SUBDIVISION SURFACES

Jeppe Oskar Meyer Larsen

Christian Krog Madsen

July 1, 2001

Mid-way Project

Department of Mathematics

Technical University of Denmark

Contents

Chapter 1

Introduction

Subdivision surfaces have become a valuable tool in geometric modeling for computer graphics and computer aided geometric design (CAGD). Their chief advantages are simplicity, efficiency and ease of implementation. The shape of a subdivision surface is determined by a structured mesh of control points and a set of subdivision rules. The control mesh forms a piecewise planar approximation to the surface and the subdivision rules prescribe a procedure for refining the mesh, giving a finer approximation. The subdivision surface itself is defined as the limit of repeated recursive refinement. Subdivision surfaces satisfy all the usual requirements for surface representations for CAGD: moving a control point should only affect the surface locally, the surface should be easy and intuitive to manipulate, and it should be easy to evaluate points and tangents on the surface. For a designer these properties mean that he does not need to understand all the mathematical intricacies, but only the relation between the control points and the surface.

Although subdivision schemes are very simple, the analysis of the limit surface quickly becomes remarkably involved. These difficulties explain why subdivision surfaces have only in recent years been integrated in commercial modeling software, despite the fact that the first subdivision surface schemes were proposed more than 20 years ago.

The currently prevailing surface representation techniques in geometric modeling are various types of spline surfaces, particularly B-spline and NURBS surfaces. These surfaces can be expressed in closed form and efficient algorithms for evaluating points and derivatives on the surface have been developed. The major deficiency of the spline surfaces is that they cannot easily model objects with non-planar topology, because their control net must form a rectangular grid. Subdivision surfaces do not have such topological limitations. Indeed, most of the early subdivision schemes were devised to generalize existing spline surfaces to arbitrary topology.

The recursive nature of their definition makes subdivision surfaces ideal for interactive multiresolution mesh editing, where the overall shape of an object is controlled by a coarse mesh, while details are added by modifying the control points of a refined mesh. For computer graphics applications where time is a concern, subdivision can be used to reduce the complexity of a scene through level-of-detail rendering. This means that distant objects are drawn as a rough polygonal model, but as objects come close to the viewer subdivision is employed to make the model smoother and more detailed. Subdivision surfaces have also been shown to be suitable as shape functions for finite element simulations of thin shell equations [24].

Subdivision surfaces were first introduced by Doo and Sabin [7], and Catmull and Clark [3] in 1978 as generalizations of uniform biquadratic and bicubic B-spline surfaces, respectively. Later, Loop [16] proposed a generalization of quartic Box spline surfaces for triangular meshes. The first interpolatory scheme was the Butterfly scheme of Dyn et al. [9]. Hoppe et al. [10] extended Loop's scheme to generate piecewise smooth surfaces. Extensions of the Doo-Sabin and Catmull-Clark schemes to non-uniform surfaces were derived by Sederberg et al. [27]. Other recent subdivision schemes include the Kobbelt interpolatory scheme [12], $\sqrt{3}$ subdivision [14] and 4-8 subdivison [31].

Analysis of subdivision surfaces was started by Doo and Sabin [7] and continued by Ball and Storry [1, 2] who derived necessary conditions for tangent plane continuity. Reif [23] gave sufficient conditions for tangent plane continuity and C^1 continuity. Prautzsch [18] and Zorin [33, 34] gave conditions for C^k continuity. Based on these conditions Prautzsch and Umlauf [19, 20] modified the Catmull-Clark and Loop schemes to be C^2 continuous at extraordinary points and modified the Butterfly scheme to be C^1 continuous at extraordinary points. An important recent development is that Stam has shown how to evaluate points and derivatives exactly on Catmull-Clark [29] and Loop surfaces [28] at arbitrary parameter values.

We begin our treatment of subdivision in the realm of B-spline curves and surfaces in chapter 2. We define the uniform B-spline functions and use these functions to define the piecewise polynomial B-spline curves. We show how to subdivide a B-spline curve and extend the method to B-spline surfaces.

In chapter 3 we take the more general view of subdivision as a method for defining curves and surfaces. We discuss the topological restrictions on spline-based surface representations and go on to present five subdivision surface schemes without such limitations.

Techniques for analysis of subdivision surfaces are introduced in chapter 4. We show how properties of the limit surface can be derived from eigenanalysis of a matrix characterizing the subdivision procedure. The characteristic map is defined and Reif's sufficient condition for smoothness [23] is presented. As a example we carry out the analysis for Loop's scheme.

Chapter 5 documents selected parts of the implementation of a program for viewing subdivision surfaces. Special attention is given to the data structure for representing polygonal meshes and the algorithms used for subdividing.

The implemented subdivision surface schemes are compared in chapter 6. We highlight certain properties of the schemes by examining a number of surfaces rendered using the viewer program. We also briefly discuss performance issues of the implementation.

A conclusion is offered in chapter 7.

The complete source code for the viewer program is listed in appendix A. A brief description of the source files along with page reference is given in the overview in section A.1.

The implemented viewer program reads objects in a special file format. The syntax of the MDL format is described in appendix B.

Chapter 2

Subdivision of Spline Curves and Surfaces

In this chapter we define the uniform B-spline basis functions and use them to construct piecewise polynomial curves. We derive methods for subdividing a curve using properties of the B-spline functions. The curve methods are then extended to surfaces.

2.1 Continuous and Discrete Convolution

We begin by defining convolutions and present some properties of convolutions we will need in the following sections.

Definition 1 *The continuous convolution of functions, $g(t)$ and $h(t)$, is defined as*

$$g(t) * h(t) = \int g(s)h(t-s)\mathrm{d}s.$$

The *translates* of a univariate function, $f(t)$, is the sequence of functions, $f_i(t) = f(t-i)$. The *dilates* of $f(t)$ is the sequence of functions, $f^j(t) = f(2^j t)$. Combining translation with dilation we get $f_i^j(t) = f(2^j t - i)$.

The following theorem shows how convolution affects dilates and translates.

Theorem 1 *Let $f(t)$, $g(t)$, $h(t)$ be functions and let $m(t) = g(t) * h(t)$ be the continuous convolution of $g(t)$ and $h(t)$, then*

$$\begin{aligned} f(t) * (g(t) + h(t)) &= f(t) * g(t) + f(t) * h(t) \\ g(t-i) * h(t-j) &= m(t-i-j) \\ g(2t) * h(2t) &= \frac{1}{2}m(2t) \end{aligned}$$

Proof. These identities are proved by substituting the definition of convolution and using simple properties of integration.

$$\begin{aligned} f(t) * (g(t) + h(t)) &= \int f(s)(g(t-s) + h(t-s))\mathrm{d}s \\ &= \int f(s)g(t-s)\mathrm{d}s + \int f(s)h(t-s)\mathrm{d}s \\ &= f(t) * g(t) + f(t) * h(t) \end{aligned}$$

$$
\begin{aligned}
g(t-i) * h(t-j) &= \int g(s-i)h(t-j-s)\mathrm{d}s = \int g(\bar{s})h(t-i-j-\bar{s})\mathrm{d}\bar{s} \\
&= \int g(\bar{s})h(\bar{t}-\bar{s})\mathrm{d}\bar{s} = g(\bar{t}) * h(\bar{t}) = m(\bar{t}) = m(t-i-j),
\end{aligned}
$$

where we change variables $\bar{s} = s - i$ and $\bar{t} = t - i - j$.

$$
\begin{aligned}
g(2t) * h(2t) &= \int g(2s)h(2(t-s))\mathrm{d}s = \frac{1}{2}\int g(\bar{s})h(2t-\bar{s})\mathrm{d}\bar{s} \\
&= \frac{1}{2}g(2t) * h(2t) = \frac{1}{2}m(2t),
\end{aligned}
$$

where $\bar{s} = 2s$. □

The next theorem shows that convolution raises the degree of differentiability.

Theorem 2 *If $f(t)$ is a C^k continuous function, then $U(t) * f(t)$ is a C^{k+1} continuous function.*

Proof. Using the definition of convolution and of $U(t)$ a direct calculation gives

$$
U(t) * f(t) = \int U(s)f(t-s)\mathrm{d}s = \int_0^1 f(t-s)\mathrm{d}s = -\int_t^{t-1} f(\bar{s})\mathrm{d}\bar{s} = \int_{t-1}^{t} f(\bar{s})\mathrm{d}\bar{s},
$$

with $\bar{s} = t - s$. The theorem then follows from the fact that the differentiability of the last integral is one higher than the differentiability of $f(t)$. □

We now turn to the discrete convolution of sequences.

Definition 2 *Let a and b be sequences, then the discrete convolution of a and b is a sequence, c, with elements*

$$
c_k = (a * b)_k = \sum_{i+j=k} a_i b_j .
$$

Certain properties of discrete convolution can conveniently be expressed in terms of *generating functions*. The generating function of a sequence, a, is the unique polynomial function, A(z), given by

$$
A(z) = \sum_k a_k z^k .
$$

It is easy to verify that the generating function of the sequence c in definition 2 can be written as

$$
C(z) = A(z)B(z),
$$

where $A(z)$ and $B(z)$ are the generating functions for the sequences a and b, respectively.

In the following sections we derive equations of the form

$$
f(t) = \sum_i c_i f(2t-i),
$$

i.e. we write some function, $f(t)$, in terms of its dilates. These equations are called *subdivision equations* or *refinement equations*. The following theorem provides the link between subdivision equations and discrete convolution.

Theorem 3 *Let $h(t) = f(t) * g(t)$ be the continuous convolution of two functions, $f(t)$ and $g(t)$, that both satisfy subdivision equations*

$$f(t) = \sum_i a_i f(2t - i),$$
$$g(t) = \sum_j b_j g(2t - j).$$

Then, $h(t)$ satisfies a similar subdivision equation

$$h(t) = \sum_k c_k h(2t - k),$$

where $c_k = \sum_{i+j=k} \frac{1}{2} a_i b_j = \frac{1}{2}(a * b)_k$.

Proof. By the linearity of continuous convolution (theorem 1)

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

From theorem 1 we also have $f(2t - i) * g(2t - j) = \frac{1}{2}h(2t - i - j)$. Inserting this in the sum we get

$$h(t) = \sum_i \sum_j \left(\frac{1}{2} a_i b_j h(2t - i - j)\right) = \sum_k \sum_{i+j=k} \frac{1}{2} a_i b_j h(2t - k) = \sum_k c_k h(2t - k),$$

where $c_k = \sum_{i+j=k} \frac{1}{2} a_i b_j$. □

Comparing the definition of the generating function with the elements in the sequence c in theorem 3 we see that the generating function of c is $C(z) = \frac{1}{2}A(z)B(z)$.

2.2 Uniform B-spline functions

Given a *uniform* knot vector, i.e. a vector

$$T^0 = [\ldots, t^0_{-2}, t^0_{-1}, t^0_0, t^0_1, t^0_2, \ldots],$$

where the knots t_i are equally spaced and $t_i < t_{i+1}$, we define the function

$$U(t) = \begin{cases} 1 & \text{if } t^0_0 \le t < t^0_1, \\ 0 & \text{otherwise.} \end{cases}$$

For simplicity, and without loss of generality, we choose to set $t^0_i = i$. The *translates* of $U(t)$ are $U_i(t) = U(t - t^0_i) = U(t - i)$. The *dilates* of $U(t)$ are defined over the refined knot vector

$$T^j = [\ldots, t^j_{-2}, t^j_{-1}, t^j_0, t^j_1, t^j_2, \ldots] = [\ldots, -\frac{1}{2^{j-1}}, -\frac{1}{2^j}, 0, \frac{1}{2^j}, \frac{1}{2^{j-1}}, \ldots],$$

which is obtained from T^0 by iterating the process of binary refinement, i.e. inserting a new knot halfway between each pair of consecutive knots, j times. The *dilates* are

$$U_i^j(t) = U(2^j(t - t_i^j)) = U(2^j t - t_i^0) = U(2^j t - i), \quad i \in \mathbb{Z}.$$

From the definition of $U(t)$ it is easy to see that

$$U(t) = U(2t) + U(2t-1). \tag{2.1}$$

We now define the B-spline functions as repeated continuous convolution of $U(t)$.

Definition 3 *The uniform B-spline function of degree k is defined as*

$$N^k(t) = \mathop{\ast}_{i=0}^{k} U(t).$$

We define the translates of the B-spline functions as we did for $U(t)$

$$N_i^k(t) = N^k(t - i), \quad i \in \mathbb{Z}.$$

The degree zero uniform B-spline is just $N^0(t) = U(t)$. The degree one uniform B-spline is

$$N^1(t) = \int_{-\infty}^{\infty} U(s)U(t-s)\mathrm{d}s = \int_0^1 U(t-s)\mathrm{d}s = -\int_t^{t-1} U(p)\mathrm{d}p = \int_{t-1}^{t} U(p)\mathrm{d}p,$$

which expands to

$$N^1(t) = \begin{cases} t & 0 \le t \le 1 \\ 2-t & 1 \le t \le 2 \\ 0 & \text{otherwise} \end{cases}$$

$N^1(t)$ is clearly continuous so from theorem 2 it follows that $N^k(t)$ is C^{k-1} continuous for $k \ge 1$. Other important properties of the B-spline functions are

- $N_i^k(t) \ge 0$ for all t.
- $\sum_i N_i^k(t) = 1$.
- $N_i^k(t)$ has support on the interval $[t_i^0, t_{i+k+1}^0]$.
- $N_i^k(t)$ is a polynomial function of degree k on each interval $[t_j^0, t_{j+1}^0]$, $j \in \mathbb{Z}$.

The first two properties imply that the translated B-spline functions form a partition of unity.

Figure 2.1 shows graphs of the uniform B-spline functions of degree zero, one, two and three.

We will now show that the subdivision equation for $U(t)$ in (2.1) is inherited by the B-spline function, i.e. the B-spline function can be decomposed into dilates and translates of itself. Let u be a sequence with $u_0 = u_1 = 1$ and $u_j = 0$ for $j \notin \{0, 1\}$, then (2.1) can be written as the sum $U(t) = \sum_j u_j U(2t - j)$. The generating function of the sequence u is

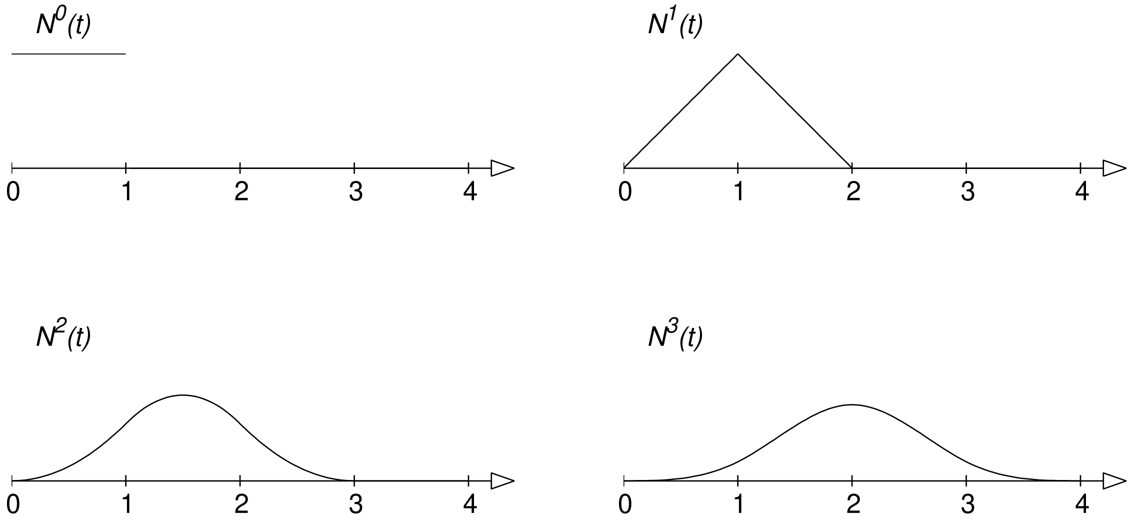


Figure 2.1: B-spline functions of degree zero, one, two and three.

$1+z$. Since the B-spline functions are defined as repeated continuous convolution of $U(t)$ it follows from theorem 3 that the B-splines satisfy the subdivision equation

$$N^k(t) = \sum_j s_j N^k(2t-j),$$

where the sequence s has the generating function $S(z) = \frac{1}{2^k}(1+z)^{k+1}$. Expanding using the binomial formula we get $S(z) = \frac{1}{2^k}\sum_{j=0}^{k+1}\binom{k+1}{j}z^j$ and $s_j = \frac{1}{2^k}\binom{k+1}{j}$. Therefore, we have

$$N^k(t) = \sum_{j=0}^{k+1} \frac{1}{2^k}\binom{k+1}{j} N^k(2t-j).$$

More generally, the translates $N_i^k(t)$ satisfy the subdivision equation

$$N_i^k(t) = \sum_{j=0}^{k+1} \frac{1}{2^k}\binom{k+1}{j} N^k(2t-j-2i). \tag{2.2}$$

2.3 B-spline Curves

Definition 4 *A B-spline curve, $\boldsymbol{r}(t)$, of degree k is defined as*

$$\boldsymbol{r}(t) = \sum_i N_i^k(t) p_i,$$

where p_i are called the control points or de Boor points. With the ordering given by the subscript the points define a polygon called the control polygon or the de Boor polygon.

We leave the index i unspecified to allow both finite and infinite curves. In the case of an infinite curve the sum is well defined since for any given value of t only a finite number of the translated B-spline functions are non-zero, due to the local support property.

An important property of a B-spline curve is the *strong convex hull property*. This states that for $t \in [t_i^0, t_{i+1}^0]$ the curve, $\boldsymbol{r}(t)$, is contained in the convex hull of $\{p_{i-k}, \ldots, p_i\}$. Globally, the B-spline curve is contained in the union of the convex hull of every set of $k+1$ consecutive control points. The strong convex hull property follows from the local support of the B-spline functions, which means that only $k+1$ of the translated B-spline functions are non-zero on the interval $[t_i^0, t_{i+1}^0]$, so only $k+1$ of the control points influence the curve in that interval.

The subdivision equation for the B-splines from the previous section can now be applied to a B-spline curve. We want to express the curve $\boldsymbol{r}(t)$ in the dilated B-spline basis. From the definition above and (2.2), we get

$$\boldsymbol{r}(t) = \sum_i N_i^k(t) p_i = \sum_i \left(\sum_{j=0}^{k+1} \frac{1}{2^k} \binom{k+1}{j} N^k(2t - j - 2i) \right) p_i .$$

We split the inner sum into even and odd terms

$$\begin{aligned} \boldsymbol{r}(t) = \frac{1}{2^k} \sum_i \Bigg(& \sum_{l=0}^{\lfloor (k+1)/2 \rfloor} \binom{k+1}{2l} N(2t - 2l - 2i) \\ & + \sum_{l=0}^{\lceil (k-1)/2 \rceil} \binom{k+1}{2l+1} N(2t - 2i - 2l - 1) \Bigg) p_i , \end{aligned}$$

and collect terms with the same B-spline functions

$$\begin{aligned} \boldsymbol{r}(t) = \frac{1}{2^k} \sum_r \Bigg(& \left(\sum_{l=0}^{\lfloor (k+1)/2 \rfloor} \binom{k+1}{2l} p_{r-l} \right) N(2t - 2r) \\ & + \left(\sum_{l=0}^{\lceil (k-1)/2 \rceil} \binom{k+1}{2l+1} \boldsymbol{p}_{r-l} \right) N(2t - 2r - 1) \Bigg) . \end{aligned} \tag{2.3}$$

Therefore, the new even and odd control points for the subdivided curve are

$$\begin{aligned} \hat{p}_{2r} &= \frac{1}{2^k} \sum_{l=0}^{\lfloor (k+1)/2 \rfloor} \binom{k+1}{2l} p_{r-l}, \\ \hat{p}_{2r+1} &= \frac{1}{2^k} \sum_{l=0}^{\lceil (k-1)/2 \rceil} \binom{k+1}{2l+1} p_{r-l}. \end{aligned}$$

In this report we are particularly interested in subdivision of quadratic and cubic B-splines. For the special case of a quadratic B-spline curve the new control points are

$$\hat{p}_{2r} = \frac{1}{4}(3p_{r-1} + p_r), \tag{2.4}$$

$$\hat{p}_{2r+1} = \frac{1}{4}(p_{r-1} + 3p_r), \tag{2.5}$$

and in the case of a cubic B-spline curve the new control points are

$$\hat{p}_{2r} = \frac{1}{8}(p_{r-2} + 6p_{r-1} + p_r), \tag{2.6}$$

$$\hat{p}_{2r+1} = \frac{1}{2}(p_{r-1} + p_r). \tag{2.7}$$

So far we have viewed subdivision as algebraic manipulations used to change basis from one set of B-spline functions to another. However, the coefficients calculated above show that we can instead interpret subdivision as a geometric construction. Given an initial set of control points, we can find new control points by taking convex combinations of the existing points. The geometric construction for quadratic subdivision is known as Chaikin's algorithm [4]. Chaikin described it as a fast way of drawing curves for computer graphics, but he did not realize that his algorithm generated quadratic B-spline curves in the limit. Riesenfeld made the connection to B-splines [25].

Two levels of subdivision of an initial set of control points using the rules for quadratic and cubic subdivision are shown in figure 2.2 and figure 2.3, respectively. The figures indicate that as the control polygon is subdivided it becomes a better and better approximation of the B-spline curve. In section 2.4.1 we will prove that this observation is correct.

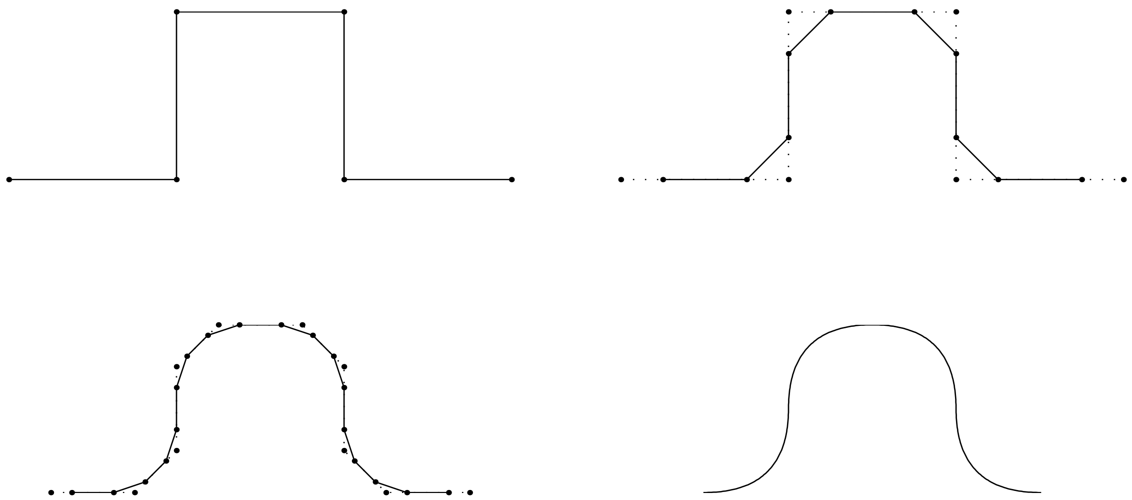


Figure 2.2: Subdivision of a quadratic B-spline curve. Top left: the initial control polygon. Top right: first subdivision. Bottom left: second subdivision. Bottom right: the B-spline curve.

2.3.1 Subdivision Matrix

It is useful to reformulate the definition of a B-spline curve in terms of matrices. We introduce a vector of B-spline functions of degree k as

$$\boldsymbol{N}^k(t) = [\ldots, N^k(t+2), N^k(t+1), N^k(t), N^k(t-1), N^k(t-2), \ldots],$$

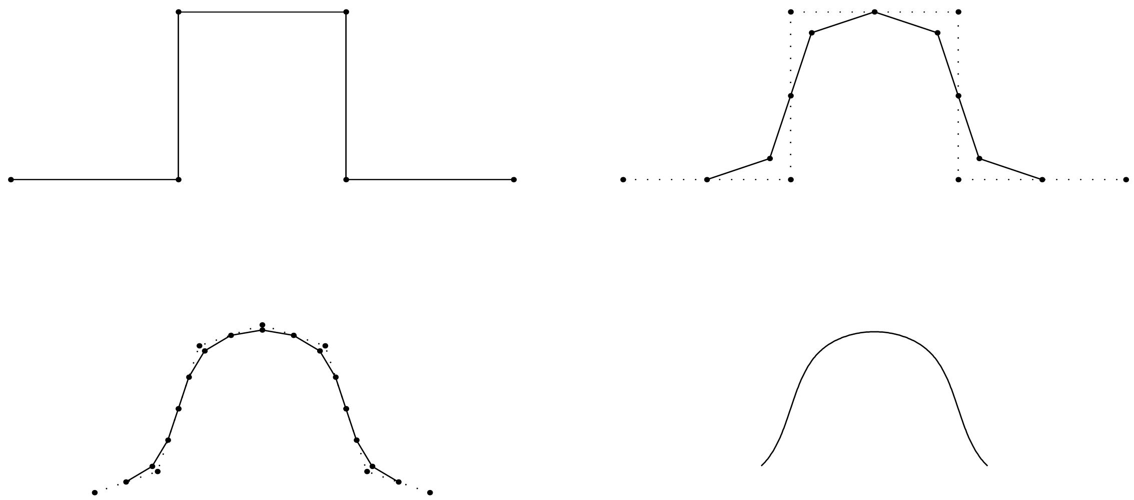


Figure 2.3: Subdivision of a cubic B-spline curve. Top left: the initial control polygon. Top right: first subdivision. Bottom left: second subdivision. Bottom right: the B-spline curve.

and a vector of control points

$$\boldsymbol{p}^0 = [\ldots, p_{-2}, p_{-1}, p_0, p_1, p_2, \ldots]^{\mathrm{T}}.$$

Then the curve, $\boldsymbol{r}(t)$, is simply

$$\boldsymbol{r}(t) = \boldsymbol{N}^k(t)\boldsymbol{p}^0 \tag{2.8}$$

The dilated B-spline functions are written as a vector

$$\boldsymbol{N}^k(2t) = [\ldots, N^k(2t+2), N^k(2t+1), N^k(2t), N^k(2t-1), N^k(2t-2), \ldots].$$

We saw in (2.2) that the B-spline functions can be written as linear combinations of dilated B-spline functions. This implies that there exists a matrix S such that

$$\boldsymbol{N}^k(t) = \boldsymbol{N}^k(2t)S. \tag{2.9}$$

The matrix S is called the *subdivision matrix*, and we see from (2.2) that its elements are

$$S_{2i+j,i} = \frac{1}{2^k}\binom{k+1}{j}.$$

An important property of the subdivision matrix for the B-splines is that the elements in each row sum to one. This property will later be shown to be a necessary and sufficient condition for affine invariance of a general subdivision scheme.

We can now rewrite (2.3) as a matrix equation by substituting (2.9) into (2.8)

$$\boldsymbol{r}(t) = \boldsymbol{N}^k(t)\boldsymbol{p}^0 = \boldsymbol{N}^k(2t)S\boldsymbol{p}^0.$$

The curve is now expressed in the dilated B-spline basis with new control points $\boldsymbol{p}^1 = S\boldsymbol{p}^0$. Since the subdivision matrix S is independent of the parameter t, (2.9) can be generalized to $\boldsymbol{N}^k(2^jt) = \boldsymbol{N}^k(2^{j+1}t)S$. Therefore, the subdivision process can be applied repeatedly to express the curve in a basis of finer and finer dilates of the B-spline functions

$$\boldsymbol{r}(t) = \boldsymbol{N}^k(t)\boldsymbol{p}^0 = \boldsymbol{N}^k(2t)S\boldsymbol{p}^0 = \boldsymbol{N}^k(2^2t)S^2\boldsymbol{p}^0 = \cdots = \boldsymbol{N}^k(2^jt)S^j\boldsymbol{p}^0.$$

From this we can see that the control points after j steps of subdivision become $\boldsymbol{p}^j = S^j\boldsymbol{p}^0$. It is then hardly surprising that the analysis of subdivision in the limit $j \to \infty$ is closely related to the subdivision matrix, as we shall see in the next section.

2.4 Analysis of B-spline Curves

2.4.1 Convergence

We will now show that with repeated subdivision the control polygon converges to a limit curve and that this limit curve is in fact the B-spline curve defined by the initial control polygon.

Once we have shown the control polygon converges to the B-spline curve, we are justified in using the control polygon as a piecewise linear approximation to the curve. We also know that by subdividing we can get as good an approximation as we desire.

To prove convergence of the control polygon to a continuous curve we examine the differences between adjacent control points. The adjacent differences are calculated as $\Delta\boldsymbol{p}$, where Δ is a bi-infinite matrix with -1 on the diagonal and 1 on the upper diagonal.

The control polygon after j subdivisions can be written as a piecewise linear curve in the dilated basis of first degree B-splines $P^j(t) = N^1(2^jt)\boldsymbol{p}^j$.

For the proof of convergence we define norms on functions, vectors and matrices

$$\|f(t)\| = \sup_t |f(t)|,$$
$$\|\boldsymbol{p}\| = \max_i |p_i|,$$
$$\|S\| = \max_j |S_{ij}|.$$

With these norms we have the Cauchy-Schwarz inequality

$$\|S\boldsymbol{p}\| \leq \|S\| \cdot \|\boldsymbol{p}\|.$$

Theorem 4 *If there exists $\beta > 0$ and $0 < \alpha < 1$ such that*

$$\|\Delta\boldsymbol{p}^j\| \leq \beta\alpha^j$$

for all $j > 0$, then $P^j(t)$ converges uniformly to a continuous limit function $P^\infty(t)$ as $j \to \infty$.

Proof. Let S be the subdivision matrix for the scheme to be investigated and let S_1 be the subdivision matrix for first degree B-splines. We noted above that for the B-splines the rows of the subdivision matrix sum to one, i.e. $S\mathbf{1} = \mathbf{1}$ and $S_1\mathbf{1} = \mathbf{1}$. Then, $(S - S_1)\mathbf{1} = \mathbf{0}$ so the rows of $S - S_1$ sum to zero. As the rows of Δ also sum to zero and Δ has full rank, there exists a matrix A such that

$$S - S_1 = A\Delta.$$

The difference between the piecewise linear curves defined by two successive control polygons is

$$\|P^{j+1}(t) - P^j(t)\| = \|\boldsymbol{N}^1(2^{j+1}t)\boldsymbol{p}^{j+1} - \boldsymbol{N}^1(w^j t)\boldsymbol{p}^j\|.$$

We use that $\boldsymbol{p}^{j+1} = S\boldsymbol{p}^j$ and $\boldsymbol{N}^1(2^j t) = \boldsymbol{N}^1(2^{j+1}t)S_1$

$$\begin{aligned}\|P^{j+1}(t) - P^j(t)\| &= \|\boldsymbol{N}^1(2^{j+1}t)S\boldsymbol{p}^j - \boldsymbol{N}^1(2^{j+1}t)S_1\boldsymbol{p}^j\| \\ &= \|\boldsymbol{N}^1(2^{j+1}t)(S - S_1)\boldsymbol{p}^j\|.\end{aligned}$$

Application of the Cauchy-Schwarz inequality and $S - S_1 = A\Delta$ yields

$$\|P^{j+1}(t) - P^j(t)\| \leq \|\boldsymbol{N}^1(2^{j+1}t)\| \cdot \|A\Delta\boldsymbol{p}^j\|,$$

where $\|\boldsymbol{N}^1(2^{j+1}t)\| \leq 1$ since the B-splines form a partition of unity. Another application of Cauchy-Schwarz gives

$$\|P^{j+1}(t) - P^j(t)\| \leq \|A\| \cdot \|\Delta\boldsymbol{p}^j\|.$$

By hypothesis $\|\Delta\boldsymbol{p}^j\| \leq \beta\alpha^j$ so

$$\|P^{j+1}(t) - P^j(t)\| \leq \|A\|\beta\alpha^j.$$

We now want to show that the sequence of functions $P^j(t)$ converges pointwise to a limit function. We expand $P^j(t)$ as the summation

$$P^j(t) = P^0(t) + \sum_{k=0}^{j-1}(P^{k+1}(t) - P^k(t)).$$

Let $\{p\}_i$ select the ith coordinate of the point p, then

$$\left\{(P^{k+1}(t) - P^k(t))\right\}_i \leq \|P^{k+1}(t) - P^k(t)\| \leq \|A\|\beta\alpha^k.$$

$\sum_{k=0}^{\infty}\|A\|\beta\alpha^k$ is a convergent geometric series, so by the comparison test

$$\sum_{k=0}^{\infty}\left\{(P^{k+1}(t) - P^k(t))\right\}_i$$

is a convergent series. This implies that each coordinate of $P^j(t)$ converges pointwise, so there exists a limit function $P^\infty(t) = \lim_{j\to\infty} P^j(t)$.

We now prove uniform convergence. The difference between the limit function and its approximation is

$$\begin{aligned}\|P^\infty(t) - P^j(t)\| = \left\|\sum_{k=j}^{\infty}(P^{k+1}(t) - P^k(t))\right\| &\leq \sum_{k=j}^{\infty}\|P^{k+1}(t) - P^k(t)\| \\ \leq \sum_{k=j}^{\infty}\|A\|\beta\alpha^k &= \alpha^j\sum_{k=0}^{\infty}\|A\|\beta\alpha^k,\end{aligned}$$

which is a convergent geometric series with sum $\frac{\|A\|\beta}{1-\alpha}\alpha^j$, i.e.

$$\|P^\infty(t) - P^j(t)\| \leq \frac{\|A\|\beta}{1-\alpha}\alpha^j.$$

Therefore, the sequence of functions $P^j(t)$ converges uniformly to $P^\infty(t)$ as $j \to \infty$. Since each function $P^j(t)$ is continuous, the limit function $P^\infty(t)$ is also continuous. □

We consider a subdivision scheme for the difference vectors $\Delta\boldsymbol{p}^j$. The subdivision matrix, D, for this scheme must satisfy

$$\Delta\boldsymbol{p}^{j+1} = D\Delta\boldsymbol{p}^j. \tag{2.10}$$

The next theorem provides a sufficient condition on D for the applicability of theorem 4.

Theorem 5 *If $\|\boldsymbol{p}^0\|$ is bounded and there exists $k \geq 1$ such that $0 < \|D^k\| < 1$, then there exists $\beta > 0$ and $0 < \alpha < 1$ such that $\|\Delta\boldsymbol{p}^j\| \leq \beta\alpha^j$ for all $j > 0$.*

Proof. From (2.10)

$$\Delta\boldsymbol{p}^j = \Delta S^j\boldsymbol{p}^0 = D^j\Delta\boldsymbol{p}^0.$$

Let $\|D^k\| = \gamma^k < 1,\ \gamma > 0$ and let $j = nk + m$, where $n = \lfloor j/k \rfloor$ and $m = j \bmod k$, then

$$\begin{aligned}
\|\Delta\boldsymbol{p}^j\| &= \|D^j\Delta\boldsymbol{p}^0\| = \|D^{nk+m}\Delta\boldsymbol{p}^0\| \\
&\leq \|D^{nk+m}\|\|\Delta\boldsymbol{p}^0\| \leq \|D^{nk}\|\|D^m\|\|\Delta\boldsymbol{p}^0\| \\
&\leq \|D^k\|^n\|D^m\|\|\boldsymbol{p}^0\| = \gamma^{kn}\|D^m\|\|\Delta\boldsymbol{p}^0\| \\
&= \gamma^j\gamma^{-m}\|D^m\|\|\Delta\boldsymbol{p}^0\|.
\end{aligned}$$

Now the theorem follows by setting

$$\begin{aligned}
&\alpha = \gamma \\
&\beta = \gamma^{-m}\|\Delta\boldsymbol{p}^0\| \max\{\ \|D^m\| \ \big|\ m = 0, \ldots, k-1\ \}.
\end{aligned}$$

□

Remark. Theorems 4 and 5 are due to Warren [32], however, the proof given by Warren for theorem 5 is not correct. The proof given above was suggested by Jens Gravesen.

To apply the above theorems to B-spline curves, we first find the norm of the difference scheme subdivision matrix. For quadratic curves we get

$$\begin{aligned}
\left\{\Delta\boldsymbol{p}^{j+1}\right\}_{2i} &= p_{2i+1}^{j+1} - p_{2i}^{j+1} = \frac{1}{4}(p_{i-1} + 3p_i) - \frac{1}{4}(3p_{i-1} + p_i) \\
&= \frac{1}{4}(-2p_{i-1} + 2p_i) = \frac{1}{2}(p_i - p_{i-1}) = \frac{1}{2}\left\{\Delta\boldsymbol{p}^j\right\}_{i-1}, \\
\left\{\Delta\boldsymbol{p}^{j+1}\right\}_{2i+1} &= p_{2i+2}^{j+1} - p_{2i+1}^{j+1} = \frac{1}{4}(3p_i + p_{i+1}) - \frac{1}{4}(p_{i-1} + 3p_i) \\
&= \frac{1}{4}(p_{i+1} - p_{i-1}) = \frac{1}{4}(p_{i+1} - p_i + p_i - p_{i-1}) = \frac{1}{4}\left\{\Delta\boldsymbol{p}^j\right\}_i + \frac{1}{4}\left\{\Delta\boldsymbol{p}^j\right\}_{i-1}.
\end{aligned}$$

With these coefficients $\|D\| = \frac{1}{2}$, so Theorem 4 and 5 show that for quadratic B-spline subdivision, the control polygon converges to a continuous limit curve. By the strong convex hull property this limit curve must be the B-spline curve.

The above argument can be repeated for cubic curves, see [6]. With a more detailed study of generating functions, it is possible to show that $\|D\| = \frac{1}{2}$ for B-spline curves of arbitrary degree [32]. This implies that the control polygon converges under subdivision, regardless of the degree.

The applicability of Theorem 4 and 5 is not restricted to subdivision of B-spline curves. The only assumption we make is that the rows of the subdivision matrix sum to one. As previously noted this condition is necessary to ensure affine invariance, so it is a very weak assumption. Another subdivision scheme that can be shown to converge using these theorems is the interpolatory four-point scheme [8, 32]. The four-point scheme is an example of a curve generating scheme that is defined by subdivision rules without reference to basis functions. We return to this more general application of subdivision in the next chapter.

2.4.2 Invariant Neighbourhood

An important question in relation to subdivision is which control points affect a given part of the limit curve? One way to answer this question is to look at the support of the basis functions. The support of the B-spline functions is determined by the knot vector and the degree. For the B-spline of degree k, the support is $k+1$ consecutive knot intervals. We let the limit curve be parameterised by the knot vector. Suppose we want to find the control points needed to define the limit curve on the two initial knot intervals next to the origin, i.e. on the parameter interval $[-1, 1]$, then we see that $k+2$ of the translated B-splines are non-zero on that interval, so we say that the invariant neighbourhood has size $k+2$.

The case for quadratic B-splines is illustrated in figure 2.4. To define the limit curve on the interval $[-1, 1]$ four of the initial control points are needed, two inside the interval and two outside. The invariant neighbourhood for quadratic B-splines is therefore of size 4.

For cubic B-splines five initial control points are needed to define the limit curve on $[-1, 1]$, see figure 2.5. The invariant neighbourhood is therefore of size 5.

If we only want to find the limit point corresponding to the parameter value 0, then a smaller neighbourhood is needed. Note that the B-spline function $N_i^k(t), k \geq 1$ is identically zero outside the interval $[t_i^0, t_{i+k+1}^0]$ so by continuity it must be zero at t_i^0 and at t_{i+k+1}^0. That implies that only k of the translates of the B-spline function are non-zero at the parameter value 0, so only k control points are necessary to define the limit point. The smaller neighbourhood is also sufficient to find the tangent at a point.

2.5 Tensor Product B-spline Surfaces

Given a representation of curves a representation of surfaces can be obtained through the concept of a tensor product. We start with a curve

$$\boldsymbol{r}(u) = \sum_i F_i(u)\boldsymbol{c}_i,$$

where $F_i(u)$ are some basis functions. We then move the curve while simultaneously allowing it to alter shape. The motion and deformation can be described by introducing a

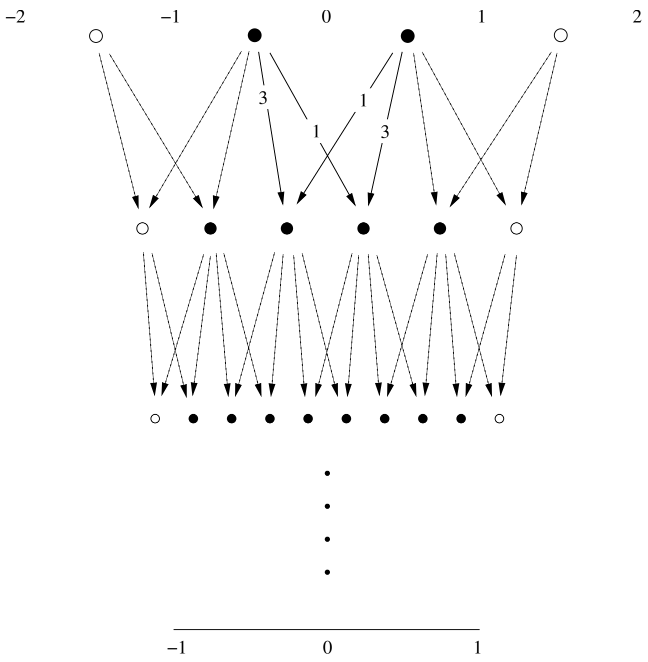


Figure 2.4: Subdivision of quadratic B-spline. Four control points are needed to define the curve on the interval $[-1, 1]$, so the invariant neighbourhood has size 4.

