Designing Subdivision Surfaces through Control Mesh Refinement

by

Haroon S. Sheikh

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of Master of Mathematics

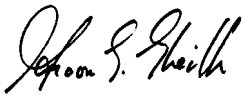
in

Computer Science

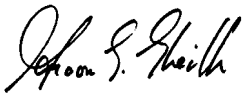
Waterloo, Ontario, Canada, 1996

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.



I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.



The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Designing Subdivision Surfaces through Control Mesh Refinement.

I present a technique for designing subdivision surfaces through the repeated process of refining the control mesh of the surface. The refinement is performed using the surface's subdivision algorithm. An abstract structure for subdivision surfaces is developed that involves the control mesh and the subdivision algorithm. This structure is used to build a generic editor that assists in the design of surfaces represented by classes of control meshes and subdivision algorithms.

The theory of subdivision surfaces, in particular, tensor product cardinal splines and box splines is described in detail. From the theory, subdivision algorithms for these surfaces are constructed and then used to create Refiners. The concept of trimming these surfaces is also introduced to aid in rendering of the surface. Several C++ spline classes used in the implementation are described. The subdivision surface editor uses these classes and Open Inventor to provide a prototype 3D surface design, manipulation and editing environment.

Acknowledgment

First of all, I wish to thank my supervisor, Richard Bartels, for his guidance, supervision, patience and his help throughout my Masters studies. He was one of the main reason that I came back from the job force for a Masters degree at the University of Waterloo. I would also like to thank my two faculty readers, Steve Mann and Grant Weddell for taking the time to go through this work.

Secondly, I would like to thank my family, especially my loving wife, Wasia, for always standing by me and helping me finish the thesis write-up.

Last, but not least, I would like to thank Allah, for giving me the strength, the wisdom, the mercy, the benevolence to help me complete this thesis. I wish to start this work in the name of God, the most Gracious, the most Merciful : Bismillah-hir Rahman-nir Raheem.

Table of Contents

List of Illustrations

1. Introduction

Spline theory is a rich field of computer graphics that provides many forms of curve and surface definitions. Curves are the fundamental starting block in spline theory. Béziers and B-splines are common examples of such curves. Spline curves are piecewise polynomial curves usually used to represent complex shapes. Surfaces are usually built using curves or by extending the theory to a higher dimension. Some examples of surfaces include tensor product NURBS and Coons patches. A good reference for these curves and surfaces is [Far93].

This work is concerned with a category of surfaces called subdivision surfaces. In simple terms, these represents surfaces that provide a subdivision rule or algorithm for refining the surface's control mesh. Some examples include tensor product cardinal spline surfaces, box spline surfaces, and Catmull-Clark surfaces.

The result of this work is a subdivision surface editor that provides a rendering and surface manipulation environment for such surfaces. The theory behind these surfaces is investigated in the following chapter. The requirements for a subdivision surface editor is presented in Chapter 3 and the implementation required for the editor is presented in Chapter 4.

1.1 Control Mesh

In general, surfaces in CAGD have a corresponding control mesh that allows the user to control the shape of the surface. In practice, to create a new surface, one starts by defining a control mesh for

the surface. This control mesh is mathematically linked to the surface and is used to design and render the surface. Another approach taken to generate surfaces is to sample real life data points on the surface. The control mesh is then generated by interpolating a surface over these data points. In either approach, a control mesh is inherently connected to the surface.

A control mesh is defined by a pair - a set of vertices and the connectivity relationship between the vertices. The former defines the shape of the mesh and is called the set of control vertices or control points. The latter defines the topology of the surface. Vertices are connected together by edges and edges enclose a face. This work is concerned with meshes on a rectangular grid or quadrilateral (quad) meshes. This restriction is imposed for the purpose of simplicity. These meshes are easiest to handle and provide a starting place to build upon in the future.

Quad meshes are also called regular meshes because the control vertices lie on a regular structure. As such, the connectivity relationship is very simple. Vertices are arranged on a two dimensional (u,v) domain represented by rows and columns. Internal vertices, those in the middle of the mesh, are connected to four other vertices in neighboring rows and columns. Vertices at the edges are connected to three others while those at the corners are connected to two other vertices. The shape of each face is a quadrilateral. See Figure 1 for a pictorial view of a quad mesh. In contrast irregular meshes do not have the restrictions associated with regular meshes and vertex connectivity can be arbitrary.

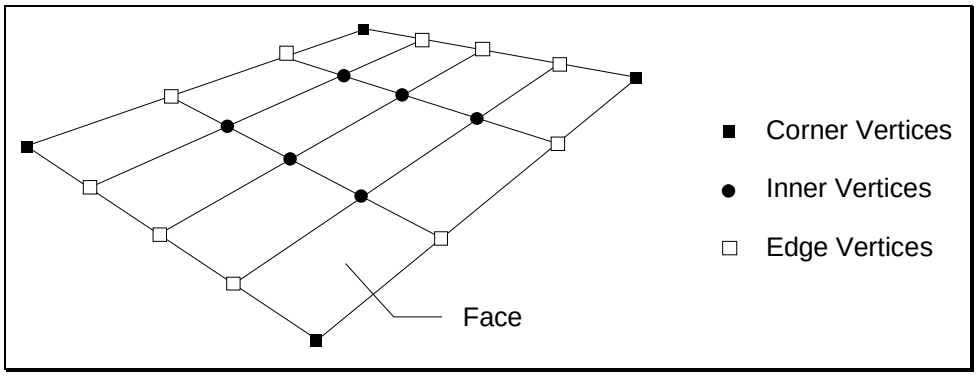


Figure 1. 5x4 Quadrilateral Mesh.

1.2 Motivation

This thesis started as a study of box spline surfaces and multivariate splines. The initial purpose was the incorporation of bivariate and multivariate box splines into the Splines project at the Computer Graphics Lab (CGL) at the University of Waterloo. In the process of this study it was found that rendering of the box splines could be accomplished using subdivision algorithms instead of explicit evaluation of the surface. The former provided a fast and straightforward approach for surface rendering while the latter was a computationally intensive method. Investigation into the subdivision algorithm led to other subdivision surfaces with corresponding subdivision algorithms. Further study of these surfaces presented an abstract structure on which a generic subdivision surface editor could be built to design, manipulate and render any subdivision surface based on a control mesh. Currently, rectangular meshes are supported, but it is hoped that the software abstractions used to build the system will support meshes of other connectivity.

1.3 Abstract structure of subdivision surfaces

The definition of subdivision surfaces is closely tied to the control mesh representing the surface. Given a control mesh and a mesh subdivision rule, the subdivision surface is generated by repeatedly refining the control mesh using the subdivision rule. The subdivision rule is specific to the subdivision surface. However, it is possible for more than one rule to exist for a subdivision surface. A subdivision rule, stated loosely, is a set of formulas by which selected collections of vertices are replaced by larger collections.

The foundation for this work materialized after working with several subdivision surfaces and their algorithms. It was found that the subdivision surfaces could be implemented and worked on at an abstract level. There are two fundamental concepts that can be abstracted. One is the control mesh and the other is the application of the subdivision rule. All subdivision surfaces are constructed by starting with a control mesh defining the surface. The control mesh plays an important part in the description of the surface. The control points in the control mesh define the shape of the underlying surface. The user has full control of the surface when manipulating the control vertices. If a single control point is modified, it changes the surface appropriately.

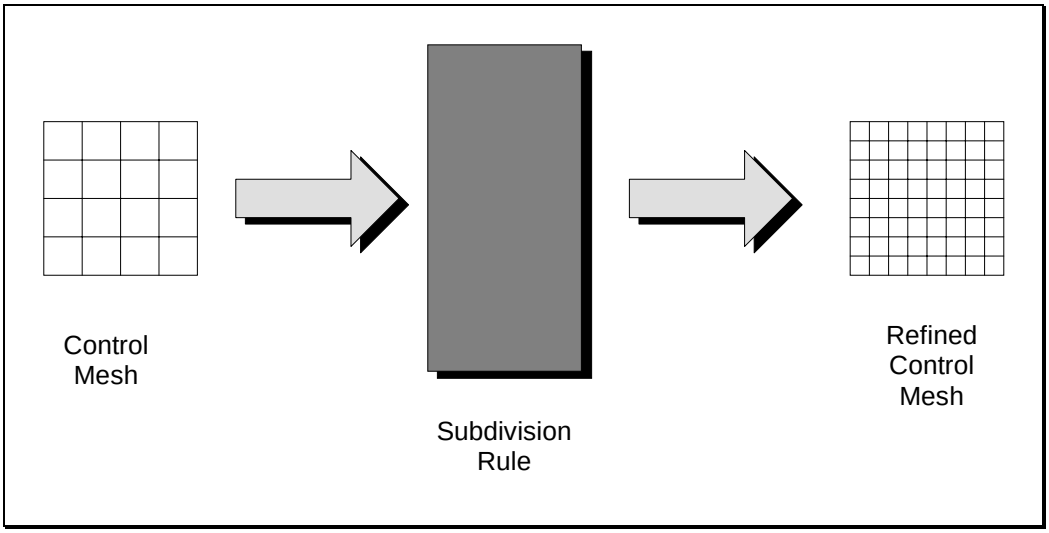


Figure 2. Refining a control mesh by a subdivision rule.

By definition, all subdivision surfaces have a subdivision rule. At an abstract level, the subdivision rule acts like a black box that takes a control mesh as input and generates a refined control mesh. This concept is illustrated in Figure 2. We start with the control mesh defining the surface. We call this the *initial* control mesh. We can apply the rule to the *initial* control mesh which results in a refined control mesh. We can further apply the rule to the refined control mesh to create a further refined control mesh. If this subdivision rule is repeatedly applied to the resultant control mesh then the refined control mesh will converge to the subdivision surface. The process can be stopped after the required degree of accuracy has been achieved. We call the resultant control mesh from this process the *final* refined control mesh. The *final* refined control mesh is not the surface but an approximation of the surface. A good approximation is usually achieved after only a few refinements. The control mesh and the subdivision rule for any subdivision surface allow us to approximate the surface to the required degree of accuracy.

1.4 Purpose

There are several applications for the *final* refined control mesh. It can be used in ray tracing to compute intersections between a ray and subdivision surfaces and for other fast intersection results between surfaces. More importantly it can be used to render the surface or more accurately an

approximation to the surface. Depending on the degree of rendering accuracy required, the level of refinement can be determined.

It is this last application that I used in this work to render the subdivision surface. The purpose of this work was to test the hypothesis that the mesh/refinement abstractions can be used to design a generic surface editor. The implementation was twofold. First, I needed to define several surface Refiner objects whose purpose is the refinement of control meshes. Each Refiner is based on a subdivision surface and its subdivision rule. Secondly, I used these Refiner objects to write a subdivision surface editor that provides a generic subdivision surface design and manipulation environment. I designed the editor to be abstract in nature in that it should handle any subdivision surface created from a quadrilateral control mesh. It should also provide support for adding new subdivision surfaces by specifying the subdivision rule through a new Refiner object. The editor provides an environment for studying subdivision surfaces, in particular bivariate box spline surfaces. The prospects for extending the editor to non-quadrilateral meshes have been supported by hiding access to the mesh vertices through the abstraction of a lattice indexing scheme.

2. Subdivision Surfaces

The two subdivision surfaces examined in this work are tensor product cardinal B-spline surfaces and box spline surfaces. The former is bivariate in nature, while the latter is multivariate. However I will only focus on the bivariate case of the latter. In order to understand the two it is necessary to start at the univariate case of splines.

Initially I will start with Schoenberg's theoretical background on univariate cardinal B-splines. The bulk of the theory and results presented in this chapter are based on [Dae91]. These splines form the building blocks of tensor product cardinal B-spline surfaces. I will then extend the univariate theory to the bivariate case and introduce box splines. These bivariate box splines can be used as the basis for box spline surfaces. The theory and results for box splines is also taken from [Dae91]. For each of the three, I will discuss the theory of subdivision to arrive at algorithms to refine a control mesh. Other works are examined at the end of the chapter.

