] C. Loop. *Smooth Subdivision Surfaces Based on Triangles*, Master's thesis, University of Utah, Department of Mathematics, 1987.

[10] J. Peters & U. Reif. *Analysis of algorithms generalizing B-spline subdivision*, SIAM Journal of Numerical Analysis, 2 (1998), pp. 728–748.

[11] U. Reif. *A unified approach to subdivision algorithms near extraordinary vertices*, Computer Aided Geometric Design, 12 (1995), pp. 153–174.

[12] P. Schröder, et al. *Subdivision for Modeling and Animatio*, Course notes for SIGGRAPH 99, `http://www.multires.caltech.edu/pubs`.

[13] J.E. Schweitzer. *Analysis and Application of Subdivision Surfaces*, Department of Computer Science and Engineering, University of Washington, Seattle, 1996, Technical report UW-CSE-96-08-02.

[14] E.J. Stollnitz, T.D. DeRose, D.H. Salesin. *Wavelets for Computer Graphics*, Morgan Kaufmann Publishers, Inc., San Fransisco, 1996.

[15] G. Umlauf. *Analyzing the Characteristic Map of Triangular Subdivision Schemes*, To appear: Constructive Approximation. Downloadable at `http://i33www.ira.uka.de/~umlauf`.

[16] J. Warren. *Subdivision methods for geometric design*, Unpublished manuscript, Dept. of Computer Science, Rice University, 1995.