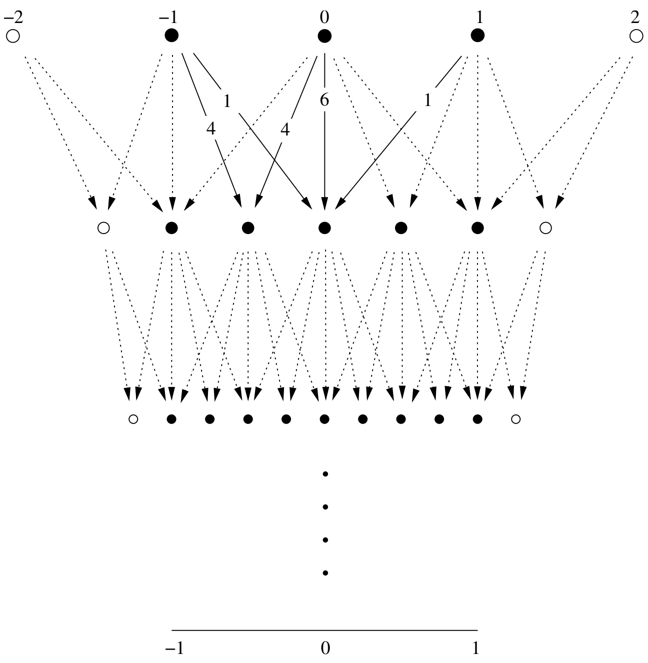


Figure 2.5: Subdivision of cubic B-spline. Five control points are needed to define the curve on the interval $[-1, 1]$, so the invariant neighbourhood has size 5.

second parameter v and letting the control points $\boldsymbol{c}_i$ be functions of v

$$\boldsymbol{c}_i(v) = \sum_j G_j(v)\boldsymbol{p}_{ij},$$

where $G_j(v)$ is another set of basis functions. In this process a surface is swept out. The surface has the parameterisation

$$\boldsymbol{r}(u,v) = \sum_i \sum_j F_i(u)G_j(v)\boldsymbol{p}_{ij}.$$

The control points $\boldsymbol{p}_{ij}$ can be thought of as being organized in a mesh structure. For a finite surface, there are a finite number of control points and the parameter values run over a rectangular domain.

If we replace the arbitrary basis functions above with B-spline functions, we get a tensor product B-spline surface.

Definition 5 *A tensor product B-spline surface, $\boldsymbol{r}(u,v)$, is defined as*

$$\boldsymbol{r}(u,v) = \sum_i \sum_j N_i(u)N_j(v)\boldsymbol{p}_{ij},$$

where $\boldsymbol{p}_{ij}$ are points in the control mesh or the de Boor mesh.

The B-spline functions need not be of the same degree, although for most practical applications they will be.

The construction for tensor product surfaces indicates that subdivision of surfaces is closely related to subdivision of curves. This proves to be true, and we can apply the subdivision process for B-spline curves in (2.3) to the control points in the u direction first, and then to the new points now in the v direction. For curves there are two rules, one for even points and one for odd points. In the case of surfaces, we then get one rule for each possible combination of the curve rules, i.e. four rules in all.

Rules for subdivision of surfaces are most easily described by *masks*. The masks are supposed to be applied to all parts in the de Boor mesh that match the structure in the mask, and for each application a new de Boor point is calculated using the coefficients in the mask.

For biquadratic tensor product B-spline surfaces the subdivision rules are particularly simple since they apply to a single face in the de Boor mesh and the coefficients are just cyclicly rotated. The masks are shown in figure 2.6.

The masks for bicubic tensor product B-spline surfaces are slightly more complicated. The masks are shown in figure 2.7.

$$\begin{array}{ccc} 1/16 & — & 3/16 \\ | & & | \\ 3/16 & — & 9/16 \end{array} \qquad \begin{array}{ccc} 3/16 & — & 1/16 \\ | & & | \\ 9/16 & — & 3/16 \end{array} \qquad \begin{array}{ccc} 9/16 & — & 3/16 \\ | & & | \\ 3/16 & — & 1/16 \end{array} \qquad \begin{array}{ccc} 3/16 & — & 9/16 \\ | & & | \\ 1/16 & — & 3/16 \end{array}$$

Figure 2.6: Masks for subdivision of quadratic B-splines.

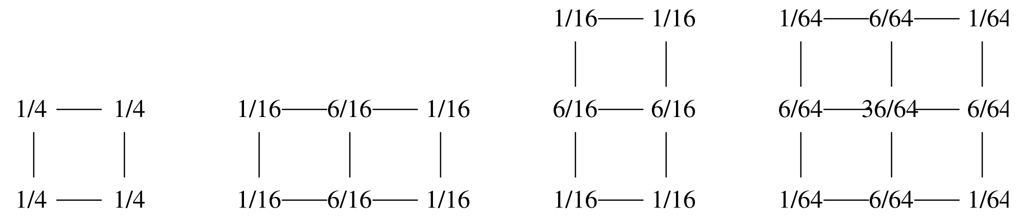


Figure 2.7: Masks for subdivision of cubic B-splines.

2.6 Triangular Box Spline Surfaces

The spline surfaces considered in the previous section are defined on a rectangular parameter domain. It is also possible to define a spline surface over a regular triangulation of the parameter plane.

One way to define a triangular spline is to start with the piecewise linear box spline which is a hexagonal pyramid with the apex at height 1. The higher degree splines are then obtained through convolution with a unit impulse along the three directions in the grid. The quartic box spline is obtained by convolving once in each direction, see [11] for details. The quartic box spline is C^2 continuous.

The triangular splines share many of the properties of B-splines, most notably they have local support, are non-zero and the translates form a partition of unity. In this case translation can be in any of the three directions in the grid.

It is possible to derive a subdivision equation for the triangular splines similar to the equation for the B-splines. The subdivision equation leads to the masks in figure 2.8. For a partial derivation of the subdivision equation, see [16].

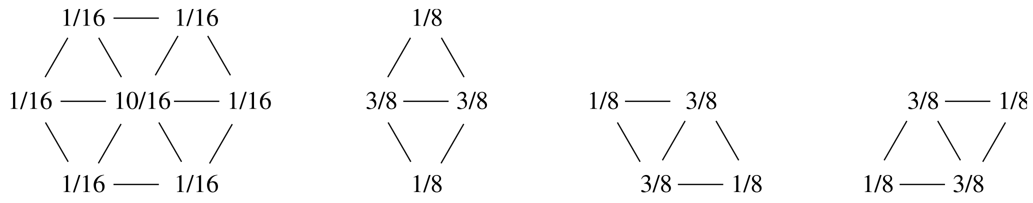


Figure 2.8: Masks for subdivison of quartic box splines.

2.7 Summary

In this chapter we have defined the B-spline functions and have seen how a B-spline function can be written as a linear combination of translated dilates of itself. This property was used to express a parametric curve in a basis of finer and finer dilates of B-spline functions. We have derived formulae for calculating the new control points after subdivision. With these formulae we are offered two views on subdivision, either as algebraic manipulations to change basis functions, or as a geometric construction, where new points are calculated by taking convex combinations of existing points. We saw that the control polygon considered as a piecewise linear curve converges to the B-spline curve. This justifies using the control polygon as a piecwise linear approximation to the curve.

We have briefly discussed how methods for curves can be extended to surfaces. We have described tensor product B-spline surfaces and triangular box spline surfaces, and the subdivision rules for these types of surfaces. It can be shown that under subdivision the control mesh for tensor product B-spline surfaces and box spline surfaces converges to the surface. This again means the control mesh can be used as an approximation to the surface.

Our discussion of subdivision started with the basis functions and from properties of these functions we derived a geometric construction. Another approach to subdivision is to start with a geometric construction and then define a curve as the limit of repeated application of the construction. We return to this view in the next chapter. In the former case many properties of the curve are determined by the basis functions, so much of the analysis of the curve reduces to analysis of the basis functions. In the latter case the basis functions are not available and other methods of analysis have to be employed. One such method is eigenanalysis of the subdivision matrix. That technique also applies to surfaces, as we shall see in chapter 4.

Chapter 3

Subdivision Surfaces

In this chapter we expand our view on subdivision to defining curves and surfaces as the limit of repeated application of subdivision rules, without reference to basis functions. We discuss the topological restrictions on spline based surface techniques and use subdivision to overcome these barriers. Five subdivision surface schemes are presented in detail.

3.1 Curves and Surfaces Defined by Subdivision

In the previous chapter we examined subdivision of spline curves and surfaces. We saw how under repeated subdivision the control polygon converged to the spline curve, and a similar result can be shown for surfaces. The subdivision equations derived offer a geometric interpretation of the subdivision process: new control points are obtained as certain convex combinations of the old control points. It is this geometric approach that provides the inspiration for using subdivision to define curves and surfaces. We have already mentioned two curve schemes defined by their subdivision rules, namely Chaikin's algorithm [4] and the four-point interpolatory scheme [8].

The problem with defining curves and surfaces through subdivision rules lies in choosing coefficients for which subdivision converges to a smooth curve or surface. There is no general method to achieve this.

3.2 Arbitrary Topology Surface Schemes

In chapter 2 we briefly described two types of surfaces – tensor product B-spline surfaces and triangular box spline surfaces. For tensor product B-spline surfaces the control mesh must a regular quadrilateral grid, while for triangular box spline surfaces the control mesh must be a regular triangular mesh. This imposes a restriction on the topology of the surfaces that can be modelled. If we view the control mesh as a graph embedded on the limit surface and assume the surface is bounded and closed, we can use Euler's formula $v - e + f = 2 - 2g$ that relates the number of vertices, v, the number of edges, e, and the number of faces, f, in the control mesh to the topological genus, g, of the surface. In a regular quadrilateral grid without boundaries each vertex can be associated with two edges and one face, so the left hand side of the formula is zero, and the genus of the surface modelled is one. Surfaces of genus one are topologically like the torus. In order to model a surface of genus zero, such as a sphere, the mesh must have either vertices with valency different from 4 or faces with other than four sides. In a regular triangular mesh, surfaces of genus other than one require vertices with valency different from 6 or faces with other than three sides.

The topological restrictions on spline based surface modelling methods are overcome with subdivision surfaces. The early subdivision surface schemes, such as those proposed by Doo and Sabin [7], Catmull and Clark [3], and Loop [16], generalize spline surface schemes to arbitrary topology by specifying additional masks for irregular faces or irregular vertices in the control mesh. The coefficients for these masks are chosen from eigenanalysis to give maximum smoothness of the limit surface.

An important property of subdivision surface schemes is that the number of irregular vertices and faces remains constant after the first subdivision. As subdivision progresses the irregular features are separated by larger and larger patches of regular faces. This means that in the analysis of irregular features, it is sufficient to analyze isolated irregular features.

The limit position of the control point associated with an irregular vertex is called an extraordinary point. Similarly, the limit position of the control points associated with the vertices in an irregular face is also called an extraordinary point.

For the subdivision schemes extending well-known spline scheme the behaviour on the regular parts of the surface is already known. Analysis of the limit surface therefore reduces to analysis of the neighbourhood of extraordinary points. In the next chapter we present the techniques used for this analysis.

3.3 Classification of Subdivision Surface Schemes

In the literature many subdivision surface schemes have been proposed. One way to classify these schemes is according to their properties on regular control meshes (the definition of "regular" is scheme specific).

Mesh type

The regular mesh should be homeomorphic to a tiling of the plane. A reasonable requirement is that all the faces in the tiling are identical. If the faces are also required to be regular polygons, there are only three possibilities: triangular, quadrilateral and hexagonal tilings. Most subdivision schemes operate on meshes that are homeomorphic to either regular triangular or regular quadrilateral tilings. When in the following we refer to triangular meshes, for example, we mean meshes that on their regular parts are homeomorphic to a regular triangular tiling.

Refinement rule

The refinement rule is an algorithm to produce from a given tiling a finer tiling of the same type. The most common refinement rules are the face split and vertex split rules, see Fig. 3.1. The face split rule can be applied to triangular and quadrilateral meshes. Each face is split into four faces of the same type. The vertex split rule can only be applied to quadrilateral meshes. With this rule every vertex is split into four new vertices. Subdivision schemes using the face split rule are called primal, while schemes using the vertex split rule are called dual.

Interpolating/Approximating

A subdivision scheme is interpolating, if all the control points in the original mesh are also control points in all the meshes obtained through subdivision, otherwise the scheme is approximating.

Smoothness

The order of continuity on the regular parts of the subdivision surface depends on the scheme used. Most schemes produce C^1 or C^2 continuous surfaces.

Note that the order of continuity near extraordinary points is usually lower than on the regular parts of the surface.

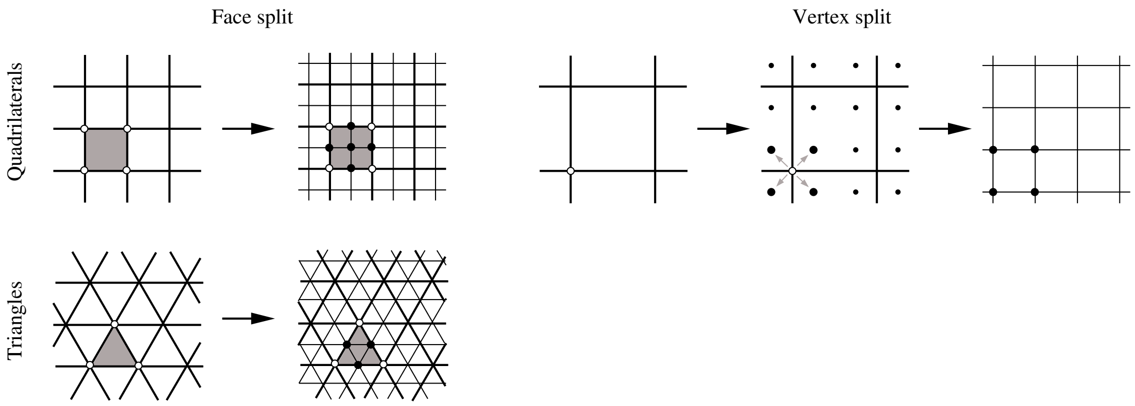


Figure 3.1: Face split and vertex split refinement rules.

The schemes considered in this paper are categorized as follows:

Doo-Sabin: quadrilateral, vertex split, approximating, C^1.

Catmull-Clark: quadrilateral, face split, approximating, C^2.

Loop: triangular, face split, approximating, C^2.

Quadratic NURSS: same as Doo-Sabin.

Cubic NURSS: same as Catmull-Clark.

3.4 Doo-Sabin Scheme

The Doo-Sabin scheme [7] is a generalization of uniform biquadratic tensor product B-spline surfaces to arbitrary topology. It is an approximating vertex split (dual) scheme. The subdivision is quite simple since there is only one mask used to compute the new vertices. Fig. 3.2 shows the general mask (right), which reduces to the ordinary mask for quadratic tensor product B-spline (left) when applied to a quadrilateral face. In the subdivision a new face is created for every face, edge and vertex in the control mesh. After one step of subdivision, every vertex has valence four, and the number of extraordinary faces (with other than four sides) does not change in subsequent subdivisions.

Analysis of the Doo-Sabin scheme was initiated by Doo and Sabin [7]. Peters and Reif proved C^1 continuity of a more general class of schemes of which Doo-Sabin is a special case [17].

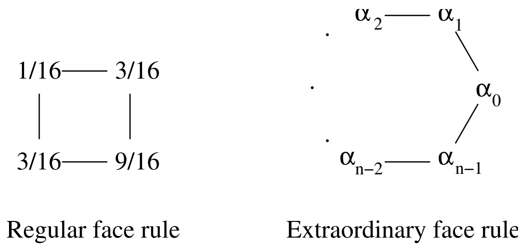


Figure 3.2: Masks for the Doo-Sabin scheme. Doo and Sabin [7] propose the coefficients $\alpha_0 = \frac{1}{4} + \frac{5}{4n}$ and $\alpha_i = \frac{3+2\cos(2i\pi/n)}{4n}$, where n is the number of sides in the face.

3.5 Catmull-Clark Scheme

The Catmull-Clark scheme [3] was proposed at about the same time as the Doo-Sabin scheme. Catmull-Clark generalizes uniform bicubic B-spline surfaces to arbitrary topology. The masks shown in figure 3.3 generate new face, edge and vertex points. The Catmull-Clark scheme is an approximating face split (primal) scheme, splitting each face with n sides into n quadrilaterals. After one step of subdivision, all faces are quadrilaterals. The number of vertices with valence other than four remains constant in subsequent subdivisions.

The masks shown in figure 3.3 cannot be applied to control meshes with irregular faces. To handle such meshes the rules are extended.

New face points are computed as the average of the control points associated with the vertices in the original face.

New edge points are computed as the average of the end points of the edge and the new face points for the two faces sharing the edge.

New vertex points are computed as

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

where F is the average of the new face vertices of all faces adjacent to the old vertex V, E is the average of the midpoints of all edges incident upon the old vertex V, and n is the valency of V.

Ball and Storry [2] proved tangent plane continuity of the Catmull-Clark scheme. Peters and Reif [17] proved C^1 continuity for valencies up to 10000. Away from extraordinary vertices, the scheme produces C^2 continuous surfaces.

3.6 Loop Scheme

Loop's scheme [16] is an approximating face split scheme defined on triangular meshes. The basis function for the scheme in the regular case is the quartic box spline. Loop surfaces are C^2 continuous at ordinary points and C^1 at extraordinary points.

The scheme was extended by Hoppe et al. [10] to incorporate sharp creases, darts and corner points. Creases are smooth curves across which the surface is C^0 continuous instead

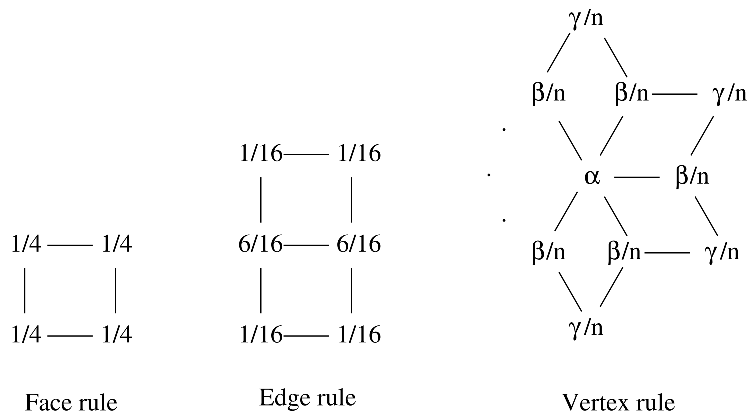


Figure 3.3: Masks for the Catmull-Clark scheme. At the extraordinary vertex Catmull and Clark [3] use the coefficients $\alpha = 1 - \frac{7}{4n}$, $\beta = \frac{3}{2n}$, $\gamma = \frac{1}{4n}$, where n is the valency.

of C^1 or C^2 continuous. Corners are points where two or more creases join. A dart is a crease that transitions into a smooth part of the surface.

Schweitzer [26] further extended the scheme with conical and cusp points. At conical and cusp points the limit surface is C^0. At a conical point the space spanned by the limiting tangent vectors has dimension three. At a cusp point the tangent vectors become parallel.

The full set of masks for the Loop scheme including the described extensions are shown in figure 3.4. To implement this scheme, there must be a facility for tagging edges and vertices in the control net, in order to determine which rule to use.

Loop showed tangent plane continuity of the original scheme. Schweitzer [26] proved C^1 continuity at extraordinary vertices up to valency 100 for the original scheme, and also examined the behaviour of the extended scheme. Zorin [34] proved C^1 continuity at extraordinary vertices for all valences for the original scheme.

3.6.1 Normals of the Loop Scheme

In computer graphics the normal vectors of surfaces are used to calculate lighting effects. In order to make a polygonal model look smooth, the normal at a vertex is usually computed by interpolating the normals of the faces joining at the vertex. For the extended Loop scheme, which is only piecewise smooth, this approach is not satisfactory, since the interpolation will cause creases and other sharp features to appear smooth. One solution to this problem is to use the exact normals of the limit surface. Schweitzer [26] derives formulas for the tangent vectors in the extended scheme using eigenanalysis. We list the formulas here for completeness.

The labelling of the vertices is shown in figure 3.5 for smooth, dart, conical and cusp vertices, and in figure 3.6 for crease and corner vertices.

At the limit position of a smooth or dart vertex with valency n, the tangent space is

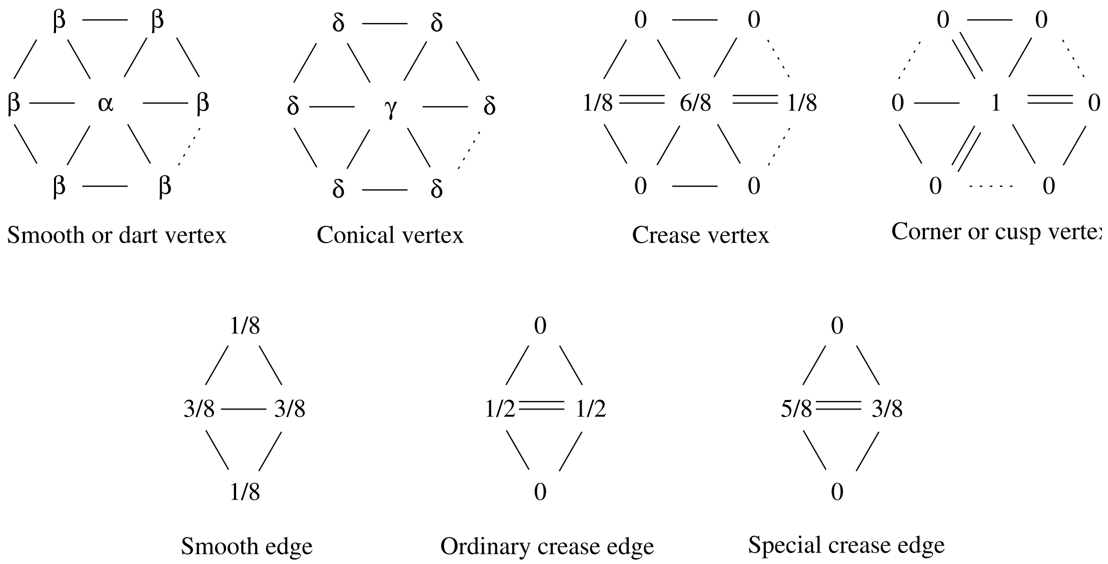


Figure 3.4: Masks for the Loop scheme. The double edges are those marked as crease edges. Here $\beta = \frac{1}{n}(\frac{5}{8} - (\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{n})^2)$, $\alpha = 1 - n\beta$, $\delta = \frac{1}{4n}(1 - \cos\frac{2\pi}{n})$ and $\gamma = 1 - n\delta$, where n is the valency.

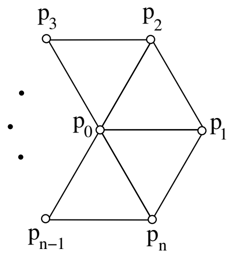


Figure 3.5: Labelling of the 1-neighbourhood of a smooth, dart, conical or cusp vertex.

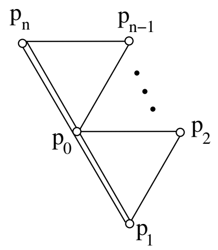


Figure 3.6: Labelling of the 1-neighbourhood of a crease or corner vertex. The double edges are edges marked as creases.

spanned by the vectors

$$\vec{t}_1 = \sum_{i=1}^{n} \cos\left(\frac{2\pi(i-1)}{n}\right) p_i,$$
$$\vec{t}_2 = \sum_{i=1}^{n} \sin\left(\frac{2\pi(i-1)}{n}\right) p_i.$$

At the limit position of a crease vertex, the tangent along the crease is

$$\vec{t}_1 = p_1 - p_n,$$

where n is the crease valency, i.e. the number of edges between and including the two edges marked as crease edges.

The cross tangent vector depends on the valency. For a valency 4 vertex it is

$$\vec{t}_2 = (p_2 + p_3)/2 - (p_1 + 2p_0 + p_4)/4.$$

For an extraordinary vertex with valency $n > 4$ it is

$$\vec{t}_2 = \sum_{i=1}^{n} \omega_i p_i,$$

where $\omega_1 = \omega_n = \sin(\frac{\pi}{n-1})/(2\cos(\frac{\pi}{n-1}) - 2)$ and $\omega_i = \sin(\frac{\pi(i-1)}{n-1})$ for $i = 2, \ldots, n-1$. For $n = 2$

$$\vec{t}_2 = p_1 + p_2 - 2p_0,$$

while for $n = 3$ the eigenanalysis fails because the subdivision matrix does not have a complete set of eigenvectors. The solution is to use generalized eigenvectors, but like Schweitzer we choose to skip this case. The formulas above assume the neighbour vertices are ordinary. If this is not the case, the formulas can be applied after subdividing once.

At a corner vertex the tangent vectors are

$$\vec{t}_1 = p_1 - p_0,$$
$$\vec{t}_2 = p_n - p_0.$$

At a conical point there is no tangent plane. At a cusp point the tangent plane degenerates into a line, which is spanned by

$$\vec{t} = p_0 - (p_1 + \cdots + p_n)/n.$$

3.7 Quadratic NURSS Scheme

The quadratic NURSS scheme proposed by Sabin et al [27] extends quadratic non-uniform tensor product B-spline surfaces to arbitrary topology. The NURSS acronym stands for "Non-Uniform Recursive Subdivision Surfaces".

For uniform subdivision the information necessary to fully specify a surface is the connectivity of the control mesh and the points in the control mesh. Non-uniform surfaces need the additional information given by knot vectors. For a non-uniform tensor product surface there are two knot vectors – one giving the knots for the u parameter curves and one giving the knots for the v parameter curves. The NURSS scheme is less restrictive since it allows different knot spacings for each edge in the control mesh. For the quadratic NURSS scheme it is possible to consider the control mesh as having two knot spacings associated with every edge.

Fig. 3.7 shows part of a control mesh for a quadratic NURSS surface. The knot spacing $d^0_{i,i+1}$ is associated with the edge from p_i to p_{i+1}. By rotating counter-clockwise about p_i, $d^k_{i,i+1}$ is the knot spacing of the kth edge encountered. Similarly, $d^{-k}_{i,i+1}$ is the knot spacing of the kth edge encountered by rotating clockwise about p_i. The subscript i should be taken modulo the number of edges in the face.

In the case where all the knot spacings are the same, the quadratic NURSS scheme is identical to the Doo-Sabin scheme. The subdivision procedure for quadratic NURSS differs from Doo-Sabin in two ways: other coefficients are used to calculate new points, and new knot spacings must be assigned to the edges in the subdivided control polygon.

The new control points, $\bar{p}_i$, are calculated as

$$\bar{p}_i = \frac{V + p_i}{2} + \left(d^0_{i+1,i+2} d^0_{i+3,i+2} + d^0_{i-1,i-2} d^0_{i-3,i-2}\right) \frac{-np_i + \sum_{j=0}^{n-1} \left(1 + 2\cos\left(\frac{2\pi|i-j|}{n}\right)\right) p_j}{8 \sum_{k=0}^{n-1} d^0_{k-1,k} d^0_{k+1,k}},$$

where

$$V = \frac{\sum_{k=0}^{n-1} d^0_{k-1,k} d^0_{k+1,k} p_k}{\sum_{k=0}^{n-1} d^0_{k-1,k} d^0_{k+1,k}},$$

and the labelling is as shown in Fig. 3.7.

Sabin et al propose two different rules for the new knot spacings

$$\begin{aligned} \bar{d}^0_{i,i+1} &= \bar{d}^{-1}_{i,i-1} = d^0_{i,i+1}/2, \\ \bar{d}^0_{i,i-1} &= \bar{d}^1_{i,i+1} = d^0_{i,i-1}/2, \end{aligned}$$

or

$$\begin{aligned} \bar{d}^0_{i,i+1} &= d^0_{i,i+1}/2, \quad \bar{d}^{-}_{i,i+1}1 = (d^0_{i,i+1} + d^{-1}_{i,i-1})/4, \\ \bar{d}^0_{i,i-1} &= d^0_{i,i-1}/2, \quad \bar{d}^1_{i,i+1} = (d^0_{i,i-1} + d^1_{i,i+1})/4. \end{aligned}$$

The former option gives a constant subdivision matrix after a few subdivisions, but Sabin et al. note that the latter option produces higher quality surfaces. In the implementation we use the latter knot spacings.

Sabin, Sederberg, Sewell and Zheng [27] claim to have proved C^1 continuity of the quadratic NURSS scheme for valences $n < 9$ with positive knot spacings and also conjectured the result to be true for all valences. However, their proof only guarantees tangent plane continuity. Furthermore, Qin and Wang [22] show that although the scheme will converge and be C^0 for valences $n < 13$, it may diverge for valences $n \geq 15$.

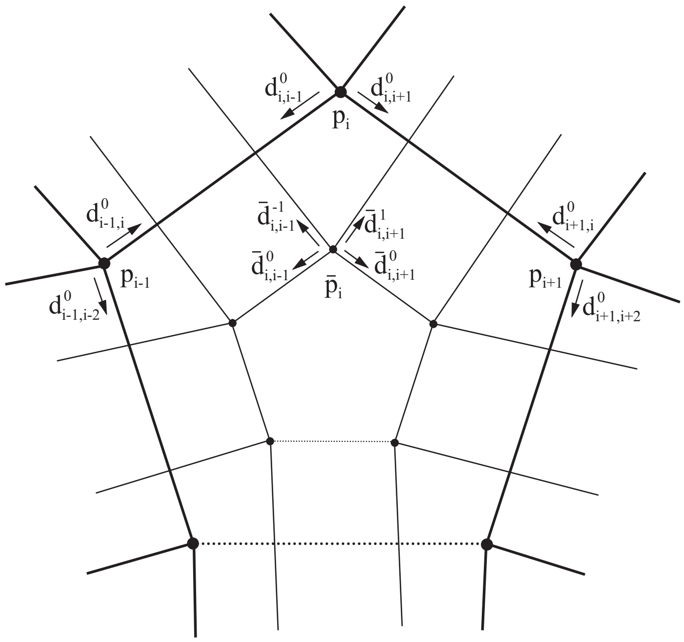


Figure 3.7: Labelling for quadratic NURSS subdivision.

3.8 Cubic NURSS Scheme

The cubic NURSS scheme was proposed in the same paper as the quadratic NURSS scheme. It generalizes non-uniform bicubic tensor product B-spline surfaces to arbitrary topology. As for the quadratic NURSS scheme knot spacings are associated with the edges in the control mesh, although in this case there is only one knot spacing per edge. The knot spacings are labelled as described for the quadratic scheme. In the cubic case $d^l_{ij} = d^l_{ji}$.

Figures 3.8, 3.9 and 3.10 show the labelling used in the formulas for the new face, edge and vertex points, respectively.

A new face point, F, is calculated as

$$F = \frac{\sum_{i=0}^{n-1} w_i p_i}{\sum_{i=0}^{n-1} w_i},$$

where

$$\begin{aligned} w_i = {} & (d^0_{i+1,i} + d^2_{i+1,i} + d^{-2}_{i+1,i} + d^0_{i-2,i-1} + d^2_{i-2,i-1} + d^{-2}_{i-2,i-1}) \\ & (d^0_{i-1,i} + d^2_{i-1,i} + d^{-2}_{i-1,i} + d^0_{i+2,i+1} + d^2_{i+2,i+1} + d^{-2}_{i+2,i+1}). \end{aligned}$$

An edge point, E, is computed as

$$E = (1 - \alpha_{ij} - \alpha_{ji})M + \alpha_{ij}F_{ij} + \alpha_{ji}F_{ji},$$

where

$$\alpha_{ij} = \frac{d^1_{ji} + d^{-1}_{ij}}{2(d^1_{ji} + d^{-1}_{ij} + d^{-1}_{ji} + d^1_{ij})},$$

if the denominator is non-zero, otherwise $\alpha_{ij} = 0$, and

$$M = \frac{(d^0_{ji} + d^2_{ji} + d^{-2}_{ji})p_i + (d^0_{ij} + d^2_{ij} + d^{-2}_{ij})p_j}{d^0_{ji} + d^2_{ji} + d^{-2}_{ji} + d^0_{ij} + d^2_{ij} + d^{-2}_{ij}}.$$

A new vertex point, V, is calculated as

$$V = cp_0 + \frac{3\sum_{i=0}^{n}(m_i M_i + f_i F_{i,i+1})}{n\sum_{i=1}^{n}(m_i + f_{i,i+1})},$$

where

$$\begin{aligned} m_i &= (d^1_{0i} + d^{-1}_{0i})(d^2_{0i} + d^{-2}_{0i})/2, \\ f_{ij} &= d^1_{0i} d^{-1}_{0j}, \\ c &= \frac{n-3}{n} \text{ if } \sum_{i=1}^{n}(m_{i0} f_{i,i+1}) \neq 0, \text{ otherwise } c = 0. \end{aligned}$$

The new knot spacings are determined as illustrated in Fig. 3.11.

The only published work on continuity of cubic NURSS surfaces are some inconclusive numerical experiments by Sabin, Sederberg, Sewell and Zheng [27], and an article by Qin and Wang [21] apparently available only in Chinese.

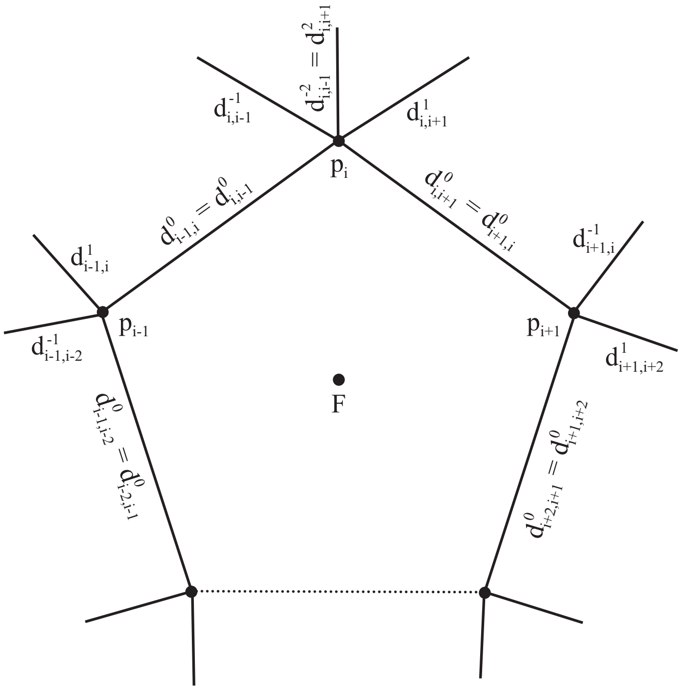


Figure 3.8: Face point for cubic NURSS subdivision.

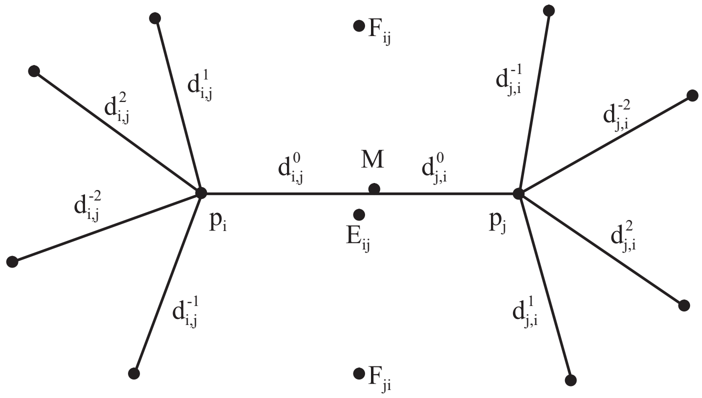


Figure 3.9: Edge point for cubic NURSS subdivision.

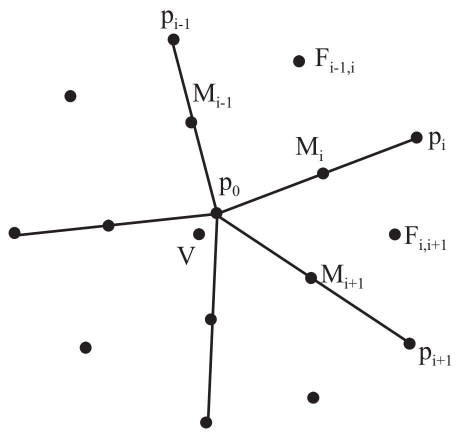


Figure 3.10: Vertex point for cubic NURSS subdivision.

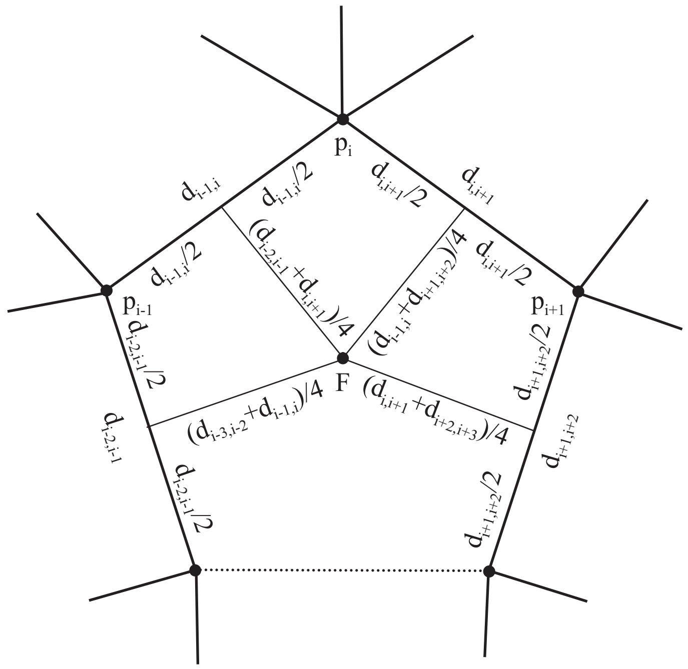


Figure 3.11: New knot spacings for cubic NURSS subdivision.

Chapter 4

Analyzing Subdivision Schemes

In this chapter we describe conditions for convergence and smoothness of subdivision surfaces. Most known schemes reduce to spline schemes on topologically regular meshes, so for these schemes convergence and continuity away from extraordinary points follow from the properties of the basis functions. Near extraordinary points, however, other methods must be employed. We only consider analysis near extraordinary points and assume the schemes are known to generate smooth surfaces away from extraordinary points.

4.1 Subdivision Matrix for Surfaces

In the following we consider only linear subdivision schemes. For these schemes, subdivision can be thought of as a linear mapping from one set of control points to a new and larger set of control points. In section 2.3.1 we showed that subdivision of curves can be written as a matrix equation where the control points after subdivision are related to the original control points by a subdivision matrix. For surfaces one step of linear subdivision can be written in the same way

$$\boldsymbol{p}^{j+1} = S\boldsymbol{p}^{j},$$

where S is the *subdivision matrix* and $\boldsymbol{p}^j$ is a vector of control points on level j.

We restrict the analysis to *stationary*, *locally defined* and *finitely supported* subdivision schemes.

- A subdivision scheme is stationary, if the subdivision rules applied to a fixed neighbourhood of a vertex remains constant after a finite number of initial subdivisions.
- A subdivision scheme is locally defined, if each of the subdivision rules in the scheme only uses a finite and constant number of control points.
- A subdivision scheme is finitely supported, if the basis functions on the regular mesh have finite support.

With these restrictions we do not have to look at the full subdivision matrix to analyze the behaviour near an extraordinary point. Instead we examine the *local subdivison matrix* which relates the neighbourhood of a vertex to the neighbourhood of the same vertex after one step of subdivision. The local subdivision matrix is a square matrix. In the rest of this chapter we denote by S the local subdivision matrix of a given subdivision scheme, and wherever we say subdivision matrix we mean the local subdivision matrix. Also, we denote by $\boldsymbol{p}^j$ the vector of control points in the same neighbourhood as used for the subdivision matrix.

As for curves, repeated subdivision of a control polygon for a surface amounts to repeated multiplication with the subdivision matrix. In general, the control points on level j are obtained from the initial control points as

$$\boldsymbol{p}^j = S^j \boldsymbol{p}^0 \tag{4.1}$$

What we want to analyze is then the behaviour of the limit $\lim_{j\to\infty} \boldsymbol{p}^j$ if it exists.

We want the subdivision process to be affine invariant, i.e. we should get the same surface if we apply an affine transformation to the control points and then subdivide as we get when we subdivide and then apply the same affine transformation. The class of affine transformations consists of translations, rotations and shears. Affine invariance means the subdivision process is independent of the coordinate system.