2.1 Univariate Cardinal Splines

Univariate cardinal splines are B-splines over a uniform knot sequence (or cardinal B-spline). The theory of B-splines in general can be found in [Far93]. Only the cardinal case is discussed here. One way of defining the cardinal B-spline of order k is by a recurrence relation,

$$M_k(u) = \int_0^1 M_{k-1}(u-t)dt,$$
$$\text{where } M_1 = \begin{cases} 1, & \text{if } 0 \le u \le 1, \\ 0, & \text{otherwse.} \end{cases} \tag{1}$$

M_1 is the characteristic function over the unit interval. M_2 is the piecewise linear hat function over the interval (0,2), with the two pieces being over (0,1) and (1,2). M_k is a piecewise polynomial over (0,k) with the i^{th} piece being over (i,i+1). The integers 0, 1, …, k are the knots and M_k is a polynomial of degree at most k–1 over the interval between two knots. Figure 3 shows examples of the M_k family of B-splines. M_3 and M_4 have been sketched in the figure to illustrate the shape of these functions.

By induction on k we can realize the following properties of cardinal B-splines.

1. Local Support : $M_k(u) = 0$ for $u \notin [0,k]$,
2. Positivity : $M_k(u) > 0$ for $u \in (0,k)$,
3. Normalization : $\int_0^1 M_k(u)du = 1$,
4. Partition of unity : $\sum_{i=L}^{U} M_k(u-i) = 1$ and $L + k - 1 \le u \le U - k + 1$,
5. $\sum_{i=L}^{U} iM_k(u-i) = u - k/2$, L and U as above,
6. Differentiation formula : $\mathrm{D}M_k(u) = \frac{d}{du} M_k(u) = M_{k-1}(u) - M_{k-1}(u-1)$.

In general, individual polynomials of M_k join with continuity C^{k-2}. This result can be derived by

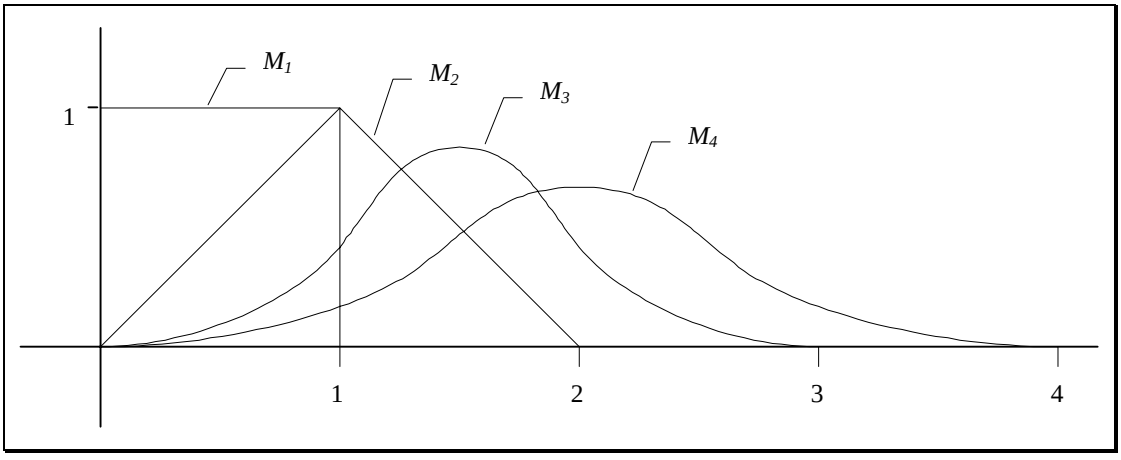


Figure 3. Several examples of Univariate cardinal B-splines.

induction on k and by using the differentiation formula.

Now that we have the fundamentals of univariate cardinal B-splines, we can construct cardinal splines by taking a linear combination of translations and dilations of a cardinal B-spline. In the following equation **R** represents the real number space and **Z** represents the integer space. A cardinal spline of order k is a function of the following form:

$$p(u) = \sum_{i \in Z} c_i M_k(wu - i),$$

where $u \in \mathbf{R}$, given a positive integer w and $c_i \in \mathbf{R}$ for all integers i.

In the previous equation c_i represents the linear combination, w the dilation and i the translation. Let $S_{k,w}$ represent the linear space of functions obtained by varying the c_i over **R**. Also set $S_k = S_{k,1}$. Only a finite number of the B-spline coefficients will be nonzero in practical applications. For this reason we only need to sum over a finite index set of consecutive integers. In general, $S_{k,w}$ consists of all piecewise polynomials of degree < k and smoothness C^{k-2} on the grid $\mathbf{Z}/w = \{i/w: i \in \mathbf{Z}\}$. This result follows from the Curry & Schoenberg Theorem [Sch81].

If $p(u) \in S_{k,w}$, then the control polygon of p in $S_{k,w}$ is given by drawing straight line segments between neighboring points $((i + k / 2) / w, c_i) \in \mathbf{R}^2$. From the positivity and partition of unity properties of cardinal B-splines we have that $p(u)$ lies in the convex hull of those points of the control polygon corresponding to B-splines which are nonzero at u. The values $u = (i + k / 2) / w$ in the domain of the spline are know as the Greville abscissas (or nodal points) of the spline.

So far we have examined the explicit case of the spline. We now focus our attention on the parametric case in order to define cardinal spline curves. If $c_i \in \mathbf{R}^N$, then

$$p(u) = \sum_{i \in Z} c_i M_k(wu - i),$$

where $u \in \mathbf{R}$, given a positive integer w, for all integers i

is called a cardinal spline curve of order k. When N = 2, $p(u)$ is a parametric curve lying on a plane and when N = 3, it is a parametric curve in 3D space. We already saw that in the explicit case the control polygon was defined by the points $((i + k / 2) / w, , c_i) \in \mathbf{R}^2$. Using property (5) of cardinal B-splines from above, we see that the parametric curve $(u, p(u))$ has the same control polygon. The control polygon of the cardinal spline curve is the polygon in $\mathbf{R}^N$ obtained by connecting neighboring c_i by straight line segments.

2.1.1 Subdivision

In the explicit case, we can see that $S_k \subset S_{k,w}$, i.e., any function $p(u) \in S_k$ can be written in terms of a B-spline in $S_{k,w}$. The reason is that p is also a piecewise polynomial of degree < k and smoothness C^{k-2} on the grid $\mathbf{Z}/w = \{i/w: i \in \mathbf{Z}\}$, for a positive integer w and $S_{k,w}$ includes all such piecewise polynomials. Therefore, we can write p in terms of a refined set of cardinal B-splines as follows.

$$p(u) = \sum_j c_j M_k(u - j) = \sum_i b_i M_k(wu - i) \quad \text{for coefficients } c_j \text{ and } b_i. \tag{2}$$

To see the relationship between c_j and b_i we use a theorem defined and proven in [Dae91]. The theorem states that

$$M_k(u) = \sum_j \beta_k^w(j) M_k(wu - j)$$

$$\text{where } q_k^w(z) = \frac{1}{w^{k-1}}\left(1 + z + z^2 + \cdots z^{w-1}\right)^k = \sum_j \beta_k^w(j) z^j.$$

Here the coefficients β represent the power form coefficients of the generating function q. The above equations gives us a formula to write a cardinal B-spline $M_k(u)$ in terms of a refined cardinal B-spline $M_k(wu–i)$.

When $w = 2$, the generating function $q_k^w(z)$ is given as

$$q_k^2 = \frac{(1+z)^k}{2^{k-1}} = 2^{1-k} \sum_{j=0}^{k} \binom{k}{j} z^j. \tag{3}$$

This allows us to write M_k in terms of a refined cardinal B-spline on the $\mathbf{Z}/2$ grid.

$$M_k(u) = \frac{1}{2^{k-1}} \sum_{j=0}^{k} \binom{k}{j} M_k(2u - j).$$

Now we can write b_i as a linear combination of c_j :

$$b_i = \sum_j c_j \alpha_{j,k}(i) \qquad \text{where } \alpha_{j,k}(i) = \beta_k^w(i + wj). \tag{4}$$

The numbers $\alpha_{j,k}$ are called *discrete* cardinal B-splines. In the specific case, when $k = 4$ and $w = 2$, the cubic cardinal B-spline can be refined as

$$M_4(u) = \tfrac{1}{8} M_4(2u) + \tfrac{1}{2} M_4(2u-1) + \tfrac{3}{4} M_4(2u-2) + \tfrac{1}{2} M_4(2u-3) + \tfrac{1}{8} M_4(2u-4)$$

$$\text{and } b_{2j} = \tfrac{1}{2} c_j + \tfrac{1}{2} c_{j+1} \text{ and } b_{2j+1} = \tfrac{1}{8} c_j + \tfrac{3}{4} c_{j+1} + \tfrac{1}{8} c_{j+2}.$$

This last result allows us to calculate the coefficients b_j of the refined basis $M_k(2u - j)$ on the $\mathbf{Z}/2$ grid from the coefficients of the original basis on $\mathbf{Z}$.

These results also apply to the parametric case of cardinal spline curves. If coefficients $c_i \in \mathbf{R}^3$, represent control vertices of a cardinal spline curve in 3-space, then using (4) we can generate a

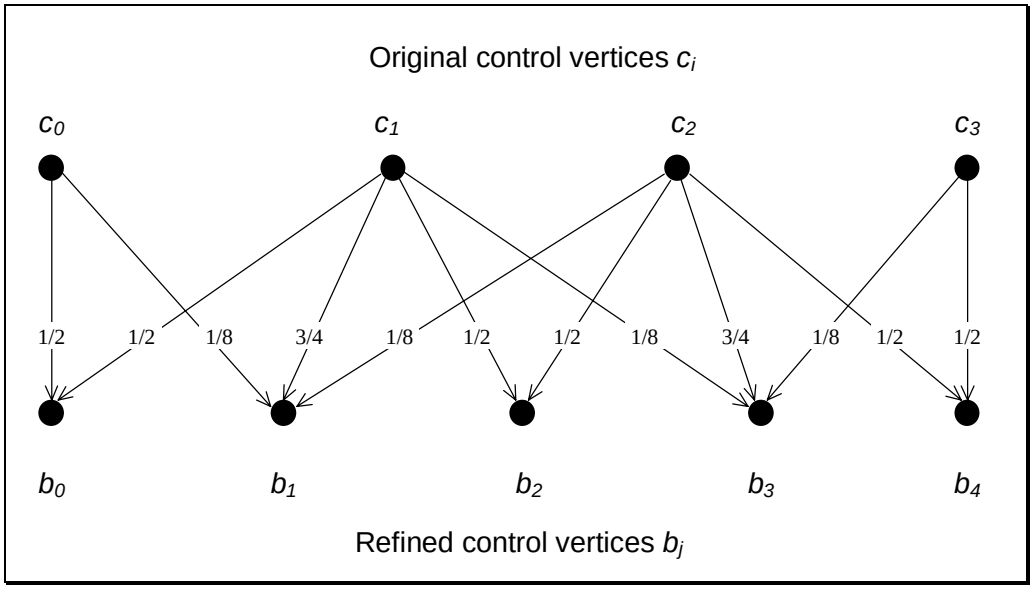


Figure 4. Coefficients c_i written in terms of coefficients b_j for $w = 2$ and $k = 4$.

refined set of control vertices representing the same curve. This refinement process is equivalent to adding more knots to the uniform knot sequence. Figure 4 illustrates this operation for the case where 4 coefficients are refined resulting in 5 coefficients. For each b_j a line is drawn to each c_i that is used in the calculation. Fractional weights are associated with each line to indicate the linear combination.

[Dae91] shows that the control polygon will be close to the curve if w is large. Alternatively, it is recommended to use $w = 2$ and repeat the process several times to avoid high computational costs. We can use the fact that as w gets large, the control polygon more closely matches the curve, for rendering and computing intersections of these curves.

2.2 Tensor Product Cardinal Splines

Although cardinal splines are not subdivision surfaces, we can use the theory discussed in the previous section to build subdivision surfaces. One surface that can be constructed from cardinal splines is the tensor product cardinal spline surface. To build such a surface we need to introduce tensor product cardinal B-splines which are constructed from cardinal B-splines along two separate dimensions.

Given two cardinal B-splines $M_k(u)$ and $M_l(v)$ of order k and l respectively, a tensor product B-spline is given by $M_{k,l}(u,v) = M_k(u)\, M_l(v)$. We can apply (1) to both $M_k(u)$ and $M_l(v)$ to define the tensor product cardinal B-spline recursively. Therefore, we can successively integrate on either the u or v variable. The recursive definition for tensor product B-splines is

$$M_{k,l}(u,v) = \int_0^1 M_{k-1,l}(u-t,v)dt = \int_0^1 M_{k,l-1}(u,v-t)dt\ .$$

The starting condition for the recurrence is given by

$$M_{1,1}(u,v) = \begin{cases} 1, & \text{if } 0 \le u \le 1 \text{ and } 0 \le v \le 1, \\ 0, & \text{otherwise.} \end{cases}$$

Some properties of these splines can be listed as follows.

1. Local Support: $M_{k,l}(u,v) = 0$ for $(u,v) \notin [0,k] \times [0,l]$,
2. Positivity : $M(u,v) > 0$ for $(u,v) \in (0,k) \times (0,l)$,
3. Normalization : $\int_0^1 \int_0^1 M_{k,l}(u,v)dudv = 1$,
4. Partition of unity: $\sum_{(i,j)\in Z^2} M_{k,l}(u-i, v-j) = 1$.

The bi-cubic tensor product cardinal spline is shown in Figure 5.

By taking linear combinations of shifts of these tensor product cardinal B-splines we can construct a tensor product cardinal spline surface. This concept is defined formally as

$$T_{k,l}(u,v) = \sum_{j\in \mathbf{Z}} \sum_{i\in \mathbf{Z}} V_{i,j}\, M_{k,l}(w_1 u - i, w_2 v - j) \text{ for positive integers } w_1 \text{ and } w_2.$$

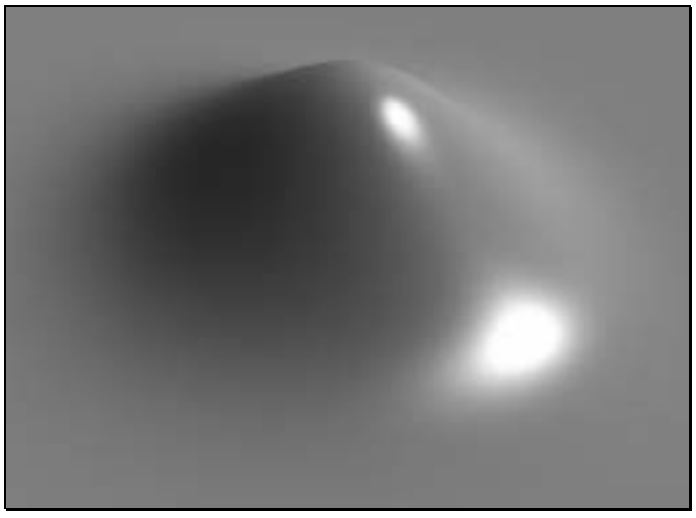


Figure 5. Bi-cubic tensor product cardinal spline.

Here $V_{i,j} \in \mathbf{R}^3$ represents the coefficients of a parametric surface. Practically, these coefficients are only defined (i.e., nonzero) for a finite contiguous index set of the 2 dimensional domain, $\mathbf{Z}^2$. The control mesh for the surface is given by connecting the points $V_{i,j}$ with the 4 neighboring points $V_{i-1,j}$, $V_{i+1,j}$, $V_{i,j-1}$, $V_{i,j+1}$. We now examine the subdivision theory behind the refinement process of this control mesh.

2.2.1 Subdivision

The subdivision rule for tensor product cardinal spline surfaces is a simple process. The theory is based on the subdivision of univariate cardinal spline curves. Given a coefficient set $V_{i,j}$, we wish to find the coefficient set, $W_{s,t}$ on the refined basis. That is, if

$$T_{k,l}(u,v) = \sum_{j\in\mathbf{Z}}\sum_{i\in\mathbf{Z}} V_{i,j} M_{k,l}(u-i, v-j) = \sum_{t\in\mathbf{Z}}\sum_{s\in\mathbf{Z}} W_{s,t} M_{k,l}(w_1 u - s, w_2 v - t),$$

then find the relationship between $W_{s,t}$ and $V_{i,j}$. To do so, we apply the subdivision rule of section 2.1.1 twice - once to M_k and once to M_l. This procedure can be applied because everything that is valid for univariate cardinal spline curves is valid for tensor product cardinal spline surfaces along each of the dimensions u and v individually. First we apply the subdivision along u to get $U_{s,j}$ from $V_{i,j}$. Then we do the same in the other direction to get $W_{s,t}$ from $U_{s,j}$. This result is formally expressed as follows.

$$T_{k,l}(u,v) = \sum_{j\in\mathbf{Z}}\sum_{i\in\mathbf{Z}} V_{i,j} M_{k,l}(u-i, v-j) = \sum_{j\in\mathbf{Z}}\sum_{i\in\mathbf{Z}} V_{i,j} M_k(u-i) M_l(v-j)$$

$$= \sum_{j\in\mathbf{Z}}\left(\sum_{s\in\mathbf{Z}} U_{s,j} M_k(w_1 u - s)\right) M_l(v-j) = \sum_{t\in\mathbf{Z}}\left(\sum_{s\in\mathbf{Z}} U_{s,j} M_k(w_1 u - s)\right) M_l(w_2 v - t).$$

Therefore we can write $T_{k,l}(u,v) = \sum_{t\in\mathbf{Z}}\sum_{s\in\mathbf{Z}} W_{s,t} M_{k,l}(w_1 u - s, w_2 v - t)$,

$$\text{where } U_{s,j} = \sum_i V_{i,j}\alpha_{i,k}(s) \quad \text{and} \quad W_{s,t} = \sum_j U_{s,j}\alpha_{j,l}(t).$$

Here, α represents the *discrete* cardinal B-splines defined in (4) of Section 2.1.1. In the specific case when $w_1 = w_2 = 2$, and $k = l = 4$, starting with the control mesh $V_{i,j}$, refinement along the i direction gives,

$$U_{2i,j} = \tfrac{1}{2}V_{i,j} + \tfrac{1}{2}V_{i+1,j} \text{ and } U_{2i+1,j} = \tfrac{1}{8}V_{i,j} + \tfrac{3}{4}V_{i+1,j} + \tfrac{1}{8}V_{i+2,j}.$$

Then, refining along the other direction, j, gives,

$$W_{i,2j} = \tfrac{1}{2}U_{i,j} + \tfrac{1}{2}U_{i,j+1} \text{ and } W_{i,2j+1} = \tfrac{1}{8}U_{i,j} + \tfrac{3}{4}U_{i,j+1} + \tfrac{1}{8}U_{i,j+2}.$$

Now we can refine a control mesh by performing the subdivision rule detailed above. Similar to the univariate cardinal spline case, it is better to repeat the control mesh refinement process with $w_1 = w_2 = 2$ instead of refining using large w_1, w_2. After such an operation, the refined control mesh will more closely represent the underlying surface. This refinement process can be used to render the surface and for computing intersections.

2.3 Box Splines

Box splines are locally supported piecewise polynomials on uniform grids. The theory of box splines has generally been motivated by Schoenberg's treatment of univariate cardinal splines. We now use theory of univariate cardinal splines discussed in section 2.1 to introduce bivariate box splines. This theory can also be extended to the multivariate case to define multivariate box splines, but only the bivariate case is discussed here. The multivariate case is discussed in [Dae91].

2.3.1 Bivariate Box Splines

Bivariate box splines can be defined as follows. Let $x^i=(x_i,y_i)$ be $\mu > 2$ vectors in $\mathbf{R}^2$. Then define $\boldsymbol{X}_2 = \{x^1,x^2\}$, where x^1 and x^2 are linearly independent and $\boldsymbol{X}_j = \boldsymbol{X}_{j-1} \cup \{x^j\}$. A bivariate box spline is a function $M(u,v \mid \boldsymbol{X}_\mu)$:$\mathbf{R}^2 \rightarrow \mathbf{R}$ defined recursively by

$$M\left(u,v\middle|\boldsymbol{X}_\mu\right) = \int_0^1 M\left(u - tx_\mu, v - ty_\mu \middle| \boldsymbol{X}_{\mu-1}\right)dt \quad \text{and}$$

$$M\left(u,v\middle|\boldsymbol{X}_2\right) = \begin{cases} \frac{1}{\left|\det\left(\boldsymbol{X}_2\right)\right|}, & \text{if } (u,v) \in \left[\boldsymbol{X}_2\right], \\ 0, & \text{otherwise.} \end{cases}.$$

$\boldsymbol{X}_\mu$ represents the direction vectors and $[\boldsymbol{X}_\mu] = \{ t_1x^1 + ... + t_\mu x^\mu : 0 \le t_j \le 1, 1 \le j \le \mu \}$ represents the support of the box spline. We note that $M(u,v)$ can only be nonzero if (u,v) belongs to the interior of this support.

Box splines are best understood by examples. If $\mu = 2$ and $X_\mu = ((1,0), (0,1))$, then we get the square support show in Figure 7a. The box spline is the characteristic function over this support. If $\mu = 3$ and $\boldsymbol{X}_\mu = ((1,0), (0,1), (1,1))$, then we get the hexagonal support shown in Figure 7b. This box spline $M(u,v \mid \boldsymbol{X}_3)$ is the piecewise linear "hat-function" which is equal to 1 at (1,1) and 0 outside the hexagon with vertices {(0,0), (1,0), (2,1), (2,2) (1,2), (0,1)} (see Figure 6). This shape is a hexagonal pyramid and this three directional box spline is known as the Courant finite element.

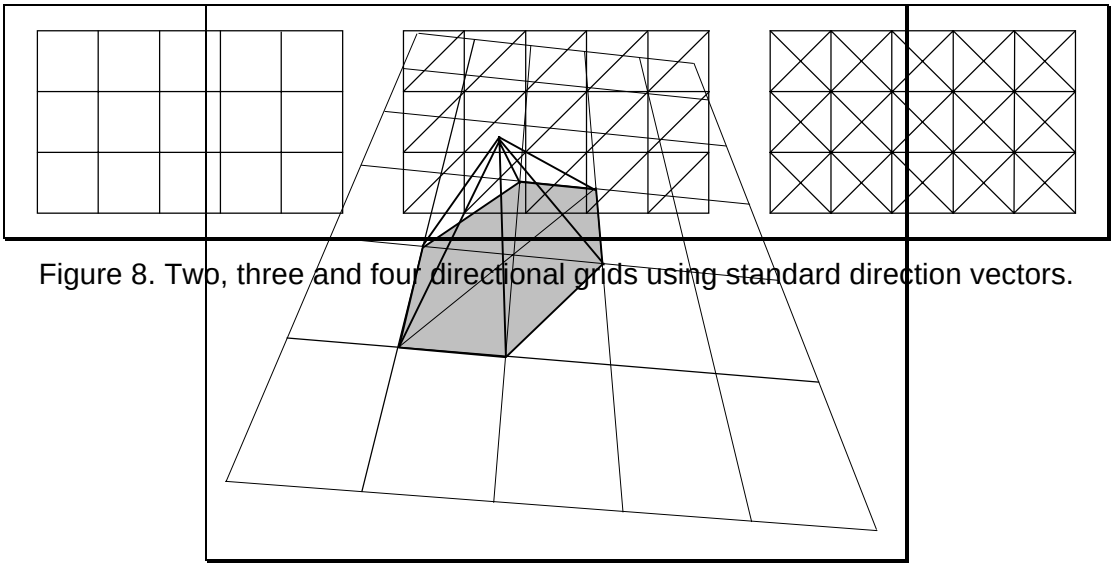


Figure 8. Two, three and four directional grids using standard direction vectors.

Figure 6. Wireframe representation of the Courant Finite Element. The shaded region represents the box spline support.

We now introduce the situation where the directions of a box spline consists of repetitions of a few nonparallel directions. Let $k_1, k_2, ..., k_r$ be positive integers with $\mu = \Sigma_i k_i$ and let r be an integer ≥ 2. Let $E = (d^1, d^2, ..., d^r)$ be pairwise nonparallel vectors in $\mathbf{R}^2$. Let $\boldsymbol{X}_\mu = (d^1, ..., d^1, d^2, ..., d^2, . . . , d^r, ..., d^r)$, where each d^i is repeated k_i times. We can write $\boldsymbol{X}_\mu = E^k = E^{(k_1, ..., k_r)}$. Then a r directional box spline is written as $M(u,v \mid E^k)$. If for $r \leq 4$, we have the standard directions given by $d^1 = (1,0)$, $d^2 = (0,1)$, $d^3 = (1,1)$ and $d^4 = (1,-1)$, then we denote $M(u,v \mid E^k)$ by $M_k(u,v)$. For example, the *Courant* finite element is represented by $M_{(1,1,1)}(u,v)$.