Lemma 6 *A linear subdivision procedure is affine invariant if and only if every row of the subdivision matrix sum to one.*

Proof. Assume first that every row of the subdivision matrix S sums to one. Applying an arbitrary affine transformation given by a matrix A and a translation vector v to the control points p_k^j and then calculating the new control point p_i^{j+1} we get

$$\sum_{k=1}^{n} s_{ik}(Ap_k^j + v) = A\sum_{k=1}^{n} s_{ik}p_k^j + \sum_{k=1}^{n} s_{ik}v = Ap_i^{j+1} + v,$$

showing affine invariance. Now assume affine invariance. Then

$$\begin{aligned} Ap_i^{j+1} + v = \sum_{k=1}^{n} s_{ik}(Ap_k^j + v) \quad &\Rightarrow \quad Ap_i^{j+1} + v = A\sum_{k=1}^{n} s_{ik}p_k^j + \sum_{k=1}^{n} s_{ik}v \\ &\Rightarrow \quad A\left(p_i^{j+1} - \sum_{k=1}^{n} s_{ik}p_k^j\right) = \sum_{k=1}^{n} s_{ik}v \; - v \end{aligned}$$

This should be true for all A and v. That implies

$$\sum_{k=1}^{n} s_{ik}v - v = 0 \quad \Rightarrow \quad \sum_{k=1}^{n} s_{ik} = 1$$

showing that all rows sum to one. □

Lemma 7 *The subdivision matrix $S \in \mathbb{R}^{n\times n}$ of an affine invariant subdivision scheme has an eigenvalue $\lambda_1 = 1$ with corresponding right eigenvector $\boldsymbol{v}_1 = [1,\ldots,1]^{\mathrm{T}}$.*

Proof. By lemma 6 every row of S sums to one. Then

$$S\begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} S_{1i} \\ \vdots \\ \sum_{i=1}^{n} S_{ni} \end{bmatrix} = 1\begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}.$$

Therefore, $\lambda_1 = 1$ is an eigenvalue of S with corresponding eigenvector $\boldsymbol{v}_1 = [1,\ldots,1]^{\mathrm{T}}$. □

4.2 Behaviour at Extraordinary Points

Our goal is to analyze the continuity of a subdivision surface near an extraordinary point. For triangular schemes, extraordinary points arise if the initial control mesh has vertices with a valence other than 6. For quadrilateral schemes, extraordinary points are caused by vertices with valence other than 4, or by faces with other than 4 sides. An important observation is that the number of extraordinary points is preserved by subdivision, i.e. the control net after any number of subdivisions induces the same number of extraordinary points in the surface as did the initial control net. Moreover, by applying subdivision any two extraordinary vertices or faces can be separated by an arbitrary number of regular vertices or faces. Combined with the locality of subdivision this implies that it is sufficient to analyze the surface resulting from a mesh with a single extraordinary vertex or face at the center.

In the following we look at an infinite control mesh, C_0, with a single extraordinary vertex of valence n (or face with n sides) at the center surrounded by r rings of ordinary vertices. The value of r should be chosen such that at least one full surface ring is defined by the basis functions on the regular parts of C_0. The surface defined by the regular parts of C_0 is called $\boldsymbol{s}_0$. This surface has a single n sided hole at the center. Subdividing C_0 m times we get a new control mesh, C_m, whose regular parts define a surface, $\boldsymbol{s}_m$. Subdivision extends the regular part of the control mesh, so $\boldsymbol{s}_{m-1} \subset \boldsymbol{s}_m$ for $m \geq 1$. The new part of the surface is a full ring $\boldsymbol{r}_m = \boldsymbol{s}_{m-1} \setminus \boldsymbol{s}_m$ joining smoothly to the n-sided hole in $\boldsymbol{s}_{m-1}$, since the scheme is assumed to generate smooth surfaces away from extraordinary points. The new rings $\boldsymbol{r}_m$ can be parameterized over the domain $\Omega \times \mathbb{Z}_n$, where Ω is either

$$\Omega^{\triangle} = \left\{ \, (u,v) \;\middle|\; u,v \geq 0 \wedge 1 \leq u+v \leq 2 \, \right\},$$

for triangular meshes or

$$\Omega^{\square} = \left\{ \, (u,v) \;\middle|\; u,v \geq 0 \wedge 1 \leq \max\{u,v\} \leq 2 \, \right\},$$

for quadrilateral meshes, and $\mathbb{Z}_n = 1, 2, \ldots, n$, see figures 4.1 and 4.2. One parameterization of the rings is given by the basis functions $B_k(u,v)$ and a sequence of control points $p_1^m, \ldots, p_K^m$ associated with the vertices in a properly selected submesh $D_m \subset C_m$. We add an extra parameter, j, to the basis functions such that $B_k(u,v,j) = B_k(u,v)$ if p_k^m is one of the control points defining the jth sector, $\boldsymbol{r}_m^j$, of the ring, and zero otherwise. We now have

$$\begin{gathered}
\boldsymbol{r}_m : \Omega \times \mathbb{Z}_n \to \mathbb{R}^3, \\
\boldsymbol{r}_m : (u,v,j) \mapsto \boldsymbol{r}_m^j = \sum_{k=1}^{M} B_k(u,v,j) p_k^m = \boldsymbol{B}(u,v,j)\boldsymbol{p}^m.
\end{gathered} \tag{4.2}$$

Because Ω is closed, adjacent sectors overlap on their common boundary curve.

For a Loop surface the basis functions would be the quartic box splines, while for Catmull-Clark and Doo-Sabin surfaces they would be bicubic and biquadratic tensor product B-splines, respectively.

We can now define convergence of a subdivision scheme at an extraordinary point in terms of the surface rings.

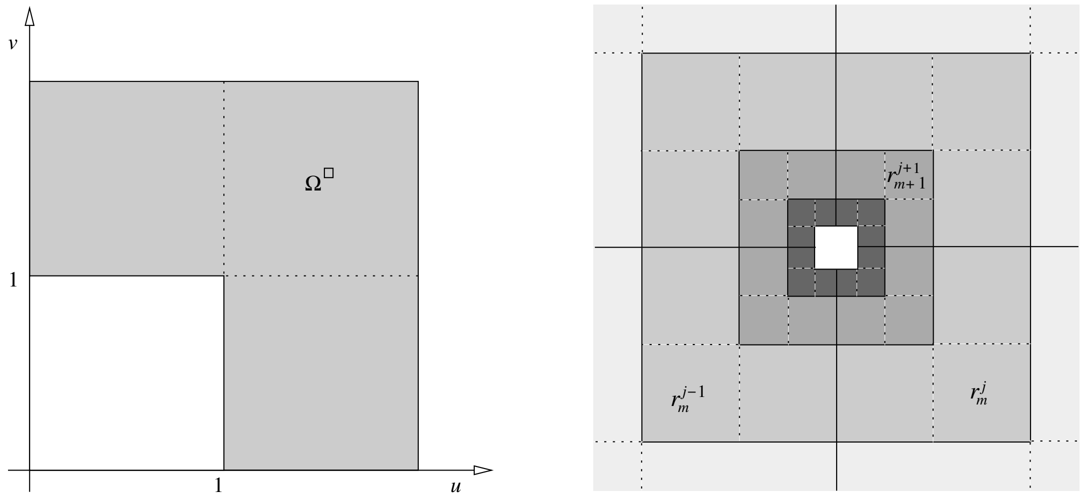


Figure 4.1: Domain $\Omega^\square$ (left) and the structure of the rings $\boldsymbol{r}_m$ for a quadrilateral control mesh at a 4 sided hole.

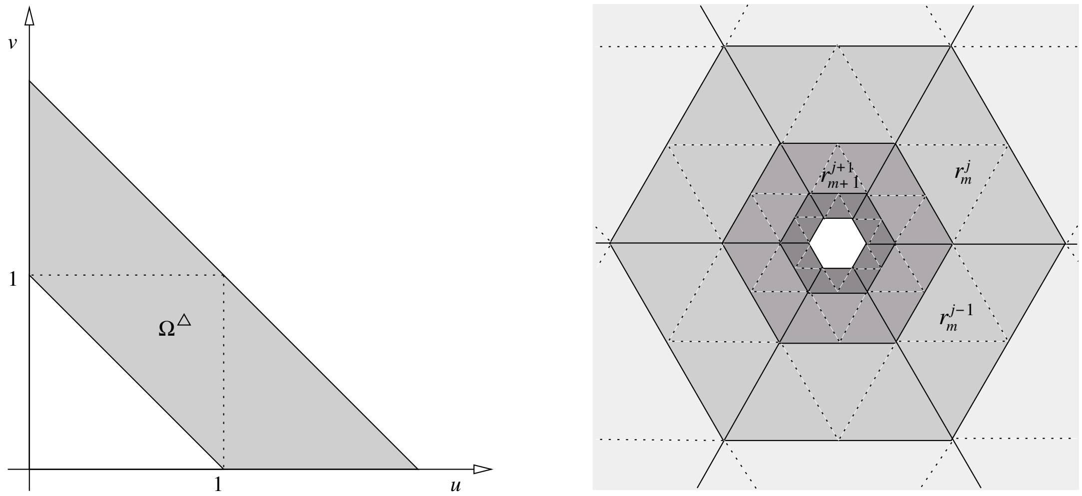


Figure 4.2: Domain $\Omega^\triangle$ (left) and the structure of the rings $\boldsymbol{r}_m$ for a triangular control mesh at a 6 sided hole.

Definition 6 *A subdivision procedure S is convergent, if there exists a unique limit point p such that*

$$\lim_{m\to\infty} x_m = p$$

for any sequence of points $x_m \in \boldsymbol{r}_m$.

For a convergent subdivision procedure the rings $\boldsymbol{r}_m$ define a surface that in the limit fills the initial n sided hole except for the point p. This surface is given by

$$\boldsymbol{r} = \bigcup_{m\in\mathbb{N}} \boldsymbol{r}_m.$$

The closure

$$\bar{\boldsymbol{r}} = \boldsymbol{r} \cup p,$$

then completely fills the n sided hole.

We now define two types of continuity, tangent plane continuity and C^1 continuity.

Definition 7 *The surface $\bar{\boldsymbol{r}}$ is called tangent plane continuous at p, if the subdivision procedure S converges and if there is a unique limit $\boldsymbol{n}(p)$ for any sequence of normal vectors*

$$\lim_{m\to\infty} \boldsymbol{n}(x_m) = \boldsymbol{n}(p), \quad x_m \in \boldsymbol{r}_m.$$

Definition 8 *The surface $\bar{\boldsymbol{r}}$ is called C^1 continuous at p, if there is a regular C^1 parameterization of $\bar{\boldsymbol{r}}$ near p.*

C^1 continuity is a stronger property than tangent plane continuity. Figure 4.3 shows the graph of the surface given by the parameterization

$$(r,t) \mapsto (r\cos 3t, r\sin 3t, r^2 \sin t), \quad (r,t) \in [0,1] \times [0, 2\pi[.$$

The surface is tangent plane continuous with limit normal $\lim_{r\to 0} \boldsymbol{n}(r,t) = (0,0,1)$, but the surface is not regular at $(0,0,0)$ since the projection of the surface on the xy-plane is not injective. A designer would not consider such a surface smooth. C^1 continuity more closely matches the intuitive notion of smoothness.

4.3 Eigenstructure Analysis

It turns out that the properties of the limit surface $\bar{\boldsymbol{r}}$ are determined by the eigenstructure of the subdivision matrix. Recall that a right eigenvector $\boldsymbol{v}$ with corresponding eigenvalue λ of a matrix A satisfies $A\boldsymbol{v} = \lambda\boldsymbol{v}$.

We order the eigenvalues of the subdivision matrix S by modulus

$$|\lambda_1| \geq |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_K|,$$

where the eigenvalues are listed with their algebraic multiplicity. The corresponding eigenvectors are denoted by $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3, \ldots, \boldsymbol{v}_K$.

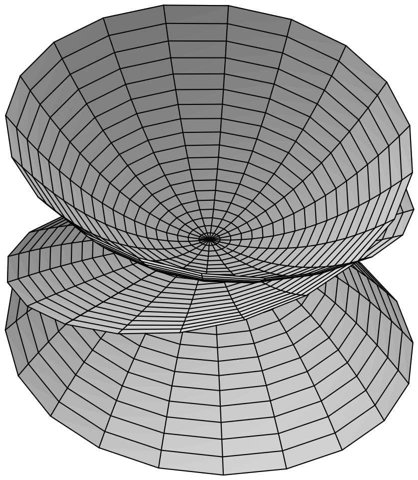


Figure 4.3: Tangent plane continuous surface without a regular parameterization.

Assuming the subdivision matrix S has a complete set of linearly independent right eigenvectors[1] $\boldsymbol{v}_i$ with corresponding eigenvalues λ_i, the initial control points can be written in the basis formed by the eigenvectors for some coefficients a_i

$$\boldsymbol{p}^0 = \sum_{i=1}^{K} a_i \boldsymbol{v}_i. \tag{4.3}$$

Then (4.1) can be rewritten in terms of the eigenvectors

$$\boldsymbol{p}^j = S^j \boldsymbol{p}^0 = S^j \sum_{i=1}^{K} a_i \boldsymbol{v}_i = \sum_{i=1}^{K} S^j a_i \boldsymbol{v}_i = \sum_{i=1}^{K} \lambda_i^j a_i \boldsymbol{v}_i. \tag{4.4}$$

This expansion in the eigenbasis leads us to a sufficient condition for the convergence of a subdivision scheme.

Theorem 8 *A subdivision algorithm converges, if* $1 = \lambda_1 > |\lambda_2|$.

Proof. The proof given here is similar to that in [23]. By lemma 7 λ_1 is an eigenvalue of S with corresponding eigenvector $\boldsymbol{v}_1 = [1, \ldots, 1]^{\mathrm{T}}$. From (4.4) we have

$$\boldsymbol{p}^m = a_1 \boldsymbol{v}_1 + \sum_{i=2}^{K} \lambda_i^m a_i \boldsymbol{v}_i.$$

Then by (4.2) the ring $\boldsymbol{r}_m$ is given by

$$\boldsymbol{r}_m(u, v, j) = \boldsymbol{B}(u, v, j) \left(a_1 \boldsymbol{v}_1 + \sum_{i=2}^{K} \lambda_i^m a_i \boldsymbol{v}_i \right).$$

[1] When the subdivision matrix is defective, i.e. it does not have a complete set of linearly independent eigenvectors, the control points can be expanded in a basis of generalized eigenvectors instead. A generalized eigenvector $\bar{\boldsymbol{v}}_i$ of S is either an eigenvector of S or satisfies $S\bar{\boldsymbol{v}}_i = \lambda_i \bar{\boldsymbol{v}}_i + \bar{\boldsymbol{v}}_{i-1}$

By hypothesis $|\lambda_i| < 1,\ i \geq 2$, so $|\lambda_i|^m \to 0$ as $m \to \infty$ for $i \geq 2$. This implies that $\boldsymbol{B}(u,v,j)\sum_{i=2}^{K}\lambda_i^m a_i \boldsymbol{v}_i$ converges uniformly to zero, since the domain $\Omega \times \mathbb{Z}_n$ is compact. We use that the basis functions form a partition of unity, so that $\boldsymbol{B}(u,v,j)\boldsymbol{v}_1 = 1$, and finally get

$$\lim_{m\to\infty} \boldsymbol{r}_m(u,v,j) = \boldsymbol{B}(u,v,j)a_1\boldsymbol{v}_1 = a_1. \tag{4.5}$$

By definition 6 the subdivision algorithm is therefore convergent and the limit point is $p = a_1$. □

Equation (4.5) gives us a formula for computing the limit point of the irregular vertex. To evaluate a_1 we need to define left eigenvectors. If we let $V = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_K]$ and $D = \text{diag}(\lambda_1, \ldots, \lambda_K)$, then the definition of a right eigenvector gives us $SV = VD$. We have assumed the subdivision matrix has a complete set of linearly independent right eigenvectors so V is a regular matrix with an inverse V^{-1}. If we multiply with V^{-1} at the front and back we get $V^{-1}S = DV^{-1}$. The rows $\tilde{\boldsymbol{v}}_i$ of V^{-1} are called left eigenvectors since $\tilde{\boldsymbol{v}}_i S = \lambda_i \tilde{\boldsymbol{v}}_i$. Because $V^{-1}V = I$, $\tilde{\boldsymbol{v}}_i \cdot \boldsymbol{v}_j = \delta_{ij}$, where δ_{ij} is the Kronecker delta symbol, which is one for $i = j$ and zero otherwise. Now we can compute a_1 from (4.3)

$$\tilde{\boldsymbol{v}}_1 \cdot \boldsymbol{p}^0 = \sum_{i=1}^{K} a_i \tilde{\boldsymbol{v}}_1 \cdot \boldsymbol{v}_i = \sum_{i=1}^{K} a_i \delta_{1,i} = a_1.$$

Thus, the limit point of the irregular vertex is

$$p = \tilde{\boldsymbol{v}}_1 \cdot \boldsymbol{p}^0. \tag{4.6}$$

Here we have chosen the left eigenvector $\tilde{\boldsymbol{v}}_1$ such that $\tilde{\boldsymbol{v}}_1 \cdot \boldsymbol{v}_1 = 1$. Obviously, we could choose any vector spanning the same subspace. In that case we must modify the formula for the limit point

$$p = \frac{\tilde{\boldsymbol{v}}_1 \cdot \boldsymbol{p}^0}{\tilde{\boldsymbol{v}}_1 \cdot \boldsymbol{v}_1}.$$

In preparation for some theorems in the next sections we return to the expansion of the control points in the eigenvectors of the subdivision matrix. We are interested in finding the tangent plane at the limit point. To do this we assume the subdivision process converges, i.e. $1 = \lambda_1 > \lambda_2$. Further, we assume that $\lambda = \lambda_2 = \lambda_3$ and $|\lambda_2| > |\lambda_4|$. From (4.4) we now have

$$\boldsymbol{p}^j = a_1\boldsymbol{v}_1 + \lambda^j(a_2\boldsymbol{v}_2 + a_3\boldsymbol{v}_3) + \sum_{i=4}^{K}\lambda_i^j a_i \boldsymbol{v}_i. \tag{4.7}$$

If we choose a coordinate system where $a_1 = 0$ and divide by λ we get

$$\frac{\boldsymbol{p}^j}{\lambda^j} = a_2\boldsymbol{v}_2 + a_3\boldsymbol{v}_3 + \sum_{i=4}^{K}\left(\frac{\lambda_i}{\lambda}\right)^j a_i\boldsymbol{v}_i.$$

As $j \to \infty$, $\left(\frac{\lambda_i}{\lambda}\right)^j \to 0$ since by assumption $|\lambda| > |\lambda_i|,\ i = 4, \ldots, K$. This means that for large j the control points in the neighbourhood of the central vertex essentially lie in the plane spanned by a_2 and a_3. For now we assume a_2 and a_3 are linearly independent. Then we can apply an affine transformation mapping a_2 and a_3 to the usual basis e_1 and e_2 in the plane. With respect to this basis the coordinates of the transformed control points are the rows in the matrix $[\boldsymbol{v}_2, \boldsymbol{v}_3]$. These coordinates are independent of the initial control mesh and depend solely on the subdivision matrix. We also see that if we subdivide once this amounts to a scaling with the eigenvalue λ.

From the above discussion we get two important results. Firstly, the tangent plane at the limit point is spanned by a_2 and a_3. Secondly, the projection of the control points onto the tangent plane can be transformed into a configuration that only depends on the subdivision matrix. Clearly, we can also project the surface onto the tangent plane and the same transformation will give a surface in $\mathbb{R}^2$ that depends only on the subdivision matrix. This indicates that this surface can reveal properties of subdivision surfaces generated by an almost arbitrary set of control points, namely those for which a_2 and a_3 are linearly independent. In the next section we define a characteristic map of which the surface in $\mathbb{R}^2$ is a representation.

The tangent vectors can be evaluated as

$$\vec{t_1} = a_2 = \tilde{\boldsymbol{v}}_2 \cdot \boldsymbol{p}^0, \tag{4.8}$$

$$\vec{t_2} = a_3 = \tilde{\boldsymbol{v}}_3 \cdot \boldsymbol{p}^0. \tag{4.9}$$

The eigenvalue λ_1 is called the dominant eigenvalue and its corresponding eigenvector $\boldsymbol{v}_1$ is called the dominant eigenvector. The eigenvalues $\lambda_2 = \lambda_3$ are called the sub-dominant eigenvalues with corresponding sub-dominant eigenvectors.

4.4 Conditions for Tangent Plane, C^1 and C^k Continuity

In this section we define the characteristic map and give conditions for tangent plane, C^1 and C^k continuity of a subdivision surface at an extraordinary point.

4.4.1 Characteristic Map

The conditions for tangent plane, C^1 and C^k continuity all rely on the characteristic map defined by Reif [23].

Definition 9 *For a subdivision algorithm S with eigenvalues $1 > |\lambda_2| = |\lambda_3| > |\lambda_4|$, the characteristic map $\Psi : \Omega \times \mathbb{Z}_n \to \mathbb{R}^2$ is defined by*

$$\Psi : (u, v, j) \mapsto \boldsymbol{B}(u, v, j)[\boldsymbol{v}_2, \boldsymbol{v}_3], \tag{4.10}$$

where $\boldsymbol{v}_2$ and $\boldsymbol{v}_3$ are eigenvectors corresponding to λ_2 and λ_3, written as column vectors.

Ψ is called regular*, if $\Delta(u, v, j) = \det \partial\Psi(u, v, j)/\partial(u, v) \neq 0$ for all $(u, v, j) \in \Omega \times \mathbb{Z}_n$.*

Remark. The right eigenvectors $\boldsymbol{v}_2$ and $\boldsymbol{v}_3$ are not uniquely determined. However, the important properties of the characteristic map, namely regularity and injectivity, are independent of the choice of $\boldsymbol{v}_2$ and $\boldsymbol{v}_3$ as long as they span the invariant subspace corresponding to λ_2 and λ_3.

Equation (4.10) allows us to view the characteristic map as a surface embedded in $\mathbb{R}^2$ defined by the basis functions $\boldsymbol{B}(u, v, j)$ and a set of control points. It was this surface we saw in the eigenanalysis above. The surface can be used to visualize the characteristic map. Figure 4.4 shows a graphical representation of the characteristic map for a smooth vertex of valency 7 in the Loop scheme. We will show other examples of characteristic maps for the Loop scheme in section 4.5.1.

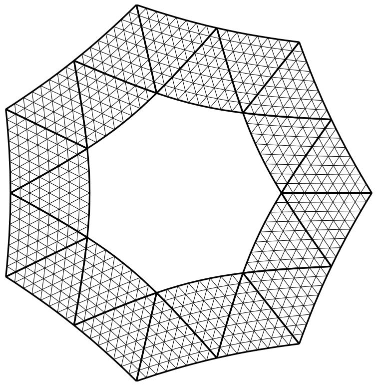


Figure 4.4: Characteristic map for a valency 7 smooth vertex in the Loop subdivision scheme.

Definition 9 is only applicable to those subdivision schemes for which $1 > |\lambda_2| = |\lambda_3| > |\lambda_4|$. Schweitzer [26] suggests that $|\lambda_2| = |\lambda_3|$ is a property of the class of symmetric subdivision schemes and extends the definition of the characteristic map to handle non-symmetric schemes for which $|\lambda_2| > |\lambda_3|$. The extended Loop scheme is an example of a scheme that is non-symmetric near a crease, since, for example, the cross tangent vector at a crease only depends on the surface on one side of the crease.

4.4.2 Sufficient Conditions for Tangent Plane Continuity

Theorem 9 *If $\lambda_2 = \lambda_3$, $1 > |\lambda_2| > |\lambda_4|$, is a real eigenvalue with algebraic and geometric multiplicity 2 and if the characteristic map is regular, then $\bar{\boldsymbol{r}}$ is tangent plane continuous for almost every initial vector $\boldsymbol{p}^0$ of control points.*

Proof. The proof follows that in [23], except for a minor correction in the case where the Jacobian of the characteristic map in negative. We use the expansion (4.7)

$$\boldsymbol{p}^m = a_1\boldsymbol{v}_1 + \lambda_2^m(a_2\boldsymbol{v}_2 + a_3\boldsymbol{v}_3) + \sum_{i=1}^{K} \lambda_i^m a_i \boldsymbol{v}_i.$$

The sectors $\boldsymbol{r}_m^j$ of a ring can be written

$$\begin{aligned}
\boldsymbol{r}_m^j(u,v) &= \boldsymbol{B}(u,v,j)\cdot\boldsymbol{p}^m \\
&= \boldsymbol{B}(u,v,j)\cdot a_1\boldsymbol{v}_1 + \lambda_2^m\boldsymbol{B}(u,v,j)\cdot(a_2\boldsymbol{v}_2 + a_3\boldsymbol{v}_3) + \sum_{i=4}^{K}\lambda_i^m\boldsymbol{B}(u,v,j)\cdot a_i\boldsymbol{v}_i \\
&= \boldsymbol{v}_1 + \lambda_2^m\boldsymbol{B}(u,v,j)\cdot(a_2\boldsymbol{v}_2 + a_3\boldsymbol{v}_3) + \sum_{i=4}^{K}\lambda_i^m\boldsymbol{B}(u,v,j)\cdot a_i\boldsymbol{v}_i.
\end{aligned}$$

The partial derivatives are

$$\boldsymbol{r}^j_{m,u}(u,v) = \lambda_2^m \boldsymbol{B}_u(u,v,j)\cdot(a_2\boldsymbol{v}_2 + a_3\boldsymbol{v}_3) + \sum_{i=4}^K \lambda_i^m \boldsymbol{B}_u(u,v,j)\cdot a_i\boldsymbol{v}_i,$$

$$\boldsymbol{r}^j_{m,v}(u,v) = \lambda_2^m \boldsymbol{B}_v(u,v,j)\cdot(a_2\boldsymbol{v}_2 + a_3\boldsymbol{v}_3) + \sum_{i=4}^K \lambda_i^m \boldsymbol{B}_u(u,v,j)\cdot a_i\boldsymbol{v}_i.$$

To ease the notation slightly we drop the parameters on the basis functions. The cross product of the partial derivatives is

$$\begin{aligned}
&\boldsymbol{r}^j_{m,u}(u,v)\times\boldsymbol{r}^j_{m,v}(u,v) = \\
&\quad \lambda_2^{2m}\Delta a_2\times a_3 \\
&\quad + \sum_{i+4}^K \left(\lambda_2^m\lambda_i^m(\boldsymbol{B}_u\cdot(a_2\boldsymbol{v}_2+a_3\boldsymbol{v}_3))\times(\boldsymbol{B}_v\cdot a_i\boldsymbol{v}_i) + (\boldsymbol{B}_u\cdot a_i\boldsymbol{v}_i)\times(\boldsymbol{B}_v\cdot(a_2\boldsymbol{v}_2+a_3\boldsymbol{v}_3))\right) \\
&\quad + \sum_{i=4}^K\sum_{j=4}^K \lambda_i^m\lambda_j^m(\boldsymbol{B}_u\cdot\boldsymbol{v}_i)(\boldsymbol{B}_v\cdot\boldsymbol{v}_j)a_i\times a_j,
\end{aligned} \tag{4.11}$$

where we have used the Jacobian of the characteristic map

$$\Delta = \det\frac{\partial\Psi(u,v,j)}{\partial(u,)} = \begin{vmatrix} \boldsymbol{B}_u\cdot\boldsymbol{v}_2 & \boldsymbol{B}_v\cdot\boldsymbol{v}_3 \\ \boldsymbol{B}_u\cdot\boldsymbol{v}_3 & \boldsymbol{B}_v\cdot\boldsymbol{v}_3 \end{vmatrix} = (\boldsymbol{B}_u\cdot\boldsymbol{v}_2)(\boldsymbol{B}_v\cdot\boldsymbol{v}_3) - (\boldsymbol{B}_v\cdot\boldsymbol{v}_3)(\boldsymbol{B}_u\cdot\boldsymbol{v}_3).$$

If we divide (4.11) with λ_2^{2m} then as $m\to\infty$ the terms with coefficients $\left(\frac{\lambda_i}{\lambda_2}\right)^m$ and $\left(\frac{\lambda_i\lambda_j}{\lambda_2^2}\right)^m$ will converge uniformly to zero in (u,v,j). By hypothesis the characteristic map is regular so Δ is non-zero and $\|a_2\times a_3\|>0$ for almost every choice of control points $\boldsymbol{p}^0$. Therefore, the limiting normalized normal vector is given by

$$\begin{aligned}
\lim_{m\to\infty}\boldsymbol{n}^j_m(u,v) &= \lim_{m\to\infty}\frac{\lambda_2^{-2m}\boldsymbol{r}^j_{m,u}(u,v)\times\boldsymbol{r}^j_{m,v}(u,v)}{\|\lambda_2^{-2m}\boldsymbol{r}^j_{m,u}(u,v)\times\boldsymbol{r}^j_{m,v}(u,v)\|} = \lim_{m\to\infty}\frac{\Delta a_2\times a_3}{|\Delta|\;\|a_2\times a_3\|} \\
&= \operatorname{sign}(\Delta)\frac{a_2\times a_3}{\|a_2\times a_3\|} = \boldsymbol{n}(p).
\end{aligned}$$

This shows tangent plane continuity of $\bar{\boldsymbol{r}}$. □

Remark. The phrase "almost every vector" requires a more precise explanation. The theorem holds for those vectors of control points whose expansion in the eigenvectors (cf. (4.3)) has linearly independent coefficients a_2 and a_3. This was also the condition we mentioned above. An analogy is a cubic Bèzier curve which is smooth for almost all control polygons, but it is possible to choose a control polygon that produces a cusp on the curve.

4.4.3 Sufficient Conditions for a Regular C^1 Parameterization

We can now state Reif's [23] sufficient conditions for the existence of a regular C^1 continuous parameterization near an extraordinary point.

Theorem 10 *If $\lambda_2 = \lambda_3$, $1 > |\lambda_2| > |\lambda_4|$ is a real eigenvalue with algebraic and geometric multiplicity 2 and if the characteristic map is regular and injective, then $\bar{\boldsymbol{r}}$ has a regular C^1 parameterization near p for almost every initial vector $\boldsymbol{p}^0$ of control points.*

Proof. See [23]. □

Remark. Injectivity of the characteristic map should be understood such that for fixed $j \in \mathbb{Z}_n$, the sector $\Psi(u, v, j)$ of the characteristic map is injective, and except for a common boundary between sectors j and $(j+1) \bmod n$ no sectors may overlap.

The interpretation of the phrase "almost every vector" is as explained in the remark for theorem 9.

4.4.4 Sufficient Conditions for a Regular C^k Parameterization

Prautzsch [18] and Zorin [34] extend theorem 10 to provide a sufficient condition for C^k continuity of a subdivision surface near an extraordinary point. We give a specialization of the theorem by Prautzsch.

Theorem 11 *If $\lambda = \lambda_2 = \lambda_3$, $1 > |\lambda_2|$ is a (possibly complex) eigenvalue with algebraic and geometric multiplicity 2 and if the characteristic map is regular and injective, and*

$$|\lambda_2|^k > |\lambda_4|, \quad k \geq 1,$$

then $\bar{\boldsymbol{r}}$ has a regular C^k parameterization near p for almost every initial vector $\boldsymbol{p}^0$ of control points.

Proof. See [18]. □

Remark. The remarks for theorem 10 apply to theorem 11 as well.

4.4.5 Proving Regularity and Injectivity of the Characteristic Map

In this section we will sketch the methods that have been used to prove regularity and injectivity of the characteristic map.

Peters and Reif [17] analyze symmetric subdivision schemes generalizing tensor product B-spline surfaces, including the Doo-Sabin and Catmull-Clark schemes. They begin by showing that for symmetric subdivision schemes it is enough to prove regularity and injectivity of a sector, $\Psi(u, v, j)$ for fixed j, of the characteristic map. Then they use the inverse function theorem to show that if the sector is regular, then injectivity of the sector is equivalent to injectivity of the boundary curve of the sector. By using symmetry and the scaling relation $S\Psi = \lambda_2\Psi$ of the characteristic map, they reduce the analysis to checking the sign of a partial derivative on half of the boundary curve common to the boundary of the sector and the inner boundary curve of the characteristic map. More precisely, let the eigenvectors defining the characteristic map be chosen such that $\Psi(2, 2, 1) = (\alpha, 0)$ for some $\alpha > 0$ and let $\Psi(u, v, 1) = (\Psi^1(u, v), \Psi^2(u, v))$, then their main theorem can be stated as below.

Theorem 12 *If $\Psi(u, v, 1)$ is regular and $\Psi^1_v(1, t), \Psi^2_v(1, t) > 0$, $t \in [0, 1]$, then the characteristic map Ψ is regular and injective.*

From the theorem they get the following corollary.

Corollary 13 *If $\Psi_v^1(u,v),\ \Psi_v^2(u,v) > 0$ for all $(u,v) \in \Omega^\square$, then the characteristic map Ψ is regular and injective.*

To apply the corollary they convert the three polynomial patches of a sector of the characteristic map from the B-spline basis to the Bernstein basis. Then the partial derivative can be expressed as a Bézier surface by taking differences in the original Bézier control mesh and scaling. By verifying that the coordinates of the control points for the derivative Bézier surface are positive, the convex hull property of a Bézier surface can be used to show that the conditions of corollary 13 are satisfied. Peters and Reif prove C^1 continuity analytically for a class of schemes including Doo-Sabin for all valencies, and prove C^1 continuity numerically for the Catmull-Clark scheme for valencies up to 10000.

Umlauf [30] extends the method of Peters and Reif to triangular schemes. He proves C^1 continuity of the Loop scheme for all valencies.

Schweitzer [26] analyzes the Loop scheme and its extensions. She writes the characteristic map as a collection of triangular Bézier surfaces. For each patch she writes the Jacobian as a sixth degree triangular Bézier surface and asserts positivity of the coordinates of the control points, thereby showing regularity, due to the convex hull property. To prove injectivity she shows that the inner boundary curve of the characteristic map does not pass through the origin and that any half line beginning in the origin will intersect the curve exactly once. To do this the curve is written in Bézier form and the variation diminishing property[2] is applied. For the extended subdivision rules additional conditions are necessary. Schweitzer checks the conditions numerically for valencies up to 100.

The most general known method for proving regularity and injectivity of the characteristic map is the one developed by Zorin [34]. Unlike the other methods mentioned, his does not require an explicit formula for the limit surface on regular meshes. Also, for symmetric schemes it can be used to show C^1 continuity for all valencies. The method for testing regularity relies on using a linear approximation of the characteristic map together with convergence rate estimates to evaluate bounds for the minimum and maximum value of the Jacobian. The injectivity is checked by verifying that the image of the characteristic map is isolated from 0 and that the winding number of a linear approximation of the outer boundary curve is 1. The conditions are evaluated numerically using interval arithmetics to guard against roundoff error.

4.5 Example Eigenanalysis for the Loop Scheme

We can now apply the results from the previous sections to an example where we carry out the eigenanalysis for a smooth vertex in the Loop scheme. We show that the subdivision matrix fulfills the requirements for convergence in theorem 8 and we compute the tangent vectors at the limit point associated with the vertex. We use the sub-dominant eigenvectors of the subdivision matrix to construct caracteristic maps. At the end of this section we show how the tangent vectors near a crease can be computer using a 1-neighbourhood subdivision matrix.

[2]The variation diminishing property for a Bézier curve states that a line cannot have more intersections with the curve than it does with the control polygon of the curve.

4.5.1 Smooth Vertex

For analysis a 2-neighbourhood is needed. With the labelling shown in figure 4.5, the local subdivision matrix around a valency n smooth vertex can be written as

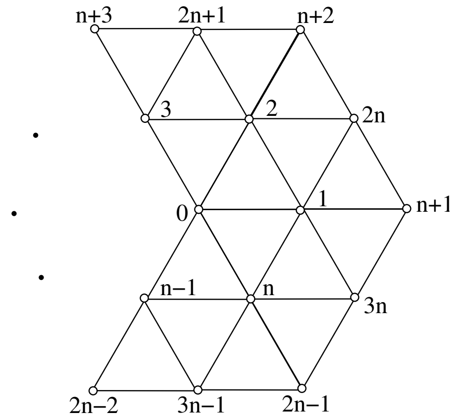


Figure 4.5: Labelling of a 2-neighbourhood of a smooth vertex in the Loop scheme.

$$S = \begin{bmatrix} S_{11} & 0 & 0 \\ S_{21} & S_{22} & S_{23} \\ S_{31} & 0 & S_{33} \end{bmatrix},$$

where

$$S_{11} = \begin{bmatrix} 1-n\beta & \beta & \beta & \beta & \beta & \cdots & \beta & \beta \\ 3/8 & 3/8 & 1/8 & 0 & 0 & \cdots & 0 & 1/8 \\ 3/8 & 1/8 & 3/8 & 1/8 & 0 & \cdots & 0 & 0 \\ 3/8 & 0 & 1/8 & 3/8 & 1/8 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 3/8 & 1/8 & 0 & 0 & 0 & \cdots & 1/8 & 3/8 \end{bmatrix},$$

$$S_{21} = \frac{1}{16}\begin{bmatrix} 1 & 10 & 1 & 0 & \cdots & 0 & 1 \\ 1 & 1 & 10 & 1 & \cdots & 0 & 0 \\ 1 & 0 & 1 & 10 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 1 & 0 & 0 & \cdots & 1 & 10 \end{bmatrix}, \quad S_{22} = \frac{1}{16} I_{n\times n},$$

$$S_{23} = \frac{1}{16}\begin{bmatrix} 1 & 0 & 0 & \cdots & 1 \\ 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}, \quad S_{31} = \frac{1}{8}\begin{bmatrix} 1 & 3 & 3 & 0 & \cdots & 0 \\ 1 & 0 & 3 & 3 & \cdots & 0 \\ 1 & 0 & 0 & 3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 3 & 0 & 0 & \cdots & 3 \end{bmatrix}, \quad S_{33} = \frac{1}{8} I_{n\times n},$$

and $\beta = (1/n)(5/8 - (3 + 2\cos(\frac{2\pi}{n}))^2/64)$.

To calculate the eigenstructure of the subdivision matrix, we first note that by permuting the rows and columns, S becomes block triangular. If we let

$$P = \begin{bmatrix} I_{n+1\times n+1} & 0 & 0 \\ 0 & 0 & I_{n\times n} \\ 0 & I_{n\times n} & 0 \end{bmatrix},$$

then

$$PSP^{-1} = \begin{bmatrix} S_{11} & 0 & 0 \\ S_{31} & S_{33} & 0 \\ S_{21} & S_{23} & S_{22} \end{bmatrix}.$$

Since a similar transform does not change the eigenvalues, S has the same eigenvalues as PSP^{-1}. We then immediately get the eigenvalues $\frac{1}{8}$ and $\frac{1}{16}$, both with algebraic multiplicity n, from S_{33} and S_{22}. The remaining $n+1$ eigenvalues are determined by S_{11}. The lower $n \times n$ submatrix of S_{11} is a circulant matrix, where each row is obtained by cyclicly shifting the previous row one position left. Because of the cyclic nature of S_{11} the Discrete Fourier Transformation is useful in finding the eigenvalues [13, 28]. Here we choose instead the method used by Schweitzer [26] based on characteristic polynomials.

We first rewrite S_{11} in block form

$$S_{11} = \begin{bmatrix} 1-n\beta & \beta & \cdots & \beta \\ 3/8 & & & \\ \vdots & & C_n & \\ 3/8 & & & \end{bmatrix},$$

where C_n is the $n \times n$ circulant matrix mentioned above. It can be shown [26] that the characteristic polynomials of C_n and S_{11} are related by the equation

$$\left(\frac{5}{8} - \lambda\right) R_{S_{11}}(\lambda) = (1-\lambda)\left(\frac{5-n\beta}{8} - \lambda\right) R_{C_n}(\lambda). \tag{4.12}$$

From the theory of circulant matrices [5] the eigenvalues of C_n are known to be

$$\gamma_j = \frac{3 + 2\cos\left(\frac{2\pi(j-1)}{n}\right)}{8}, \tag{4.13}$$

for $j = 1, \cdots, n$.

By insertering $j = 1$ in (4.13) it is seen that $\gamma_1 = \frac{5}{8}$ is an eigenvalue of C_n. When n is even $\gamma_{1+n/2} = 1/8$. The rest of the eigenvalues have multiplicity 2 because $\gamma_j = \gamma_{n-j+2}$, which is seen by substituting $n-j+2$ for j in (4.13)

$$\gamma_{n-j+2} = \frac{3 + 2\cos\left(\frac{2\pi(n-j+2-1)}{n}\right)}{8} = \frac{3 + 2\cos\left(2\pi - \frac{2\pi(j-1)}{n}\right)}{8}.$$

Since $\cos(2\pi - a) = \cos(a)$ we get $\gamma_j = \gamma_{n-j+2}$ as claimed, and therefore

$$\gamma_j = \gamma_{n-j+2} = \frac{3 + 2\cos\left(\frac{2\pi(j-1)}{n}\right)}{8}, \quad j = 2, \ldots, \left\lceil \frac{n}{2} \right\rceil.$$

The dominant eigenvalue of C_n is $\gamma_1 = 5/8$. The sub-dominant eigenvalues are $\gamma_2 = \gamma_n = (3 + 2\cos(\frac{2\pi}{n}))/8$.

We now find the eigenvalues of S_{11} from (4.12). We see that S_{11} inherits all the eigenvalues of C_n except for $5/8$. S_{11} has two additional eigenvalues: 1 and $(5-n\beta)/8 = \gamma_2^2$. Since $1 > 5/8 = \gamma_1 > \gamma_2$, $\lambda_1 = 1$ is the dominant eigenvalue. Also, $\gamma_2 < 1$ so $\gamma_2^2 < \gamma_2$, which leaves $\lambda_2 = \lambda_3 = \gamma_2$ as the sub-dominant eigenvalues.

We have now shown that Loop's subdivision scheme converges when working on a smooth vertex, because the eigenvalues of the subdivision matrix fulfills the requirements stated in theorem 8.