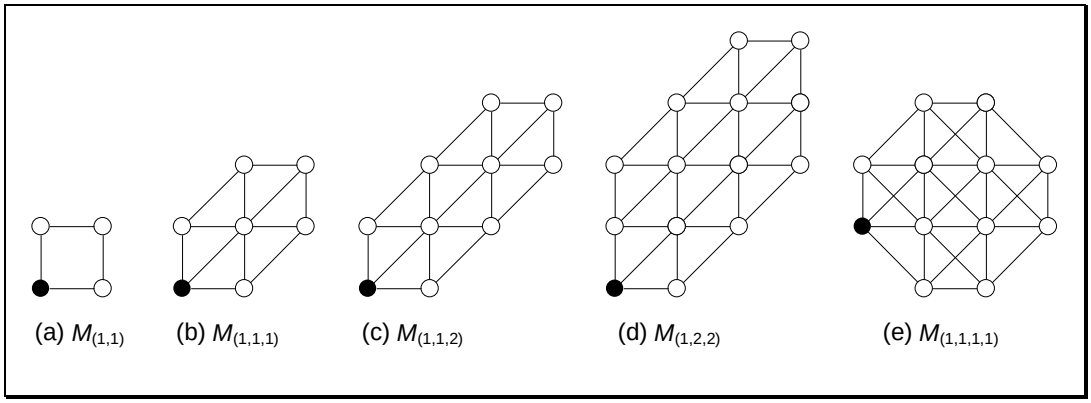


Figure 7. Supports for box splines.

These standard directions generate regular r directional grids G_r for r = 2, 3, 4 (see Figure 8). Figure 7 shows a series of supports for several box splines. These box splines are $M_{(1,1)}$, $M_{(1,1,1)}$, $M_{(1,1,2)}$, $M_{(1,2,2)}$ and $M_{(1,1,1,1)}$ respectively. Note that the dark circle represents the origin. The last box spline, $M_{(1,1,1,1)}$ (Figure 7e) is called the the *Zwart* Element. The standard directions d^1, d^2, d^3, d^4 generate some of the more important and simple grids seen in Figure 8. However, other directions d^5 = (0.5, 1), d^6 = (1, 0.5), d^7 = (1, –0.5) and d^8 = (0.5, –1) can be introduced to build other grids. Also non-standard directions can also be used to create various supports and grids including triangular and other non-rectangular supports. Two box splines, $M_{(1,1,1,1)}$ and $M_{(2,2,1,1)}$ used in the this work are shown in Figure 9 and Figure 10 and respectively.

As in section 2.1, we can list some properties of box splines for $M(u,v) = M(u,v|\,\boldsymbol{X}_\mu)$:

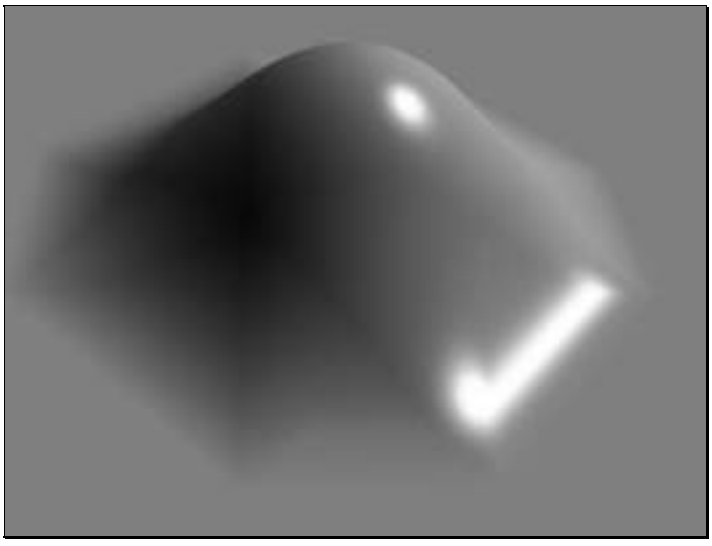


Figure 9. The $M_{(1,1,1,1)}$ box spline.

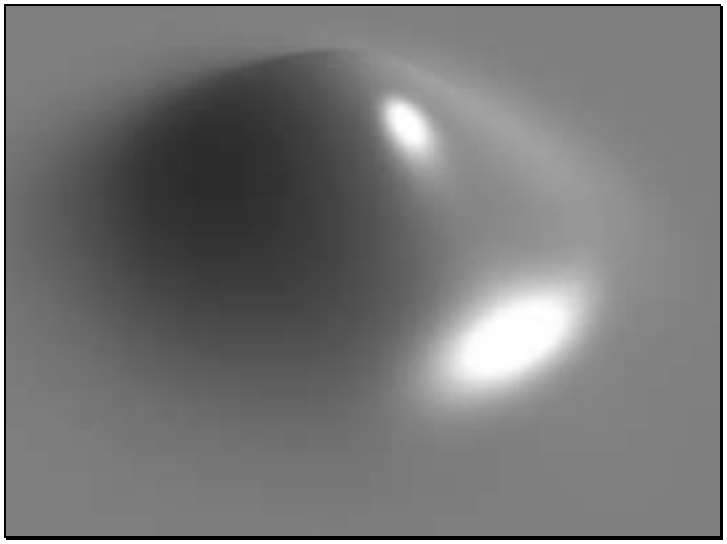


Figure 10. The $M_{(2,2,1,1)}$ box spline.

- Local Support: $M(u,v) = 0$ for $(u,v) \notin [\boldsymbol{X}_\mu]$,
- Positivity : $M(u,v) > 0$ for $(u,v) \in]\boldsymbol{X}_\mu[$, where $]\boldsymbol{X}_\mu[= \{t_1x^1 + ... + t_\mu x^\mu : 0 < t_j < 1, 1 \le j \le \mu\}$,
- Normalization : $\int\int_{[X_\mu]} M(u,v)dudv = 1$,
- Partition of unity: $\sum_{(i,j)\in Z^2} M(u-i, v-j \,|\, \boldsymbol{X}_\mu) = 1$.

We now state a theorem from [Dae91] that gives us a better insight on the shape of a box spline on the r directional grids G_r, r = 2, 3, 4. M_k is a piecewise polynomial of degree at most $\mu-2$. Also $M_k \in C^{\mu-2-k_i}$ across a direction d_i. Therefore by introducing multiplicity along a specific direction we reduce the continuity across that direction.

2.3.2 Box Spline Surfaces

We can construct box spline surfaces by summation over several translated and dilated versions of a box spline. A *box spline surface* is defined as

$$p(u,v) = \sum_{(i,j)\in Z^2} c_{i,j} M\left(wu - i, wv - j \,|\, \boldsymbol{X}_\mu\right) \text{ where } c_{i,j} \in \mathbf{R}^N \text{ and } u,v \in \mathbf{R}.$$

If N = 1 then the $p(u,v)$ represents a bivariate cardinal spline (explicit case) and if N = 3, the surface is a parametric box spline surface.

Let $S_{k,w}$ represent the linear space of surfaces obtained by varying the $c_{i,j}$ over $\mathbf{R}^3$ and set $S_{k,1} = S_k$. We note that the box spline surface is a piecewise polynomial on the grid G_r / w, $r > 2$. Although the explicit control polygon equation for bivariate box spline is not stated in this document, we note that similar to the univariate cardinal splines case, as w goes to infinity, the control polygon will converge to the surface. Therefore, the control polygon can be used as an approximation of the surface. For more details on the control polygon see [Dae91]. An example box spline surface based on the $M_{(1,1,1,1)}$ box spline is shown in Figure 15. The underlying box spline support is clearly visible in this surface.

2.3.3 Subdivision

Given a box spline surface in the space S_k on the grid G_r, we can refine it to a surface in $S_{k,w}$ on the grid G_r / w. This refinement is possible by applying the following result taken from [Cav89].

For $w \in \mathbf{N}$ and $(u,v) \in \mathbf{R}^2$, $x^l = (x_l, y_l) \in \boldsymbol{X}_\mu$, $l = 1, 2, \ldots, \mu$,

$$M\left(u,v\middle|\boldsymbol{X}_\mu\right) = \sum\nolimits_{(i,j)\in Z^2} \beta^w\left(i,j\middle|\boldsymbol{X}_\mu\right) M\left(wu-i, wv-j\middle|\boldsymbol{X}_\mu\right), \tag{5}$$

where the generating function for $\beta^w\left(i,j\middle|\boldsymbol{X}_\mu\right)$ is

$$q\left(y,z\middle|\boldsymbol{X}_\mu\right) = \sum_{(i,j)} \beta^w\left(i,j\middle|\boldsymbol{X}_\mu\right) y^i z^j = \frac{1}{w^{\mu-2}} \prod_{l=1}^{\mu} \left(1 + y^{x_l} z^{y_l} + y^{2x_l} z^{2y_l} + \ldots + y^{(w-1)x_l} z^{(w-1)y_l}\right). \tag{6}$$

For the box spline $M(u,v \mid \boldsymbol{X}_\mu)$, $\left\{(i,j), \beta^w\left(i,j\middle|\boldsymbol{X}_\mu\right)\right\}$ is called the *mask* for the box spline.

We can now apply this box spline subdivision rule to refine a box spline surface. If $p \in S_k$, then from (5) we know that $p \in S_{k,w}$ for all $w \in \mathbf{N}$. Therefore, given coefficients $c_{m,n}$, we need to determine the coefficients $b_{i,j}$ in

$$p(u,v) = \sum_{m,n} c_{m,n} M\left(u-m, v-n\middle|\boldsymbol{X}_\mu\right) = \sum_{i,j} b_{i,j} M\left(wu-i, wv-j\middle|\boldsymbol{X}_\mu\right). \tag{7}$$

From (5) we find that since

$$p(u,v) = \sum_{m,n} c_{m,n} \sum_{i,j} \beta^w\left(i,j\middle|\boldsymbol{X}_\mu\right) M\left(w(u-m)-i, w(v-n)-j\middle|\boldsymbol{X}_\mu\right),$$

we can give the relationship between the coefficients:

$$b_{i,j} = \sum_{m,n} \alpha^w_{m,n}\left(i,j\middle|\boldsymbol{X}_\mu\right) c_{m,n}\ . \tag{8}$$

where

$$\alpha^w_{m,n}\left(i,j\middle|\boldsymbol{X}_\mu\right) = \beta^w\left(i-wm, j-wn\middle|\boldsymbol{X}_\mu\right)$$

is called a *discrete box spline*. For more information on discrete box spline, see [Dae91]. In the parametric case of box spline surfaces, the refinement process is applied simultaneously in each component of the coefficients.

Specific Case

Here we give a subdivision algorithm for the specific case, box spline surfaces based on the quartic $M_{(2,2,1,1)}$ box spline. For this box spline, from (6) the generating function is

$$\begin{aligned} q(y,z) &= (1+y)^2 (1+z)^2 (1+yz)(1+y/z) \\ &= 1 + 4y + 2z + 6yz + y/z + 6y^2 + z^2 + 4yz^2 + 8y^2z + 6y^2z^2 + yz^3 + 2y^2z^3 \\ &\quad + 6y^3z + 4y^3z^2 + y^3z^3 + 2y^2/z + 4y^3 + y^3/z + y^4 + 2y^4z + y^4z^2. \end{aligned}$$

This generating function gives us a formula for writing $M_{(2,2,1,1)}$ in terms of several refined box splines on the refined grid $G_r / 2$.

$$\begin{aligned} M(u,v) = {} & M(2u,2v) + 4M(2u-1,2v) + 2M(2u,2v-1) + 6M(2u-1,2v-1) + M(2u-1,2v+1) + 6M(2u-2,2v) \\ & + M(2u,2v-2) + 4M(2u-1,2v-2) + 8M(2u-2,2v-1) + 6M(2u-2,2v-2) + M(2u-1,2v-3) \\ & + 2M(2u-2,2v-3) + 6M(2u-3,2v-2) + M(2u-3,2v-3) + \\ & 2M(2u-2,\ 2v+1) + 4M(2u-3,2v) + M(2u-3,2v+1) + M(2u-4,2v) + 2M(2u-4,2v-1) + \\ & M(2u-4,2v-2). \end{aligned}$$

From the generating function we can generate the mask for the box spline. The mask is shown in Figure 11.