Now we will calculate the limit point and the tangent vectors at a smooth vertex. Recall that we need the left eigenvectors for this, and that we need only a 1-neighbourhood, i.e. we only have to find the eigenvectors for S_{11}. We also calculate the right eigenvectors because we need them later to contruct the characteristic map. By affine invariance of the subdivision matrix, the eigenvector corresponding to the eigenvalue $\lambda_1 = 1$ is $\boldsymbol{v}_1 = [1, \ldots, 1]$ (cf. lemma 7). We can verify that the corresponding left eigenvector is $\tilde{\boldsymbol{v}}_1 = [\frac{3}{8\beta}, 1, \ldots, 1]$

$$\tilde{\boldsymbol{v}}_1 S_{11} = \left[\frac{3(1-n\beta)}{8\beta} + \frac{3n}{8}, \frac{3}{8\beta}\beta + \frac{1+3+1}{8}, \ldots, \frac{3}{8\beta}\beta + \frac{1+3+1}{8}\right] = \left[\frac{3}{8\beta}, 1, \ldots, 1\right].$$

Similarly, we verify that the two sub-dominant eigenvalues can be chosen as $\boldsymbol{v}_2 = [0, c_1, c_2, \ldots, c_n]$ and $\boldsymbol{v}_3 = [0, s_1, s_2, \ldots, s_n]$, where $c_k = \cos\left(\frac{2\pi(k-1)}{n}\right)$ and $s_k = \sin\left(\frac{2\pi(k-1)}{n}\right)$. To do this we let $\omega = e^{2\pi}ni$ with $i = \sqrt{-1}$ and introduce the complex vector $\hat{\boldsymbol{v}} = [0, \omega^0, \omega^1, \ldots, \omega^{n-1}]^{\mathrm{T}} = \boldsymbol{v}_2 + i\boldsymbol{v}_3$. Then

$$S_{11}\hat{\boldsymbol{v}} = \left[\beta \sum_{i=1}^{n} \omega^1, \varphi_0, \varphi_1, \ldots, \varphi_{n-1}\right]^{\mathrm{T}}$$

where $\varphi_k = \frac{1}{8}\omega^{(k-1) \bmod n} + \frac{3}{8}\omega^k + \frac{1}{8}\omega^{(k+1) \bmod n}$, $k = 0, \ldots, n-1$. Firstly, the complex roots of unity sum to zero, so the first coordinate of the vector is zero. Noting that $\omega^{-1} = \omega^{n-1}$ and $\omega^n = \omega^0$ we can drop the modulus in φ_k and get

$$\begin{aligned}\varphi_k &= \frac{1}{8}\omega^{k-1} + \frac{3}{8}\omega^k + \frac{1}{8}\omega^{k+1} = \frac{1}{8}e^{i\frac{2\pi(k-1)}{n}} + \frac{3}{8}e^{i\frac{2\pi k}{n}} + \frac{1}{8}e^{i\frac{2\pi(k+1)}{n}} \\ &= \frac{1}{8}e^{i\frac{2\pi k}{n}}\left(e^{-i\frac{2\pi}{n}} + 3 + e^{i\frac{2\pi}{n}}\right).\end{aligned}$$

The real and imaginary parts are then

$$\begin{aligned}\Re(\varphi_k) &= \frac{1}{8}\cos\frac{2\pi k}{n}\left(\cos\frac{-2\pi}{n} + 3 + \cos\frac{2\pi}{n}\right) + \frac{1}{8}\sin\frac{2\pi k}{n}\left(\sin\frac{-2\pi}{n} + \sin\frac{2\pi}{n}\right) \\ &= \frac{1}{8}\cos\frac{2\pi k}{n}\left(3 + 2\cos\frac{2\pi}{n}\right), \\ \Im(\varphi_k) &= \frac{1}{8}\sin\frac{2\pi k}{n}\left(\cos\frac{-2\pi}{n} + 3 + \cos\frac{2\pi}{n}\right) + \frac{1}{8}\cos\frac{2\pi k}{n}\left(\sin\frac{-2\pi}{n} + \sin\frac{2\pi}{n}\right) \\ &= \frac{1}{8}\sin\frac{2\pi k}{n}\left(3 + 2\cos\frac{2\pi}{n}\right).\end{aligned}$$

Since $\lambda_2 = \frac{1}{8}\left(3 + 2\cos\frac{2\pi}{n}\right)$ we finally get

$$\begin{aligned}\Re(S_{11}\hat{\boldsymbol{v}}) &= [0, \lambda_2 c_1, \lambda_2 c_2, \ldots, \lambda_2 c_n]^{\mathrm{T}} = \lambda_2 \boldsymbol{v}_2, \\ \Im(S_{11}\hat{\boldsymbol{v}}) &= [0, \lambda_2 s_1, \lambda_2 s_2, \ldots, \lambda_2 s_n]^{\mathrm{T}} = \lambda_2 \boldsymbol{v}_3,\end{aligned}$$

showing that $\boldsymbol{v}_2$ and $\boldsymbol{v}_3$ are eigenvectors corresponding to the eigenvalueas $\lambda_2 = \lambda_3$ as claimed.

A similar argument will show that the two left eigenvectors are $\tilde{\boldsymbol{v}}_2 = \boldsymbol{v}_2$ and $\tilde{\boldsymbol{v}}_3 = \boldsymbol{v}_3$.

The tangent vectors of the limit surface can then be written as (cf. (4.8) and (4.9))

$$\vec{t}_1 = a_2 = \tilde{\boldsymbol{v}}_2 \cdot \boldsymbol{p}^0 = \sum_{i=1}^{n} \cos\left(\frac{2\pi(i-1)}{n}\right) p_i$$
$$\vec{t}_2 = a_3 = \tilde{\boldsymbol{v}}_2 \cdot \boldsymbol{p}^0 = \sum_{i=1}^{n} \sin\left(\frac{2\pi(i-1)}{n}\right) p_i.$$

This result is used in our implementation of the Loop scheme to calculate the exact normals in the case of a smooth vertex. The exact normals produce a smoother surface in fewer subdivision steps in comparison to the number of steps required when using interpolated normals. The tangent vectors for the remaining types of vertices were given in section 3.6.1.

As a final step we now find the two sub-dominant eigenvectors of the full 2-neighbourhood subdivision matrix. We use the permuted block triangular subdivision matrix and extend the right eigenvectors of the 1-neighbourhood subdivision matrix by solving the systems

$$\begin{bmatrix} S_{11} & 0 & 0 \\ S_{31} & S_{33} & 0 \\ S_{21} & S_{23} & S_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_2 \\ \boldsymbol{a} \\ \boldsymbol{b} \end{bmatrix} = \lambda_2 \begin{bmatrix} \boldsymbol{v}_2 \\ \boldsymbol{a} \\ \boldsymbol{b} \end{bmatrix}, \tag{4.14}$$

$$\begin{bmatrix} S_{11} & 0 & 0 \\ S_{31} & S_{33} & 0 \\ S_{21} & S_{23} & S_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_3 \\ \boldsymbol{c} \\ \boldsymbol{d} \end{bmatrix} = \lambda_3 \begin{bmatrix} \boldsymbol{v}_3 \\ \boldsymbol{c} \\ \boldsymbol{d} \end{bmatrix}. \tag{4.15}$$

for $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$ and $\boldsymbol{d}$. After swapping $\boldsymbol{a}$ and $\boldsymbol{b}$, and $\boldsymbol{c}$ and $\boldsymbol{d}$ we then have the right eigenvectors of the original subdivision matrix.

We skip the evaluation and just give the resulting two full sub-dominant right eigenvectors (see [26])

$$\boldsymbol{V}_2 = [0, c_1, c_2, \ldots, c_n, yc_1, yc_2, \ldots, yc_n, z(c_1+c_2), z(c_2+c_3), \ldots, z(c_n+c_1)],$$
$$\boldsymbol{V}_3 = [0, s_1, s_2, \ldots, s_n, ys_1, ys_2, \ldots, ys_n, z(s_1+s_2), z(s_2+s_3), \ldots, z(s_n+s_1)],$$

where

$$c_k = \cos\frac{2\pi k}{n},$$
$$s_k = \sin\frac{2\pi k}{n},$$
$$y = \frac{13 + 2\cos\frac{2\pi}{n}}{5 + 4\cos\frac{2\pi}{n}},$$
$$z = \frac{3}{2 + 2\cos\frac{2\pi}{n}}.$$

We can use the computed right eigenvectors corresponding to the two sub-dominant eigenvalues to draw representations of some characteristic maps for the Loop scheme.

Figure 4.6 shows three such maps. These maps were constructed by converting the box spline control points given by the eigenvectors into control points for triangular Bézier surfaces and then evaluating points on the surface using the generalized de Casteljau algorithm.

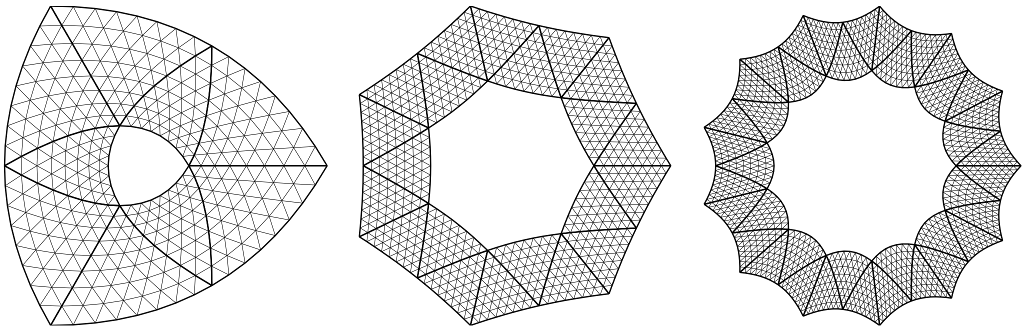


Figure 4.6: Characteristic maps for a smooth vertex in the Loop scheme. The valencies are 3,5 and 13.

Ordinary Crease Vertex

An ordinary crease vertex has valence 4. With the labelling in figure 4.7 the 1-neighbourhood subdivision matrix is given by

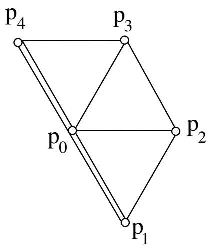


Figure 4.7: Labelling of a 1-neighbourhood of an ordinary crease vertex in the extended Loop scheme.

$$S_{\text{crease}} = \frac{1}{8}\begin{bmatrix} 6 & 1 & 0 & 0 & 1 \\ 4 & 4 & 0 & 0 & 0 \\ 3 & 1 & 3 & 1 & 0 \\ 3 & 0 & 1 & 3 & 1 \\ 4 & 0 & 0 & 0 & 4 \end{bmatrix}.$$

By finding the roots of the characteristic polynomial, we get the eigenvalues

$$\lambda_1 = 1, \quad \lambda_2 = \frac{1}{2}, \quad \lambda_3 = \frac{1}{2}, \quad \lambda_4 = \frac{1}{4}, \quad \lambda_5 = \frac{1}{4}.$$

The right eigenvectors corresponding to the two sub-dominant eigenvalues $\lambda_2 = \lambda_3 = \frac{1}{2}$ can be chosen to be

$$\boldsymbol{v}_2 = [0, 0, 1, 1, 0]^{\mathrm{T}}, \quad \boldsymbol{v}_3 = [0, 1, 1/2, -1/2, -1]^{\mathrm{T}}.$$

The left eigenvectors can be chosen as

$$\tilde{\boldsymbol{v}}_2 = [0, 1, 0, 0, -1], \quad \tilde{\boldsymbol{v}}_3 = [-1/2, -1/4, 1/2, 1/2, -1/2].$$

Near the central vertex, the tangent vector along the crease is then given by

$$\vec{t}_1 = p_1 - p_4,$$

while the one-sided cross tangent vector is

$$\vec{t}_2 = (p_2 + p_3)/2 - (p_1 + 2p_0 + p_4)/4.$$

Chapter 5

Implementation of Subdivision Surfaces

5.1 Data Structure

5.1.1 Introduction

One of the fundamental steps we had to cover before starting to implement any of the subdivision algorithms was to design a suitable data structure. A major challenge was to find a data structure that could handle the large and constantly changing data sets involved in subdivision. The chosen structure should besides representing the data in its native form, also show the connectivity of the mesh in question, while keeping storage at a minimum.

Specifically for our purpose it was important that the data structure could represent almost arbitrary meshes (as defined in the next paragraph) since we worked with differing types of subdivision schemes. The schemes operate on mesh types that consist primarily of either triangular or quadrilateral faces. It would be very impractical to have separate data structures for the different mesh types, so an important criterion is the support for almost arbitrary meshes.

We make a few restrictions on the mesh. Firstly, no edge can be shared by more than two faces. Secondly, the faces sharing a vertex can be ordered such that every two consecutive faces also share an edge. Thirdly, each face has an orientation and faces sharing an edge must have the same orientation. The last condition precludes meshes representing non-orientable surfaces, for example a Möbius band.

The connectivity of the mesh should be represented easily since this changes often during subdivision when new additional vertices are introduced and therefore a new connectivity structure has to be generated. A good implementation of connectivity also ensures easy traversal of the data structure. This is again important in our implementation since we recur over the mesh when subdividing.

We have chosen to focus less on other interesting and important areas of a good data structure, such as computational cost and memory consumption, the reason being that todays computers can rather effortlessly support large data sets and that the focus of this report is directed elsewhere. Nevertheless, our data structure still ensures that basic traversal of the mesh is reasonably efficient, but there is certainly room for improvement in other areas of our implementation.

5.1.2 The Lath

We were inspired by the data structure described in [15]. The main idea behind all the ideas in the article is a concept called the lath. This data item can elegantly encapsulate the structure of the mesh and it is supported by a fairly straightforward implementation. The lath is used to represent all the important elements of a mesh namely the vertices, edges, faces and the connectivity between them. Furthermore the lath supports the development of a basic set of traversal functions that are used during subdivision.

The paper describes several variations of the lath. The one we choose is the corner lath. This type of lath, like the others, is associated with exactly one vertex, one edge and one face of the mesh. Each lath contains a link to the vertex, one to the lath contained in the same face but in the counter-clockwise direction and one to the next lath in a counter-clockwise traversal of the associated vertex. A visualization of the corner lath data structure is shown in figure 5.1. The corner lath structure derives its name from the fact that the laths are visualized as begin placed in the corners of each face. The two things that must be stored are the laths and the vertices. The vertices contain the coordinates of the associated control points.

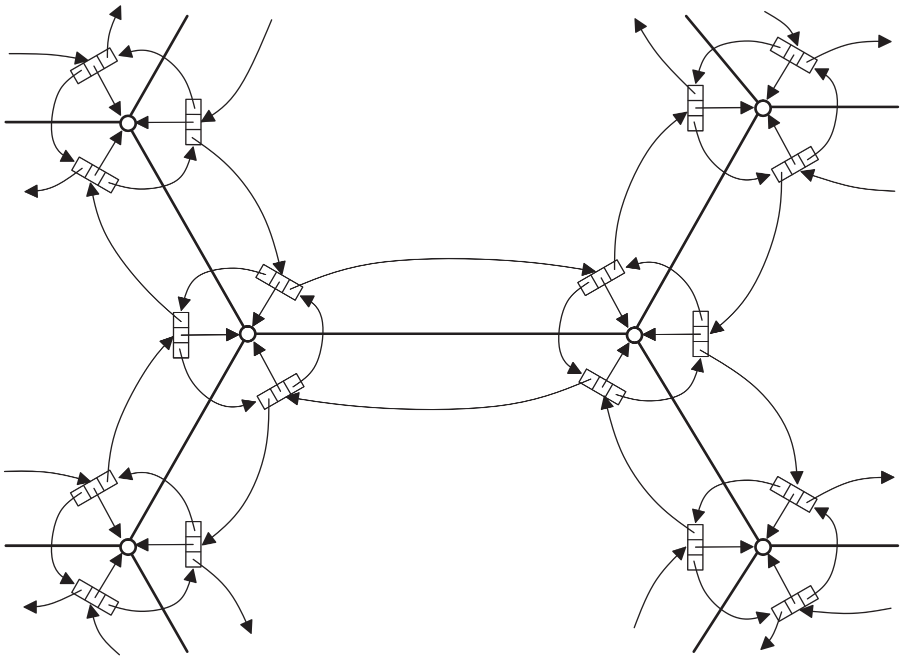


Figure 5.1: Corner lath data structure. The circles represent vertices, the bold lines represent edges, the rectangles divided into three squares represent the lath and the arrows represent pointers.

By focusing on a larger portion of the mesh (figure 5.2) two loops formed by the lath pointers are easily identified. The first loop allows a counter-clockwise traversal of the laths in a face and the second loop allows a counter-clockwise traversal of the laths around

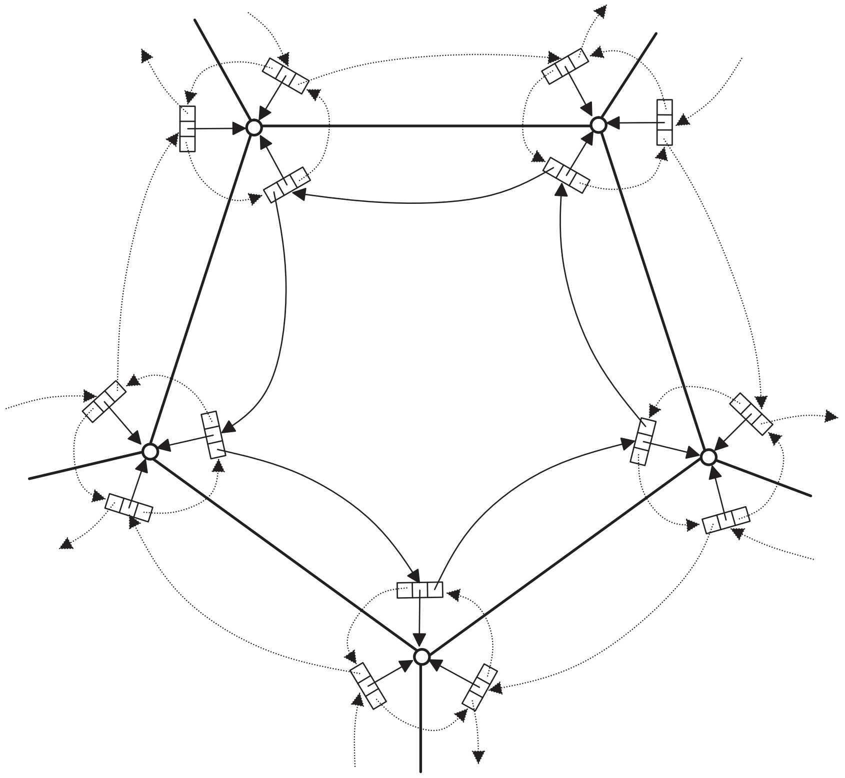


Figure 5.2: Corner lath data structure. The pointers form a loop around each face and around each vertex.

a vertex. Applying simple functions that use the lath pointers to navigate it is possible to traverse these loops. Given a lath L the most basic functions are,

L.ccwv() – returns the lath that follows L in a counter-clockwise traversal around the vertex that L points to.

L.ccwf() – returns the lath that follows L in a counter-clockwise traversal of the face that contains L.

L.cwv() – returns the lath that follows L in a clockwise traversal around the vertex that L points to. This can be computed either by using L.ccwv() until the lath L.ccwv() equals L, or by using the sequence L.ccwf().ccwv().ccwf(). Special consideration has to be taken when working on the boundary. See the paragraph below.

L.cwf() – returns the lath that follows L in a clockwise traversal around the face that contains L. This can be computed either by using L.ccwf() until the lath L.ccwf() equals L, or by using the sequence L.ccwv().ccwf().ccwv(). On the boundary the second option cannot be used.

On the boundary special care has to be taken when traversing around a vertex or in other words when switching from one face to another. The destination face might not exist so this has to be clearly identified in the data structure. By storing a Null pointer in the vertex link of a lath, and checking that no such one exists when traversing the data structure.

5.1.3 The Implementation

Our implementation of the corner lath data structure is represented by the class SSCornerLath shown in listing A.9 on page 86. The basic functions discussed above are implemented just as described as member functions of the class. The following will outline as well as underline certain details that either differ from the discussion above or are special measures we have taken in order to implement our data structure.

A very important but very simple extension to our implementation of the corner lath structure is needed to connect the subdivision levels that arise during subdivision. The reason being that reversal from and to different levels is needed during subdivision. Each lath therefore has a pointer to a lath in the next subdivided level which is accessed by the function

L.nextlevel() - returns the lath in the next subdivided level.

To support the non-uniform subdivision schemes discussed in sections 3.6–3.8 the lath contains member variables to store two knot spacings for the vertex it points to. One knot spacing is associated with the edge between the vertex and its counter-clockwise neighbour in the face. The other knot spacing is associated with the edge between the vertex and its clockwise neighbour.

To facilitate connecting faces we introduce a numbering of the edges in a face relative to a given lath. The edges are numbered consecutively from 0 in counter-clockwise order, starting with the edge crossed by the counter-clockwise vertex pointer from the given lath, see figure 5.2.

Besides the above mentioned extensions the data structure implementation is fairly straightforward and is easily understood by studying the source code in appendix A. Consult the overview on page 73 for a brief description of the source code files.

5.2 Implementation of Subdivision Schemes

This section will describe the algorithms we have implemented in order to support each of the schemes. The discussion will not concentrate on the actual code but instead it will describe the main ideas behind each of the implementations. In the code each of the schemes are contained in their own classes that all inherit from an abstract base class called SSSubdivisionMethods. This makes it possible to switch methods at run time because it avoids type binding at compile time.

5.2.1 Loop – class SSLoop

Subdivision

The Loop scheme works on triangular meshes so the first assumption for the implementation of the scheme is that it is applied to meshes consisting entirely of triangles. If this assumption is not satisfied for a model, it must be triangulated before subdividing.

In the following we talk about a point being marked as a cusp for example. Strictly speaking, what we mean is that a vertex in the mesh has a marking that makes the limit point associated with the vertex a cusp point.

The implemented scheme extends Loop's original scheme [16] with facilities for creating crease edges, cusp, conical, corner and dart points as discussed in section 3.6. These features are marked in the data structure as special knot spacings. Recall that each edge has two knot spacings associated with it – one in each end of the edge. Table 5.1 lists the available knot spacing values and their interpretation. If a point has conflicting markings, e.g. it is marked as both a conical and a cusp point, the rule used is determined by the order of precedence: corner, conical, cusp, dart/ordinary.

Knot spacing value	*Interpretation*
0	Crease edge. The two knot spacings on the edge should have the same value. A point is a dart if exactly one edge emanating from its corresponding vertex is marked as a crease. The dart vertex rule is identical to the ordinary vertex rule.
1	Ordinary edge. The two knot spacings on the edge should have the same value.
2	Conical point. A point is conical if at least one edge emanating from its corresponding vertex is marked as conical.
3	Cusp point. A point is a cusp if at least one edge emanating from its corresponding vertex is marked as a cusp.
4	Corner point. A point is a corner if at least one edge emanating from its corresponding vertex is marked as a corner, or if three or more creases are incident upon a vertex. The edge rule will treat edges marked as corners as though they were marked as creases.

Table 5.1: Special knot spacing values for marking features for the Loop scheme.

The Loop subdivision process is implemented in the same recursive fashion as the other schemes. The recursion traverses vertices in the mesh, building the subdivided 1-neighborhood of each vertex, and connecting these at the end of the recursion.

The order of events when processing a single vertex is outlined below.

1. If the vertex has already been processed, nothing is done.
2. It is checked whether the vertex is on the boundary of the mesh. If so, pointers to the clockwise-most and counter-clockwise-most laths are saved and the regular valence for the vertex is recorded as 4.
3. The control points of the neighbor vertices are collected and the knot spacings are inspected to determine the type of vertex rule to use.
4. The new vertex point is calculated by selecting the appropriate vertex rule based on the findings in the previous step.
   (a) If the vertex is marked as a corner or more than two crease edges are incident upon the vertex, the corner vertex rule is used.

   (b) If exactly two crease edges are incident upon the vertex, the crease vertex rule is used.
   (c) If the vertex is marked as conical, the conical vertex rule is used.
   (d) If the vertex is marked as a cusp, the cusp vertex rule is used.
   (e) If none of the above conditions apply, the vertex is either a dart or an ordinary vertex, and the ordinary vertex rule is used.

5. A new point is calculated for each edge emanating from the vertex. The vertex at the opposite end of the edge is inspected to finds its valence and check whether it is a corner. The edge rule is selected based on the findings.
   (a) If the edge is smooth or the start vertex is conical, a cusp or a dart, or if the opposite vertex is a dart, the smooth edge rule is used.
   (b) If the edge is marked as a crease and if either the start vertex or the opposite vertex, but not both, is extraordinary, the special crease edge rule is used. Here, a vertex is extraordinary if it has a valence other than 6 in the interior of the mesh or other than 4 on the boundary, or if the vertex is a corner.
   (c) If the edge is marked as a crease and if the start vertex and the opposite vertex are both either regular or extraordinary, the ordinary crease edge rule is used.
6. The triangles comprising the 1-neighborhood of the start vertex on the subdivided level are constructed and connected. The knot spacings on the original edges are inherited by the corresponding edges on the subdivided level.
7. The subdivision procedure is applied recursively to all the neighboring vertices.
8. The 1-neighborhood triangle patches are connected by filling in the remaining triangles. These are the triangles at the center of the original triangles.

Normals

When Loop surfaces with creases and special points are visualized, the use of interpolated polygonal normals is unsatisfactory since this will tend to smooth out these feature. To get the correct visual appearance the exact normals of the limit surface are used instead. The formulas for calculating the exact normals were shown in section 3.6.1.

5.2.2 Doo-Sabin – class SSDooSabin

Subdivision

As mentioned in section 3.4 the Doo-Sabin scheme is relatively simple to use and implement. There is essentially only one mask which is applied to all faces in the control mesh. The subdivision process is illustrated in figure 5.3 and includes the following steps.

1. If the vertex has already been processed, nothing is done.
2. The first step in the refinement calculates the new face vertices. In general it takes each vertex of each face in the mesh and applies the mask shown in 3.2 in section 3.4 to calculate the new vertex. See figure 5.3(a).

3. Then the generated vertices are connected to form the new faces inside the old faces. See figure 5.3(b).

4. Connections from the coarser mesh to the newly created laths in the next level are made.

5. The subdivision procedure is applied recursively to all the neighbouring faces.

6. New faces are constructed for each edge in the old face by connecting the four new vertices adjacent to an old edge. These new faces are connected to the faces constructed in step 3. See figure 5.3(c).

7. The final step is to construct new faces for each vertex in the old mesh by connecting the new vertices adjacent to each old vertex. These new faces are connected to the faces constructed in steps 3 and 6. See figure 5.3(d).

Normals

We did not implement the exact normals of the limit surface as in the Loop scheme. Instead we use interpolated polygonal normals on each vertex.

5.2.3 Catmull-Clark – class SSCatmullClark

Subdivision

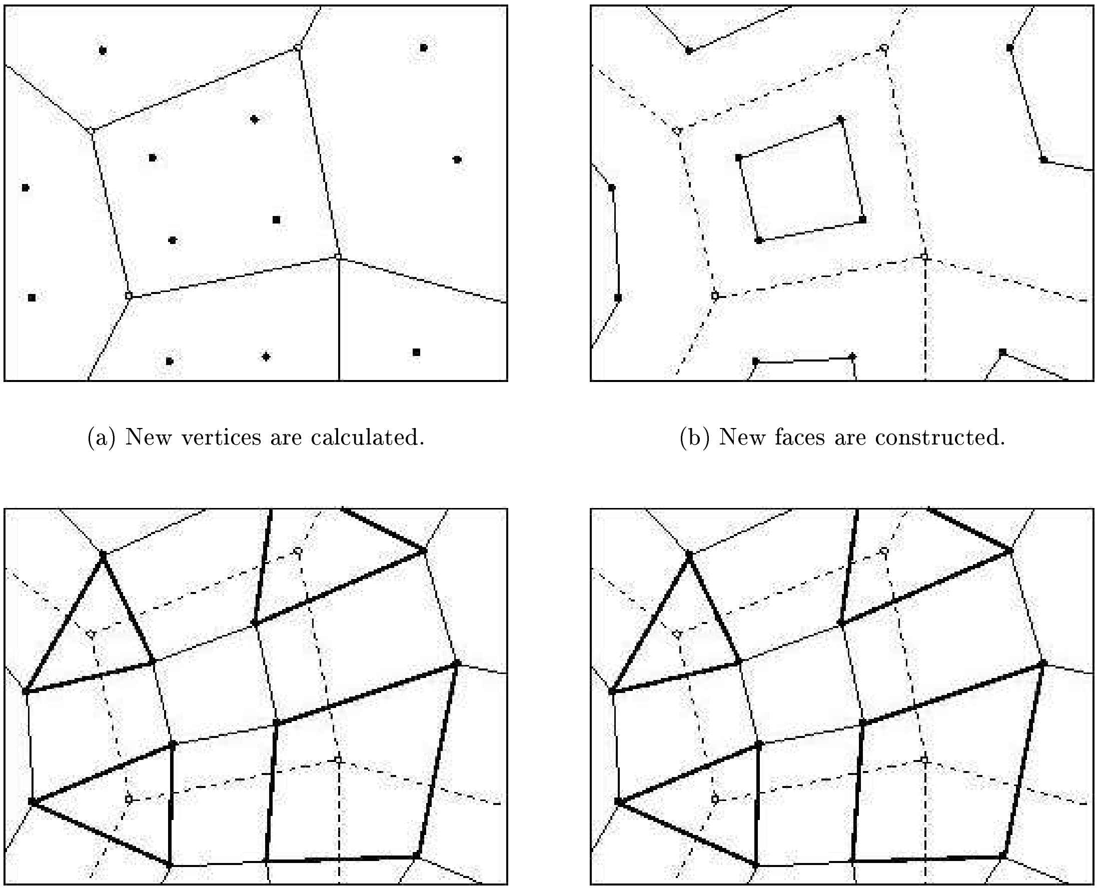
The procedure for subdividing a mesh with the Catmull-Clark masks is a little more complicated than the case of Doo-Sabin. This is because a 2-neighbourhood is necessary to calculate the new points. There are three types of new points: face points, edge points and vertex points. The newly generated points then have to be connected in a proper way. The following shows the order of events in the Catmull-Clark scheme and the accompanying figures help to clarify each step. We work on the mesh with irregular faces shown in figure 5.4(a).

1. If the vertex has already been processed, nothing is done.

2. The new face points are calculated in this step. That is we circle our vertex and generate a new face point for each face containing the vertex. The face point is the average of all the original points defining the face. If we hit the boundary and can not circle any further we start the recursive subdivsion here on the next counter clockwise vertex. See figure 5.4(b).

3. The new edge points are calculated by averaging the end points of the edge and the newly computed face points in the two faces sharing the edge. The end points are placed on level j and the newly created face points on level $j+1$ so we are using two levels. See figure 5.4(c).

4. The new center point can be calculated in different ways but we use the formula $\frac{F}{n} + \frac{2E}{n} + \frac{(n-3)V}{n}$, where F is the average of the new face points of all faces adjacent to the old vertex V, E is the average of the midpoints of all edges incident upon the old vertex V, and n is the valency of V. See figure 5.4(d).



(a) New vertices are calculated.

(b) New faces are constructed.

(c) The faces are connected across edges.

(d) The faces around vertices are connected.

Figure 5.3: Procedure for Doo-Sabin subdivision.

5. The faces surrounding the newly created center vertex are created by using the new edge and face vertices. Each face consists of the new center vertex, the two new adjacent edge vertices and the face vertex created in each face.

6. The subdivision procedure is applied recursively to all the neighboring vertices.

7. The new faces around each vertex are connected to each other as a final step. The result of the scheme is seen in figure 5.4(e). It should be noted that the refined mesh only contains faces with four edges regardless of the structure of the original mesh.

Normals

We did not implement the exact normals of the limit surface as in the Loop scheme. Instead we use interpolated polygonal normals on each vertex.

5.2.4 Quadratic NURSS – class SSQuadraticNURSS

The subdivision procedure for the quadratic NURSS scheme closely follows the procedure for the Doo-Sabin scheme from which it is derived. The differences are that the rule for calculating new points is modified to include knot spacings, and that new knot spacings must be calculated for the edges in the subdivided mesh. The modified rule and new knot spacings were described in section 3.7.

5.2.5 Cubic NURSS – class SSCubicNURSS

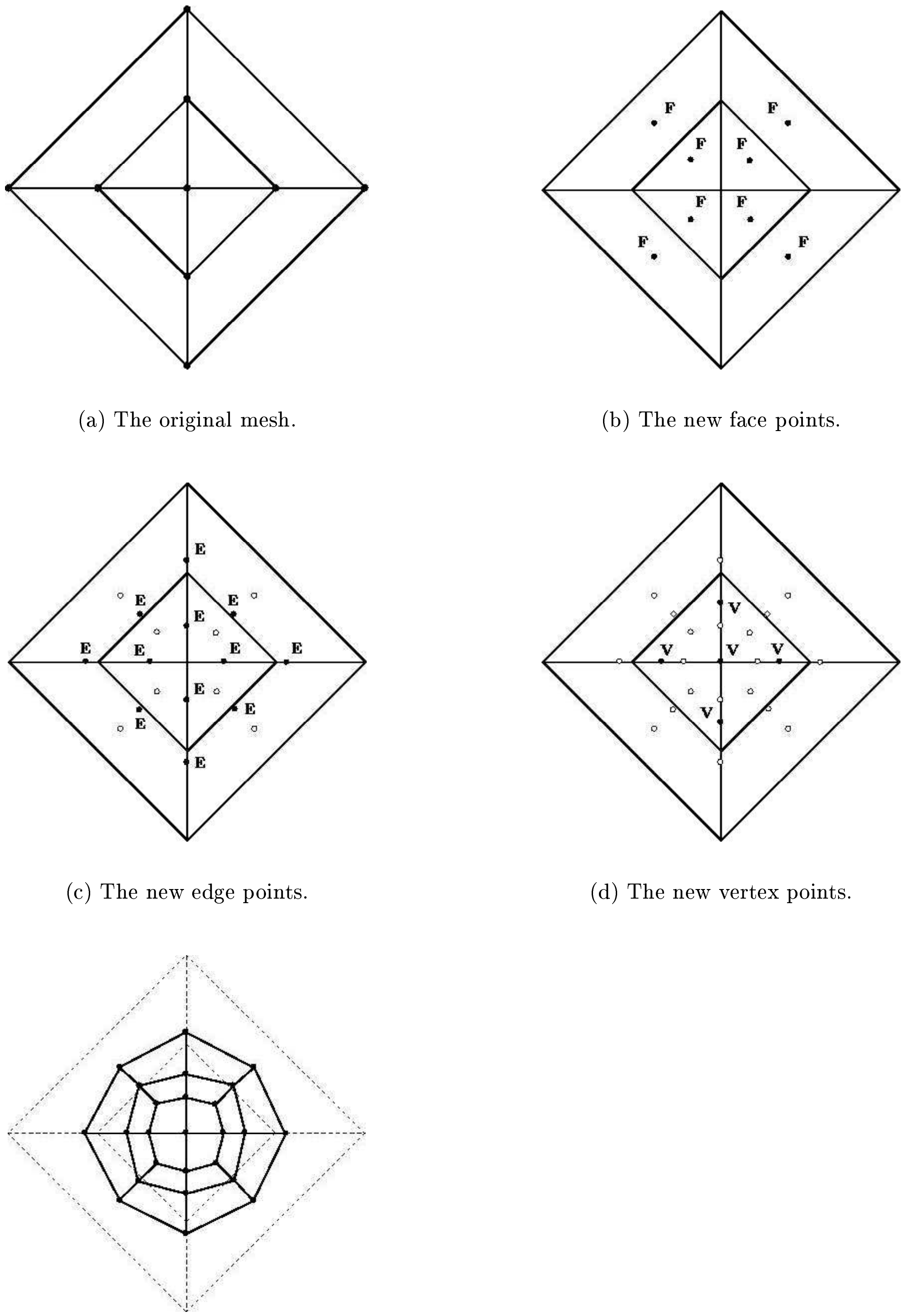
The cubic NURSS subdivision procedure is very similar to the subdivision procedure for Catmull-Clark surfaces. Again, the only differences are the modified face, edge and vertex rules, and the calculation of the new knot spacings. The rules and new knot spacings for the cubic NURSS scheme were discussed in section 3.8.

5.3 Overview of the remaining implementation

The above described areas are by far the most critical in the our implementation because they form the foundation. But in order to do the subdivision a lot of work has to be done to develop support functions that perform the legwork of each subdivision scheme. These functions and the classes containing them will be discussed in this section. This section does not discuss every area of our implementation but only focuses on the most interesting, seen from a technical point of view and on the most frequently used features. It should be noted that our project has been developed in C++ and takes full advantage of the object oriented support present in that language. The graphical rendering is done with OpenGL and therefore our program is platform independent – that is we have used it on a Win32 platform as well as a Linux based platform.

5.3.1 The Readers

As part of the program we developed a file format that describes the structure of a polygonal control mesh. The format describes the vertices and their associated coordinates, the faces,



(a) The original mesh.

(b) The new face points.

(c) The new edge points.

(d) The new vertex points.

(e) The subdivided mesh.

Figure 5.4: Procedure for Catmull-Clark subdivision.

and the connections between the faces. These three types of information are divided into sections in the file, with appropriate section headers. The headers are used in the reader to keep track of what the current section is and when a new section is encountered and thus new information is to be read. In addition it is possible to specify knot spacings. The syntax of the MDL file format is shown in appendix B. The file format is read by the class SSMdlReader and it made it a lot easier to design different models and also to switch between different models.

We also developed a reader for a subset of the more complicated but widely used format VRML2. This is exported by most 3D design programs and we specifically used this format to read a complicated 3D model designed by IOI Software. Once again the format specifies the three necessary pieces of information as mentioned above seperated into sections. The reader is implemented in the class SSHeroReader.

5.3.2 Support for the Model

In the class SSQModel we have placed the code that supports the model. The model is specified by a polygonal control mesh represented as a number of laths that define the structure of the mesh as described in 5.1.2. The class therefore has to account for the laths in an effective manner that enables easy application of the subdivision shemes on the mesh. This is done by storing the laths that make up a level in the subdivision algorithm in a vector and in turn storing the levels in a data strucutre called a map. A map is similar to a hash table where a key is associated to an element – in this case the key is the level number that returns access to the laths on that particular level.

The class also implements a number of methods that manipulate the model via the laths that comprise it. The methods are called from the various subdivision algorithms whenever changes are made to a mesh during subdvision. Two of the most important methods are

ConnectFaces takes two laths, belonging to different faces and connects these faces across the edges whose numbers are supplied as parameters. A face has a predefined numbering of the edges making it possible to apply this method correctly. The numbering was described in section 5.1.3.

BuildFace makes new laths corresponding to the vertices that are supplied as parameters. The laths are connected in a counter-clockwise manner to form a face.

SSQModel also controls the current subdivision method that is applied on the model. The current method can be switched at run time to a new one by calling the method SwitchMehtod and the subdivision is invoked by calling Subdivide.

The final method of some interest is Triangulate that triangulates each face in a quadrilateral mesh so for example the Loop scheme can applied without any probles.

5.3.3 OpenGL Implemenation

We have used OpenGL to render the models on the screen. Most of the OpenGL specific code is placed in a file called viewer.cpp, which is shown in listing A.32 on page 153. It contains code that initializes OpenGL, sets up the proper projections, ligthing, material properties etc. There is some additional OpenGL code in SSQModel inside the function for drawing each face.

Chapter 6

Example Subdivision Surfaces

This chapter presents an empirical comparison of properties of subdivision schemes by examining images of example surfaces rendered using the viewer program.

6.1 Comparison of the Doo-Sabin, Catmull-Clark and Loop Schemes

Figure 6.1 shows surfaces generated by the Doo-Sabin, Catmull-Clark and Loop schemes from a common control mesh in the shape of a tetrahedron. Notice how much the mesh shrinks under subdivision, particularly for the Loop and Catmull-Clark surfaces. Another thing to notice is that the shape of the control polyhedron can be recognized in the Doo-Sabin surface and to a lesser degree in the Loop surface, while the Catmull-Clark surface bears no such resemblance.

The Catmull-Clark and Loop schemes use the same type of refinement rule, namely face split, while Doo-Sabin uses the vertex split rule. The difference between the rules can be seen in figures 6.2–6.4, which show two refinements of an initial control polyhedron. The limit surfaces are shown in figure 6.5.

In order to use the quadrilateral control mesh for the Loop method, the mesh has to be triangulated. The triangulation strategy employed is very simple: each face is split into as many triangles as there are sides in the face by connecting every pair of consecutive vertices to a new vertex at the centroid of the face. This simple strategy has a number of implications; firstly, it raises the valency of the original vertices, possibly making them irregular; secondly, the new vertices inserted will be irregular unless the corresponding face is a hexagon; and, thirdly, the triangles produced tend to be flat, i.e. to have two very small angles and one large angle. Ideally, the triangulation should not increase the number of irregular features and all triangles should have three approximately equal angles.

The extra control points inserted by the triangulation explain why the Loop surface exhibits less shrinkage than the Catmull-Clark surface in figure 6.5, contrary to the situation in figure 6.1.

The flat triangles can cause visual artifacts in the limit surface. Figure 6.6 shows a Loop surface, where depressions appear along the sides of the model and meet at the joins. The Catmull-Clark surface does not have such depressions. To explain this phenomenon, look at the control polyhedron in figure 6.6. First note that all the coloured control points lie in a plane. After one subdivision the new point corresponding to the green point will

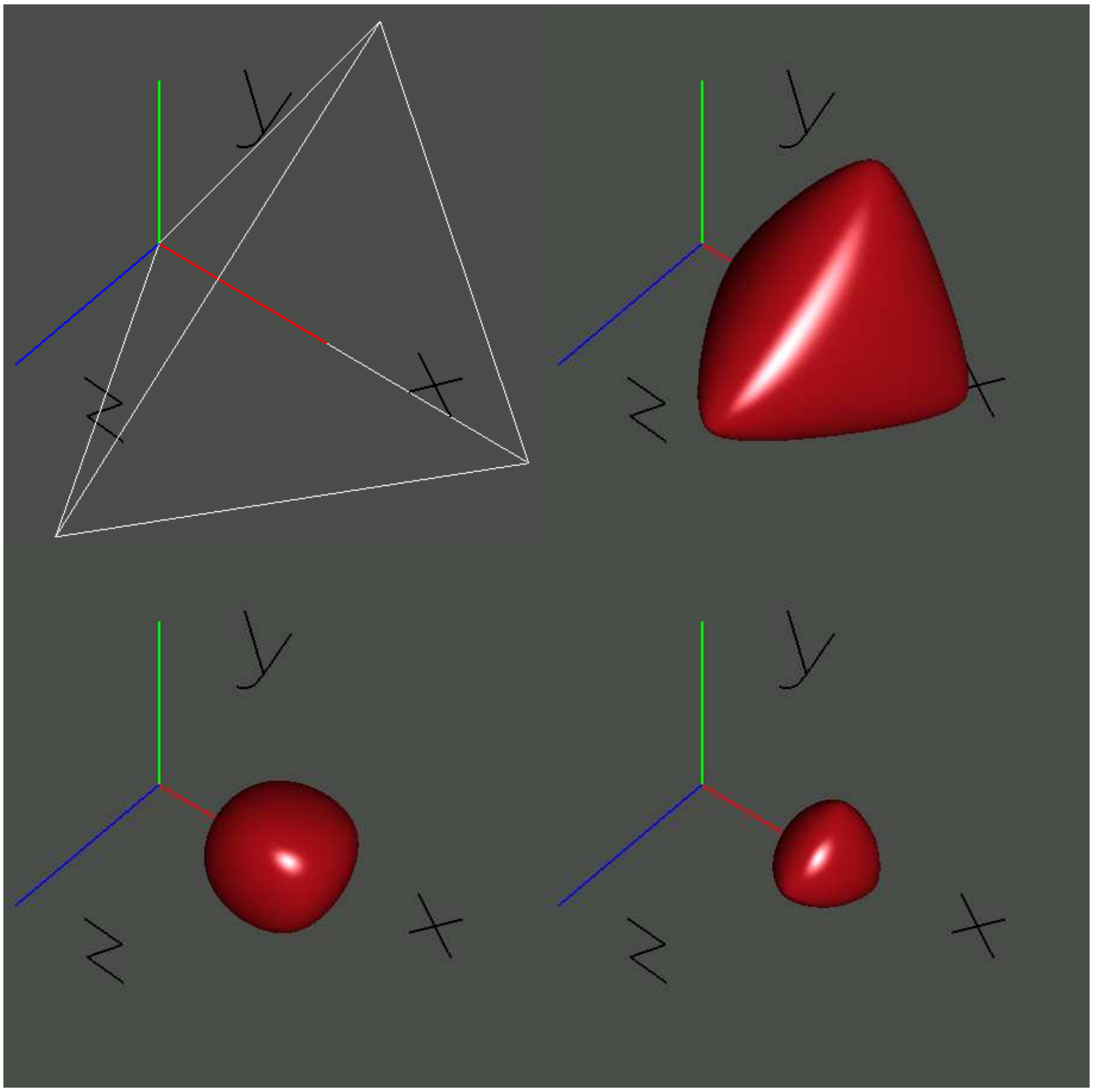


Figure 6.1: Subdivision surfaces with the same initial control mesh. Top left: Initial control mesh. Top right: Doo-Sabin surface. Bottom left: Catmull-Clark surface. Bottom right: Loop surface.

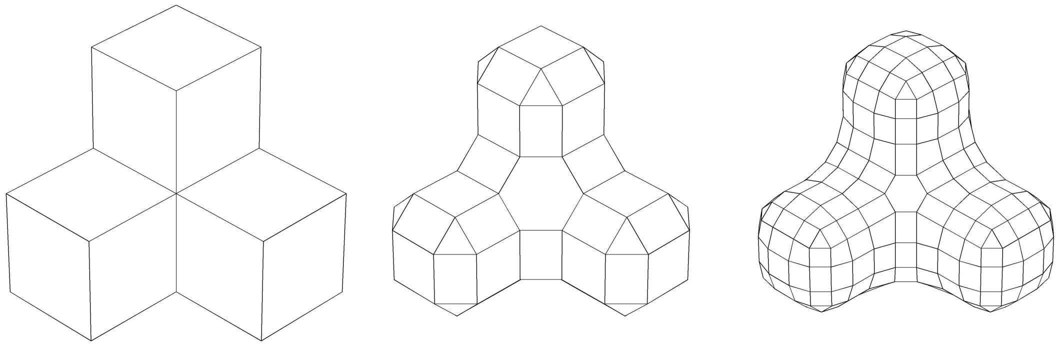


Figure 6.2: Two steps of Doo-Sabin subdivision.

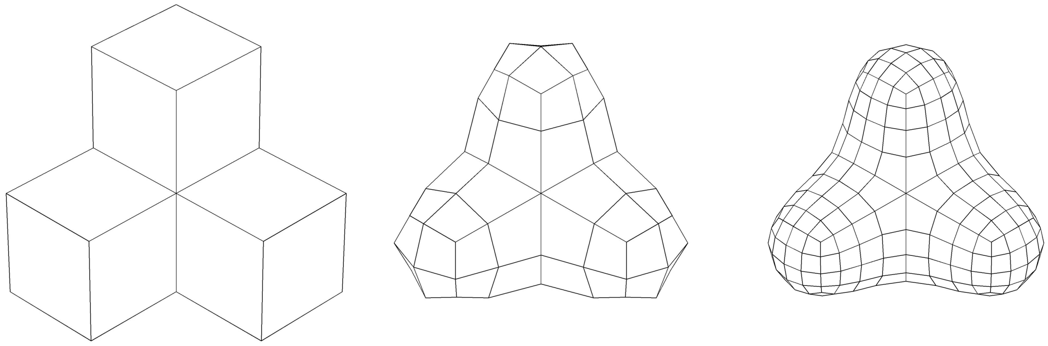


Figure 6.3: Two steps of Catmull-Clark subdivision.

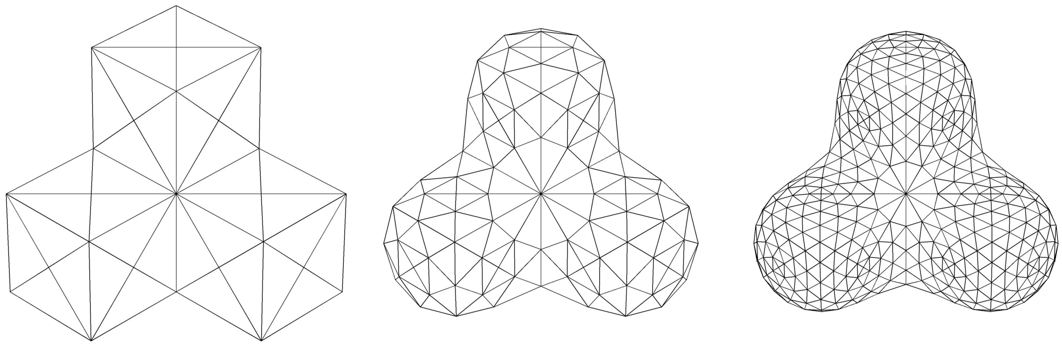


Figure 6.4: Two steps of Loop subdivision after triangulation of the initial control mesh.

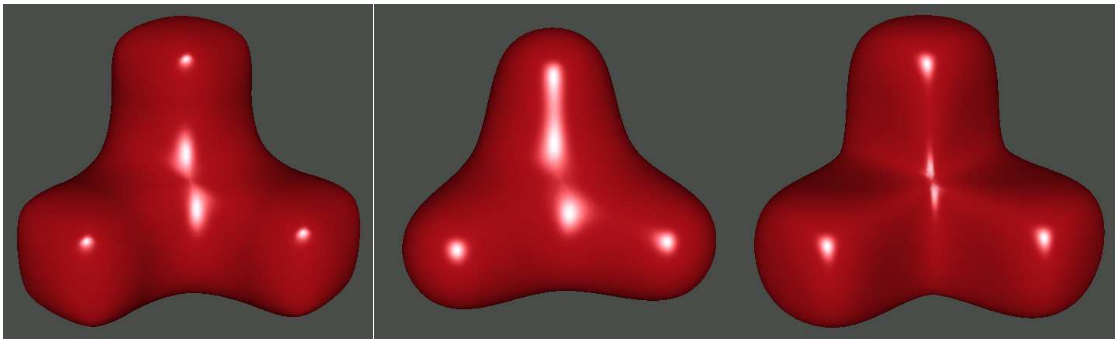


Figure 6.5: The limit surface of the control meshes in Fig. 6.2–6.4. Left: Doo-Sabin. Center: Catmull-Clark. Right: Loop.

also lie in this plane, and so will the control points inserted on the edges between the green and blue points. The valency of the vertices associated with the blue points is 12. The coefficients for the vertex subdivision rule are then

$$\beta = \frac{1}{12}\left(\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{12}\right)^2\right) = \frac{28 - 6\sqrt{3}}{768} \approx 0.023,$$
$$\alpha = 1 - 12\beta \approx 0.72.$$

Although the contribution of the individual neighbouring control points is small because of the small β value, the new points corresponding to the blue points will move out of the plane and closer to the center of the tetrahedron. The new point inserted on the edge between the blue points will also move closer to the center of the tetrahedron. The result is that a small depression is formed along the edge between the blue points and this is maintained by subsequent subdivisions. The problem is that the blue points contract too much. The situation could be improved by a trangulation strategy that inserted a new vertex on the edge between the blue points, since this would cause the blue points to contract less.

Figure 6.7 is an example of applying a method for quadrilateral meshes to a triangular mesh. Ripples form on the Doo-Sabin surface and it is visibly less smooth than the Loop surface. Clearly, it is not desireable for a surface to reveal the connectivity of the control mesh in this fashion. The lesson to be learnt here is that the number of irregular features in the control mesh should be kept to an absolute minimum. Note that the topological genus of the cup is 1, so it can actually be represented by a control mesh without irregular features at all.

Some insight into the performance of the different subdivision schemes can be gained by examining the number of polygons computed per time unit by the viewer program. The absolute values are naturally highly dependent on the specifications of the computer used. On a 600 MHz Pentium III system the number of polygons produced per second is about 31000 for Doo-Sabin, 25000 for Catmull-Clark and 42000 for Loop. In this measurement quadrilateral meshes were used for the two former schemes and a triangular mesh was used for the latter scheme. When the program is compiled with compiler optimization these figures are almost doubled. The reason for the comparatively high performance of the Loop

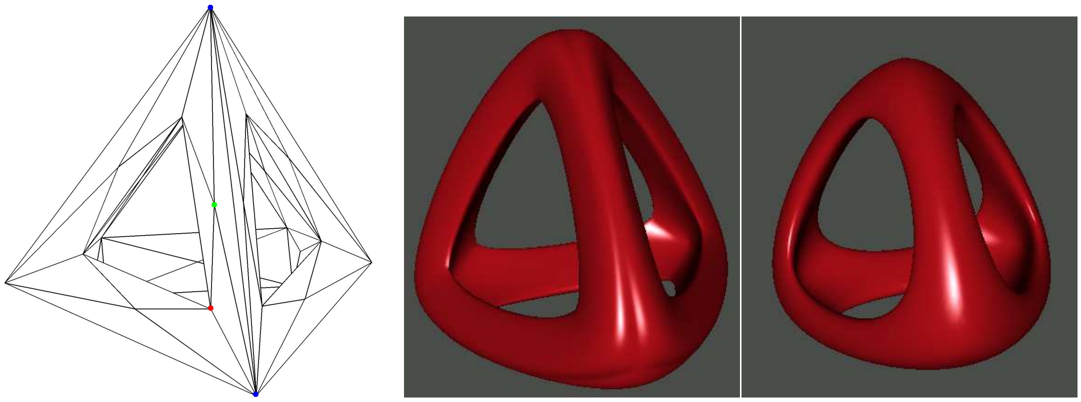


Figure 6.6: A hollow tetrahedron. Left: Control polyhedron for Loop surface. Left: Loop surface. Right: Catmull-Clark surface.

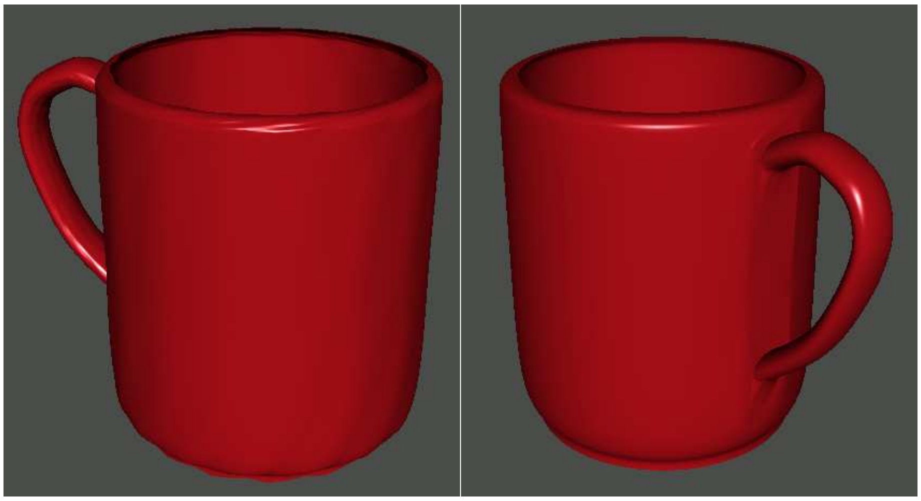


Figure 6.7: Two views of a cup. Left: Doo-Sabin surface. Right: Loop surface.

scheme is that constructing triangles is faster than constructing quadrilaterals in our data structure. Also, in the construction of the subdivided control mesh triangles generally have to be connected to three neighbours, while quadrilaterals have four neighbours. Finally, for a given number of vertices a triangular mesh will have twice as many polygons as a quadrilateral mesh. All things considered the Loop scheme is actually slightly slower than the other schemes. The superiority of the Doo-Sabin scheme over the Catmull-Clark scheme is explained by the smaller masks in the former scheme.

It should be stressed that the focus of the current implementation is correctness rather then speed. A fully optimized implementation will yield substantially higher polygon frequencies.

It is not possible to appoint the best subdivision scheme based on the evidence given in this section. For a given application the choice of scheme must depend on the type of control mesh available, i.e. triangular or quadrilateral, and requirements for smoothness and performance. For sheer performance the Doo-Sabin scheme is the best choice. In some applications the C^2 continuity of Catmull-Clark and Loop on regular meshes may be the deciding factor.

6.2 Comparison of the Extended Loop and NURSS Schemes

For many applications in computer graphics it is necessary to have more control over the surface than is offered by the uniform schemes. The most useful feature is to create creases and corners. It is desireable to have semi-sharp creases whose sharpness can be adjusted. One approach to modelling such features is to make the control mesh denser around the features. To produce a sharp crease some of the faces in the control mesh should be collapsed to lines. From an implementation point of view this solution is not satisfactory, since it increases the amount of memory needed to store the control mesh and the time spent on subdividing the mesh. The extended Loop scheme and the NURSS schemes propose two different solutions.

The NURSS schemes offer fine-grained control over the limit surface and allow sharp as well as semi-sharp creases. Unfortunately, choosing the knot spacings to achieve a particular effect is rather difficult and requires some experimentation. Conversely, the extended Loop scheme has a very intuitive way of introducing creases and similar features, however, it is less flexible, since, for example, it cannot create semi-sharp features. This could be remedied by introducing additional subdivision rules and extending the tagging scheme for the data structure. Obviously, introducing more rules makes the implementation more complicated and less efficient. Pixar Animation Studios use a variant of the Catmull-Clark scheme extended with rules for creases [6], in a manner similar to the extended Loop scheme. To create semi-sharp creases they first apply the sharp crease rule a number of times and then apply an interpolation of the sharp and smooth rule. This scheme allows absolute control over the sharpness.

Figure 6.8 shows a quadratic NURSS surface of topological genus 2 where some of the knot spacings have been set to zero to produce a sharp crease around the hole on the right. The curvature is visibly reduced on the part of the surface next to the crease.

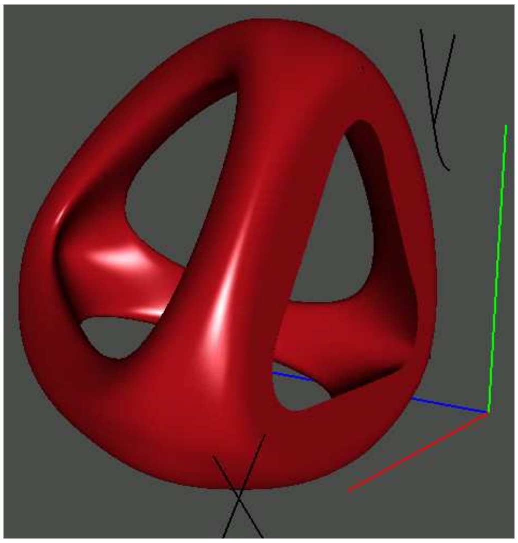


Figure 6.8: Quadratic NURSS surface. The hole to the right has a sharp crease, while the the remaining holes are similar to those of a Doo-Sabin surface.

Figures 6.9 and 6.10 show examples of the flexibility of the NURSS schemes. By varying the knot spacings a wide range of surfaces can be obtained from a common control polyhedron. Figure 6.11 shows a collection of Loop surfaces with the same initial control polyhedron but different markings to produce creases and conical, corner and cusp points.

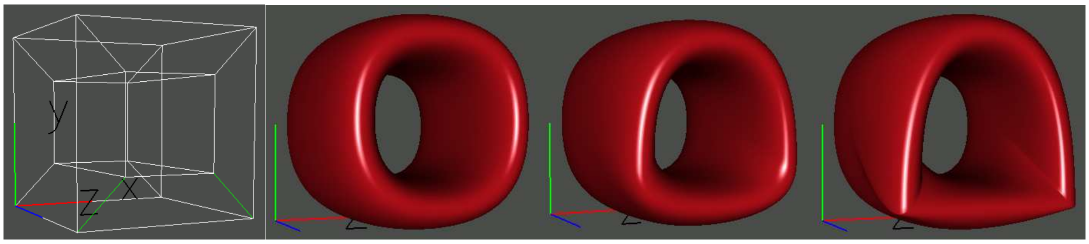


Figure 6.9: Quadratic NURSS surfaces. The control polygon is shown on the left. The knot spacings on the white edges are 1. The knot spacings on the two green edges vary. From left to right they are 1, 3 and 50.

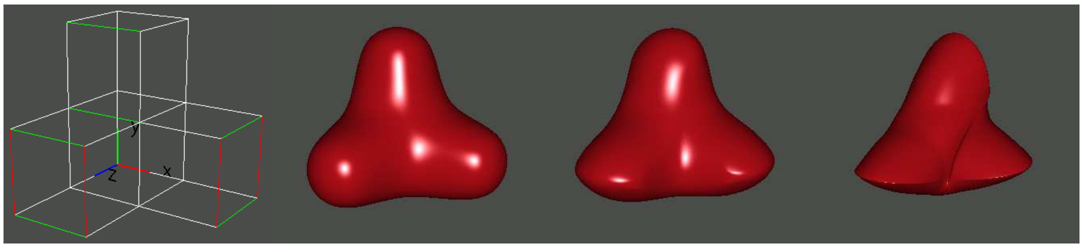


Figure 6.10: Cubic NURSS surfaces. The control polygon is shown on the left. The knot spacings on the white edges are 1. The knot spacings on the green and red edges vary. From left to right they are 1, 3 and 50 for the green and 1, 0.3 and 0.02 for the red.

For the extended Loop scheme we have implemented calculation of the exact normals of the limit surface. Figure 6.12 illustrates the difference between using interpolated normals and exact normals. With interpolated normals the sharp creases are smoothed, and although in some situations that may give the effect we are looking for, it should also be possible to maintain the sharpness. By further subdivision the effect of the smoothing can be reduced but never entirely eliminated. With exact normals the sharpness of the crease is accentuated. Additionally, the exact normals make the approximation to the surface look better than it actually is, so it may be possible to save one subdivision.

In the current unoptimized implementation the polygon frequency of the quadratic NURSS scheme is roughly 20000 and for the cubic NURSS scheme it is about 13000. This means the Doo-Sabin scheme is about 50% faster than the quadratic NURSS scheme, while the Catmull-Clark scheme is about twice as fast as the cubic NURSS scheme. Since the overall structure of the program code for the Doo-Sabin scheme is identical to the structure of the quadratic NURSS code, and similarly for the Catmull-Clark and cubic NURSS schemes, it shows that the evaluation of the subdivision coefficients for the NURSS schemes is computationally fairly expensive. A possible improvement would be to check

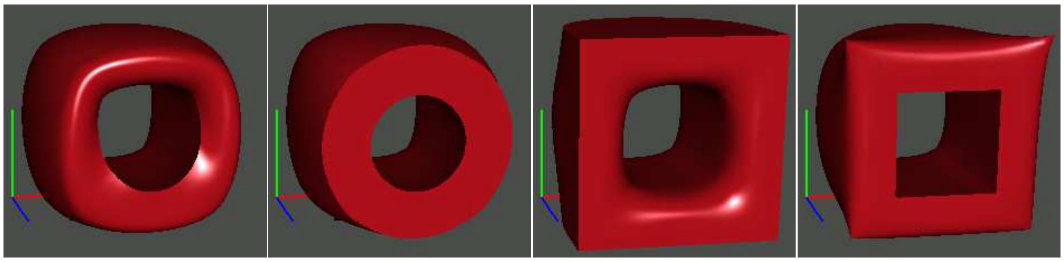


Figure 6.11: Extended Loop surfaces. The control polyhedron is the same as in figure 6.9. These surfaces were obtained by marking some of the edges as crease edges. In the right surface the upper corners are cusp points and the lower corners are conical points.

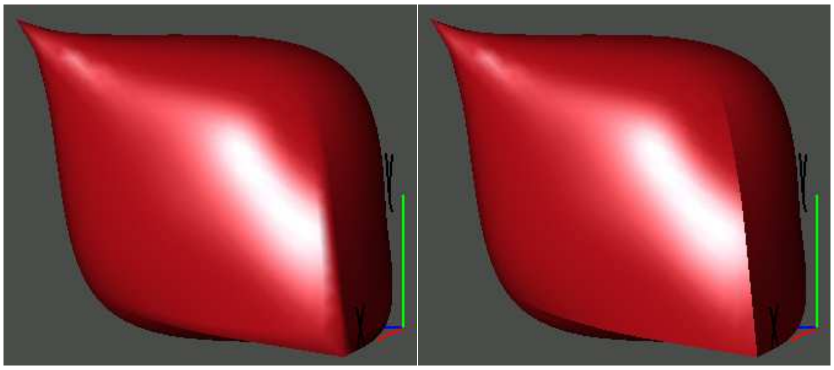


Figure 6.12: Loop surface with sharp creases, a cusp point and a corner point. To the left interpolated normals are used, to the right the exact normals are used.

whether the knot spacings are locally uniform, and if so, then use the uniform masks, which can be precomputed.

The polygon frequency of the extended Loop scheme is identical to the figure given for Loop in the previous section, namely 42000. As discussed in the previous section, this number should be divided by two for comparison with the NURSS schemes.

The extended Loop scheme can not match the flexibility of the NURSS schemes, but it is superior in most other aspects. It is marginally faster and much easier to control. Also, the behaviour of the extended Loop scheme is fully understood[1], whereas the analysis of the NURSS schemes still leaves many questions unanswered. The fact that the quadratic NURSS scheme may actually diverge for valencies $n \geq 15$ will disqualify it for most applications.

[1] There are a few special valencies for which the subdivision matrix is defective and the behaviour is not known. As mentioned previously generalized eigenanalysis should be able to answer these remaining questions.

Chapter 7

Conclusion

In the recent years the subject of subdivision surfaces has grown considerably in size and complexity. The mathematical concepts have been extended so they can verify that subdivision schemes are capable of producing limit surfaces that fulfill conditions for smoothness, regardless of the topology in question.

Especially the varying topology supported by the subdivison schemes have shown they are very useful. The well known spline surfaces have been the dominating way to represent surfaces in geometric modeling but they have the major drawback of being limited by the topology of the surface – they can not easily model objects with non-planar topology.

The subdivision surfaces studied in this paper are generalizations of the spline surfaces and therefore we wanted to discuss the mathematical concepts behind the spline curves and spline surfaces. We defined the uniform B-spline curve in chapter 2 and showed how it can be subdivided. Especially important we showed that the subdivision of B-spline curves indeed converges to a limit curve and that the curve in fact is the B-spline curve defined by the control points. We then showed how to extend this to B-spline surfaces and described the tensor product B-spline surfaces and the triangular box spline surfaces.

As mentiond the spline surfaces imposed limits on the topology so in chapter 3 we introduced the five subdivision schemes studied in the paper – Doo-Sabin, Catmull-Clark, Loop, Quadratic NURSS and Cubic NURSS. The NURSS schemes are non-uniform and therefore introduce the concept of knot spacings but in the case where the knot spacings are equal the Quadratic NURSS scheme is identical to Doo-Sabin and the Cubic NURSS scheme is identical to Catmull-Clark. Doo-Sabin, Catmull-Clark and Loop generalize spline surfaces to arbitrary topology by specifying additional masks for irregular faces and irregular vertices in the control mesh. These masks were all shown in chapter 3 together with a general classification of the schemes based on the control mesh and how they operate on it. Chapter 3 is meant as a non-technical description of the schemes and as such it is sufficient to read only that part if it is desired to implement the schemes without further mathematical understanding.

In the following chapter we discussed the mathematical tools employed in analysis of the subdivision schemes. We showed criteria for convergence of subdivision and smoothness of the limit surface. We focused on the extraordinary points in the scheme since away from these the schemes reduce to spline schemes and we showed that the number of extraordinary points is preserved by subdivision – that is the limit surface contains the same number of extraordinary points as the number of irregularities in the original mesh.

The main concepts we utilized in the our analysis was eigenanalysis of the subdivision matrix and the characteristic map. The eigenanalysis is used to verify the convergence criterion and derive a tangent plane, and the charateristic map can then be used to show that the limit surfaces has a regular C^1 parameterization. We then used these two concepts in an example. The example showed their application in the case of a smooth and a crease vertex in the Loop scheme and demonstrated our understanding of the concepts beyond the theory - as seen in the characteristic maps we have drawn.

We put some thought into designing our implementation so that it can be extended and optimized. Although the main concern was to make sure that we generated surfaces that were correct and not so much on perfomance we feel that we have a sound foundation. The data structure was of major importance because we had to handle various mesh topologies and we had to be able to traverese the mesh in a intuitive and efficient manner. The lath data structure fulfilled these requirements and seen as an isolated object in the implementation it can be reused in other instances. We also implemented some of the theory, beyond the masks, namely exact surface normals in the Loop scheme. The normals were found in the eigenanalysis for the Loop scheme and their effect, in comparison to interpolated normals, is especially visible at creases and corner, cusp and conical points. Because of that fewer steps in the subdivision are needed to produce satisfying results, which saves time and memory consumption.

As mentioned we did not focus so much on a fast and memory efficient implementation but that is something that could be worked on by using the present code. Another approach to efficent subdivision is given by Stam in [29, 28] were he shows how to do an exact evaluation of the surface, i.e. it is done in one step and therefore it is very fast. This could be combined with standard optimized algorithms for drawing spline surfaces on the regular parts of a control mesh.

In closing we feel that we have gotten a good understanding of the rather complex subject of subdivision surfaces. We have omitted some areas such as showing that the characteristic map is regular and injective near non-uniform features in the Loop scheme and we have not done a general eigenanalysis in those case either. This paper, though, gives us a good starting point to investigate even further into subdivision surfaces and their mathematical properties.

Appendix A

Source Code Listings

A.1 Overview

The following table provides an overview over the source code files that constitute the model viewer program.

Source file	*Description*	*Page*
SSTypes.h	Common type definitions and constant values used throughout the program	74
SSMaster.h	Includes often used standard header files.	75
ErrorDescription.h	Defines the class ErrorDescription used to manage error messages internally.	75
SSVertex.h	Defines the class SSVertex representing a vertex in a control mesh. The vertex contains the coordinates of the point associated with it.	76
SSVertex.cpp	Defines some methods in the class SSVertex.	78
SSVector.h	Defines the class SSVector which implements arithmetic operators on vectors.	78
SSVector.cpp	Method definitions for class SSVector.	84
SSLath.h	Defines the class SSLath representing a lath as described in section 5.1.2.	85
SSCornerLath.h	Defines the class SSCornerLath specializing SSLath for the corner lath data structure.	86
SSCornerLath.cpp	Defines some methods in the class SSCornerLath.	88
SSSubdivideMethods.h	Defines the class SSSubdivideMethods which is the ancestor for all classes implementing subdivision schemes.	91
SSModel.h	Defines the class SSModel.	92
SSQModel.h	Defines the class SSQModel inheriting from SSModel. The SSQModel class encapsulates a 3D model and features convenience methods for building models as well as methods for drawing and triangulating a model.	93
SSQModel.cpp	Defines most of the methods in SSQModel.	94
SSDooSabin.h	Defines the class SSDooSabin implementing Doo-Sabin subdivision.	105

A.2 Listings

Listing A.1: SSTypes.h.

```
#if !defined _SSTypes_HEADER
#define _SSTypes_HEADER

typedef double CoordType;
typedef long ErrorType;
typedef double CoefficientType;
typedef double KnotSpacingType;
typedef int MethodType;
typedef int LevelType;

const int c_DooSabin=1;
const int c_CatmullClark=2;
const int c_Loop=3;
const int c_QuadNURSS=4;
const int c_CubicNURSS=5;

```

```
const double c_PI = 3.14159265359;
const double c_Small = 1e-12;

// Flags for rendering
#define SS_DRAW_EXACT_NORMALS 0x0001
#define SS_DRAW_OUTLINE       0x0002
#define SS_DRAW_TRACE         0x8000
#endif
```

Listing A.2: SSMaster.h.

```
#if !defined _SSMaster_HEADER
#define _SSMaster_HEADER

#include <vector>
#include <iostream>
#include <string>
#include <map>
#include <memory>
using namespace std;

#include "SSTypes.h"

#pragma warning( disable : 4786 )
#endif
```

Listing A.3: ErrorDescription.h.

```
#if !defined _ErrorDescription_HEADER
#define _ErrorDescription_HEADER

#ifndef _SSMaster_HEADER
#include "SSMaster.h"
#endif

class ErrorDescription
{

public:
  ErrorDescription()
  {
    m_info = "No Error";
    m_type = 1000;
    m_where = "";
    m_error = false;
  };

  ErrorDescription(ErrorType n, string info, string where)
  {
    m_type = n;
    m_info = info;
```

```
    m_where = where;
    m_error = true;
  };

  virtual ∼ErrorDescription() {};

  string getinfo () { return m_info; };
  string getwhere() { return m_where; };
  ErrorType gettype() { return m_type; };
  bool error() { return m_error; };

private:
  bool m_error;
  string m_info;
  ErrorType m_type;
  string m_where;
};

#endif
```

Listing A.4: SSVertex.h.

```
#if !defined _SSVertex_HEADER
#define _SSVertex_HEADER

#ifndef _SSMaster_HEADER
#include "SSMaster.h"
#endif

class SSVertex
{

public:
  SSVertex();

  SSVertex(CoordType x,
           CoordType y,
           CoordType z);

  virtual ∼SSVertex();

  CoordType x() const { return m_x;};
  CoordType y() const { return m_y;};
  CoordType z() const { return m_z;};

  bool operator==(const SSVertex& rhs)
  {
    return (m_x == rhs.x() && m_y == rhs.y() && m_z == rhs.z());
  }

  bool operator≠(const SSVertex& rhs)
  {
```

```
    return (m_x ≠ rhs.x() || m_y ≠ rhs.y() || m_z ≠ rhs.z());
  }

  SSVertex& operator= (const SSVertex& rhs)
  {
    if(*this == rhs)
      return *this;

    m_x = rhs.x();
    m_y = rhs.y();
    m_z = rhs.z();

    return *this;
  };

  SSVertex operator+ (const SSVertex& rhs)
  {
    return SSVertex(m_x + rhs.m_x, m_y + rhs.m_y, m_z + rhs.m_z);
  };

  void operator+= (const SSVertex& rhs)
  {
    m_x += rhs.m_x;
    m_y += rhs.m_y;
    m_z += rhs.m_z;
  }

  SSVertex operator* (const CoefficientType& c)
  {
    return SSVertex(m_x * c, m_y * c, m_z * c);
  }

  SSVertex operator/ (const CoefficientType& c)
  {
    if(c == 0) {
      cout ≪ "Division by zero"≪ endl;
      throw "Div by zero";
    }

    return SSVertex(m_x / c, m_y / c, m_z / c);
  }

  void operator/= (const CoefficientType& c)
  {
    m_x /= c;
    m_y /= c;
    m_z /= c;
  }

private:
  CoordType m_x, m_y, m_z;

  friend SSVertex operator*(const CoefficientType& c, const SSVertex& v);
```

```
};

inline SSVertex operator*(const CoefficientType& c, const SSVertex& v)
{
  return SSVertex(c*v.m_x, c*v.m_y, c*v.m_z);
}

ostream& operator<<(ostream&, const SSVertex & v);
ostream& operator<<(ostream&, const vector<SSVertex> & vec);

#endif
```

Listing A.5: SSVertex.cpp.

```
#include "SSVertex.h"

SSVertex::SSVertex()
{
  m_x = m_y = m_z = 0.0;
}

SSVertex::~SSVertex()
{
  return;
}

SSVertex::SSVertex(CoordType x, CoordType y, CoordType z)
{
  m_x = x;
  m_y = y;
  m_z = z;
}

ostream& operator<<(ostream& o, const SSVertex & v)
{
  return o << "("<< v.x() << ","<< v.y() << ","<< v.z() << ")";
}

ostream& operator<<(ostream& o, const vector<SSVertex> & vec)
{
  for(vector<SSVertex>::const_iterator iter = vec.begin(); iter != vec.end(); iter++)
    {
      o << *iter << " ";
    }
  return o;
}
```

Listing A.6: SSVector.h.

```
#ifndef _SSSSVector_HEADER
#define _SSSSVector_HEADER
```

```cpp

#include "SSMaster.h"

#include <stdlib.h>
#include <math.h>

#include "SSVertex.h"

// Class representing a vector in homogeneous coordinates
class SSVector
{

public:
  // Constructors
  SSVector(SSVertex f, SSVertex t);
  SSVector(SSVertex f);
  SSVector( const double _a, const double _b, const double _c );
  SSVector( const double ar[3]);
  SSVector( const double _a, const double _b, const double _c, const double _d );
  SSVector(void);

  // Non-destructive arithmetic operators
  SSVector operator + ( const SSVector &x ) const;
  SSVector operator - ( const SSVector &x ) const;
  SSVector operator - ( void ) const;
  SSVector operator * ( const double &k ) const;

  // Destructive arithmetic operators
  void operator -= ( const SSVector &x );
  void operator += ( const SSVector &x );
  void operator *= ( const SSVector &x );
  void operator *= ( const double &k );
  void operator /= ( const double &k );

  // Comparison operators
  int operator < ( const SSVector &x ) const;
  int operator <= ( const SSVector &x ) const;
  int operator > ( const SSVector &x ) const;
  int operator >= ( const SSVector &x ) const;
  int operator != ( const int &val ) const;
  int operator != ( const SSVector &v ) const;
  int operator == ( const SSVector &v ) const;

  // Cross product, Euclidean length
  void cross( const SSVector &x, const SSVector &y );
  double len( void ) const;

  // Normalization and homogenization
  void normalize( void );
  void homogenize()
  {
    a = a/d;
    b = b/d;
```

```
    c = c/d;
    d = 1;
  };

  // Reverse direction of vector
  void invert();

  // Make vector x orthogonal to projection of current vector onto XY-plane
  void get_ortogonal( SSVector &x ) const;

  // Indexing and assignment
  void copy_to_array( double dest[3] ) const;
  double& operator [] ( const int index );
  double operator [] ( const int index ) const;
  void operator = (const SSVector &v);

private:
  double a,b,c,d;

  // Non-destructive dot and cross product
  friend inline double dot( const SSVector &x, const SSVector &y );
  friend inline double dot( const SSVector &x, const double v[3] );
  friend inline SSVector cross( const SSVector &x, const SSVector &y );
};

// Constant null vector
const SSVector NULLSSVector;

inline SSVector::SSVector(SSVertex f)
{
  a = f.x();
  b = f.y();
  c = f.z();
  d = 1.0;
}

inline SSVector::SSVector(SSVertex f, SSVertex t)
{
  a = f.x() - t.x();
  b = f.y() - t.y();
  c = f.z() - t.z();
  d = 1.0;
}

inline SSVector::SSVector(void)
{
  a=b=c=0.0;
  d=1.0;
}

inline SSVector::SSVector( const double _a, const double _b, const double _c ):
  a(_a),b(_b),c(_c)
{
```

```
  d = 1.0;
}

inline SSVector::SSVector( const double ar[3]) : a(ar [0]), b(ar [1]), c(ar [2])
{
  d = 1.0;
}

inline SSVector::SSVector( const double _a, const double _b,
                           const double _c, const double _d):
  a(_a),b(_b),c(_c),d(_d) {}

inline SSVector SSVector::operator + ( const SSVector &x ) const
{
  return SSVector( x.a+a, x.b+b, x.c+c);
}

inline SSVector SSVector::operator − ( const SSVector &x ) const
{
  return SSVector( a−x.a, b−x.b, c−x.c );
}

inline SSVector SSVector::operator − ( void ) const
{
  return SSVector( −a, −b, −c );
}

inline SSVector SSVector::operator ∗ ( const double &k ) const
{
  return SSVector( a∗k, b∗k, c∗k );
}

inline void SSVector::operator −= ( const SSVector &x )
{
  a−=x.a;
  b−=x.b;
  c−=x.c;
}

inline void SSVector::operator += ( const SSVector &x )
{
  a+=x.a;
  b+=x.b;
  c+=x.c;
}

inline void SSVector::operator ∗= ( const SSVector &x )
{
  a∗=x.a;
  b∗=x.b;
  c∗=x.c;
}

```

```
inline void SSVector::operator *= ( const double &k )
{
  a*=k;
  b*=k;
  c*=k;
}

inline void SSVector::operator /= ( const double &k )
{
  a/=k;
  b/=k;
  c/=k;
}

inline int SSVector::operator < ( const SSVector &x ) const
{
  return (a<x.a && b<x.b && c<x.c);
}

inline int SSVector::operator ≤ ( const SSVector &x ) const
{
  return (a≤x.a && b≤x.b && c≤x.c);
}

inline int SSVector::operator > ( const SSVector &x ) const
{
  return (a>x.a && b>x.b && c>x.c);
}

inline int SSVector::operator ≥ ( const SSVector &x ) const
{
  return (a≥x.a && b≥x.b && c≥x.c);
}

inline int SSVector::operator ≠ ( const int &val ) const
{
  if (val==0 && a==0 && b==0 && c==0)
    return 0;
  else
    return 1;
}

inline int SSVector::operator ≠ ( const SSVector &v ) const
{
  if (a≠v.a || b≠v.b || c≠ v.c)
    return 1;
  else
    return 0;
}

inline int SSVector::operator == ( const SSVector &v ) const
{
  if (a==v.a && b==v.b && c== v.c)
```

```
    return 1;
  else
    return 0;
}

inline void SSVector::cross( const SSVector &x, const SSVector &y )
{
  a = x.b*y.c-x.c*y.b;
  b = x.c*y.a-x.a*y.c;
  c = x.a*y.b-x.b*y.a;
  d = 1;
}

inline double SSVector::len( void ) const
{
  return sqrt( a*a + b*b + c*c );
}

inline void SSVector::normalize( void )
{
  double len;
  if ((len = sqrt(a*a+b*b+c*c)) == 0.0)
    {
      cerr << "Can't normalize nullSSVector! Using [1,0,0]."<< endl;
      a=1.0; b=0.0; c=0.0;
      return;
    }
  double invlen = 1.0/len;
  a *= invlen; b *= invlen; c *= invlen;
}

inline void SSVector::invert()
{
  a = -a;
  b = -b;
  c = -c;
}

inline void SSVector::copy_to_array( double dest[3] ) const
{
  dest [0] = a; dest [1] = b; dest [2] = c;
}

inline double& SSVector::operator [] ( const int index )
{
  switch( index ) {
  case 0: return a;
  case 1: return b;
  case 2: return c;
  case 3: return d;
  default:
    cerr << "SSVector index must be in [0,3]";
    exit (1);
```

```
  }
  return a; // To avoid warnings - we don't get here
}

inline double SSVector::operator [] ( const int index ) const
{
  switch( index )
    {
    case 0: return a;
    case 1: return b;
    case 2: return c;
    case 3: return d;
    default:
      cerr << "SSVector index must be in [0,3]";
      exit (1);
    }
  return 0.0;
}

inline void SSVector::operator = (const SSVector &v)
{
  a = v.a; b = v.b; c = v.c;
}

inline double dot( const SSVector &x, const SSVector &y )
{
  return x.a*y.a + x.b*y.b + x.c*y.c;
}

inline double dot( const SSVector &x, const double v[3] )
{
  return x.a*v[0] + x.b*v[1] + x.c*v[2];
}

inline SSVector cross( const SSVector &x, const SSVector &y )
{
  return SSVector( x.b*y.c-x.c*y.b, x.c*y.a-x.a*y.c, x.a*y.b-x.b*y.a );
}

ostream& operator<<(ostream& os, const SSVector& x );

#endif
```