If the surface $p(u,v)$ in (7) is based on the $M_{(2,2,1,1)}$ box spline, then by using (8) we can refine the surface. Usually the refinement is performed several times with $w = 2$ instead of using a large value of w. We will use the same approach here and use $w = 2$ in (8). Therefore the control mesh $c_{m,n} \in \mathbf{R}^3$, $m,n \in \mathbf{Z}$, defining the surface can be refined to a control mesh $b_{i,j}$ on the grid $G_r / 2$. The relationship between the control meshes is

$$\begin{aligned} b_{i,j} &= c_{m-2,n-1} + c_{m-2,n} + 6\,c_{m-1,j-1} + 6\,c_{m-1,j} + c_{m,j-1} + c_{m,j}, \\ b_{i+1,j} &= 4\,c_{m-1,j-1} + 4\,c_{m-1,j} + 4\,c_{m,j-1} + 4\,c_{m,j}, \\ b_{i,j+1} &= 2\,c_{m-2,j} + 2\,c_{m-1,j-1} + 8\,c_{m-1,j} + 2\,c_{m-1,j+1} + 2\,c_{m,j}, \\ b_{i+1,j+1} &= c_{m-1,j-1} + 6\,c_{m-1,j} + c_{m-1,j+1} + c_{m,j+1} + 6\,c_{m,j} + c_{m,j+1}, \quad \text{where } i = 2m \text{ and } j = 2n. \end{aligned}$$

This result gives us a specific subdivision algorithm for box spline surfaces based on the $M_{(2,2,1,1)}$ box spline.

Figure 12 and Figure 13 show a 8x8 control mesh (shown in wireframe) and the refined control mesh (shown as a shaded surface). The control mesh was refined once and five times respectively.

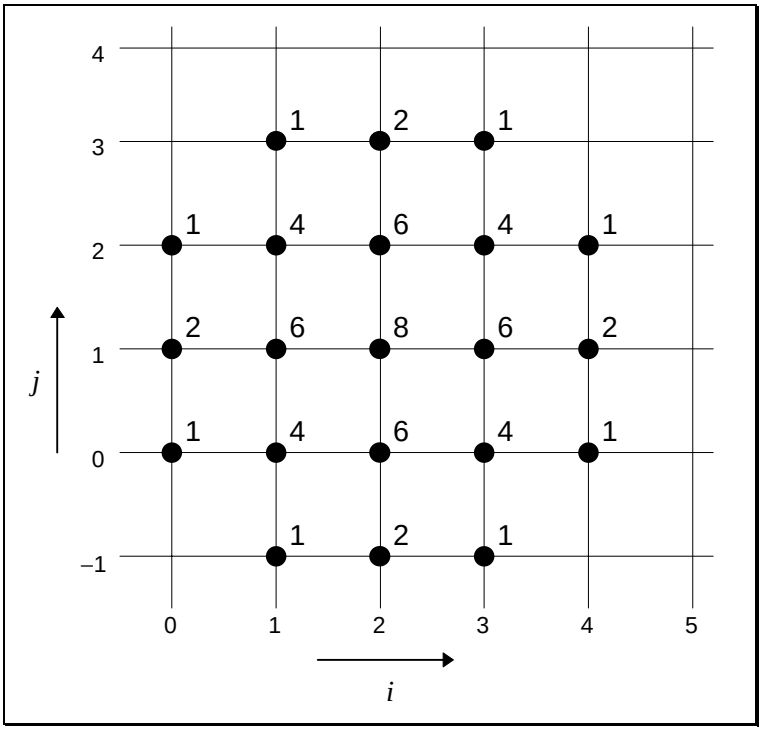


Figure 11. Mask for the $M_{(2,2,1,1)}$ box spline.

In Figure 13, the refined control mesh is almost indistinguishable from the underlying box spline surface.

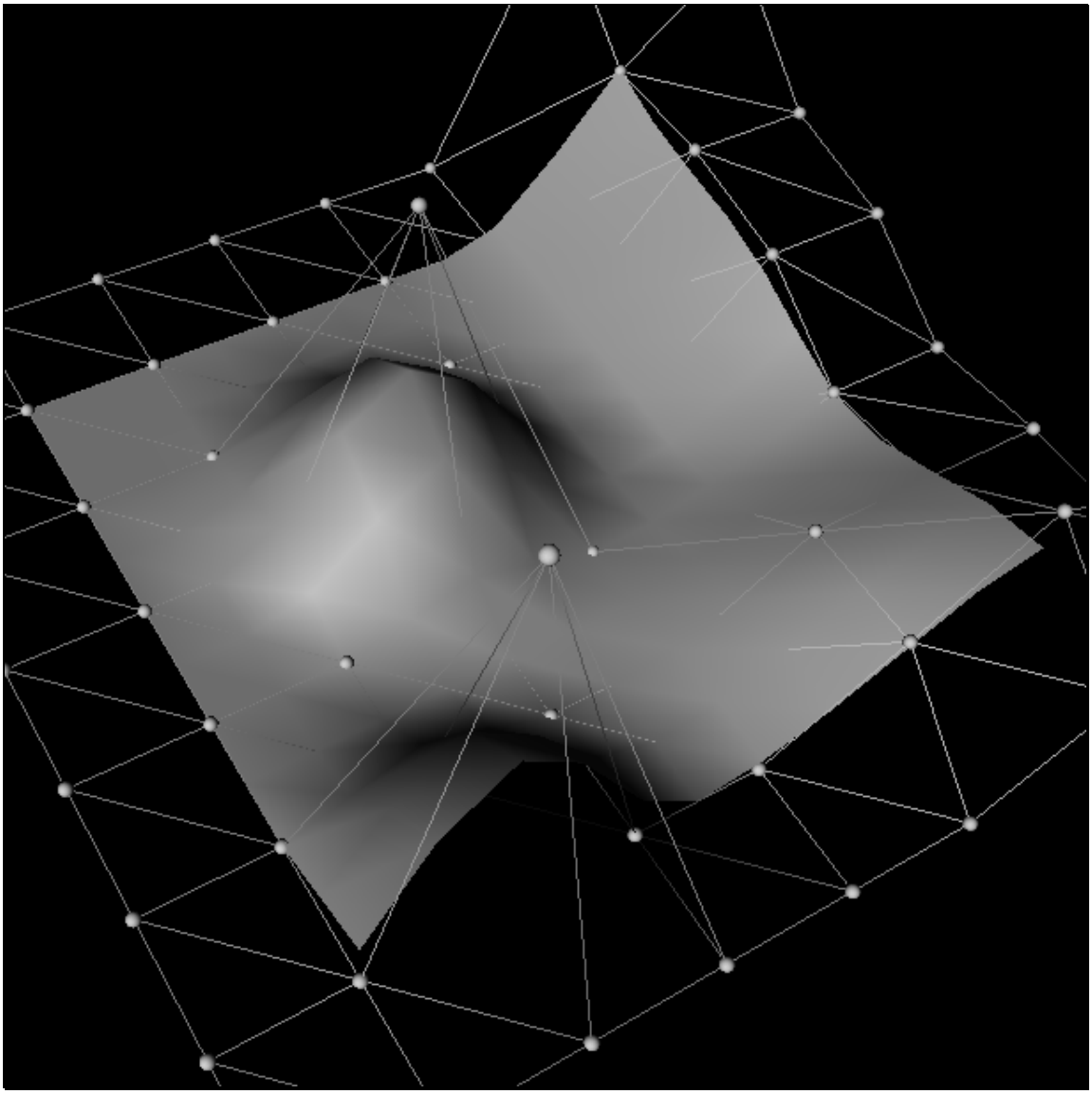


Figure 12. Box spline surface based on the $M_{(2,2,1,1)}$ box spline. This surface approximation is generated by refining the control mesh once.

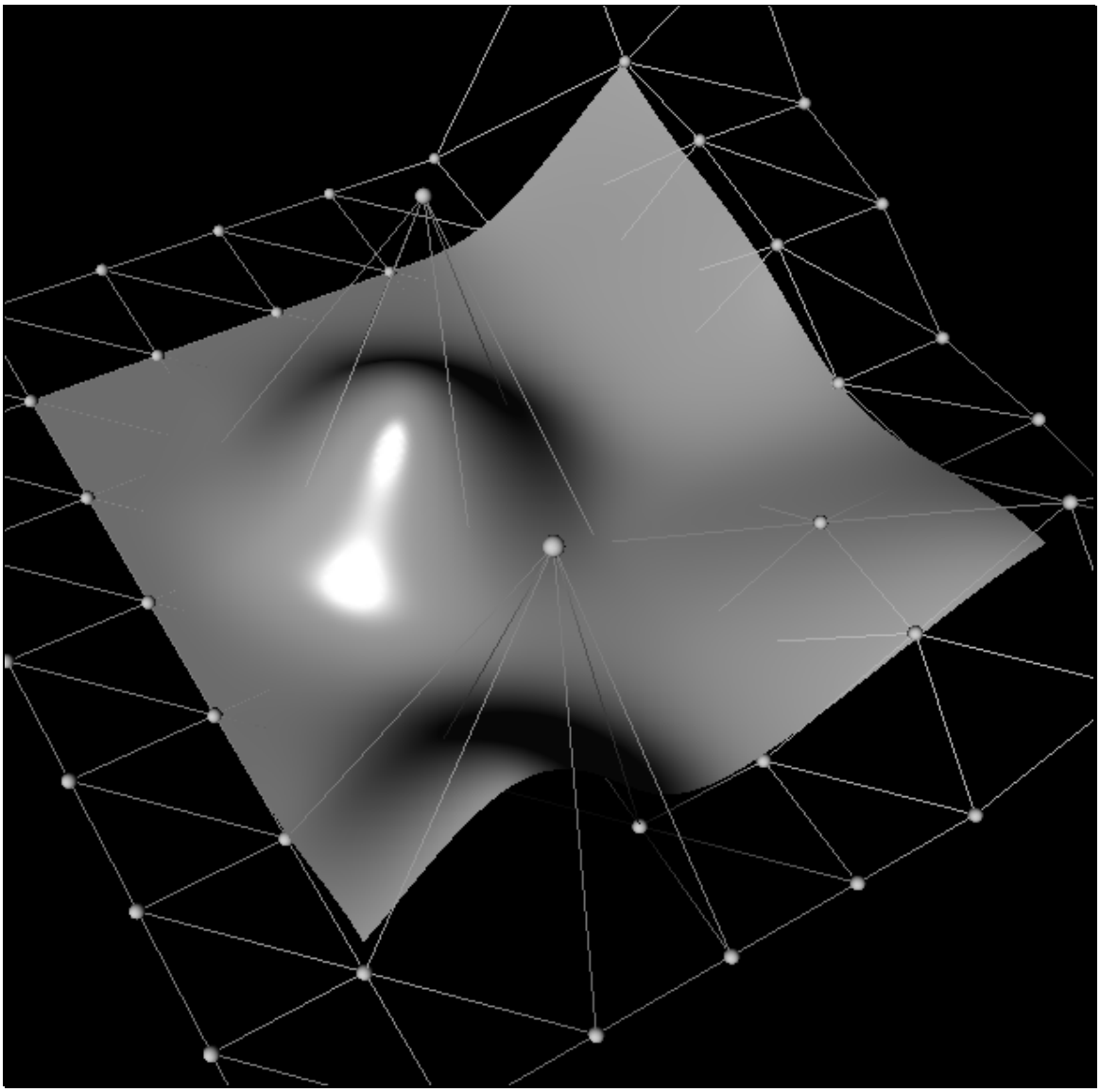


Figure 13. Box spline surface based on the $M_{(2,2,1,1)}$ box spline. The surface is generated by refining the control mesh five times.

Direction Vector based Subdivision Algorithm

We now present another algorithm for generating the same subdivision presented earlier. This subdivision algorithm for box spline surfaces is based on direction vectors of the box spline. There are several variations of this algorithm. [Dae91] presents a version limited to box splines composed of the four standard direction vectors. [Hol89] presents one limited to $w = 2$. [deB95] gives a generic version applicable to any multivariate box spline surface. We present the latter simplified for the bivariate case. The derivation, which is based on generalizing the results from the previous section, is not presented here and can be found in [Hol89] and [deB95].

In simplified terms, the algorithm takes a control mesh and averages it along each direction vector of the box spline. Formally, given $w \in \mathbf{N}$, $w > 1$, coefficients $c_{m,n}$ of the original control mesh, then the refined control mesh $b_{i,j}$ is given in two steps.

1. $b_{i,j} = w^2 c_{i,j}$ where $i = m/w$, $j = n/w$
2. For each direction vector $x^l = (x_l, y_l) \in \boldsymbol{X}_\mu$, $l = 1, \ldots, \mu$

$$b_{i,j} = \frac{1}{w}\left(b_{i,j} + b_{i+x_l, j+y_l} + b_{i+2x_l, j+2y_l} \ldots + b_{i+(w-1)x_l, j+(w-1)y_l}\right)$$

An important point for step 1 is that $c_{i,j} = 0$ where $i,j \notin \mathbf{Z}^2 / w$. In the case when $w = 2$, step 2 simplifies to $b_{i,j} = \frac{1}{2}\left(b_{i,j} + b_{i+x_l, j+y_l}\right)$.

The algorithm is fast, numerically stable and straightforward. If we repeat the algorithm several times, the refined control mesh converges to the underlying surface. We can apply this algorithm to any box spline surface based on a box spline with arbitrary direction vectors. If we were to apply it to a box spline surface based on the $M_{(2,2,1,1)}$ box spline we would have achieved the same subdivision from the previous section.

The results from this algorithm can be seen in Figure 14 and Figure 15. They show a 8x8 control mesh (shown in wireframe and the refined control mesh (shown as a shaded surface). The control mesh is based on the $M_{(1,1,1,1)}$ box spline and was refined two and four times respectively. In the latter figure, the refined control mesh is almost indistinguishable from the underlying box spline surface. In both figures the underlying $M_{(1,1,1,1)}$ box spline is clearly visible.

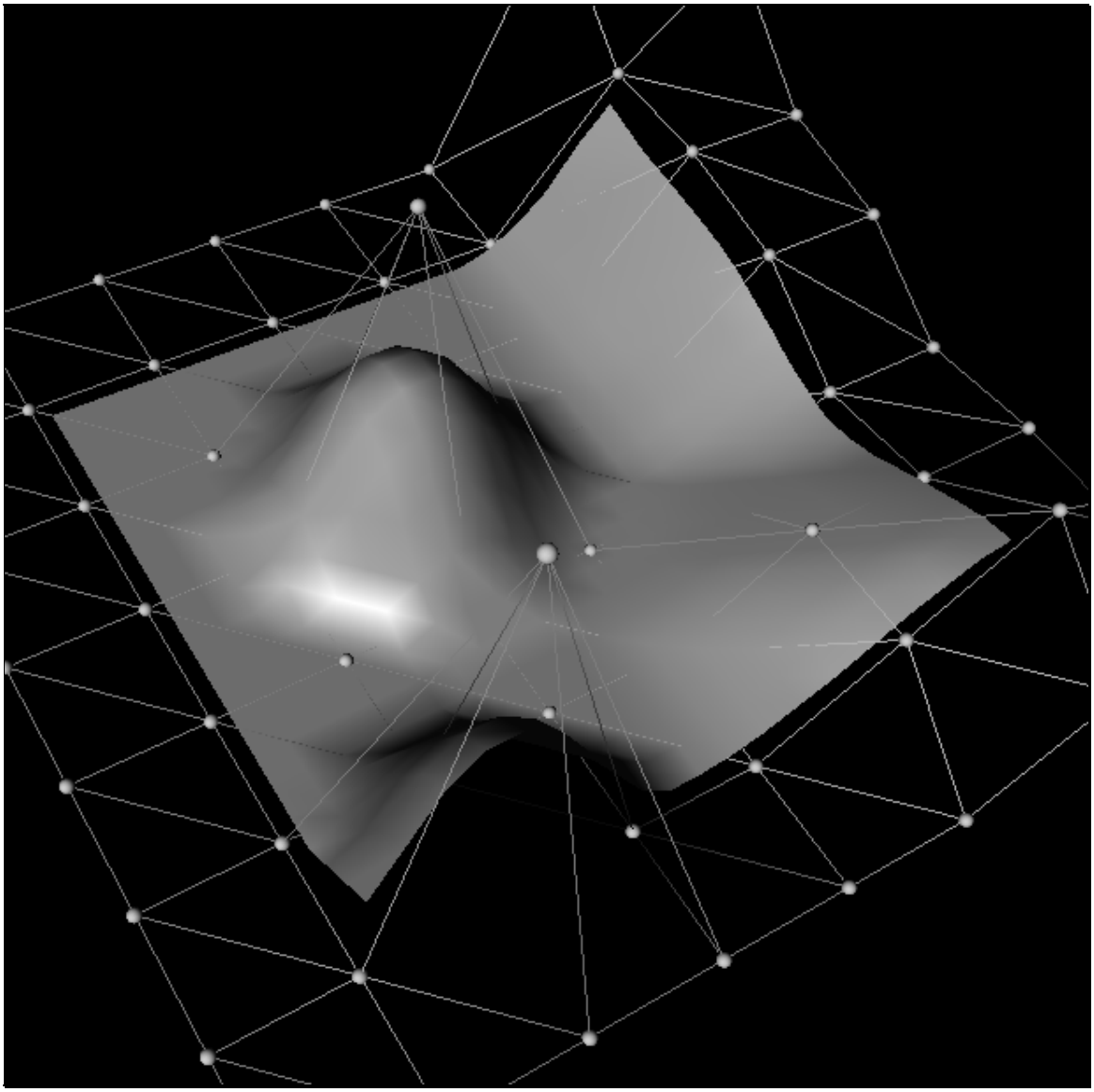


Figure 14. Box spline surface based on the $M_{(1,1,1,1)}$ box spline. The surface is generated by refining the control mesh twice.

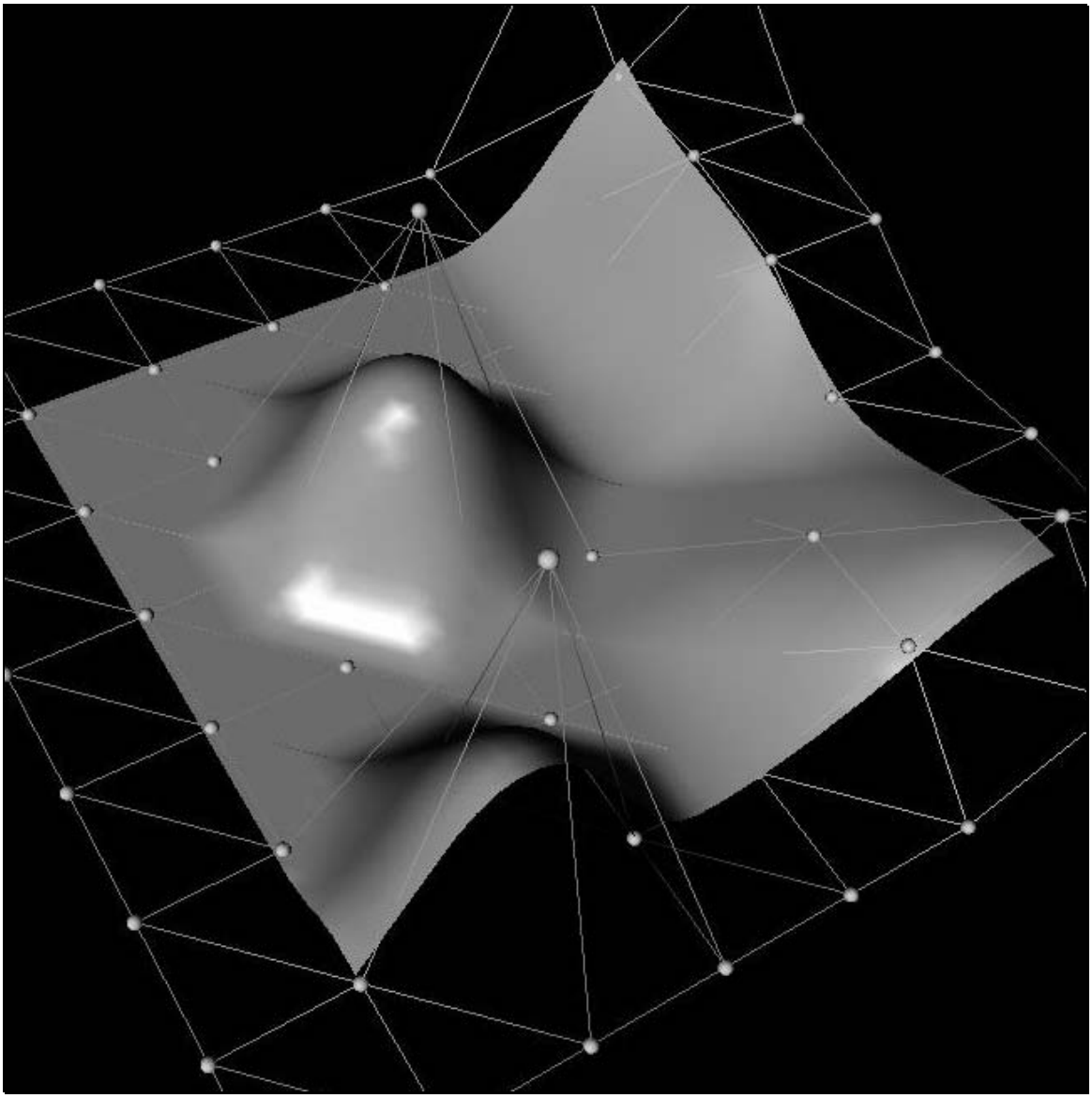


Figure 15. Box spline surface based on the $M_{(1,1,1,1)}$ box spline. The surface is generated by refining the control mesh four times.

2.4 Other Works

Subdivision algorithms first appeared in CAGD with Chaikin's work on corner cutting [Cha74]. It involved subdivision of quadratic cardinal B-splines curves. These curves are the same as $M_3(u)$ in equation (1). The idea was to generate smooth curves by recursively cutting the corners from a control polygon. The general case of this algorithm for an arbitrary k was presented in Section 2.1.1.

The works discussed below deal with subdivision algorithms for surfaces. The algorithms are applied to a mesh to generate a refined mesh. However, they deal with meshes where the surface interpolates the mesh vertices instead of representing the control mesh as we have discussed in this work. Vertices on a control mesh may or may not lie on the surface. All vertices on the mesh discussed in these other works lie on the surface. These meshes are usually generated by sampling real life data. The mesh is subdivided based on a subdivision rule and repeated subdivision of the mesh generates the surface. The control mesh of the surface is not explicitly used.

Work on subdivision algorithms for surfaces was first done by Doo/Sabin in [Doo78] and Catmull/Clark in [Cat78]. The primary purpose of the work was to overcome the limitations of the rectangular control topology. Their work generalized bi-quadratic and bi-cubic tensor product B-splines. Catmull/Clark surfaces are subdivision B-spline surfaces on arbitrary topological meshes. Both bi-cubic and bi-quadratic B-spline subdivision algorithms are presented for rectangular meshes. These algorithm are then generalized for meshes of arbitrary topology. The subdivision rule involves generating new vertices by locally averaging vertices of the initial control mesh. For irregular meshes, the generated surface reduces to a standard B-spline surface where the vertex is regular. That is, the algorithm yields the standard B-spline subdivision algorithm when the number of edges incident to a vertex is four. Irregular vertices, those with other than four edges incident on them, are called extraordinary points and need to be handled separately.

Other work that involved subdivision algorithms include [Loo94], G-splines in [Hol90] and [Pet93]. Once again the primary focus involved working with irregular meshes. The algorithms presented are generalization of bi-quadratic B-splines. If they are applied to a regular mesh, the generated surface will be the standard B-spline surface. An important aspect of these works is the assumption that irregular vertices are isolated, that is, they are surrounded by quadrilaterals and regular vertices. In [Loo94] and [Pet93], the refinement step assures this isolation of irregular

vertices. In [Loo94], the refined mesh is converted into triangular Bezier surface patches by interpolation for rendering. The isolation of the irregular vertices results in fewer patches with higher degrees (at most polynomial degree 4)

Finally, subdivision surfaces are also used in [DeR94]. The purpose of this work is to model surfaces of arbitrary topology from scattered range data. The subdivision algorithm works with triangular mesh data, where faces on the mesh are composed of triangles. The primary contribution of the work is defining subdivision rules that model sharp features such as creases and corners and also model boundary conditions. The subdivision is a generalization of C^2 quartic triangular B-splines.

3. Subdivision Surface Editing and Display

I can use the subdivision algorithms presented in the previous chapter to construct Refiner objects. I designed a Refiner as an abstraction (a base class) that takes a control mesh as input and generates a refined control mesh. Each Refiner subclass has a corresponding subdivision surface and algorithm associated with it. I incorporated a collection of such Refiners in an application, called *RefEdit*, that I created to assist in rendering the subdivision surface. It is a subdivision surface editor that manipulates surfaces in 3D. This editor has special requirements to aid the user in handling subdivision surfaces. I will discuss these requirements in this chapter.