Listing A.7: SSVector.cpp.

```
#include "SSMaster.h"

#include <math.h>

#include "SSVector.h"

void SSVector::get_ortogonal( SSVector &x ) const
```

```
{
  if ( fabs(a) > c_Small )
    {
      if ( fabs( b ) > c_Small )
        {
          x.a = b; x.b = −a; x.c = 0.0;
        }
      else
        {
          x.a = 0.0; x.b = −1.0; x.c = 0.0;
        }
    }
  else
    {
      x.a = 1.0; x.b = 0.0; x.c = 0.0;
    }
}

ostream& operator≪(ostream& os, const SSVector& x )
{
  os ≪ "["≪ x[0] ≪ ", "≪ x[1] ≪ ", "≪ x[2] ≪ "]";
  return os;
}
```

Listing A.8: SSLath.h.

```
#if !defined _SSLath_HEADER
#define _SSLath_HEADER

#include "SSVertex.h"

// Class representing a lath
class SSLath
{
public:
  // Next lath pointing to same vertex, clockwise and counter−clockwise, resp.
  virtual SSLath ∗cwv()=0;
  virtual SSLath ∗ccwv()=0;

  // Next lath in same face, clockwise and counter−clockwise resp.
  virtual SSLath ∗cwf()=0;
  virtual SSLath ∗ccwf()=0;

  // Associated lath on lower level
  virtual SSLath ∗nl()=0;

  // Vertex pointed to by this lath
  virtual SSVertex corner() const =0;

  // Output a string representation
  virtual ostream& put(ostream &s) const = 0;

```

```
  // Set next lath counter-clockwise around face or vertex, resp.
  virtual void ccwnextf(SSLath *lath)=0;
  virtual void ccwnextv(SSLath *lath)=0;

  // Set lath on lower level
  virtual void nextlevel(SSLath *lath)=0;

  // Check and mark lath as visited
  virtual bool isvisited() const { return false; };
  virtual void visited(bool b) {};

  // Update knot spacings, clockwise and counter-clockwise, resp.
  virtual void setcwknotspc(KnotSpacingType spc)=0;
  virtual void setccwknotspc(KnotSpacingType spc)=0;

  // Get knot spacings, clockwise and counter-clockwise, resp.
  virtual KnotSpacingType cwknotspc() const=0;
  virtual KnotSpacingType ccwknotspc() const=0;

  // Get knot spacing according to labelling defined by Sederberg et al.
  virtual KnotSpacingType knotspc(int s, int e, int r) const=0;

  // Destructor for clean-up
  virtual ~SSLath() {};
};

#endif
```

Listing A.9: SSCornerLath.h.

```
#if !defined _SSCornerLath_HEADER
#define _SSCornerLath_HEADER

#include "SSLath.h"
#include "SSVertex.h"

// Specialization of lath for the corner lath structure
// See method descriptions in file SSLath.h
class SSCornerLath : public SSLath
{
public:
  virtual SSLath *cwv()
  {
    SSLath *tmp = this;
    tmp = tmp->ccwf()->ccwv();
    if(tmp) {
      return tmp->ccwf();
    } else {
      tmp = this;
      while(tmp && tmp->ccwv() != this) { tmp = tmp->ccwv(); }
      return tmp;   // May be null => no clockwise neighbour
    }
```

```
};

virtual SSLath *ccwv()
{
  return m_ccwnextv;
};

virtual SSLath *cwf()
{
  SSLath *tmp = this;

  while(tmp && tmp→ccwf() ≠ this) { tmp = tmp→ccwf(); };

  return tmp;
};

virtual SSLath *ccwf()
{
  return m_ccwnextf;
};

virtual SSLath *nl()
{
  return m_nextlevel;
};

virtual SSVertex corner() const
{
  return m_corner;
};

virtual void ccwnextf(SSLath *lath)
{
  m_ccwnextf = lath;
};

virtual void ccwnextv(SSLath *lath)
{
  m_ccwnextv = lath;
};

virtual void nextlevel(SSLath *lath)
{
  m_nextlevel = lath;
};

virtual ostream& put(ostream &s) const;

virtual bool isvisited () const
{
  return m_visited;
}
```

```
  virtual void visited(bool b)
  {
    m_visited = b;
  }

  virtual void setcwknotspc(KnotSpacingType spc)
  {
    m_cwknot = spc;
  }

  virtual void setccwknotspc(KnotSpacingType spc)
  {
    m_ccwknot = spc;
  }

  virtual KnotSpacingType ccwknotspc() const
  {
    return m_ccwknot;
  }

  virtual KnotSpacingType cwknotspc() const
  {
    return m_cwknot;
  }

  virtual KnotSpacingType knotspc(int,int,int) const;

  virtual ~SSCornerLath();

  SSCornerLath(SSVertex corner);

private:
  SSVertex m_corner;
  SSLath *m_nextlevel;
  SSLath *m_ccwnextv;
  SSLath *m_ccwnextf;
  KnotSpacingType m_ccwknot, m_cwknot;
  bool m_visited;
};

ostream& operator<<(ostream &s, SSLath &l);
ostream& operator<<(ostream &s, SSLath *l);

#endif
```

Listing A.10: SSCornerLath.cpp.

```
#if defined(WIN32)
#pragma warning( disable : 4786 )
#endif

#include "SSCornerLath.h"
```

```
#include "SSVertex.h"

// See Sederberg et al for labelling of knot spacings
// Request for non-existant knot spacing returns -1
KnotSpacingType SSCornerLath::knotspc(int s, int e, int r) const
{
  SSLath *tmp = const_cast<SSCornerLath*>(this);
  int c;

  // Find correct start vertex
  if(s>=0)
    {
      for(c=s; c>0; c--) { tmp = tmp->cwf(); }

      // Rotate
      if(r>0)
        for(c=r; c>0; c--) {
          tmp = tmp->ccwv();
          if(!tmp) {
            cerr << "1 Returned -1.0 s "<< s << " e "<< e
                 << " r " << r << endl;
            return (KnotSpacingType) -1.0;
          }
        }
      else
        for(c=r; c<0; c++) {
          tmp = tmp->cwv();
          if(!tmp) {
            cerr << "2 Returned -1.0 s "<< s << " e "<< e
                 << " r " << r << endl;
            return (KnotSpacingType) -1.0;
          }
        }
      if(e<s) {
        return tmp->ccwknotspc();
      } else if(e>s) {
        return tmp->cwknotspc();
      } else {
        cerr << "3 Returned -1.0 s "<< s << " e "<< e
             << " r " << r << endl;
        return (KnotSpacingType) -1.0;
      }
    }
  else // s<0
    {
      for(c=s; c<0; c++) { tmp = tmp->ccwf(); }

      // Rotate
      if(r>0)
        for(c=r; c>0; c--) {
          tmp = tmp->ccwv();
          if(!tmp) {
            cerr << "4 Returned -1.0 s "<< s << " e "<< e
```

```
                    << "  r " << r << endl;
                return (KnotSpacingType) -1.0;
            }
        }
      else
        for(c=r; c<0; c++) {
            tmp = tmp->cwv();
            if (!tmp) {
                cerr << "5 Returned -1.0 s "<< s << " e "<< e
                    << "  r " << r << endl;
                return (KnotSpacingType) -1.0;
            }
        }
      if(e<s) {
        return tmp->ccwknotspc();
      } else if(e>s) {
        return tmp->cwknotspc();
      } else {
        cerr << "6 Returned -1.0 s "<< s << " e "<< e
            << "  r " << r << endl;
        return (KnotSpacingType) -1.0;
      }
    }
}

SSCornerLath::SSCornerLath(SSVertex corner)
{
  m_corner = corner;
  m_ccwnextv = m_ccwnextf = m_nextlevel = 0;
  m_visited = false;
  m_ccwknot = m_cwknot = 1.0;
}


SSCornerLath::~SSCornerLath()
{
}


ostream& SSCornerLath::put(ostream &s) const
{
  s << "V: "<< m_corner << " ";

  s << "CCWV ";
  if(m_ccwnextv != 0)
    s << m_ccwnextv->corner() << " ";
  else
    s << "null ";

  s << "CCWF ";
  if(m_ccwnextf != 0)
    s << m_ccwnextf->corner() << " ";
  else
```

```
    s << "null ";

  s << "NL ";
  if(m_nextlevel != 0)
    s << m_nextlevel->corner();
  else
    s << "null";

  return s;
}

ostream& operator<<(ostream &s, SSLath &l)
{
  return l.put(s);
}

ostream& operator<<(ostream &s, SSLath *l)
{
  return l->put(s);
}
```

Listing A.11: SSSubdivideMethods.h.

```
#if !defined _SSSubdivideMethods_HEADER
#define _SSSubdivideMethods_HEADER

#include "ErrorDescription.h"
#include "SSCornerLath.h"
#include "SSVector.h"

// Base class for implementing subdivision schemes
class SSSubdivideMethods
{

public:
  virtual SSLath* Subdivision(vector<SSLath*> &vec, SSLath *start)=0;

  // Generic interpolated polygonal normal
  // Overridden by classes providing scheme specific normals
  virtual SSVector calcNormal(SSLath *face)
  {
    SSVector norm (0.0, 0.0, 0.0);
    SSLath *tmp;
    tmp = face;
    int count = 0;
    do {
      SSVector v1(tmp->corner(), tmp->ccwf()->corner());
      SSVector v2(tmp->corner(), tmp->cwf()->corner());
      v1.normalize(); v2.normalize(); // Normalize to reduce numerical instability
      SSVector c = cross(v1, v2);
      norm += c;
      tmp = tmp->ccwv();
```

```
      count++;
      if(count > 200)
        exit (0);   // Valency > 200 indicates a malformed mesh and infinite loop
    } while(tmp && tmp != face);
    norm.normalize();
    return norm;
  };

  virtual ~SSSubdivideMethods() {};
};

#endif
```

Listing A.12: SSModel.h.

```
#if !defined _SSModel_HEADER
#define _SSModel_HEADER

#include "SSMaster.h"
#include "SSLath.h"
#include "SSVertex.h"
#include "ErrorDescription.h"

// Base class for representing 3D polygonal models
class SSModel
{
public:

  // Convenience methods for building a face with or without knot spacings
  // The new laths are appended to vec
  virtual SSLath *BuildFace(const vector<SSVertex> vertices, vector<SSLath*> &vec)=0;
  virtual SSLath *BuildFace(const vector<SSVertex> vertices,
                            const vector<KnotSpacingType> knots,
                            vector<SSLath*> &vec)=0;

  // Convenience method for stitching together two faces across numbered edges
  // Edges are numbered counterclockwise starting with 0 on the clockwise edge
  // from the lath pointer
  virtual ErrorDescription ConnectFaces(SSLath* f1, int edgeno1,
                                        SSLath* f2, int edgeno2)=0;

  // Collects the vertices and optionally knot spacings in the face given by start
  virtual ErrorDescription FaceVertices(vector<SSVertex> &vertices, SSLath* start)=0;
  virtual ErrorDescription FaceVertices(vector<SSVertex> &vertices,
                                        vector<KnotSpacingType> &knots,
                                        SSLath* start)=0;

  // Loads a model from the file filename
  virtual ErrorDescription Init(const string &filename)=0;

  // Prepares the model for a new subdivision scheme
  virtual ErrorDescription SwitchMethod(MethodType newmethod)=0;
```

```

  // Subdivides the model using the specified scheme
  virtual ErrorDescription Subdivide(MethodType m, int levels)=0;

  // Draws a specific subdivision level of the model using OpenGL
  // Flags control the rendering
  virtual ErrorDescription Draw(int level, int flags)=0;

  // Return the number of faces in the given level of the model
  virtual long NumberOfFaces(int level) { return 0; };

  // Delete a level of the model
  virtual void DeleteLevel(LevelType l)=0;

  // Destructor for clean-up
  virtual ~SSModel() {};

protected:
  SSModel() {};

};

#endif
```

Listing A.13: SSQModel.h.

```
#if !defined _SSQModel_HEADER
#define _SSQModel_HEADER

#include "SSMaster.h"
#include "SSSubdivideMethods.h"
#include "SSModel.h"
#include "SSVector.h"
#include <GL/glut.h>

typedef vector<SSLath*> LathVector;
typedef pair<SSLath*, LathVector> Level;

// Specialization of SSModel
// See SSModel.h for method descriptions
class SSQModel : public SSModel
{
public:
  virtual SSLath *BuildFace(const vector<SSVertex> vertices,
                            vector<SSLath*> &vec);
  virtual SSLath *BuildFace(const vector<SSVertex> vertices,
                            const vector<KnotSpacingType> knots,
                            vector<SSLath*> &vec);

  virtual ErrorDescription ConnectFaces(SSLath* f1, int edgeno1,
                                        SSLath* f2, int edgeno2);

```

```
  virtual ErrorDescription FaceVertices(vector<SSVertex> &vertices, SSLath* start);
  virtual ErrorDescription FaceVertices(vector<SSVertex> &vertices,
                                        vector<KnotSpacingType> &knots,
                                        SSLath* start);

  virtual ErrorDescription Init(const string &filename);

  virtual SSVector calcNormal(SSLath *face);

  virtual ErrorDescription Draw(int level, int flags );

  virtual ErrorDescription SwitchMethod(MethodType newmethod);

  virtual ErrorDescription Subdivide(MethodType m, int levels);

  virtual void DeleteLevel(LevelType l);

  virtual long NumberOfFaces(int level)
  {
    if(level < 0 || level > m_curlevel)
      level = 0;

    return m_Levels[level].second.size();
  };

  static SSQModel *Instance();

  virtual ~SSQModel();

  // Triangulates the current model by inserting new vertices at the
  // centroid of all faces
  virtual void Triangulate();

protected:
  SSQModel();

private:
  void DrawFace(SSLath *start, int flags);

  static auto_ptr<SSQModel> m_instance;
  SSSubdivideMethods *m_method;
  LevelType m_curlevel;
  map<LevelType, Level> m_Levels;
  MethodType m_curmethod;
  string m_filename;
};

#endif
```

Listing A.14: SSQModel.cpp.

```
#if defined(WIN32)
```

```
#pragma warning( disable : 4786 )
#endif

#include "SSQModel.h"
#include <numeric>
#include <time.h>
#include "SSCornerLath.h"
#include "SSDooSabin.h"
#include "SSCatmullClark.h"
#include "SSLoop.h"
#include "SSQuadraticNURSS.h"
#include "SSCubicNURSS.h"
#include "SSMdlReader.h"
#include "SSVector.h"
#include "SSTypes.h"
#include "SSHeroReader.h"

#if defined(WIN32)
#include <windows.h>
#include <winbase.h>
#else
#include <unistd.h>
#endif

auto_ptr<SSQModel> SSQModel::m_instance = auto_ptr<SSQModel>(0);

SSQModel::SSQModel()
{
  m_method = 0;
  m_curlevel = 0;
}

SSQModel::~SSQModel()
{
  if(m_method)
    delete m_method;

  for(LevelType i = 0; i <= m_curlevel; i++)
    {
      DeleteLevel(i);
    }

  m_Levels.clear();
}

void SSQModel::DeleteLevel(LevelType l)
{
  if(l < 0 || l > m_curlevel)
    return;

  LathVector laths = m_Levels[l].second;

  for(int j = 0; j < laths.size(); j++)
```

```
    {
      SSLath *start=laths[j], *tmp=0, *tmp1=0;

      tmp = start;
      int size = 0;
      do
        {
          tmp=tmp→ccwf();
          size++;
        }
      while(tmp ≠ start);

      tmp1 = tmp = start;
      while(size−−)
        {
          tmp1 = tmp→ccwf();

          delete tmp;

          tmp = tmp1;
        }
    }
  laths.clear();
}

SSQModel *SSQModel::Instance()
{
  if(m_instance.get() == 0)
    {
      auto_ptr<SSQModel> ptr(new SSQModel());
      m_instance = ptr;
    }
  return m_instance.get();
}

ErrorDescription SSQModel::Init(const string& filename)
{
  vector<SSLath*> v;
  int pos = filename.find_last_of('.');

#if defined(WIN32)
  if(pos<filename.size() && !filename.compare(filename.find_last_of('.'), 4, ".mdl"))
#else
    if(pos<filename.size() && !filename.compare(".mdl", filename.find_last_of('.'), 4))
#endif
      {
        SSMdlReader r;
        r.read(filename, v);
        m_curlevel = 0;
        m_Levels.insert(make_pair(m_curlevel, make_pair(v[0], v)));
      }
#if defined(WIN32)
    else if(pos<filename.size() && !filename.compare(filename.find_last_of('.'), 4, ".wrl"))
```

```
#else
      else if(pos<filename.size() && !filename.compare(".wrl", filename.find_last_of('.'), 4))
#endif
        {
          SSHeroReader r;
          r.read(filename, v);
          m_curlevel = 0;
          m_Levels.insert(make_pair(m_curlevel, make_pair(v[0], v)));
        } else {
          cerr << "Unknown file format.\nOnly .mdl and .wrl files are supported"<< endl;
          exit (1);
        }
  m_filename = filename;
  return ErrorDescription();
}

SSLath *SSQModel::BuildFace(vector<SSVertex> vertices, vector<SSLath*> &vec)
{

  SSLath *base, *curr, *next, *tmp;

  if( vertices.size() < 3 )
    return 0;
  base = new SSCornerLath(vertices[0]);
  curr = base;

  for(int i = 1; i<vertices.size(); i++)
    {
      next = new SSCornerLath(vertices[i]);
      curr->ccwnextf(next);
      curr = next;
    }

  curr->ccwnextf(base);

  tmp = base;
  int j=0;
  do
    {
      if( vertices[j] != tmp->corner())
        cout << "Error: "<< vertices[j] << " / "<< tmp->corner() << endl;
      tmp = tmp->ccwf();
      j++;
    }
  while(tmp != base);
  vec.push_back(base);

  return base;
}

SSLath *SSQModel::BuildFace(const vector<SSVertex> vertices,
                            const vector<KnotSpacingType> knots,
                            vector<SSLath*> &vec)
```

```
{
  SSLath *base, *curr, *next;
  int i;

  if( vertices.size() < 3 )
    throw string( "Wrong number of vertices");

  base = new SSCornerLath(vertices[0]);
  curr = base;

  for(i = 1; i<vertices.size(); i++)
    {
      next = new SSCornerLath(vertices[i]);
      curr->ccwnextf(next);
      curr->setcwknotspc(knots[(i-1)*2]);
      curr->setccwknotspc(knots[i*2-1]);
      curr = next;
    }

  curr->ccwnextf(base);
  curr->setcwknotspc(knots[(i-1)*2]);
  curr->setccwknotspc(knots[i*2-1]);

  vec.push_back(base);
  return base;
}

ErrorDescription SSQModel::ConnectFaces(SSLath* f1, int edgeno1, SSLath* f2, int edgeno2)
{
  while(edgeno1 > 0)
    {
      f1 = f1->ccwf();
      edgeno1--;
    }

  while(edgeno2 > 0)
    {
      f2 = f2->ccwf();
      edgeno2--;
    }

  f1->ccwnextv(f2->cwf());
  f2->ccwnextv(f1->cwf());

  return ErrorDescription();
}

ErrorDescription SSQModel::FaceVertices(vector<SSVertex> &vertices, SSLath* start)
{
  SSLath *tmp;

  tmp = start;
  do
```

```
    {
      vertices.push_back(tmp->corner());
      tmp = tmp->ccwf();
    }
  while(start != tmp);

  return ErrorDescription();
}

ErrorDescription SSQModel::FaceVertices(vector<SSVertex> &vertices,
                                        vector<KnotSpacingType> &knots,
                                        SSLath* start)
{
  SSLath *tmp;

  tmp = start;
  do
    {
      vertices.push_back(tmp->corner());
      knots.push_back(tmp->cwknotspc());
      knots.push_back(tmp->ccwknotspc());
      tmp = tmp->ccwf();
    }
  while(start != tmp);

  return ErrorDescription();
}

ErrorDescription SSQModel::SwitchMethod(MethodType newmethod)
{
  if(m_curmethod != newmethod)
    {
      if(m_method)
        delete m_method;

      for(LevelType i = 0; i <= m_curlevel; i++)
        {
          DeleteLevel(i);
        }

      m_Levels.clear();

      LevelType tmpl = m_curlevel;

      time_t s = clock();
      Init(m_filename);
      time_t e = clock();

      cout << "Init: " << e - s << " msec " << endl;

      s = clock();
      ErrorDescription ed = Subdivide(newmethod, tmpl);
      e = clock();
```

```
#if defined(WIN32)
      cout << "Subdivide: "<< e - s << " msec / levels: "<< tmpl << endl;
#else
      cout << "Subdivide: "<< ((double)(e - s))/CLOCKS_PER_SEC
           << " msec / levels: "<< tmpl << endl;
#endif
      return ed;
    }
  return ErrorDescription();
}

void SubdivideMethod(int nolevels, LevelType &curlevel,
                     map<LevelType, Level> &Levels, SSSubdivideMethods *method)
{
  vector<SSLath*> v;
  SSLath *levelstart=0;

  levelstart = Levels[curlevel].first;

  for(int i = 1; i <= nolevels; i++)
    {
      time_t start = clock();
      levelstart = method->Subdivision(v, levelstart);

      // Subdivide unconnected parts
      vector<SSLath*> faces = Levels[curlevel].second;
      for(int q=0; q<faces.size(); q++)
        {
          if(!faces[q]->nl())
            method->Subdivision(v, faces[q]);
        }

      time_t stop = clock();
      cout << "Level "<<i << " Polygons per sec: "
           << ((double)v.size())/(((double)(stop-start))/((double)CLOCKS_PER_SEC)) << endl;

      curlevel++;
      Levels.insert(make_pair(curlevel, make_pair(levelstart, v)));
      v.clear();
    }
}

ErrorDescription SSQModel::Subdivide(MethodType m, int levels)
{
  try
    {
      vector<SSLath*> v;
      SSLath *levelstart=0;

      m_curmethod = m;

      switch(m)
```

```
        {
        case c_DooSabin:
          m_method = new SSDooSabin();
          break;

        case c_CatmullClark:
          m_method = new SSCatmullClark();
          break;

        case c_Loop:
          m_method = new SSLoop();
#ifdef TRIANGULATE
          Triangulate();
#endif
          break;

        case c_QuadNURSS:
          m_method = new SSQuadraticNURSS();
          break;

        case c_CubicNURSS:
          m_method = new SSCubicNURSS();
          break;
        }
      SubdivideMethod(levels, m_curlevel, m_Levels, m_method);
    } catch (...)
      {
        cout << "Subdiv error"<< endl;
        return ErrorDescription(3000, "Subdivision Error", "");
      }
  return ErrorDescription();
}

ErrorDescription SSQModel::Draw(int level, int flags)
{
  if( level < 0 || level > m_curlevel)
    level = 0;

  LathVector laths = m_Levels[level].second;
  LathVector::iterator viter = laths.begin();

  while(viter != laths.end())
    {
      DrawFace(*viter, flags);

      if( flags & SS_DRAW_TRACE)
        {
          glutSwapBuffers();
#if defined(WIN32)
          Sleep(30 / laths.size());
#else
          usleep(30000 / laths.size());
#endif
```

```
        }
      viter++;
    }

  return ErrorDescription();
}

SSVector SSQModel::calcNormal(SSLath *face)
{
  SSVector norm (0.0, 0.0, 0.0);
  SSLath *tmp;
  tmp = face;
  int count = 0;
  do {
    SSVector v1(tmp->corner(), tmp->ccwf()->corner());
    SSVector v2(tmp->corner(), tmp->cwf()->corner());
    v1.normalize(); v2.normalize();
    SSVector c = cross(v1, v2);
    norm += c;
    tmp = tmp->ccwv();
    count++;
    if(count > 200)
      { // Possible infinite loop, most likely because of wrong connections
        return SSVector(1.0,0,0);
      }
  } while(tmp && tmp != face);
  norm.normalize();
  return norm;
}

void SSQModel::DrawFace(SSLath *start, int flags)
{
  SSLath *l1;
  l1 = start;
  SSVector c;
  SSVertex midpoint;

  if( flags & SS_DRAW_OUTLINE)
    {
      do
        {
          glBegin(GL_LINES);

          midpoint = 0.5 * (l1->corner() + l1->ccwf()->corner());

          if(l1->ccwknotspc() == 0.0)
            glColor3f (1.0,0.0,0.0);
          else if(l1->ccwknotspc() == 2.0)
            glColor3f (1.0,1.0,0.0);
          else if(l1->ccwknotspc() == 3.0)
            glColor3f (0.0,1.0,0.0);
          else if(l1->ccwknotspc() == 4.0)
            glColor3f (0.0,0.0,1.0);
```

```
        else
          glColor3f (1.0,1.0,1.0);

        glVertex3f((GLfloat) l1→corner().x(),
                   (GLfloat) l1→corner().y(),
                   (GLfloat) l1→corner().z());
        glVertex3f((GLfloat) midpoint.x(),
                   (GLfloat) midpoint.y(),
                   (GLfloat) midpoint.z());

        l1 = l1→ccwf();

        if(l1→cwknotspc() == 0.0)
          glColor3f (1.0,0.0,0.0);
        else if(l1→cwknotspc() == 2.0)
          glColor3f (1.0,1.0,0.0);
        else if(l1→cwknotspc() == 3.0)
          glColor3f (0.0,1.0,0.0);
        else if(l1→cwknotspc() == 4.0)
          glColor3f (0.0,0.0,1.0);
        else
          glColor3f (1.0,1.0,1.0);

        glVertex3f((GLfloat) l1→corner().x(),
                   (GLfloat) l1→corner().y(),
                   (GLfloat) l1→corner().z());
        glVertex3f((GLfloat) midpoint.x(),
                   (GLfloat) midpoint.y(),
                   (GLfloat) midpoint.z());

      }
    while(l1 ≠ start);
    glEnd();
  }
else
  {
    glBegin(GL_POLYGON);
    l1 = start;

    do
      {
        if(flags & SS_DRAW_EXACT_NORMALS)
          c = m_method→calcNormal(l1);
        else
          c = calcNormal(l1);

        glNormal3f(c[0], c[1], c[2]);

        glVertex3f((GLfloat) l1→corner().x(),
                   (GLfloat) l1→corner().y(),
                   (GLfloat) l1→corner().z());
```

```
            l1 = l1->ccwf();

          }
        while(l1 != start);

        glEnd();
      }
}

void SSQModel::Triangulate()
{
  int i,j;
  SSLath *face;
  LathVector laths = m_Levels[0].second;
  LathVector newlaths;
  for(i=0; i<laths.size(); i++)
    {
      face = laths[i];
      vector<SSVertex> v;
      FaceVertices(v, face);
      SSVertex center = accumulate(v.begin(), v.end(), SSVertex())/v.size();
      SSLath *base, *curr, *next, *saved, *nextbase;
      base = new SSCornerLath(center);
      curr = base;
      for(j=0; j<v.size()-1; j++)
        {
          next = new SSCornerLath(center);
          newlaths.push_back(next);
          curr->ccwnextv(next);
          curr = next;
        }
      next->ccwnextv(base);
      newlaths.push_back(base);

      curr = face;
      do {
        next = new SSCornerLath(curr->corner());
        next->ccwnextv(curr->ccwv());
        curr->ccwnextv(next);
        // Maintain knot spacings
        next->setcwknotspc(curr->cwknotspc());
        next->setccwknotspc(1.0);
        curr->setcwknotspc(1.0);

        curr = curr->ccwf();
      }
      while(curr != face);

      curr = face;
      nextbase = base;
      do {
        saved = curr->ccwf();
        curr->ccwnextf(saved->ccwv());
```

```
            curr→ccwf()→ccwnextf(nextbase);
            nextbase→ccwnextf(curr);
            nextbase = nextbase→ccwv();
            curr = saved;
          }
          while(curr ≠ face);

      }
    m_Levels[0] = make_pair(newlaths[0], newlaths);
}
```

Listing A.15: SSDooSabin.h.

```
#if !defined _SSDooSabin_HEADER
#define _SSDooSabin_HEADER

#include "SSMaster.h"
#include "SSSubdivideMethods.h"

// Implements Doo−Sabin subdivision
class SSDooSabin : public SSSubdivideMethods
{
public:
  virtual SSLath *Subdivision(vector<SSLath*> &vec, SSLath *start);

  SSDooSabin();

  virtual ~SSDooSabin();

private:
  vector<CoefficientType> CalcCoeff(int valence);
  void Subdivide(vector<SSLath*> &vec, SSLath *start);

  // Coefficients for regular case, i.e. quadrilaterals
  static const CoefficientType regcoeff [];
};

#endif
```

Listing A.16: SSDooSabin.cpp.

```
#if defined(WIN32)
#pragma warning( disable : 4786 )
#endif

#include <cmath>
#include "SSDooSabin.h"
#include "SSQModel.h"

const CoefficientType SSDooSabin::regcoeff[] = { 0.5625, 0.1875, 0.0625, 0.1875 };

```

```
SSDooSabin::SSDooSabin()
{
}

SSDooSabin::~SSDooSabin()
{
}

SSLath *SSDooSabin::Subdivision(vector<SSLath*> &vec, SSLath *start)
{

  SSDooSabin::Subdivide(vec, start);
  return start->nl();
}

vector<CoefficientType> SSDooSabin::CalcCoeff(int valence)
{
  vector<CoefficientType> coeff;

  if(valence == 4)
    {
      for(int j = 0; j < valence; j++)
        coeff.push_back(regcoeff[j]);

      return coeff;
    }

  coeff.push_back((CoefficientType) (0.25 + 1.25/(CoefficientType)valence));

  for(int i = 1; i < valence; i++)
    coeff.push_back((CoefficientType) ((3.0 + 2.0*cos(2.0*((CoefficientType)i)*c_PI
                        /((CoefficientType) valence)))/((CoefficientType) 4.0*valence)));

  return coeff;
}

void SSDooSabin::Subdivide(vector<SSLath*> &vec, SSLath *start)
{
  // If null or already subdivided just return
  if(start == 0 || start->nl())
    return;

  vector<SSVertex> vs, evs(4, SSVertex()), nvs;
  vector<SSLath *> els(4, (SSLath *) 0);
  vector<CoefficientType> coeff;

  SSLath *nf=0, *tmp1=0, *tmp2=0, *tmp3=0;

  int i, j, c, k;

  SSQModel::Instance()->FaceVertices(vs, start);
  k = vs.size();
  coeff = CalcCoeff(k);
```

```
for(i=0; i<k; i++)
  {
    SSVertex ny;

    for(j=0; j<k; j++)
      {
        c = (i+j)%k;
        SSVertex tmp = vs[c]*coeff[j];
        ny = ny + tmp;
      }
    nvs.push_back(ny);
  }

nf = SSQModel::Instance()→BuildFace(nvs, vec);

// Connect to subdivided level
start→nextlevel(nf);

tmp1 = start;
tmp2 = nf;
do
  {
    tmp1→nextlevel(tmp2);
    tmp1 = tmp1→ccwf();
    tmp2 = tmp2→ccwf();
  }
while(tmp1 ≠ start);

// Start subdivision chain reaction
tmp3 = start;
do
  {
    SSDooSabin::Subdivide(vec, tmp3→ccwv());
    tmp3 = tmp3→ccwf();
  }
while(start ≠ tmp3);

// Build and connect faces corresponding to old edges
tmp1 = start;
for(i = 0; i < k; i++, tmp1 = tmp1→ccwf())
  {
    if(tmp1→ccwv() && !tmp1→nl()→ccwv())
      {
        tmp2 = tmp1;
        evs[3] = tmp2→nl()→corner();
        els[0] = tmp2→nl();
        tmp2 = tmp2→ccwv();
        evs[2] = tmp2→nl()→corner();
        tmp2 = tmp2→ccwf();
        evs[1] = tmp2→nl()→corner();
        els[1] = tmp2→nl();
        tmp2 = tmp2→ccwv();
```

```
            evs [0]  = tmp2→nl()→corner();

            tmp2 = SSQModel::Instance()→BuildFace(evs, vec);

            SSQModel::Instance()→ConnectFaces(els[0], 0, tmp2, 0);
            SSQModel::Instance()→ConnectFaces(els[1], 0, tmp2, 2);
        }
    }

// Build and connect faces corresponding to old vertices
tmp1 = start;

do
  {
    if(tmp1→ccwv() && !tmp1→nl()→ccwv()→ccwv())
      { // If not already connected, build & connect face
        tmp2 = tmp1;
        vs. clear ();
        do
          {
            tmp3 = tmp2→nl();
            if (!tmp3 || !tmp3→ccwv())
              {
                /* All faces sharing edges with f have been subdivided in the
                 * recursion, but faces sharing only vertices with f may not
                 * have been. Solution: Just skip affected vertex -- it will
                 * be processed further down the recursion.
                 **/
                goto loop;
              }
            else
              vs.push_back(tmp3→corner());

            tmp2 = tmp2→ccwv();

            if (!tmp2)
              goto loop;
          }
        while(tmp1 ≠ tmp2);

        tmp3 = SSQModel::Instance()→BuildFace(vs, vec);

        do
          {
            if(tmp2→nl()→ccwv())
              SSQModel::Instance()→ConnectFaces(tmp2→nl()→ccwv(), 0, tmp3, 1);

            tmp2 = tmp2→ccwv();
            tmp3 = tmp3→ccwf();
          }
        while(tmp1 ≠ tmp2);
      }
  loop:
```

```
      tmp1 = tmp1->ccwf();
    }
  while(tmp1 != start);
}
```

Listing A.17: SSCatmullClark.h.

```
#if !defined _SSCatmullClark_HEADER
#define _SSCatmullClark_HEADER

#include "SSMaster.h"
#include "SSSubdivideMethods.h"

// Implements Catmull-Clark subdivision
class SSCatmullClark : public SSSubdivideMethods
{
public:
  virtual SSLath *Subdivision(vector<SSLath*> &vec, SSLath *start);

  SSCatmullClark();

  virtual ~SSCatmullClark();

private:
  void Subdivide(vector<SSLath*> &vec, SSLath *start);
};

#endif
```

Listing A.18: SSCatmullClark.cpp.

```
#if defined(WIN32)
#pragma warning( disable : 4786 )
#endif

#include <cmath>
#include "SSCatmullClark.h"
#include "SSQModel.h"

SSCatmullClark::SSCatmullClark()
{
}

SSCatmullClark::~SSCatmullClark()
{
}

SSLath *SSCatmullClark::Subdivision(vector<SSLath*> &vec, SSLath *start)
{
  SSCatmullClark::Subdivide(vec, start);
  if(vec.size()>0)
```

```
    return vec[0];
  return 0;
}

void SSCatmullClark::Subdivide(vector<SSLath*> &vec, SSLath *start)
{
  if(start == 0 || start->isvisited())
    return;

  static vector<SSVertex> vs, nvs(4, SSVertex()), fvs, evs;
  static SSVertex tmp, ny;
  static SSLath *nf=0, *tmp2=0, *tmp3=0;
  SSLath *tmp1;
  static int i, c=0, k, in;

  //New Face Vertices
  tmp1 = start;
  do
    {
      tmp1->visited(true);
      vs.clear();
      SSQModel::Instance()->FaceVertices(vs, tmp1);
      k = vs.size();
      ny = SSVertex();
      for(i=0; i<k; i++)
        {
          tmp = vs[i]/(CoordType)k;
          ny = ny + tmp;
        }
      fvs.push_back(ny);
      if(!tmp1->ccwv())
        {
          fvs.clear();
          do
            {
              tmp1->visited(true);
              SSCatmullClark::Subdivide(vec, tmp1->ccwf());
              tmp1 = tmp1->cwv();
            }
          while(tmp1);

          return;
        }
      tmp1 = tmp1->ccwv();
    }
  while(tmp1 != start);

  //New Edge Vertices
  tmp1 = start;
  k = fvs.size();
  c = 0;
  do
    {
```

```
        ny = SSVertex();
        tmp = fvs[c]/4.0;
        ny = ny + tmp;
        tmp = fvs[(c+1) % k]/4.0;
        ny = ny + tmp;
        tmp = tmp1→corner()/4.0;
        ny = ny + tmp;

        tmp = tmp1→cwf()→corner()/4.0;
        ny = ny + tmp;

        evs.push_back(ny);
        tmp1 = tmp1→ccwv();
        c++;
    }
  while(tmp1 ≠ start);

  //New Center Vertex
  ny = start→corner()∗(CoordType)((CoordType)(k−2)/(CoordType)k);

  tmp1 = start;
  tmp = SSVertex();
  do
    {
        tmp = tmp + tmp1→ccwf()→corner();
        tmp1 = tmp1→ccwv();
    }
  while(tmp1 ≠ start);

  ny = ny + tmp/(CoordType)(k∗k);
  tmp = SSVertex();

  k = fvs.size();
  for(i = 0; i < k; i++)
    tmp = tmp + fvs[i];

  ny = ny + tmp/(CoordType)(k∗k);

  //Build the faces adjacent to the Center vertex
  nvs[0] = ny;
  tmp1 = start;
  for(i = 0; i < k; i++)
    {
        nvs[1] = evs[(k−1+i) % k];
        nvs[2] = fvs[i];
        nvs[3] = evs[i];

        nf = SSQModel::Instance()→BuildFace(nvs, vec);

        tmp1→nextlevel(nf→ccwf()→ccwf());
        tmp1 = tmp1→ccwv();
    }

```

```
  //Connect the faces adjacent to the Center vertex
  tmp1 = start;
  do
    {
      SSQModel::Instance()→ConnectFaces(tmp1→nl(), 2, tmp1→ccwv()→nl(), 3);
      tmp1 = tmp1→ccwv();
    }
  while(tmp1 ≠ start);

  // Start subdivision chain reaction
  tmp1 = start;
  fvs.clear ();
  evs.clear ();
  do
    {
      SSCatmullClark::Subdivide(vec, tmp1→ccwf());
      tmp1 = tmp1→ccwv();
    }
  while(tmp1 ≠ start);

  //Connect new faces
  tmp1 = start;
  do
    {
      if(tmp1→ccwf()→nl())
        {
          SSQModel::Instance()→ConnectFaces(tmp1→nl(), 0, tmp1→ccwf()→nl(), 1);
        }
      if(tmp1→cwf()→nl())
        {
          SSQModel::Instance()→ConnectFaces(tmp1→nl(), 1, tmp1→cwf()→nl(), 0);
        }
      tmp1 = tmp1→ccwv();
    }
  while(tmp1 ≠ start);
}
```

Listing A.19: SSLoop.h.

```
#if !defined _SSLoop_HEADER
#define _SSLoop_HEADER

#include "SSMaster.h"
#include "SSSubdivideMethods.h"

// Implements Loop subdivision
class SSLoop : public SSSubdivideMethods
{
public:
  virtual SSLath *Subdivision(vector<SSLath*> &vec, SSLath *start);

  virtual SSVector calcNormal(SSLath *face);
```

```

SSLoop();

virtual ~SSLoop();

private:
  void Subdivide(vector<SSLath*> &vec, SSLath *start);
};

#endif
```