The previous chapter presented three subdivision algorithms of control meshes. I have created three corresponding Refiners out of these algorithms. These Refiners include the tensor product cardinal spline Refiner, the box spline Refiner based on the $M_{(2,2,1,1)}$ box spline and the generic box spline Refiner which uses the direction vector based algorithm. These three Refiners are the first to be incorporated into the *RefEdit* application. Other Refiners can be added as required.

3.1 Control Mesh and Surface

In a surface editing environment, a surface is primarily created by specifying the control mesh of the surface. For the purpose of simplicity, the control mesh used in my editor is limited to lie on a rectangular grid. To aid in the creation of a surface, the user always begins with a n x m control mesh where the control points lie on a plane. From this point, the surface can be modified to the desired shape by manipulating the control points in 3-space. The user should also be able to create

a new surface where the number of rows (*n*) and columns (*m*) can be user specified. For simplicity, the editor only works with one surface at a time.

It is important to note that the user only interacts with the control mesh. The user does so by manipulating the control points of the control mesh. This is the only method of control the user has for modifying the surface and this is true for most surface editing environments. The user is not given the chance to modify the surface directly. The surface changes implicitly by the manipulation applied to the control mesh. Therefore, given any control mesh it is possible to render the underlying subdivision surface using one of many subdivision algorithms by applying the corresponding Refiner to the control mesh. *RefEdit* provides an environment where the Refiner can be chosen by the user. This concept is unique to *RefEdit* and it allows the user to visualize several different underlying subdivision surfaces by interactively choosing the Refiner. This concept formed an important design issue for the editor. The ability to switch between multiple Refiners interactively, given a control mesh, gives the editor its strength.

Rendering

The editor does not need to render the subdivision surface explicitly, but it does need some way of rendering quad meshes. The *initial* control mesh needs to be rendered as a set of distinctly visible control vertices. These vertices, joined together with line segments to show the topology, will represent the *initial* control mesh. The surface will be approximated by the *final* refined control mesh. It will be rendered as a shaded object using some standard lighting model. The control vertices and the interconnecting lines should not be visible.

The subdivision algorithms mentioned in the previous chapter work with a refinement magnitude of 2 at each step of the refinement, i.e., $w = 2$. This is done for simplicity and for reducing the complexity of the computation. The refinement process is then repeated several times to achieve the desired degree of refinement. The number of iterations of the refinement is called the level of refinement. This process is rapidly converging for most subdivision surfaces and requires relatively few iterations.

For accurate representation of a surface on a screen display, the control mesh can be refined until the error between the refined control mesh and the actual surface is at the sub-pixel level. Or the level of refinement can be a user specified option. This latter approach allows the user to determine the rendering quality of the surface and is the approach I have taken with the *RefEdit* editor.

3.2 Specifications for the Editor

What objects the editor will work with need to be defined. The editor will contain only two objects, one of which is the control mesh for the surface. The other object will be the surface, which will be rendered using the *final* refined control mesh. As mentioned earlier, there will be no other surface in the 3D environment for the purpose of simplicity. Only the control mesh will can be manipulated. The rendered surface will not allow direct manipulation with the input device. It will respond to changes in the control mesh.

The editor should provide viewpoint manipulation of the scene so that the surface is visible from any direction. This manipulation can be accomplished by rotations along the three major axes, and zooming in and out of the scene. Creating a new surface, saving and loading of the surface are also necessary parts of any editor. Aside from saving the surface information, it is also necessary to save the current working environment. One important setting in the environment is the current viewpoint. Saving of the current viewpoint is necessary so that the viewpoint last saved is load-able in the future. Because of the single surface limitation in the editor, another instance of the application can be run to overcame the issue. In the second instance, loading of the same data file from the first instance, with the viewpoint intact, allows the user to experiment with the control mesh and surface and compare results.

Control vertex manipulation in 3 dimensions is always a complex problem to address. The solution should be intuitive and easy to use. Multiple control vertices should also be able to be selected and dragged in 3 space. This avoids having to select each control vertex individually and dragging them separately. This feature is not a major requirement but a convenience. The surface should respond to the control vertex manipulation as soon as possible. This feature provides interactive feedback to the user.

3.3 Trimming : What to Display

The rendering of the subdivision surface has been accomplished by refining the original control mesh. The refined control mesh approximates the surface and does not represent the surface itself. For reasons specific to each subdivision algorithm, the *final* refined control mesh contains extra vertices that do not correspond to the subdivision surface. For the *final* refined control mesh to approximate the subdivision surface accurately, these control points need to be trimmed from the

control mesh. What needs to be trimmed from the *final* refined control mesh is specific to the subdivision algorithm and is discussed individually below. I discovered the need for trimming after an initial pass of the implementation was completed. The *final* refined control mesh extended outside the convex hull of the control points along the edges. In Figure 16, a simple refined control mesh is shown trimmed along the top and bottom rows and along the left and right columns by a certain amount. The trimmed control mesh is represented by the thick outlined control mesh.

The concept of trimming a surface introduces us to Trimmer objects. I designed these objects to take a control mesh and trim it around the edges by certain amounts. The trimming amounts are specific to each Trimmer and can also be dependent on the level of refinement. As a result of this discussion, two simple Trimmers can be introduced. First there is the Constant Trimmer which trims a control mesh around all edges by a constant amount for all level of refinements. Secondly, there is the Custom Trimmer which allows the user to specify the trimming amount along each edge for every level of refinement explicitly.

With the introduction of trimming, every Refiner will require a corresponding Trimmer to be associated with it as part of the Refiner's specification. A Trimmer can be applied to any control mesh, but for our purposes we will only apply the Trimmer to refined control mesh generated by the Refiner. For subdivision algorithms that do not require that the refined control mesh be

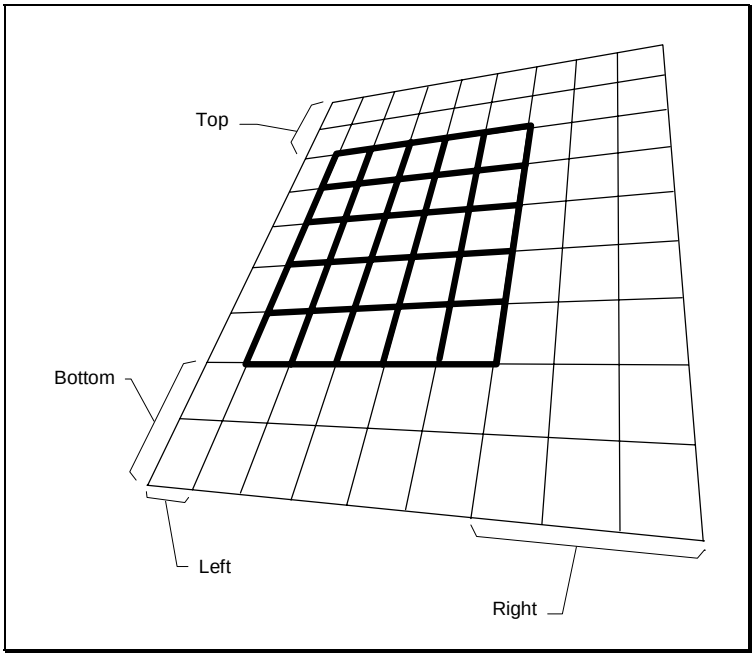


Figure 16. Trimming of a refined control mesh.

trimmed, their Refiners can associate a Constant Trimmer with a zero trim value to themselves. The Custom Trimmer is used by the *RefEdit* application to allow the user to have full control over trimming of a refined control mesh. Every surface with or without a need for trimming will be capable of trimming in the editor. This functionality allows the user to override the Trimmer associated with the current Refiner for the purpose of experimentation. The Custom Trimmer should also be one of the environment settings saved to the data file. This feature allows user specified trimming to be saved with the surface.

For an example of a surface where the refined control mesh has not been trimmed, refer to Figure 17. As can be seen from the surface, the refined control mesh (the shaded surface) wraps back upon itself outside the region where the surface is defined. The black patches on the surface are caused by trying to render overlapping regions of the refined control mesh. The figure represents a box spline surface based on the $M_{(2,2,1,1)}$ box spline. The surface is rendered by a refined control mesh that has been refined four times. However, no trimming has been performed on the surface. The figure should be compared with Figure 12 and Figure 13 from the previous chapter which represent the same surface and control mesh that has been suitably trimmed.

3.3.1 Tensor Product Cardinal Spline Surfaces

In order to determine the trimming required for a tensor product cardinal spline surface, it is important to examine the univariate cardinal spline case. From the previous chapter it is known that cardinal splines are B-splines over a uniform knot sequence. Some B-Spline properties from [Far93] are listed below. Detailed derivations of these properties can be found in the same reference.

For a non-parametric B-Spline, n represents the maximal degree of each polynomial segment, L is the number of curve segments if all knots in the domain are simple. Then the following properties hold.

- The knot sequence is given by $\{u_0, u_1, \cdots, u_{L+2n-2}\}$.
- The curve is only defined over $[u_{n-1}, \ldots, u_{L+n-1}]$.
- The Greville abscissas, which are averages of knots, are given by $\varepsilon_i = \frac{1}{n}(u_i + \ldots + u_{i+n-1})$.
- The control polygon is given by (ε_i, c_i) where $i = 0, \ldots, L + n - 1$.

These properties can now be simplified for the cardinal cubic case and modified to use the notation and theory used in Chapter 2. Given a set of control vertices $c_i \in \mathbf{R}$ for a cardinal spline $p(u) \in S_k$,

the number of curve segments defined by the curve can be specified by L. Note that $k = n + 1$ and $u_i = u_0 + i$, where u_0 is the first knot in the knot sequence. Therefore,

- the knot sequence is given by $\{u_0, u_1, \cdots, u_{L+2k-4}\} = \{u_0, u_0 + 1, \cdots, u_0 + L + 2k - 4\}$;
- the curve is only defined over $[u_{k-2}, \ldots, u_{L+k-2}]$;
- the Greville abscissas are given by by $\varepsilon_i = \frac{1}{k-1}(u_i + \ldots + u_{i+n-1}) = u_i + \frac{k-2}{2}$;
- the number of control vertices is $L + k - 1$;
- each control vertex is given by (ε_i, c_i) where $i = 0, \ldots, L + k - 2$.

The above properties are also true for the parametric case where $c_i \in \mathbf{R}^N$.

For the cubic case, $k = 4$, the number of control vertices in the control polygon is $L + 3$ given by $c_0, \ldots, c_{L+2}$. The number of knots is $L+5$ given by $u_0, \ldots u_{L+5}$. When the curve is refined using the cardinal spline subdivision algorithm from Section 2.1.1 using $w = 2$, then the $L + 3$ control vertices are replaced by $2L + 3$ control vertices. However, not all of the new refined control vertices represent the curve. The curve is only defined over the interval $[u_2, \ldots, u_{L+2}]$. Only those vertices with a Greville abscissa over that interval will represent the curve. The Greville abscissas for the control vertices c_i are $\varepsilon_i = u_i + 1$. Therefore only the control vertices $c_1, \ldots, c_{L+1}$ define the surface. The other two control vertices c_0 and c_{L+2} can be discarded for the purpose of rendering. This implies a trimming of one control point from both sides of the curve.

The previous result can be extended to the tensor product case easily. For the surface $T_{k,l}(u,v)$ with control vertices $V_{i,j}$, we can repeat the trimming result in each of the two dimensions u and v. Therefore after an *initial* control mesh has been refined, we can trim away one control vertex from each edge of the control mesh. This results indicates that we need to associate a Constant Trimmer with a trim value of one for each level of refinement to render the surface.

3.3.2 Box Spline Surfaces

In general, the box spline subdivision algorithms for surfaces discussed in the previous chapter work on a bi-infinite control mesh. Given a control mesh for a surface, to generate the bi-infinite one, additional control points are set to the value **0**. As a result, the process needs to be carried out near the original control points. On the refined control mesh, it is important to keep track of which generated control points are relevant for the underlying surface described by the original control mesh.

The specific trimming constraints for the box spline surfaces are unknown. For the Refiner based on the $M_{(2,2,1,1)}$ box spline, I determined empirical values for the refined control mesh at each level of refinement. From these empirical values I created a simple formula to generate the Trimmer for this Refiner.

The solution for the direction vector based box spline Refiner was not so simple. There are no known algorithms that specify the necessary trimming for a box spline surface. Empirical values can be used define trimming for specific surfaces, but in general a generic trimming algorithm was not available. As a result no trimming was performed for the control mesh. In this case, a Constant Trimmer with a trim value of zero was used to signify no trimming.