Listing A.20: SSLoop.cpp.

```
#if defined(WIN32)
#pragma warning( disable : 4786 )
#endif

#include <cmath>
#include "SSLoop.h"
#include "SSQModel.h"

SSLoop::SSLoop()
{
}

SSLoop::~SSLoop()
{
}

SSLath *SSLoop::Subdivision(vector<SSLath*> &vec, SSLath *start)
{
  SSLoop::Subdivide(vec, start);
  return start->nl();
}

SSVector SSLoop::calcNormal(SSLath *face)
{
  SSLath *tmp = face, *tmp1, *tmp2, *last;
  SSVector v1, v2, norm;
  int i= 0, n = 0, countNonSmooth=0, isConical = 0, isCorner = 0, isCusp = 0;

  // Check for boundary.
  tmp1 = face;
  do {
    last = tmp1;
    tmp1 = tmp1->ccwv();
  } while(tmp1 && tmp1 != face);
  if(tmp1)
    last = NULL;

  tmp1 = face;
  do
```

```
    {
      n++;
      if(tmp1→cwknotspc() ≠ 1.0) {
        if(tmp1→cwknotspc() == 0.0)
          countNonSmooth++;
        else if(tmp1→cwknotspc() == 2.0)
          isConical = 1;
        else if(tmp1→cwknotspc() == 3.0)
          isCusp = 1;
        else if(tmp1→cwknotspc() == 4.0)
          isCorner = 1;
      }
      tmp1 = tmp1→ccwv();
    }
  while(tmp1 && tmp1 ≠ face);
  if(!tmp1) {
    tmp1 = face;
    do
      {
        n++;
        if(tmp1→ccwknotspc() ≠ 1.0) {
          if(tmp1→ccwknotspc() == 0.0)
            countNonSmooth++;
          else if(tmp1→ccwknotspc() == 2.0)
            isConical = 1;
          else if(tmp1→ccwknotspc() == 3.0)
            isCusp = 1;
          else if(tmp1→ccwknotspc() == 4.0)
            isCorner = 1;
        }
        tmp1 = tmp1→cwv();
      }
    while(tmp1);
  }

  if(countNonSmooth > 2 || isCorner) {
    // CORNER CASE

    tmp1 = face;
    while(tmp1→cwv() ≠ face && tmp1→ccwknotspc() ≠ 0.0 && tmp1→ccwknotspc() ≠ 4.0)
      tmp1 = tmp1→cwv();
    v1 = SSVector(tmp1→ccwf()→corner(), face→corner());
    tmp2 = tmp1;

    tmp1 = face;
    while(tmp1→ccwv() ≠ face && tmp1→cwknotspc() ≠ 0.0 && tmp1→cwknotspc() ≠ 4.0)
      tmp1 = tmp1→ccwv();
    v2 = SSVector(tmp1→cwf()→corner(), face→corner());

    if(v1.len() < 1e−10 || v2.len() < 1e−10 || cross(v1,v2).len() < 1e−10) {
      v1 = SSVector(face→ccwf()→corner(), face→corner());
      v2 = SSVector(face→cwf()→corner(), face→corner());
    }
```

```
} else if(countNonSmooth == 2) {
  // CREASE CASE

  SSVertex sv1, sv2;
  vector<SSVertex> v;
  double w;

  v.push_back(face->corner());
  tmp1 = face;
  while(tmp1 && tmp1->ccwknotspc() != 0.0)
    {
      tmp2 = tmp1;
      tmp1 = tmp1->cwv();
    }
  if (!tmp1) {
    v.push_back(tmp2->ccwf()->corner());
    tmp1 = tmp2;
  } else {
    v.push_back(tmp1->ccwf()->corner());
  }
  do {
    v.push_back(tmp1->cwf()->corner());
    tmp1 = tmp1->ccwv();
  } while(tmp1 && tmp1->cwknotspc() != 0.0);
  if(tmp1) {
    v.push_back(tmp1->cwf()->corner());}

  v1 = SSVector(v[1],v[v.size()-1]);
  switch(v.size()) {
  case 3:
    v2 = SSVector(v[1],v[0]) + SSVector(v[2],v[0]);
    break;
  case 5:
    v2 = (SSVector(v[2]) + SSVector(v[3])) * 0.5
      - (SSVector(v[1]) + SSVector(v[0])*2.0 + SSVector(v[4])) * 0.25;
    break;
  case 4:
    // Subdivision matrix defective - no set of spanning eigenvectors
    // Best solution: analyze with general eigenvectors.
    v2 = SSVector(v[1],v[0]) + SSVector(v[2],v[0]) + SSVector(v[3],v[0]);
    break;
  default:
    w = sin(c_PI/((double) (v.size()-2)))/(2.0*cos(c_PI/((double) (v.size()-2)))-2.0);
    v2 = w*(v[1]+v[v.size()-1]);
    for(i = 2; i<v.size()-1; i++)
      v2 += sin(c_PI*((double) (i-1))/(double)(v.size()-2)) * v[i];
    break;
  }
} else if(isConical) {
  // CONICAL CASE

  // No tangent plane exists at central vertex
  // We use non-interpolated polygonal normal.
```

```
    v1 = SSVector(face→ccwf()→corner(), face→corner());
    v2 = SSVector(face→cwf()→corner(), face→corner());

  } else if(isCusp) {
    // CUSP CASE

    // Tangent plane collapses to tangent line.
    v1 = SSVector(face→corner());
    tmp1 = face;
    do {
      v1 −= SSVector(tmp1→ccwf()→corner()) ∗ (1.0 / ((double) n));
      tmp1 = tmp1→ccwv();
    } while(tmp1 && tmp1 ≠ face);
    v2 = SSVector(tmp1→ccwf()→corner(), tmp1→ccwf()→ccwf()→corner());

  } else {
    // SMOOTH OR DART CASE

    tmp1 = face;
    do {
      SSVector vi(tmp1→ccwf()→corner());

      v1 += vi ∗ cos((2.0∗c_PI∗(double)i)/(double)n);
      v2 += vi ∗ sin((2.0∗c_PI∗(double)i)/(double)n);
      i++;
      tmp1 = tmp1→cwv();
    } while(tmp1 && tmp1 ≠ face);
    if(!tmp1) {
      tmp1 = face;
      do {
        SSVector vi(tmp1→cwf()→corner());

        v1 += vi ∗ cos((2.0∗c_PI∗(double)i)/(double)n);
        v2 += vi ∗ sin((2.0∗c_PI∗(double)i)/(double)n);
        i++;
        tmp1 = tmp1→ccwv();
      } while(tmp1 && tmp1 ≠ face);
    }
    v1 = −v1;

  }
  norm = cross(v1, v2);
  norm.normalize();
  return norm;
}

void SSLoop::Subdivide(vector<SSLath∗> &vec, SSLath ∗start)
{
  // If null or already subdivided just return
  if(start == 0 || start→isvisited())
    return;


```

```
vector<SSVertex> vs, evs(3, SSVertex()), nvs;
vector<SSLath *> els(4, (SSLath *) 0);

SSLath *nf=0, *tmp1=0, *tmp2=0, *tmp3=0, *last=0;
SSVertex ny, tmp;

int i ,k, k2, neighbourcountNonSmooth=0, neighbourIsCorner=0, countNonSmooth=0,
  isConical=0, isCusp=0, isCorner=0, regularValency=6, neighbourRegularValency=6;
CoefficientType cosinus, coeff;

// Check for boundary. If found, move start to clockwise-most non-null lath
tmp1 = start;
do
  {
    tmp2 = tmp1;
    tmp1 = tmp1->cwv();
  }
while(tmp1 && tmp1 != start);
if(!tmp1) {
  do {
    last = start;
    start = start->ccwv();
  } while(start);
  start = tmp2;
  regularValency = 4;
}

// Collect the vertices around our start point
// Count number of non-smooth edges incident
// Check for conical point
tmp1 = start;
do
  {
    tmp1->visited(true);
    vs.push_back(tmp1->ccwf()->corner());

    if(tmp1->ccwknotspc() != 1.0) {
      if(tmp1->ccwknotspc() == 0.0)
        countNonSmooth++;
      else if(tmp1->ccwknotspc() == 2.0)
        isConical = 1;
      else if(tmp1->ccwknotspc() == 3.0)
        isCusp = 1;
      else if(tmp1->ccwknotspc() == 4.0)
        isCorner = 1;
    }

    tmp1 = tmp1->ccwv();
  }
while(tmp1 && tmp1 != start);
// If on the boundary remember last neighbour
if(last) {
  vs.push_back(last->cwf()->corner());
```

```
    if(last→cwknotspc() ≠ 1.0) {
      if(last→cwknotspc() == 0.0)
        countNonSmooth++;
      else if(last→cwknotspc() == 2.0)
        isConical = 1;
      else if(last→cwknotspc() == 3.0)
        isCusp = 1;
      else if(last→cwknotspc() == 4.0)
        isCorner = 1;
    }
  }

  k = vs.size();
  tmp = SSVertex();

  // Find appropriate vertex rule. Options are:
  // CREASE − on a crease
  // CORNER − endpoint of two or more creases
  // CONICAL − conical point
  // CUSP    − cusp point
  // DART    − end point of one crease
  // SMOOTH − smooth point

  if(countNonSmooth > 2 || isCorner) {
    // CORNER CASE

    isCorner = 1;
    nvs.push_back(start→corner());
  } else if(countNonSmooth == 2) {
    // CREASE CASE

    tmp1 = start;
    do {
      if(tmp1→ccwknotspc() == 0.0)
        tmp = tmp + 0.125 ∗ tmp1→ccwf()→corner();
      tmp1 = tmp1→ccwv();
    } while(tmp1 && tmp1 ≠ start);
    // If on boundary remember last edge
    if(last) {
      if(last→cwknotspc() == 0.0)
        tmp = tmp + 0.125 ∗ last→cwf()→corner();
    }
    tmp = tmp + 0.75 ∗ start→corner();
    nvs.push_back(tmp);
  } else if(isConical) {
    // CONICAL CASE

    coeff = ((2.0/(8.0∗(CoefficientType) k)) ∗
              (1.0 − cos(2.0∗c_PI/(CoefficientType) k)));

    // Calculate the new vertex
    for(i=0; i<k; i++)
```

```
        tmp = tmp + vs[i]*coeff;

      tmp = tmp + start->corner()*(1.0 - (CoefficientType)k*coeff);
      nvs.push_back(tmp);

    } else if(isCusp) {
      // CUSP CASE

      // New vertex identical to old
      nvs.push_back(start->corner());

    } else {
      // SMOOTH OR DART CASE (DART IF countNonSmooth==1)

      // Calculate new even vertex coefficient
      cosinus = 0.375 + 0.25 * cos((2.0*c_PI)/(CoefficientType)k);
      coeff = (CoefficientType) (1.0/(CoefficientType)k) * (0.625 - cosinus*cosinus);

      // Calculate the new vertex
      for(i=0; i<k; i++)
        tmp = tmp + vs[i]*coeff;

      tmp = tmp + start->corner()*(1.0 - (CoefficientType)k*coeff);
      nvs.push_back(tmp);
    }

    // Calculate the new edge vertices. Options:
    // SMOOTH - smooth edge, possibly incident upon dart, conical or cusp vertex
    // ORDINARY CREASE - edge between two ordinary vertices,
    //                   or between two extraordinary vertices
    // SPECIAL CREASE - edge between ordinary and extraordinary vertices

    tmp1 = start;
    do
      {
        k2 = neighbourIsCorner = neighbourcountNonSmooth = 0;
        neighbourRegularValency = 6;
        tmp2 = tmp1->ccwf();
        do {
          // Neighbour is on the boundary if this is true
          if(!tmp2->ccwv())
            {
              neighbourRegularValency = 4;
              tmp3 = tmp1->ccwf();
              do
                {
                  k2++;
                  if(tmp3->ccwknotspc() == 0.0)
                    neighbourcountNonSmooth++;
                  if(tmp3->ccwknotspc() == 4.0)
                    neighbourIsCorner = 1;
                  tmp3 = tmp3->cwv();
                }
```

```
            while(tmp3);
          }

        k2++; // Valency of neighbour vertex

        if(tmp2→cwknotspc() == 0.0)
          neighbourcountNonSmooth++;
        if(tmp2→cwknotspc() == 4.0)
          neighbourIsCorner = 1;
        tmp2 = tmp2→ccwv();
      } while(tmp2 && tmp2 ≠ tmp1→ccwf());
      if(neighbourcountNonSmooth > 2)
        neighbourIsCorner = 1;

      if((neighbourcountNonSmooth==1 || isConical || isCusp || countNonSmooth==1
          || tmp1→ccwknotspc()==1.0) && tmp1→ccwf()→ccwv()) {
        // SMOOTH CASE

        tmp = tmp1→ccwf()→corner()*0.375;
        tmp = tmp + tmp1→cwf()→corner()*0.125;
        tmp = tmp + tmp1→ccwf()→ccwv()→ccwf()→ccwf()→corner()*0.125;
        tmp = tmp + tmp1→corner()*0.375;
        nvs.push_back(tmp);
      } else if((tmp1→ccwknotspc() == 4.0 || tmp1→ccwknotspc() == 0.0) &&
                (isCorner || k≠regularValency) && k2==neighbourRegularValency &&
                !neighbourIsCorner) {
        // SPECIAL CREASE CASE 1 − central vertex extraordinary

        tmp = tmp1→corner() * 0.375 + tmp1→ccwf()→corner() * 0.625;
        nvs.push_back(tmp);
      } else if((tmp1→ccwknotspc() == 4.0 || tmp1→ccwknotspc() == 0.0) && !isCorner &&
                k==regularValency && (neighbourIsCorner || k2≠neighbourRegularValency)) {
        // SPECIAL CREASE CASE 2 − adjacent vertex extraordinary

        tmp = tmp1→corner() * 0.625 + tmp1→ccwf()→corner() * 0.375;
        nvs.push_back(tmp);
      } else {
        // ORDINARY CREASE

        tmp = tmp1→corner() * 0.5 + tmp1→ccwf()→corner() * 0.5;
        nvs.push_back(tmp);
      }
      tmp1 = tmp1→ccwv();
    }
  while(tmp1 && tmp1 ≠ start);
  // If on boundary remember last edge
  if(last) {
    tmp1 = last;
    k2 = neighbourIsCorner = neighbourcountNonSmooth = 0;
    neighbourRegularValency = 4;
    tmp2 = tmp1→cwf();
    k2++;
    if(tmp2→ccwknotspc() == 0.0)
```

```
      neighbourcountNonSmooth++;
    if(tmp2→ccwknotspc() == 4.0)
      neighbourIsCorner = 1;
    do {
      k2++; // Valency of neighbour vertex

      if(tmp2→cwknotspc() == 0.0)
        neighbourcountNonSmooth++;
      if(tmp2→cwknotspc() == 4.0)
        neighbourIsCorner = 1;
      tmp2 = tmp2→ccwv();
    } while(tmp2);
    if(neighbourcountNonSmooth > 2)
      neighbourIsCorner = 1;

    if((tmp1→cwknotspc() == 4.0 || tmp1→cwknotspc() == 0.0)
       && (isCorner || k≠regularValency) && k2==neighbourRegularValency
       && !neighbourIsCorner) {
      // SPECIAL CREASE CASE 1 − central vertex extraordinary

      tmp = tmp1→corner() ∗ 0.375 + tmp1→cwf()→corner() ∗ 0.625;
      nvs.push_back(tmp);
    } else if((tmp1→cwknotspc() == 4.0 || tmp1→cwknotspc() == 0.0)
              && !isCorner && k==regularValency
              && (neighbourIsCorner || k2≠neighbourRegularValency)) {
      // SPECIAL CREASE CASE 2 − adjacent vertex extraordinary

      tmp = tmp1→corner() ∗ 0.625 + tmp1→cwf()→corner() ∗ 0.375;
      nvs.push_back(tmp);
    } else {
      // ORDINARY CREASE

      tmp = tmp1→corner() ∗ 0.5 + tmp1→cwf()→corner() ∗ 0.5;
      nvs.push_back(tmp);
    }
  }

  // Build the new faces around the start vertex and connect to the next level
  tmp1 = start;
  evs [0] = nvs[0];
  i=1; k = vs.size ();
  do
    {
      evs [1] = nvs[i ];
      evs [2] = nvs[i%k+1];
      nf = SSQModel::Instance()→BuildFace(evs, vec);
      nf→setcwknotspc(tmp1→cwknotspc());
      nf→setccwknotspc(tmp1→ccwknotspc());
      if(tmp1→cwknotspc() == 0.0 || tmp1→cwknotspc() == 4.0)
        nf→cwf()→setccwknotspc(0.0);
      if(tmp1→ccwknotspc() == 0.0 || tmp1→ccwknotspc() == 4.0)
        nf→ccwf()→setcwknotspc(0.0);
      tmp1→nextlevel(nf→ccwf());
```

```
      tmp1 = tmp1→ccwv();
      i++;
    }
  while(tmp1 && tmp1 ≠ start);

  // Connect the new faces
  tmp1 = start;
  do
    {
      if(tmp1→ccwv())
        SSQModel::Instance()→ConnectFaces(tmp1→nl(), 2, tmp1→ccwv()→nl(), 0);
      tmp1 = tmp1→ccwv();
    }
  while(tmp1 && tmp1 ≠ start);


  // Start subdividing
  tmp1 = start;
  do
    {
      SSLoop::Subdivide(vec, tmp1→ccwf());
      tmp1 = tmp1→ccwv();
    }
  while(tmp1 && start ≠ tmp1);
  if(last)
    SSLoop::Subdivide(vec, last→cwf());

  // Build and Connect new face in the middle
  tmp1 = start;
  do
    {
      if (!tmp1→nl()→ccwf()→ccwv() && tmp1→ccwf()→nl() && tmp1→ccwf()→ccwf()→nl())
        {
          evs[0] = tmp1→nl()→corner();
          evs[1] = tmp1→ccwf()→nl()→corner();
          evs[2] = tmp1→ccwf()→ccwf()→nl()→corner();
          nf = SSQModel::Instance()→BuildFace(evs, vec);

          SSQModel::Instance()→ConnectFaces(tmp1→nl(), 1, nf, 0);
          SSQModel::Instance()→ConnectFaces(tmp1→ccwf()→nl(), 1, nf, 1);
          SSQModel::Instance()→ConnectFaces(tmp1→ccwf()→ccwf()→nl(), 1, nf, 2);
        }

      tmp1 = tmp1→ccwv();
    }
  while(tmp1 && tmp1 ≠ start);

}
```

Listing A.21: SSQuadraticNURSS.h.

```
#if !defined _SSQuadraticNURSS_HEADER
```

```
#define _SSQuadraticNURSS_HEADER

#include "SSMaster.h"
#include "SSSubdivideMethods.h"

// Implements Quadratic NURSS subdivision
class SSQuadraticNURSS : public SSSubdivideMethods
{
public:
  virtual SSLath *Subdivision(vector<SSLath*> &vec, SSLath *start);

  SSQuadraticNURSS();

  virtual ~SSQuadraticNURSS();

private:
  void Subdivide(vector<SSLath*> &vec, SSLath *start);
};

#endif
```

Listing A.22: SSQuadraticNURSS.cpp.

```
#include <cmath>
#include "SSQuadraticNURSS.h"
#include "SSQModel.h"
#include "SSCornerLath.h"

SSQuadraticNURSS::SSQuadraticNURSS()
{
}

SSQuadraticNURSS::~SSQuadraticNURSS()
{
}

SSLath *SSQuadraticNURSS::Subdivision(vector<SSLath*> &vec, SSLath *start)
{
  SSQuadraticNURSS::Subdivide(vec, start);
  return start->nl();
}

void SSQuadraticNURSS::Subdivide(vector<SSLath*> &vec, SSLath *start)
{
  // If null or already subdivided just return
  if(!start || start->nl())
    return;


  vector<SSVertex> vs, evs(4, SSVertex()), nvs;
  vector<SSLath *> els(4, (SSLath *) 0);
  vector<CoefficientType> coeff;
```

```
vector<KnotSpacingType> knots, nknots;

SSLath ∗nf=0, ∗tmp1=0, ∗tmp2=0, ∗tmp3=0;

int i , j , c , k;
SSVertex V, S, ny;
KnotSpacingType denom = 0, t;


SSQModel::Instance()→FaceVertices(vs, knots, start);
k = vs.size ();

// Calculate common coefficients
tmp1 = start;
for(i=0; i<k; i++)
  {
    t = tmp1→knotspc(−1,0,0) ∗ tmp1→knotspc(1,0,0);
    denom += t;
    V += vs[i]∗t;
    coeff .push_back(1.0+2.0∗cos(2.0∗((double) i)∗c_PI/((double) k)));
    tmp1 = tmp1→ccwf();
  }
if(abs(denom) ≥ 1e−10)
  V /= denom;


tmp1 = start;
for(i=0; i<k; i++)
  {
    S = SSVertex();
    for(j=0; j<k; j++)
      {
        c = (i+j)%k;
        S += coeff[j ] ∗ vs[c];
      }
    if(abs(denom) < 1e−10)
      // If denominator zero → point is not changed
      nvs.push_back(vs[i]);
    else {
      // Sederberg et al. eq (10)
      t = tmp1→knotspc(1,2,0) ∗ tmp1→knotspc(3,2,0)
        + tmp1→knotspc(−1,−2,0) ∗ tmp1→knotspc(−3,−2,0);

      ny = 0.5∗(V+vs[i]) + t ∗ (−k ∗ vs[i ] + S)/(8.0∗denom);
      nvs.push_back(ny);
    }
    tmp1 = tmp1→ccwf();
  }

// Calculate new knot spacings
for(i=0; i<k; i++)
  {
    nknots.push_back(knots[2∗i]∗0.5);
```

```
      nknots.push_back(knots[2∗i+1]∗0.5);
    }

nf = SSQModel::Instance()→BuildFace(nvs, nknots, vec);

// Connect to subdivided level
start→nextlevel(nf);

tmp1 = start;
tmp2 = nf;
do
  {
    tmp1→nextlevel(tmp2);
    tmp1 = tmp1→ccwf();
    tmp2 = tmp2→ccwf();
  }
while(tmp1 ≠ start);

// Start subdivision chain reaction
tmp1 = start;
do
  {
    SSQuadraticNURSS::Subdivide(vec, tmp1→ccwv());
    tmp1 = tmp1→ccwf();
  }
while(start ≠ tmp1);

// Build and connect faces corresponding to old edges
tmp1 = start;
do
  {
    nknots = vector<KnotSpacingType>(8,0);
    if(tmp1→ccwv() && !tmp1→nl()→ccwv())
      {
        tmp2 = tmp1;
        evs[3] = tmp2→nl()→corner();
        els[0] = tmp2→nl();
        nknots[6] = 0.25 ∗ (tmp2→knotspc(0,−1,0) + tmp2→knotspc(0,1,1));
        nknots[7] = 0.5 ∗ tmp2→knotspc(0,1,0);
        tmp2 = tmp2→ccwv();
        evs[2] = tmp2→nl()→corner();
        nknots[4] = 0.5 ∗ tmp2→knotspc(0,−1,0);
        nknots[5] = nknots[6];
        tmp2 = tmp2→ccwf();
        evs[1] = tmp2→nl()→corner();
        els[1] = tmp2→nl();
        nknots[2] = 0.25 ∗ (tmp2→knotspc(0,−1,0) + tmp2→knotspc(0,1,1));
        nknots[3] = 0.5 ∗ tmp2→knotspc(0,1,0);
        tmp2 = tmp2→ccwv();
        evs[0] = tmp2→nl()→corner();
        nknots[0] = 0.5 ∗ tmp2→knotspc(0,−1,0);
        nknots[1] = nknots[2];

```

```
            tmp2 = SSQModel::Instance()→BuildFace(evs, nknots, vec);

            SSQModel::Instance()→ConnectFaces(els[0], 0, tmp2, 0);
            SSQModel::Instance()→ConnectFaces(els[1], 0, tmp2, 2);
          }
        tmp1 = tmp1→ccwf();
      }
    while(tmp1 ≠ start);

    // Build and connect faces corresponding to old vertices
    tmp1 = start;

    do
      {
        if(tmp1→ccwv() && !tmp1→nl()→ccwv()→ccwv())
          { // If not already connected, build & connect face
            tmp2 = tmp1;
            vs.clear ();
            nknots.clear ();
            do
              {
                tmp3 = tmp2→nl();
                if (!tmp3 || !tmp3→ccwv())
                  {
                    /* All faces sharing edges with f have been subdivided in the
                     * recursion, but faces sharing only vertices with f may not
                     * have been. Solution: Just skip affected vertex -- it will
                     * be processed further down the recursion.
                     * Unknown why second condition necessary!!!
                     **/
                    goto loop;
                  }
                vs.push_back(tmp3→corner());
                nknots.push_back(tmp3→cwv()→ccwknotspc());
                nknots.push_back(tmp3→ccwv()→cwknotspc());

                tmp2 = tmp2→ccwv();

                if (!tmp2)
                  goto loop;
              }
            while(tmp1 ≠ tmp2);

            tmp3 = SSQModel::Instance()→BuildFace(vs, nknots, vec);

            do
              {
                if(tmp2→nl()→ccwv())
                  SSQModel::Instance()→ConnectFaces(tmp2→nl()→ccwv(), 0, tmp3, 1);

                tmp2 = tmp2→ccwv();
                tmp3 = tmp3→ccwf();
              }
```

```
            while(tmp1 ≠ tmp2);

          }
      loop:
        tmp1 = tmp1→ccwf();
      }
  while(tmp1 ≠ start);
}
```

Listing A.23: SSCubicNURSS.h.

```
#if !defined _SSCubicNURSS_HEADER
#define _SSCubicNURSS_HEADER

#include "SSMaster.h"
#include "SSSubdivideMethods.h"

// Implements Cubic NURSS subdivision
class SSCubicNURSS : public SSSubdivideMethods
{
public:
  virtual SSLath *Subdivision(vector<SSLath*> &vec, SSLath *start);

  SSCubicNURSS();

  virtual ~SSCubicNURSS();

private:
  void Subdivide(vector<SSLath*> &vec, SSLath *start);
};

#endif
```

Listing A.24: SSCubicNURSS.cpp.

```
#if defined(WIN32)
#pragma warning( disable : 4786 )
#endif

#include <cmath>
#include "SSCubicNURSS.h"
#include "SSQModel.h"
#include "SSCornerLath.h"

SSCubicNURSS::SSCubicNURSS()
{
}

SSCubicNURSS::~SSCubicNURSS()
{
}
```

```
SSLath *SSCubicNURSS::Subdivision(vector<SSLath*> &vec, SSLath *start)
{
  Subdivide(vec, start);
  if(vec.size()>0)
    return vec[0];
  return 0;
}

void SSCubicNURSS::Subdivide(vector<SSLath*> &vec, SSLath *start)
{
  if(start == 0 || start->isvisited())
    return;

  vector<SSVertex> vs, evs, nvs(4), fvs, M;
  vector<SSLath *> els(4, (SSLath *) 0);
  vector<CoefficientType> coeff;
  SSVertex tmp, ny, Mt;
  SSLath *nf=0, *tmp1=0, *tmp2=0;
  int i, c=0, k;
  vector<KnotSpacingType> w, knots, nknots(8);
  KnotSpacingType wi, ws=0, aij, aji, ad, ci, cj, mi, fi;

  //New Face Vertices
  tmp1 = start;
  do
    {
      ws=0;
      tmp1->visited(true);
      vs.clear();
      SSQModel::Instance()->FaceVertices(vs, tmp1);
      k = vs.size();
      ny = SSVertex();

      w.clear();
      // Calculate coefficients w_i, and sum of w_i* P_i
      tmp2 = tmp1;
      tmp = SSVertex();
      for(i=0; i<k; i++)
        {
          // See formula (12)+(11) in Sederberg et al.
          wi = (tmp2->knotspc(1,0,0) +
                tmp2->knotspc(1,0,2) +
                tmp2->knotspc(1,0,-2) +
                tmp2->knotspc(-2,-1,0) +
                tmp2->knotspc(-2,-1,2) +
                tmp2->knotspc(-2,-1,-2)) *
            (tmp2->knotspc(-1,0,0) +
             tmp2->knotspc(-1,0,2) +
             tmp2->knotspc(-1,0,-2) +
             tmp2->knotspc(2,1,0) +
             tmp2->knotspc(2,1,2) +
             tmp2->knotspc(2,1,-2));
```

```
          w.push_back(wi);
          ws += wi;
          tmp += wi * vs[i];
          tmp2 = tmp2→ccwf();
        }

      if(ws < 1e−10)
        {
          tmp = SSVertex();
          for(i=0; i<k; i++)
            {
              tmp += vs[i];
            }
          tmp /= (CoefficientType) k;
          fvs.push_back(tmp);
        }
      else
        fvs.push_back(tmp/ws);

      // If ccwv does not exist, we are on the boundary and there is nothing to do,
      // except the recursive calls on the neighbouring vertices.
      if (!tmp1→ccwv())
        {
          do
            {
              tmp1→visited(true);
              Subdivide(vec, tmp1→ccwf());
              tmp1 = tmp1→cwv();
            }
          while(tmp1);

          return;
        }
      tmp1 = tmp1→ccwv();
    }
  while(tmp1 ≠ start);

  //New Edge Vertices
  tmp1 = start;
  k = fvs.size();
  c = 0;
  do
    {
      // Compute coefficients, see Sederberg et al. eqns. (14−15)
      ad = 2.0 * (tmp1→knotspc(1,0,1) +
                  tmp1→knotspc(0,1,−1) +
                  tmp1→knotspc(1,0,−1) +
                  tmp1→knotspc(0,1,1));

      if(ad < 1e−10) {
        aij = aji = 0.0;
      } else {
        aij = (tmp1→knotspc(1,0,1) + tmp1→knotspc(0,1,−1)) / ad;
```

```
        aji = (tmp1->knotspc(0,1,1) + tmp1->knotspc(1,0,-1)) / ad;
      }

      ci = tmp1->knotspc(1,0,0) + tmp1->knotspc(1,0,2) + tmp1->knotspc(1,0,-2);
      cj = tmp1->knotspc(0,1,0) + tmp1->knotspc(0,1,2) + tmp1->knotspc(0,1,-2);

      if(ci+cj < 1e-10)
        Mt = (tmp1->corner() + tmp1->cwf()->corner())*0.5;
      else
        Mt = (tmp1->corner() * ci + tmp1->cwf()->corner() * cj) / (ci+cj);

      M.push_back(Mt);

      // Sederberg et al. eq (13)
      ny = Mt * (1.0-aij-aji) + fvs[(c+1)%k] * aij + fvs[c] * aji;

      evs.push_back(ny);
      tmp1 = tmp1->ccwv();
      c++;
    }
  while(tmp1 != start);

  //New Center Vertex
  tmp = SSVertex();
  tmp1 = start;
  ad = 0;
  i = 0;
  do
    {
      // Sederberg et al. eqs (16-19)
      mi = (tmp1->knotspc(0,1,1) + tmp1->knotspc(0,1,-1)) *
        (tmp1->knotspc(0,1,2) + tmp1->knotspc(0,1,-2)) * 0.5;

      fi = tmp1->knotspc(0,1,1) * tmp1->knotspc(0,-1,-1);

      ad += mi+fi;
      tmp += mi * M[i] + fi * fvs[i];
      tmp1 = tmp1->ccwv();
      i++;
    }
  while(tmp1 != start);

  if(ad < 1e-10)
    ny = start->corner();
  else
    ny = ((i-3.0)/i) * start->corner() + (3.0 * tmp) / (((double) i) * ad);

  //Build the faces adjacent to the Center vertex
  nvs[0] = ny;
  tmp1 = start;
  for(i = 0; i < k; i++)
    {
      nvs[1] = evs[(k-1+i) % k];
```

```
      nvs[2] = fvs[i];
      nvs[3] = evs[i];

      nknots[0] = 0.5 ∗ tmp1→knotspc(0,1,0);
      nknots[1] = 0.5 ∗ tmp1→knotspc(0,−1,0);
      nknots[2] = nknots[1];
      nknots[3] = 0.25 ∗ (tmp1→knotspc(0,1,0) + tmp1→knotspc(−1,−2,0));
      nknots[4] = nknots[3];
      nknots[5] = 0.25 ∗ (tmp1→knotspc(0,−1,0) + tmp1→knotspc(1,2,0));
      nknots[6] = nknots[5];
      nknots[7] = nknots[0];

      nf = SSQModel::Instance()→BuildFace(nvs, nknots, vec);

      tmp1→nextlevel(nf→ccwf()→ccwf());
      tmp1 = tmp1→ccwv();
    }

  //Connect the faces adjacent to the Center vertex
  tmp1 = start;
  do
    {
      SSQModel::Instance()→ConnectFaces(tmp1→nl(), 2, tmp1→ccwv()→nl(), 3);
      tmp1 = tmp1→ccwv();
    }
  while(tmp1 ≠ start);

  // Start subdivision chain reaction
  tmp1 = start;
  do
    {
      Subdivide(vec, tmp1→ccwf());
      tmp1 = tmp1→ccwv();
    }
  while(tmp1 ≠ start);


  //Connect new faces
  tmp1 = start;
  do
    {
      if(tmp1→ccwf()→nl())
        {
          SSQModel::Instance()→ConnectFaces(tmp1→nl(), 0, tmp1→ccwf()→nl(), 1);
        }
      if(tmp1→cwf()→nl())
        {
          SSQModel::Instance()→ConnectFaces(tmp1→nl(), 1, tmp1→cwf()→nl(), 0);
        }
      tmp1 = tmp1→ccwv();
    }
  while(tmp1 ≠ start);
}
```

Listing A.25: SSReader.h.

```
#if !defined _SSReader_HEADER
#define _SSReader_HEADER

#include <string>
#include "SSModel.h"
#include "SSLath.h"

// Base class for model readers
class SSReader
{
public:
  // Reads a model from the file filename, storing one lath pointer in faces
  // for each face in the polygonal model
  virtual void read(const string &filename, vector<SSLath*> &faces) = 0;

  // Destructor for clean-up
  virtual ~SSReader() {}
};

#endif
```

Listing A.26: SSMdlReader.h.

```
#if !defined _SSMdlReader_HEADER
#define _SSMdlReader_HEADER

#include <fstream>
#include <string>
#include <utility>
#include <map>
#include <vector>
#include "SSLath.h"
#include "SSVertex.h"
#include "SSReader.h"

// Reader for the MDL format
class SSMdlReader : SSReader
{
 private:
  char buf[1024];
  string modedata;
  double coorddata1, coorddata2, coorddata3;
  int numdata, vnumdata;
  map<int, vector<int> > facedata;
  map<int, SSVertex> vertices;
  vector<pair<int, int> > connections;
  map<int, vector<double> > knotspacings;
```

```
  int lineno;
  int stat, pushbackflag;
  enum status { IO_GOOD = 0, IO_EOF = 1, IO_FAIL = 2 };
  status pushbackstat;

  void pushback();
  status readLine(ifstream &in);
  void error(string msg, int io);
  void readVertices(ifstream &in);
  void readFaces(ifstream &in);
  void readConnections(ifstream &in);
  void readKnotSpacings(ifstream &in);
  pair<int, int> findEdgeNo(const vector<int> &vec1, const vector<int> &vec2, int f1, int f2);
  void buildModel(vector<SSLath*> &faces);

 public:
  virtual void read(const string &filename, vector<SSLath*> &faces);
  virtual ~SSMdlReader() {}
};

#endif
```

Listing A.27: SSMdlReader.cpp.

```
// Due to incompatiblities between Windows and GNU libraries this file
// can be compiled in three versions depending on preprocessor constants

#if defined(WIN32)
#pragma warning( disable : 4786 )
#endif

#include <iostream>
#include <strstream>
#include <cstdio>
#include "SSMdlReader.h"
#include "SSQModel.h"
#include "SSTypes.h"

void SSMdlReader::pushback()
{
  pushbackflag = 1;
}

SSMdlReader::status SSMdlReader::readLine(ifstream &in)
{
  if(pushbackflag) {
    pushbackflag = 0;
    return pushbackstat;
  }
  do {
    in.getline(buf, 1024);
    lineno++;
```

```
  } while(in.good() && (buf[0]=='\0' || buf[0]=='#'));
  if(in.eof()) {
    pushbackstat = IO_EOF;
    return pushbackstat;
  } if(in.fail()) {
    pushbackstat = IO_FAIL;
    return pushbackstat;
  }
  pushbackstat = IO_GOOD;
  return pushbackstat;
}

void SSMdlReader::error(string msg, int io)
{
  if(io)
    cerr << "I/O error on line " << lineno << ":\n" << msg << endl;
  else
    cerr << "Error:\n" << msg << endl;
  exit(EXIT_FAILURE);
}

void SSMdlReader::readVertices(ifstream &in)
{
  while(true) {
    stat = readLine(in);
    if(stat) {
      pushback();
      return;
    }

#if defined(WIN32)
    string test(buf);
    if(test == "vertices" || test == "faces"
        || test == "connections" || test == "knotspacings")
      {
        modedata = test;
        pushback();
        return;
      }

    istrstream is(buf);
    is >> numdata >> coorddata1 >> coorddata2 >> coorddata3;
#elif defined(__SCAN_BUG_WORKAROUND)
    int n = sscanf(buf, "%d %le %le %le", &numdata, &coorddata1, &coorddata2, &coorddata3);
    if(n != 4) {
      pushback();
      return;
    }
#else
    istrstream is(buf);
    is >> numdata >> coorddata1 >> coorddata2 >> coorddata3;
    if(is.fail() || is.eof()) {
      is.clear();
```

```
        is ≫ modedata;
        if(is . fail ()) {
          error(string("Illegal file format.\n"
                       "Vertices must be specified as a line\n"
                       "  <vno> <x> <y> <z>\n"
                       "where <vno> is the vertex number, and <x> <y> <z> \n"
                       "are the coordinates"), 1);
        }
        pushback();
        return;
      }
#endif
      vertices . insert (make_pair(numdata, SSVertex(coorddata1, coorddata2, coorddata3)));
  }
}

void SSMdlReader::readFaces(ifstream &in)
{
  map<int, vector<int> >::iterator it;
  int count;
  while(true) {
    count = 0;
    stat = readLine(in);
    if(stat) {
      pushback();
      return;
    }
    istrstream is(buf);
#if defined(WIN32)
    string test(buf);
    if(test == "vertices"|| test == "faces"
        || test == "connections"|| test == "knotspacings")
      {
        modedata = test;
        pushback();
        return;
      }

    is ≫ numdata;
#elif defined(__SCAN_BUG_WORKAROUND)
    int n = sscanf(buf, "%i", &numdata);
    if(n ≠ 1) {
      pushback();
      return;
    }
#else
    is ≫ numdata;
    if(is . fail ()) {
      is . clear ();
      is ≫ modedata;
      if(is . fail ())
        error("Faces must be specified as a line\n"
              "  <fno> <vno> <vno> <vno> [<vno> ...]\n"
```

```
            "where <fno> is the face number, and <vno> is a "
            "valid vertex number.\n", 1);
      pushback();
      return;
    }
#endif

#if defined(WIN32)
    while(is >> vnumdata)
      {
#elif defined(__SCAN_BUG_WORKAROUND)
    char *pos = buf + strspn(buf, " ");
    pos = pos + strcspn(pos, " ");
    while(true) {
      int n = sscanf(pos, "%i", &vnumdata);
      if(n!=1) {
        break;
      }
      pos = pos + strspn(pos, " ");
      pos = pos + strcspn(pos, " ");
#else
    while(true) {
      is >> vnumdata;
      if(is.eof()) {
        if(count < 3)
          error("Faces must be specified as a line\n"
                "  <fno> <vno> <vno> <vno> [<vno> ...]\n"
                "where <fno> is the face number, and <vno> is a "
                "valid vertex number.\n", 1);
        break;
      }
      if(is.fail()) {
        error("Faces must be specified as a line\n"
              "  <fno> <vno> <vno> <vno> [<vno> ...]\n"
              "where <fno> is the face number, and <vno> is a "
              "valid vertex number.\n", 1);
      }
#endif
      it = facedata.find(numdata);
      if(it == facedata.end()) {
        vector<int> v;
        v.push_back(vnumdata);
        facedata.insert(make_pair(numdata, v));
      } else {
        it->second.push_back(vnumdata);
      }
#if !defined(WIN32)
      count++;
#endif
    }
  }
}
```

```
void SSMdlReader::readConnections(ifstream &in)
{
  while(true) {
    stat = readLine(in);
    if(stat) {
      pushback();
      return;
    }
    istrstream is(buf);
#if defined(WIN32)
    string test(buf);
    if(test == "vertices"|| test == "faces"
        || test == "connections"|| test == "knotspacings")
      {
        modedata = test;
        pushback();
        return;
      }

    is ≫ numdata ≫ vnumdata;
#elif defined(__SCAN_BUG_WORKAROUND)
    int n = sscanf(buf, "%d %d", &numdata, &vnumdata);
    if(n ≠ 2) {
      pushback();
      return;
    }
#else
    is ≫ numdata ≫ vnumdata;
    if(is.fail()) {
      is.clear();
      is ≫ modedata;
      if(is.fail()) {
        error("Connections must be specified as a line\n"
              "  <fno> <fno>\n"
              "where <fno> is a valid face number.\n", 1);
      }
      pushback();
      return;
    }
#endif
    connections.push_back(make_pair(numdata, vnumdata));
  }
}

void SSMdlReader::readKnotSpacings(ifstream &in)
{
  map<int,vector<double> >::iterator it;
  while(true) {
    stat = readLine(in);
    if(stat) {
      pushback();
      return;
    }
```

```
    istrstream is(buf);
#if defined(WIN32)
    string test(buf);
    if(test == "vertices"|| test == "faces"
       || test == "connections"|| test == "knotspacings")
      {
        modedata = test;
        pushback();
        return;
      }

    is ≫ numdata;
#elif defined(__SCAN_BUG_WORKAROUND)
    int n = sscanf(buf, "%d", &numdata);
    if(n≠1) {
      pushback();
      return;
    }
#else
    is ≫ numdata;
    if(is.fail()) {
      is.clear();
      is ≫ modedata;
      if(is.fail()) {
        error("Knot spacings must be specified as a line\n"
              "  <fno> <ksp> <ksp> [<ksp> ...]\n"
              "where <fno> is a valid face number and <ksp> is a "
              "knot spacing.\n", 1);
      }
      pushback();
      return;
    }
#endif
#if defined(WIN32)
    while(is ≫ coorddata1 ≫ coorddata2)
      {
#elif defined(__SCAN_BUG_WORKAROUND)
    char *pos = buf + strspn(buf, " ");
    pos = pos + strcspn(pos, " ");

    while(true) {
      int n = sscanf(pos, "%le %le", &coorddata1, &coorddata2);
      if(n≠2) {
        break;
      }
      pos = pos + strspn(pos, " ");
      pos = pos + strcspn(pos, " ");
      pos = pos + strspn(pos, " ");
      pos = pos + strcspn(pos, " ");
#else
    while(true) {
      is ≫ coorddata1 ≫ coorddata2;
      if(is.eof() || is.fail()) {
```

```
          if(is.eof())
            break;
          is.clear();
          is >> modedata;
          if(is.fail()) {
            error("Knot spacings must be specified as a line\n"
                  "  <fno> <ksp> <ksp> [<ksp> ...]\n"
                  "where <fno> is a valid face number and <ksp> is a knot spacing.\n"
                  "There must be two knot spacing values for each edge in the face.\n", 1);
          }
          pushback();
          return;
        }
#endif
        it = knotspacings.find(numdata);
        if(it == knotspacings.end()) {
          vector<double> v;
          v.push_back(coorddata1);
          v.push_back(coorddata2);
          knotspacings.insert(make_pair(numdata, v));
        } else {
          it->second.push_back(coorddata1);
          it->second.push_back(coorddata2);
        }
      }
    }
}

pair<int, int> SSMdlReader::findEdgeNo(const vector<int> &vec1,
                                       const vector<int> &vec2,
                                       int f1,
                                       int f2)
{
  ostrstream ost;
  int n1 = vec1.size(), n2 = vec2.size(), c11 = 0, c12, c21, c22;
  while(c11 < n1)
    {
      c12 = 0;
      while(c12 < n2)
        {
          if(vec1[c11] == vec2[c12])
            goto cont;
          c12++;
        }
      c11++;
    }
  ost << "Trying to connect faces " << f1 << " and " << f2
      << " which share no vertices.\n";
  error(ost.str(), 0);
 cont:
  c21 = c11+1 % n1;
  c22 = c12 == 0 ? n2-1 : c12-1;
  if(vec1[c21] == vec2[c22])
```

```
    return make_pair(c21, c12);
  c21 = c11 == 0 ? n1−1 : c11−1;
  c22 = c12+1 % n2;
  if(vec1[c21] == vec2[c22])
    return make_pair(c11, c22);
  ost ≪ "Trying to connect faces "≪ f1 ≪ " and "≪ f2
      ≪ " which share no edge.\nMaybe one of the faces has the wrong orientation.\n";
  error(ost.str (), 0);
  return make_pair(0,0); // This statement is never reached
}

void SSMdlReader::buildModel(vector <SSLath∗> &faces)
{
  // Faces need not be numbered consecutively, so store here while building.
  map<int, SSLath∗> facestmp;
  map<int, vector<int> >::const_iterator cit1, cit2;
  vector<int>::const_iterator cit3;
  map<int, SSVertex>::const_iterator cit4;
  vector<pair<int,int> >::const_iterator cit5;
  map<int, SSLath∗>::const_iterator cit6;
  map<int, vector<double> >::const_iterator cit7;
  vector<double>::const_iterator cit8;
  vector<SSVertex> v;
  vector<int> vi;
  pair<int, int> conn;
  SSLath ∗base, ∗tmp;

  // Build faces
  cit1 = facedata.begin();
  while(cit1 ≠ facedata.end())
    {
      vi = cit1→second;
      cit3 = vi.begin();
      while(cit3 ≠ vi.end())
        {
          cit4 = vertices.find(∗cit3);
          if(cit4 == vertices.end())
            {
              ostrstream ost;
              ost ≪ "Invalid vertex "≪ ∗cit3 ≪ " used in face "≪ (cit1→first) ≪ "\n";
              error(ost.str (), 0);
            }
          v.push_back(cit4→second);
          cit3++;
        }
      facestmp.insert(make_pair(cit1→first,SSQModel::Instance()→BuildFace(v,faces)));
      v.clear ();
      cit1++;
    }

  // Connect faces
  cit5 = connections.begin();
  while(cit5 ≠ connections.end())
```

```
  {
    cit1 = facedata.find(cit5→first);
    cit2 = facedata.find(cit5→second);
    if(cit1 == facedata.end() || cit2 == facedata.end())
      {
        ostrstream ost;
        ost ≪ "Invalid face "≪ cit5→first ≪ " or "≪ cit5→second
            ≪ " used in connection\n";
        error(ost.str (), 0);
      }
    conn = findEdgeNo(cit1→second, cit2→second, cit5→first, cit5→second);
    SSQModel::Instance()→ConnectFaces(facestmp.find(cit5→first)→second, conn.first,
                                      facestmp.find(cit5→second)→second, conn.second);
    cit5++;
  }

// Add knot spacings
cit7 = knotspacings.begin();
while(cit7 ≠ knotspacings.end()) {
  cit8 = cit7→second.begin();
  cit6 = facestmp.find(cit7→first);
  if(cit6 == facestmp.end()) {
    ostrstream ost;
    ost ≪ "Invalid face "≪ cit7→first ≪ " specified for knot spacings.\n";
    error(ost.str (), 0);
  }
  if(2*facedata.find(cit7→first)→second.size() ≠ cit7→second.size()) {
    ostrstream ost;
    ost ≪ "Illegal number of knot spacings specified for face "
        ≪ cit6→first ≪ ".\n"
        ≪ "Two knot spacings must be specified for each edge in the face.\n";
    error(ost.str (), 0);
  }
  base = cit6→second;
  do {
    base→setcwknotspc(*(cit8++));
    base→setccwknotspc(*(cit8++));
    // Check that knot spacings match on either side of the edge,
    // and update if existing knot spacings are 1.0.
    tmp = base→ccwv();
    if(tmp && base→cwknotspc() ≠ tmp→ccwknotspc()) {
      if(tmp→ccwknotspc() ≠ 1.0) {
        ostrstream ost;
        ost ≪ "Knot spacings for face "≪ cit6→first ≪ " do not match knot spacings\n"
            ≪ "in adjacent face across edge "≪ (cit8−cit7→second.begin()) ≪ ".\n";
        error(ost.str (), 0);
      } else {
        tmp→setccwknotspc(base→cwknotspc());
      }
    }
    tmp = base→cwv();
    if(tmp && base→ccwknotspc() ≠ tmp→cwknotspc()) {
      if(tmp→cwknotspc() ≠ 1.0) {
```

```
          ostrstream ost;
          ost << "Knot spacings for face " << cit6->first << " do not match knot spacings\n"
              << "in adjacent face across edge " << (cit8-cit7->second.begin())+1 << ".\n";
          error(ost.str(), 0);
        } else {
          tmp->setcwknotspc(base->ccwknotspc());
        }
      }
      base = base->ccwf();
    } while(cit8 != cit7->second.end());
    cit7++;
  }
}

void SSMdlReader::read(const string &file, vector<SSLath*> &faces)
{
  lineno = 0;
  pushbackflag = 0;
  cout << "Opening file: " << file << endl;

  ifstream in(file.c_str());
  if(!in)
    error("Cannot open file "+file+"\n", 0);

  while(true) {
    stat = readLine(in);

    switch(stat) {
    case IO_EOF:
      goto exit;
    case IO_FAIL:
      error("I/O error. Failed to read.", 1);
    case IO_GOOD:
      break;
    }
#if defined(__SCAN_BUG_WORKAROUND)
    if(!strcmp(buf,"vertices"))
      readVertices(in);
    else if(!strcmp(buf,"faces"))
      readFaces(in);
    else if(!strcmp(buf,"connections"))
      readConnections(in);
    else if(!strcmp(buf,"knotspacings"))
      readKnotSpacings(in);
    else
      error("Illegal file format", 1);
#else
    istrstream is(buf);
    is >> modedata;

    if(modedata=="vertices")
      readVertices(in);
    else if(modedata=="faces")
```

```
      readFaces(in);
    else if(modedata=="connections")
      readConnections(in);
    else if(modedata=="knotspacings")
      readKnotSpacings(in);
    else
      error("Illegal file format", 1);
#endif
  }
 exit:
  in.close();
  buildModel(faces);
}
```

Listing A.28: SSHeroReader.h.

```
#if !defined _SSHeroReader_HEADER
#define _SSHeroReader_HEADER

#include <fstream>
#include <string>
#include <utility>
#include <map>
#include <vector>
#include "SSLath.h"
#include "SSVertex.h"
#include "SSReader.h"

// Reader for a subset of VRML2
class SSHeroReader
{

public:
  virtual bool read(const string filename, vector<SSLath*> &faces);

  virtual ~SSHeroReader();
  SSHeroReader();

private:
  enum io_status { IO_GOOD, IO_EOF, IO_FAIL, IO_NEXTSECTION };
  struct Match { int edge; SSLath *face; };
  typedef struct Match Match;


  io_status ReadLine(ifstream &is);
  void DoNextSection(ifstream &is);
  void ReadVertices(ifstream &is);
  void ReadFaces(ifstream &is);
  bool BuildModel(vector<SSLath*> &faces);
  void GetVertices();
  void GetFaceIndex();
  void DumpAll();
```

```
  int hash(int v1, int v2);
  void ConnectFaces(vector< vector<int> > &hashtable, vector<SSLath*> &faces);
  Match FindMatch(vector<vector<int> > &hashtable, vector<SSLath*> &faces, int v1, int v2);
  char m_currentline[1024];
  string m_nextsection;
  long m_lineno;
  int m_numberoffaces;

  vector<SSVertex> m_vertices;
  map<int, vector<int> > m_faces;

};

#endif
```

Listing A.29: SSHeroReader.cpp.

```
#if defined(WIN32)
#pragma warning( disable : 4786 )
#endif

#include <iostream>
#include <strstream>
#include <cstdio>
#include "SSHeroReader.h"
#include "SSQModel.h"
#include "SSTypes.h"

const string c_VertexSection = "Coordinate";
const string c_FaceSection = "coordIndex";
const string c_Stop = "Stop";
const int c_HashSize = 1000;
double maxx, minx, maxy, miny, maxz, minz;
int c;

#define MAX(a,b) (a>b ? a : b)
#define MIN(a,b) (a<b ? a : b)

SSHeroReader::SSHeroReader()
{
  m_numberoffaces = 0;
}

SSHeroReader::~SSHeroReader()
{
}


bool SSHeroReader::read(const string filename, vector<SSLath*> &faces)
{
  cout << "Opening file: "<< filename << endl;

```

```
  ifstream file (filename.c_str());

  if(! file .is_open())
    return false;

  m_nextsection = c_VertexSection;
  m_lineno = 0;

  while(true)
    {
      switch(ReadLine(file))
        {
        case IO_EOF:
          DumpAll();
          return BuildModel(faces);
        case IO_FAIL:
          cout << "IO Fail"<< endl;
          return false;
        case IO_GOOD:
          break;
        case IO_NEXTSECTION:
          cout << "New Section "<< m_nextsection << endl;
          DoNextSection(file);
          break;
        default:
          return false;
        }
    }

  return false;
}

SSHeroReader::io_status SSHeroReader::ReadLine(ifstream &is)
{
  is.getline(m_currentline, 1024);

  if(is.eof() || m_nextsection == c_Stop)
    return IO_EOF;

  if(is.fail())
    return IO_FAIL;

  istrstream str(m_currentline);
  string test;
  str >> test;
  if(test == m_nextsection)
    return IO_NEXTSECTION;


  return IO_GOOD;
}

void SSHeroReader::DoNextSection(ifstream &is)
```

```
{
  if(m_nextsection == c_VertexSection)
    {
      m_nextsection = c_FaceSection;
      ReadVertices(is);
    }
  else if(m_nextsection == c_FaceSection)
    {
      ReadFaces(is);
      m_nextsection = c_FaceSection;
    }
}

void SSHeroReader::ReadVertices(ifstream &is)
{
  ReadLine(is);

  c = 0;
  while(ReadLine(is) == IO_GOOD)
    {
      istrstream strtmp(m_currentline);
      string test;
      strtmp >> test;

      if(test == "]")
        break;

      GetVertices();
      c++;

    }

  cout << "vertices: "<< 3*c << endl;
}

void SSHeroReader::GetVertices()
{
  string curline(m_currentline), tmp;

  int lastp=0, newp=0;
  double x,y,z;

  do
    {
      newp = curline.find(",", lastp);

      if(newp == -1)
        return;

      tmp = curline.substr(lastp, newp - lastp);
      istrstream strtmp(tmp.c_str());
      strtmp >> x >> y >> z;
      m_vertices.push_back(SSVertex(x,y,z));
```

```
        lastp = newp + 1;
        if(c==0) {
          maxx = minx = x; maxy = miny = y; maxz = minz = z;
        } else {
          maxx = MAX(maxx,x); minx = MIN(minx,x);
          maxy = MAX(maxy,y); miny = MIN(miny,y);
          maxz = MAX(maxz,z); minz = MIN(minz,z);
        }
    }
  while(true);
}

void SSHeroReader::ReadFaces(ifstream &is)
{
  ReadLine(is);

  while(ReadLine(is) == IO_GOOD)
    {
      istrstream strtmp(m_currentline);
      string test;
      strtmp >> test;

      if(test == "]")
        break;

      GetFaceIndex();
    }
}

void SSHeroReader::GetFaceIndex()
{
  string curline(m_currentline), tmp;
  int lastp=0, newp=0, x = 0;
  vector<int> facedata;

  do
    {
      do
        {
          newp = curline.find(",", lastp);

          if(newp == -1)
            return;

          tmp = curline.substr(lastp, newp - lastp);

          if(tmp == "-1")
            {
              lastp = newp + 2;
              break;
            }

          istrstream strtmp(tmp.c_str());
```

```
            strtmp >> x;
            facedata.push_back(x);
            lastp = newp + 2;
          }
        while(true);
        m_faces.insert(make_pair(m_numberoffaces, facedata));
        facedata.clear ();
        m_numberoffaces++;
      }
  while(true);
}

bool SSHeroReader::BuildModel(vector<SSLath*> &f)
{
  map<int, vector<int> >::const_iterator c_mapiter;
  vector<SSVertex> face_vertices;
  vector<int> facenumbers;
  vector<int>::const_iterator c_veciter;
  vector< vector<int> > hashtable( c_HashSize, vector<int>() );
  SSLath *facelath=0;

  int c=0, hashentry=0;
  double dmax, dx, dy, dz, scale, tx, ty, tz;

  c_mapiter = m_faces.begin();

  // Scale and translate model
  dx = maxx-minx;
  dy = maxy-miny;
  dz = maxz-minz;
  dmax = MAX(MAX(dx,dy),dz);
  scale = 3.0/dmax;
  tx = -(minx+dx/2.0);
  ty = -(miny+dy/2.0);
  tz = -(minz+dz/2.0);

  cout << "maxx = "<< maxx << " minx = "<< minx
       << " scale = "<< scale << " tx = "<< tx << " ty = "<< ty
       << " tz = "<< tz << endl;

  while(c_mapiter != m_faces.end())
    {
      facenumbers = c_mapiter->second;
      c_veciter = facenumbers.begin();
      while(c_veciter != facenumbers.end())
        {
          face_vertices.push_back((m_vertices[*c_veciter] + SSVertex(tx,ty,tz))*scale);
          c_veciter++;
        }

      facelath = SSQModel::Instance()->BuildFace(face_vertices,f);
      for(int i = 0; i < facenumbers.size(); i++)
        {
```

```
        int h = hash(facenumbers[i], facenumbers[(i+1) % facenumbers.size()]);
        hashtable[h].push_back(c_mapiter→first);
      }

    face_vertices.clear ();

    c_mapiter++;
    c++;
  }

  ConnectFaces(hashtable, f);
  cout ≪ "Faces: "≪ c ≪ endl;
  return true;
}

SSHeroReader::Match SSHeroReader::FindMatch(vector<vector<int> > &hashtable,
                                            vector<SSLath∗> &faces,
                                            int v1,
                                            int v2)
{
  int h = hash(v1, v2);
  vector<int> faceindices = hashtable[h];
  Match m;

  for(vector<int>::const_iterator it=faceindices.begin(); it≠faceindices.end(); it++) {
    vector<int> vertexnums = m_faces[∗it];
    for(int i=0; i<vertexnums.size(); i++)
      if(vertexnums[i] == v2 && vertexnums[(i+1)%vertexnums.size()] == v1) {
        // Match
        m.edge = (i+1)%vertexnums.size();
        m.face = faces[∗it ];
        return m;
      }
  }
  m.edge = 0;
  m.face = NULL;
  return m;
}

void SSHeroReader::ConnectFaces(vector< vector<int> > &hashtable, vector<SSLath∗> &faces)
{
  map<int, vector<int> >::const_iterator c_mapiter;
  vector<int> vertexnumbers;
  SSLath ∗face3;
  c_mapiter = m_faces.begin();

  while(c_mapiter ≠ m_faces.end())
    {
      vertexnumbers = c_mapiter→second;

      for(int i = 0; i < vertexnumbers.size(); i++)
        {
          SSLath ∗face = faces[c_mapiter→first];
```

```
            Match m = FindMatch(hashtable, faces, vertexnumbers[i],
                                vertexnumbers[(i+1)%vertexnumbers.size()]);
            if (!m.face) {
              for(int j=0; j<i+1; j++) face = face→ccwf();
              face→setcwknotspc(0.0);
              face→cwf()→setccwknotspc(0.0);
              continue;
            } else {
              SSLath *f1 = face, *f2 = m.face;
              for(int j=0; j<i; j++) f1 = f1→ccwf();
              if(m.edge == 0) f2 = m.face→cwf();
              else
                for(int j=0; j<m.edge−1; j++) f2 = f2→ccwf();
              if(f1→ccwf()→ccwv()) {
                if(f1→ccwf()→ccwv() ≠ f2) {
                  f1→setccwknotspc(0.0);
                  f1→ccwf()→setcwknotspc(0.0);
                  f2→setccwknotspc(0.0);
                  f2→ccwf()→setcwknotspc(0.0);
                  f1→ccwf()→ccwv()→setccwknotspc(0.0);
                  f1→ccwf()→ccwv()→ccwf()→setcwknotspc(0.0);
                }
                continue;
              }
              if(f2→ccwf()→ccwv()) {
                if(f2→ccwf()→ccwv() ≠ f1) {
                  f1→setccwknotspc(0.0);
                  f1→ccwf()→setcwknotspc(0.0);
                  f2→setccwknotspc(0.0);
                  f2→ccwf()→setcwknotspc(0.0);
                  f2→ccwf()→ccwv()→setccwknotspc(0.0);
                  f2→ccwf()→ccwv()→ccwf()→setcwknotspc(0.0);
                }
                continue;
              }
              SSQModel::Instance()→ConnectFaces(face, (i+1)%vertexnumbers.size(), m.face, m.edge);

            }
          }
        c_mapiter++;
      }
}

void SSHeroReader::DumpAll()
{
  ofstream ofile ;

  ofile .open("dump.txt");

  if (! ofile .is_open())
    {
      cout ≪ "Dump file open error"≪ endl;
      return;
```

```
    }


  map<int, vector<int> >::const_iterator c_mapiter;
  vector<int> facenumbers;
  vector<int>::const_iterator c_veciter;
  vector<SSVertex>::const_iterator c_vertiter;

  c_mapiter = m_faces.begin();

  while(c_mapiter != m_faces.end())
    {

      ofile << "Face " << c_mapiter->first << ": ";

      facenumbers = c_mapiter->second;

      c_veciter = facenumbers.begin();
      while(c_veciter != facenumbers.end())
        {
          ofile << *c_veciter << " ";
          c_veciter++;
        }

      ofile << endl;

      c_mapiter++;
    }

  ofile << endl << "Vertices " << m_vertices.size() << endl;

  c_vertiter = m_vertices.begin();
  int c;

  while(c_vertiter != m_vertices.end())
    {
      c = 0;
      while(c <= 2 && c_vertiter != m_vertices.end())
        {
          ofile << *c_vertiter << " ";
          c++;
          c_vertiter++;
        }

      ofile << endl;
    }
}

int SSHeroReader::hash(int v1, int v2)
{
  return (v1 + v2) % c_HashSize;
}
```

Listing A.30: SSText.h.

```
#ifndef _TEXT_H
#define _TEXT_H

#include "SSVector.h"

// Utility class for drawing text using OpenGL
class SSGLText
{

public:
  SSGLText(SSVector pos, SSVector scale, char *message);
  ~SSGLText();

  void draw();

private:
  SSVector m_pos;
  SSVector m_scale;
  char m_message[128];
};

#endif
```

Listing A.31: SSText.cpp.

```
#include <string.h>
#include <GL/glut.h>
#include "SSText.h"

SSGLText::SSGLText(SSVector pos, SSVector scale, char *message) : m_pos(pos), m_scale(scale)
{
  strncpy(m_message, message, 128);
}

SSGLText::~SSGLText()
{
}

void SSGLText::draw()
{
  glPushMatrix();
  glDisable(GL_LIGHTING);
  glTranslatef(m_pos[0], m_pos[1], m_pos[2]);
  glScalef(m_scale[0], m_scale[1], m_scale[2]);

  char *p = m_message;

  while (*p) glutStrokeCharacter(GLUT_STROKE_ROMAN, *p++);

  glEnable(GL_LIGHTING);
```

```
  glPopMatrix();
}
```

Listing A.32: viewer.cpp.

```
#if defined(WIN32)
#pragma warning( disable : 4786 )
#endif

#ifndef _SSMaster_HEADER
#include "SSMaster.h"
#endif

#include <GL/glut.h>
#include "SSQModel.h"
#include "SSVector.h"
#include "SSText.h"
#include <iostream>
#if !defined(WIN32)
#include <unistd.h>
#include <cstdio>
#endif
#include <cmath>

#define VIEWER_VERSION "0.3"

static GLint ay = 0, ar = 0;
static GLfloat tz = -10, tx = 0.0, ty = 0.0;
int flags = SS_DRAW_OUTLINE;
bool hiddenlines = true,
  depth = true,
  lighting = true,
  outline = true,
  perspectiveproj=true,
  trace = false,
  FullScreen = false,
  exactnormal = false;

int maxlevel=4;
int curlevel=0;
int method=c_DooSabin;
int winWidth = 800, winHeight = 600;

enum MENU_ENUM
{
  M_DOOSABIN = 0,
  M_CATMULLCLARK,
  M_LOOP,
  M_QUADNURSS,
  M_CUBICNURSS,
  M_CULL,
  M_LIGHTING,
```

```
  M_PERSPECTIVE,
  M_OUTLINE,
  M_DEPTH,
  M_TRACE,
  M_FULL_SCREEN,
  M_QUIT
};

void draw_axis(SSVector axlen)
{
  SSVector zero;
  SSVector tip1, tip2, tip3;
  double colorR[] = {1.0, 0.0, 0.0, 1.0};
  double colorG[] = {0.0, 1.0, 0.0, 1.0};
  double colorB[] = {0.0, 0.0, 1.0, 1.0};

  glPushAttrib(GL_ALL_ATTRIB_BITS);

  glDisable(GL_LIGHTING);
  glDisable(GL_COLOR_MATERIAL);
  glLineWidth(2.0f);

  glBegin(GL_LINES);
  glColor3ub(255,  0,  0);
  tip1 = SSVector(axlen[0], 0.0f, 0.0f);
  glVertex3f(zero[0], zero[1], zero[2]);
  glVertex3f(tip1[0], tip1[1], tip1[2]);

  glColor3ub(0, 255,  0);
  tip2 = SSVector(0.0f, axlen[1], 0.0f);
  glVertex3f(zero[0], zero[1], zero[2]);
  glVertex3f(tip2[0], tip2[1], tip2[2]);

  glColor3ub(0,  0, 255);
  tip3 = SSVector(0.0f, 0.0f, axlen[2]);
  glVertex3f(zero[0], zero[1], zero[2]);
  glVertex3f(tip3[0], tip3[1], tip3[2]);
  glEnd();

  glColor3ub(0,0,0);
  SSGLText *textX = new SSGLText(tip1, SSVector(.002f, .002f, .002f), " x");
  SSGLText *textY = new SSGLText(tip2, SSVector(.002f, .002f, .002f), " y");
  SSGLText *textZ = new SSGLText(tip3, SSVector(.002f, .002f, .002f), " z");

  textX->draw();
  textY->draw();
  textZ->draw();

  delete textX;
  delete textY;
  delete textZ;
  glPopAttrib();
}
```

```

void display_text(GLuint x, GLuint y, double scale, char* text)
{
  double font_scale = 119.05f + 33.33f;

  glPushMatrix();
  glMatrixMode(GL_PROJECTION);
  glPushMatrix();
  glLoadIdentity();

  gluOrtho2D(0, winWidth, 0, winHeight);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  glTranslatef(x, y, 0);
  glScalef(scale / font_scale, scale / font_scale, 1);

  glPushAttrib(GL_CURRENT_BIT);
  glDisable(GL_LIGHTING);
  glColor3f (1, 1, 1);
  while (*text) glutStrokeCharacter(GLUT_STROKE_ROMAN, *text++);

  glPopMatrix();

  glMatrixMode(GL_PROJECTION);
  glPopMatrix();

  glMatrixMode(GL_MODELVIEW);
  glPopAttrib();
  if(lighting)
    glEnable(GL_LIGHTING);

}

void set_projection(void)
{
  double aspect = (double) winWidth / (double) winHeight;

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (perspectiveproj)
    {
      gluPerspective(35, aspect, 0.1, 700.0);
    }
  else
    {
      glOrtho(-3*aspect, 3*aspect, -3, 3, 0.1, 20.0);
    }

  glMatrixMode(GL_MODELVIEW);
}

```

```
void init(const string &filename)
{
  GLfloat light_diffuse []  = {0.6f, 0.6 f , 1.0 f , 1.0 f};
  GLfloat light_position [] = {0.0, 0.0, 1000.0, 1};

  float ambient[] = {0.5, 0.0, 0.0, 1.0};
  float diffuse [] = {1.0, 0.1, 0.1, 1.0};
  float specular [] = {1.0,1.0, 1.0, 1.0};
  float shine [] = {100.0};
  float emission [] = {0, 0, 0, 0};

  glMaterialfv(GL_FRONT, GL_EMISSION, emission);
  glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);
  glMaterialfv(GL_FRONT, GL_SHININESS, shine);
  glMaterialfv(GL_FRONT, GL_SPECULAR, specular);

  glShadeModel(GL_SMOOTH);
  glEnable(GL_LIGHTING);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);

  glEnable(GL_LIGHT0);
  glClearColor (.3, .3, .3, 1.0 F);

  glFrontFace(GL_CCW);

  cout << "Reading model"<< endl;
  SSQModel::Instance()->Init(filename);
  cout << "Model read"<< endl;


  SSQModel::Instance()->Subdivide(method, maxlevel);

  cout << "Model subdivided"<< endl;
}

void display(void)
{
  char text[128];

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(tx, ty, tz);

  glRotatef((GLfloat) ay , 0.0, 1.0, 0.0);
  glRotatef((GLfloat) ar, (GLfloat) cos(-ay*c_PI/180.0), 0.0, (GLfloat) -sin(-ay*c_PI/180.0));

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  if(hiddenlines)
    glEnable(GL_CULL_FACE);

```

```
if(depth)
  glEnable(GL_DEPTH_TEST);

if (outline)
  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

if(lighting)
  {
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
  }

SSVector axis = SSVector (0.5, 0.5, 0.5);
draw_axis(axis);

if(method == c_DooSabin)
  sprintf(text, "%s", "DooSabin");
else if (method == c_CatmullClark)
  sprintf(text, "%s", "CatmullClark");
else if (method == c_Loop)
  sprintf(text, "%s", "Loop");
else if (method == c_QuadNURSS)
  sprintf(text, "%s", "Quad Nurss");
else if (method == c_CubicNURSS)
  sprintf(text, "%s", "Cubic Nurss");
else
  sprintf(text, "%s", "Unknown");

display_text (10, 30, 20, text);

SSQModel::Instance()->SwitchMethod(method);

SSQModel::Instance()->Draw(curlevel, flags);

long faces = SSQModel::Instance()->NumberOfFaces(curlevel);

sprintf(text, "%d", faces);

display_text (10, 10, 20, text);

if(trace)
  {
    trace = false;
    flags ^= SS_DRAW_TRACE;
  }

glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glDisable(GL_CULL_FACE);
glDisable(GL_LIGHT0);
glDisable(GL_DEPTH_TEST);

glutSwapBuffers();
```

```
}

void reshape(int width, int height)
{
  glViewport(0, 0, width, height);
  winWidth = width;
  winHeight = height;
  set_projection();
}

void keyboard(unsigned char key, int x, int y)
{
  switch(key) {
  case 27:
    exit(0);
    break;
  case 'h':
    hiddenlines = !hiddenlines;
    break;
  case 'd':
    depth = !depth;
    break;
  case 'o':
    outline = !outline;
    flags ^= SS_DRAW_OUTLINE;
    break;
  case 'p':
    perspectiveproj = !perspectiveproj;
    set_projection();
    break;
  case 'n':
    exactnormal = !exactnormal;
    flags ^= SS_DRAW_EXACT_NORMALS;
    break;

  case 'l':
    lighting = !lighting;
    break;
  }
  glutPostRedisplay();
}

void specKeyboard(int key, int x, int y)
{
  int s = glutGetModifiers();
  switch(key) {
  case GLUT_KEY_LEFT:
    if(s == GLUT_ACTIVE_CTRL)
      tx -= 0.2;
    else
      ay = (ay + 5) % 360;
    break;
  case GLUT_KEY_RIGHT:
```

```
    if(s == GLUT_ACTIVE_CTRL)
      tx += 0.2;
    else
      ay = (ay - 5) % 360;
    break;
  case GLUT_KEY_UP:
    if(s == GLUT_ACTIVE_CTRL)
      ty += 0.2;
    else
      ar = (ar + 5) % 360;
    break;
  case GLUT_KEY_DOWN:
    if(s == GLUT_ACTIVE_CTRL)
      ty -= 0.2;
    else
      ar = (ar - 5) % 360;
    break;
  case GLUT_KEY_PAGE_UP:
    tz -= 0.2;
    break;
  case GLUT_KEY_PAGE_DOWN:
    tz += 0.2;
    break;
  case GLUT_KEY_HOME:
  case GLUT_KEY_F12:
    if(curlevel < maxlevel)
      curlevel++;
    else
      curlevel = 0;
    glutPostRedisplay();
    break;
  case GLUT_KEY_END:
  case GLUT_KEY_F11:
    if(curlevel > 0)
      curlevel--;
    else
      curlevel = 0;
    glutPostRedisplay();
    break;
  }
  glutPostRedisplay();
}

void menu_fun(int value)
{
  switch (value)
    {
    case M_LOOP:
      method = c_Loop;
      break;
    case M_CATMULLCLARK:
      method = c_CatmullClark;
      break;
```

```
    case M_DOOSABIN:
      method = c_DooSabin;
      break;
    case M_QUADNURSS:
      method = c_QuadNURSS;
      break;
    case M_CUBICNURSS:
      method = c_CubicNURSS;
      break;

    case M_CULL:
      hiddenlines = !hiddenlines;
      break;
    case M_LIGHTING:
      lighting = !lighting;
      break;

    case M_OUTLINE:
      outline = !outline;
      flags ^= SS_DRAW_OUTLINE;
      break;

    case M_PERSPECTIVE:
      perspectiveproj = !perspectiveproj;
      set_projection();
      break;

    case M_DEPTH:
      depth = !depth;
      break;
    case M_TRACE:
      trace = !trace;
      flags ^= SS_DRAW_TRACE;
      break;

    case M_FULL_SCREEN:
      FullScreen = !FullScreen;

      if (FullScreen)
        glutFullScreen();
      else
        glutReshapeWindow(800, 600);
      break;

    case M_QUIT:
      exit(0);
      break;
    }

  glutPostRedisplay();
}

void init_menu()
```

```
{
  glutCreateMenu(menu_fun);
  glutAddMenuEntry("Doo Sabin", M_DOOSABIN);
  glutAddMenuEntry("Catmull Clark", M_CATMULLCLARK);
  glutAddMenuEntry("Loop", M_LOOP);
  glutAddMenuEntry("Quad Nurss", M_QUADNURSS);
  glutAddMenuEntry("Cubic Nurss", M_CUBICNURSS);
  glutAddMenuEntry("", -1);
  glutAddMenuEntry("", -1);
  glutAddMenuEntry("Toggle Face Culling", M_CULL);
  glutAddMenuEntry("Toggle Lighting", M_LIGHTING);
  glutAddMenuEntry("Toggle Outlines", M_OUTLINE);
  glutAddMenuEntry("Toggle Perspective Proj", M_PERSPECTIVE);
  glutAddMenuEntry("Toggle Depth Test", M_DEPTH);
  glutAddMenuEntry("", -1);
  glutAddMenuEntry("Trace", M_TRACE);
  glutAddMenuEntry("Full screen", M_FULL_SCREEN);
  glutAddMenuEntry("", -1);
  glutAddMenuEntry("Exit", M_QUIT);
  glutAttachMenu(GLUT_RIGHT_BUTTON);
}

int main(int argc, char **argv)
{
  glutInit (&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH );
  glutInitWindowSize(winWidth,winHeight);
  glutCreateWindow("Viewer " VIEWER_VERSION);
  if(argc > 2)
    maxlevel = strtol(argv [2], 0, 10);
  if(argc > 1)
    init (argv [1]);
  else {
    cerr << "No input file specified.\n"<< endl;
    exit (1);
  }
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutSpecialFunc(specKeyboard);

  init_menu();

  glutMainLoop();
  return 0;
}
```

Appendix B

MDL File Format Specification

The viewer program can read control polygons specified in a subset of the standardized VRML2 (Virtual Reality Markup Language) format or in the MDL format designed specifically for the viewer. The syntax of the MDL file format is described in this chapter.

An MDL file consists of three or four sections each initiated with a text string. The first section specifies the vertices in the control mesh. It is initiated with the string `vertices`. Each subsequent line should be of the form

$$\textit{vnum} \quad \textit{xcoord} \quad \textit{ycoord} \quad \textit{zcoord}$$

and defines the vertex with number *vnum* to have coordinates $(\textit{xcoord}, \textit{ycoord}, \textit{zcoord})$. The vertex number must be an integer, while the coordinates are real numbers. The fields in a line are separated by one or more spaces.

The second section specifies the faces in the control mesh. It is initiated by the string `faces`. A face is defined by a line of the form

$$\textit{fnum} \quad \textit{vnum}_1 \quad \textit{vnum}_2 \quad \ldots \quad \textit{vnum}_n$$

where *fnum* is a number identifying the face and $\textit{vnum}_i$ is the number of a vertex defined in the previous section of the file. The vertices should be listed in counter-clockwise order to give the correct orientation of the face.

The third section specifies the connections between faces. It is initiated with the string `connections`. Each line is of the form

$$\textit{fnum}_1 \quad \textit{fnum}_2$$

where $\textit{fnum}_1$ and $\textit{fnum}_2$ are both faces defined in the previous section of the file. The two faces must share two vertices and must be connected such that the orientation is the same.

The fourth, optional, section specifies knot spacings for edges in the control polygon. It is initiated by the string `knotspacings`. The knot spacings are given by a line

$$\textit{fnum} \quad \textit{cwspc}_1 \quad \textit{ccwspc}_1 \quad \textit{cwspc2} \quad \textit{ccwspc2} \quad \ldots \quad \textit{cwspc}_n \quad \textit{ccwspc}_n$$

giving the knot spacings for face number *fnum*. There are two knot spacings for each vertex, one on the edge in the clockwise direction and one on the edge in the counter-clockwise direction. The clockwise and counter-clockwise knot spacings for the ith vertex in the face (with the ordering used to specify the face) are $cwspc_i$ and $ccwspc_i$, respectively. The knot spacings are real numbers. Edges for which no knot spacings are specified are assigned knot spacings of 1.0. If knot spacings for the same edge are specified in two adjacent faces, the values must match.

Lines beginning with a hash (#) character are considered to be comments and ignored by the viewer program.

A simple example of an MDL file is shown below.

```
# Tetrahedron model

vertices
1   0.0 0.0 0.0
2   1.0 0.0 0.0
3   0.5 0.866 0.0
4   0.5 0.289  0.866

faces
1   1 2 4
2   2 3 4
3   1 4 3
4   1 3 2

connections
1 2
1 3
1 4
2 3
2 4
3 4

knotspacings
1  0.1 1.0 1.0 0.1 0.1 0.1
2  0.1 1.0 1.0 0.1 0.1 0.1
3  1.0 0.1 0.1 0.1 0.1 1.0
```

Bibliography

[1] Ball, A. A., and Storry, D. J. T. A matrix approach to the analysis of recursively generated B-spline surfaces. *Computer Aided Design 18*, 8 (1986), 437–42.

[2] Ball, A. A., and Storry, D. J. T. Conditions for tangent plane continuity over recursively generated B-spline surfaces. *ACM Transactions on Graphics 7*, 2 (1988), 83–102.

[3] Catmull, E., and Clark, J. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design 10*, 6 (1978), 350–5.

[4] Chaikin, G. M. An algorithm for high-speed curve generation. *Computer Graphics and Image Processing 3*, 4 (1974), 346–349.

[5] Davis, P. J. *Circulant Matrices*. John Wiley and Sons, 1979.

[6] DeRose, T., Kobbelt, L., Levin, A., Schröder, P., Sweldens, W., and Zorin, D. Subdivision for modeling and animation, 2000. SIGGRAPH 2000 Course Notes.

[7] Doo, D., and Sabin, M. Behaviour of recursive division surfaces near extraordinary points. *Computer Aided Design 10*, 6 (1978), 356–60.

[8] Dyn, N., Levin, D., and Gregory, J. A. A 4-point interpolatory subdivision scheme for curve design. *Computer-Aided Geometric Design 4*, 4 (1987), 257–68.

[9] Dyn, N., Levin, D., and Gregory, J. A. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Transactions on Graphics 9*, 2 (1990), 160–169.

[10] Hoppe, H., DeRose, T., Duchamp, T., Halstead, M., Jin, H., McDonald, J., Schweitzer, J., and Stuetzle, W. Piecewise smooth surface reconstruction. In *Computer Graphics Proceedings* (1994), Annual Conference Series, pp. 295–302.

[11] Hoschek, J., and Lasser, D. *Fundamentals of Computer Aided Geometric Design*. A K Peters, Ltd., 1993.

[12] Kobbelt, L. Interpolatory subdivision on open quadrilateral nets with arbitrary topology. In *Computer Graphics Forum, Proceedings of Eurographics '96* (1996), vol. 15, pp. 409–420.

[13] Kobbelt, L. Using the discrete Fourier transform to analyze the convergence of subdivision schemes. *Applied and Computational Harmonic Analysis 5*, 1 (1998), 68–91.

[14] Kobbelt, L. $\sqrt{3}$ subdivision. In *Computer Graphics Proceedings, SIGGRAPH 2000* (2000), Annual Conference Series.

[15] Legakis, J., MacCracken, R., and Joy, K. I. Data structures for unstructured meshes. *The Visual Computer* (1999). Submitted for publication. Preprint available at http://www.cs.umd.edu/class/fall2000/cmsc725/mesh-paper.pdf.

[16] Loop, C. Smooth subdivision surfaces based on triangles. Master's thesis, Department of Mathematics, University of Utah, 1987.

[17] Peters, J., and Reif, U. Analysis of algorithms generalizing B-spline subdivision. *SIAM Journal on Numerical Analysis 35*, 2 (1998), 728–48.

[18] Prautzsch, H. Analysis of C^k subdivision surfaces at extraordinary points. Tech. rep., Fakultät für Informatik, Universität Karlsruhe, 1995. Presented at Oberwolfach, June 5, 1995.

[19] Prautzsch, H., and Umlauf, G. A G^2-subdivision algorithm. *Computing - Wien - Supplements*, 13 (1998), 217–224.

[20] Prautzsch, H., and Umlauf, G. Improved triangular subdivision schemes. In *Proceedings of Computer Graphics International 1998* (1998), pp. 626–632.

[21] Qin, K., and Wang, H. Continuity of non-uniform recursive subdivision surfaces. *Science in China (Series E)* (1999). Submitted for publication.

[22] Qin, K., and Wang, H. Eigenanalysis and continuity of non-uniform Doo-Sabin surfaces. In *Proceedings of the Seventh Pacific Conference on Computer Graphics and Applications* (1999).

[23] Reif, U. A unified approach to subdivision algorithms near extraordinary vertices. *Computer Aided Geometric Design 12*, 2 (1995), 153–174.

[24] Reif, U., and Schröder, P. Curvature integrability of subdivision surfaces. *Advances in Computational Mathematics 14*, 2 (2001), 157–174.

[25] Riesenfeld, R. F. On Chaikin's algorithm. *Computer Graphics and Image Processing 4*, 3 (1975), 304–310.

[26] Schweitzer, J. *Analysis and Application of Subdivision Surfaces.* PhD thesis, Department of Computer Science and Engineering, University of Washington, 1996.

[27] Sederberg, T. W., Zheng, J., Sewell, D., and Sabin, M. Non-uniform recursive subdivision surfaces. *Computer Graphics 32*, Annual Conference Series (1998), 387–394.

[28] Stam, J. Evaluation of Loop subdivision surfaces. In *SIGGRAPH '98 CDROM Proceedings* (1998).

[29] Stam, J. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. ACM, pp. 395–404.

[30] Umlauf, G. Analyzing the characteristic map of triangular subdivision schemes. *Constructive Approximation 16*, 1 (2000), 145–155.

[31] Velho, L., and Zorin, D. 4-8 subdivision. *Computer Aided Geometric Design 18*, 5 (2001).

[32] Warren, J. Subdivision methods for geometric design. Unpublished manuscript available at http://www.cs.rice.edu/jwarren/papers/book.ps.gz, 1995.

[33] Zorin, D. C^k continuity of subdivision surfaces. Tech. Rep. CS-TR-96-23, California Institute of Technology, 1996.

[34] Zorin, D. *Stationary Subdivision and Multiresolution Surface Representations.* PhD thesis, California Institute of Technology, 1997.

Index