This introduces us to an important concept of subdivision surfaces that a Trimmer is not necessarily available for every Refiner. As a result, the *RefEdit* application should handle this situation. It does so by allowing the user to connect a Custom Trimmer with any Refiner, whether it has a Trimmer or not.

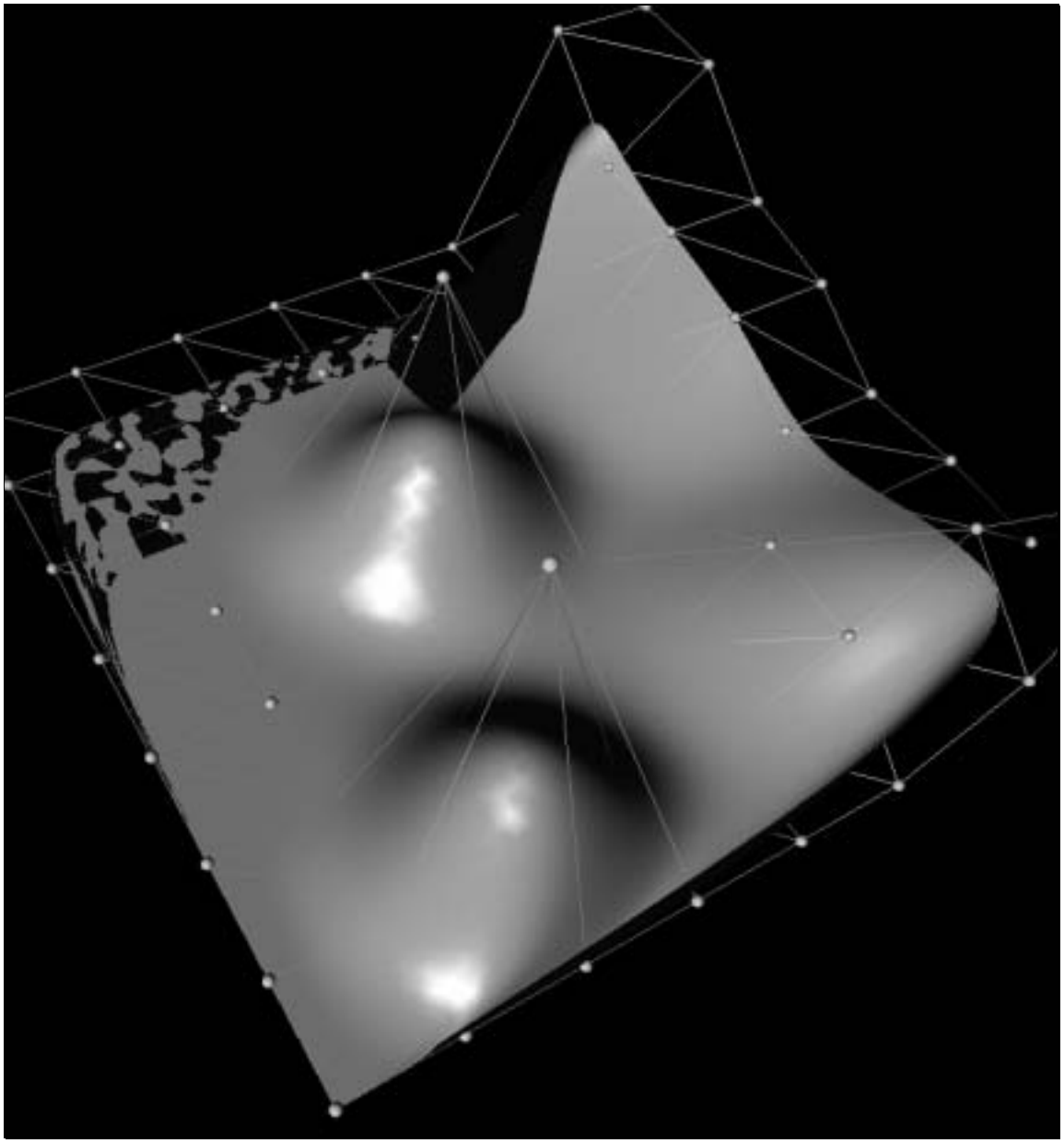


Figure 17. A surface that has not been trimmed. The visual aberrations are caused by overlapping pieces of the surface.

4. Implementation

The goal of this work is to manipulate and render subdivision surfaces. I accomplished the rendering by implementing a set of Refiner and Trimmer C++ classes that subdivide a control mesh to a refined mesh. My implementation involved two phases. The first was the creation of these Refiner classes. The second involved the implementation of a subdivision surface editor that utilized these classes. The purpose of the editor, *RefEdit*, was to present a prototype 3D surface design, manipulation and editing environment for subdivision surfaces. In such an environment, a control mesh can be refined by user selectable subdivision algorithms that are based on the Refiner spline classes.

4.1 Spline Classes

The Refiner classes became a part of the Spline classes library at the Computer Graphics Lab at the University of Waterloo. These spline classes are a collection of C++ classes used to implement several ideas in spline theory. The result of such implementations have been prototype applications useful for research.

The spline classes fall into three categories. Some may or may not have direct relations to spline theory and are nothing more than support classes. Others have a direct relationship with a spline theory concept and may or may not use the support classes. The final category are application specific classes that were necessary for the development of an application and are bundled with the rest to provide a common repository of classes. For this work some support classes already present

were used, while others needed to be modified before they could be used. I added the Trimmer class and the templated version of the Refiner class to the repository.

Some of the new classes introduced to the repository follow a simple naming convention. Class names are prefixed by '`S`' to indicate that the class belongs to the Spline classes set. If the class is templated, the '`S`' is followed by a '`T`'. If a class, initially templated, is re-written in a non-templated implementation then it is prefixed by '`NT.`' Not all classes follow this naming convention, because this naming convention did not exist when the Spline classes were first introduced and in some cases the naming convention was not adhered to.

At the start of the implementation, I wrote all new classes for this work in a template format. However, a compiler problem arose with the use of templated classes. The compiler I used at that time did not support templates very well. As a result, I rewrote several templated classes in non-templated form for this work. Newer compilers do not exhibit the same problems and there is no reason not to switch back to templated code for the non-templated classes.

4.1.1 Support Classes

Support classes are needed to avoid duplicating the work in each application. The classes discussed here were already part of the Splines class libraries written by various authors. In some cases, certain modifications were required before I could use these classes for this work. I made these modifications such that previous work dependent on these support classes would still function without any change. The man pages for these classes are given in Appendix A.

Rogue Wave

The RogueWave library is a commercial library of commonly used class for manipulation of data. The classes used by this work are the `IntVec`, `DoubleVec`, and the `IntGenMat` classes. `IntVec` is a class representing a single dimension array of integers. `DoubleVec` represents a single dimension array of doubles. Lastly, `IntGenMat` is a multidimensional matrix of integers. All these class have the added benefit of being able to dynamically change the size of the array or matrix after construction. Detailed information on each of the classes in this library is available in [Rog91].

Tuple Classes

A *tuple* is a set of data elements. An *n-tuple* contains *n* data elements. The classes under the `Tuple` hierarchy include a set of classes that work with *tuples* of data. In 3D Computer Graphics, work is primarily done with floating point data. The `Triple` class, based on `Tuple`, is used to represent a *3-tuple* containing floating point data elements. Using this class, one can represent a coordinate in 3D space. Standard arithmetic and indexing operators are defined on this class. Similarly working in 2 dimensions requires use of `Duple`, another class based on `Tuple`, that is used to represent a *2-tuple*. By design, a `Tuple` is a fixed length `DoubleVec`.

The primary class of importance in the `Tuple` hierarchy is the `TupleVec` class. This class holds a series of vectors in *tuple* format. A specific vector can be referenced by indexing into the *tuple* array. Each vector is represented by a `DoubleVec` in the `TupleVec` class. This class was one of the pre-existing support classes in the splines library. The only modification that I made to the `TupleVec` class was the addition of an assignment operator. This operator allows the user to assign a `double` to the entire `TupleVec`. This assignment is a convenient way of initializing the data to a constant value.

Lattice Classes

The data stored in a `TupleVec` can only be accessed linearly. However, it is more convenient to access the data using a multidimensional indexing scheme. The `Lattice` class allows for the access of multidimensional data through a common indexing scheme. It does this by creating a map from the user defined multidimensional space to a linear space. The index returned by the `Lattice` class can then be used to index the linear space to access the data element. The `Lattice` family of classes was implemented by Hickey in [Hic94]. See Appendix A for the man page.

The core functionality of the `Lattice` class lies in the virtual `multiToSingle()` operator. Each subclass of `Lattice` is required to implement this operator. The only subclass I used in the `Lattice` hierarchy is the `TensorIndex` class. It provides a multidimensional indexing scheme for rectangular lattices. This implementation provides an ideal indexing scheme for control meshes of surfaces. A specific control point can be indexed directly. The following code sample shows how to construct a `TensorIndex` lattice which indexes into a 2x3x5 array.

```
IntVec sizes(3);
sizes[0] = 2; sizes[1] = 3; sizes[2] = 5;
TensorIndex ti(sizes);  // Indexes into a 2x3x5 array
```

The `TensorIndex` class takes the size of each dimension as a parameter to construct the indexing scheme. One of the requirements of the Box Spline surface is that it can also index the control mesh using negative indices. Therefore, it was necessary to modify the `Lattice` and `TensorIndex` class to support negative indices. I accomplished this modification by adding a second parameter for the constructor which specifies the origin. The default value for this parameter is to use the zero origin. The following code sample shows how to construct a `TensorIndex` lattice with an overall size of 4x6x5 and an origin at (-1,-2,0).

```
IntVec sizes(3);
IntVec origin(3);
sizes[0] = 2;   sizes[1] = 3;   sizes[2] = 5;
origin[0] = -1; origin[1] = -2; origin[2] = 0;
TensorIndex ti(sizes, origin);
int index = ti(2,-1,3);
```

The value in `index` will be a mapped from the 3-dimensional index of (2,-1,3) to the linear space. Another modification to `TensorIndex` related to negative indices that I had to make was the addition of member functions that allowed for the retrieval and modification of the origin.

NTSIdxTuple Class

Data can exist in the `TupleVec` class and can be indexed using the `TensorIndex` class. However, another class, `SIdxTuple`, is required to tie the two together. The `SIdxTuple` class was implemented by B. Hickey in [Hic94]. `SIdxTuple` is a templated class that holds data that can be indexed by a variety of existing lattices. Additionally the data can be of several types. The user must specify the data type and the element type at instantiation.

Some possible template instantiations are:

```
SIdxTuple< "Lattice", Triple,  TupleVec >
SIdxTuple< "Lattice", SPoint,  SPointVec >
SIdxTuple< "Lattice", SVector, SVectorVec>
```

where "`Lattice`" can be any class derived from `Lattice.`

For my purposes, a non-templated version of the class, `NTSIdxTuple`, is used. Initially all work was done with the templated version of the class. However, due to compiler issues, the non-templated version was implemented. This class holds `TupleVec` (each vector a `Triple`) data that can be indexed by `TensorIndex` lattice. `NTSIdxTuple` is just a non-templated version of `SIdxTuple` using the following instantiation: `SIdxTuple<TensorIndex, Triple, TupleVec>`.

The Box Spline requirement for negative indices prompted a similar modification to the `NTSIdxTuple` class as was done for the `TensorIndex` and `Lattice` classes. Another modification was adding member functions to access the underlying lattice's origin and size directly. This modification was required by the refiner class.

4.1.2 Trimmer Class

The need for the Trimmer class was discussed in section 3.3. This class hierarchy was one that I added to the Spline classes repository. Every Refiner (discussed in the next section) will have a Trimmer associated with it. The Trimmer only works on a rectangular grid. This restriction is based on the restriction imposed on the topology of the surfaces in Section 3.1. The rectangular grid represents the control mesh for the surface. The Trimmer class does not have a restriction on the dimension of the rectangular grid on which it can work. Practically, two or three dimensions are used. The class hierarchy of the Trimmer classes is shown in Figure 18. Each class in this hierarchy is discussed below.

Abstract Trimmer Class

`STrimmer` is the abstract base surface trimmer class. For every refinement level it defines the trimming of a surface along each control mesh dimension. This trimming is represent by a *Start* vector and an *End* vector, where the size of the vector is equal to the control mesh dimension. Each element in these vectors represent the number of control vertices that need to be trimmed from the corresponding side.

For example, on a two dimensional domain space, the control mesh can be trimmed from the left,

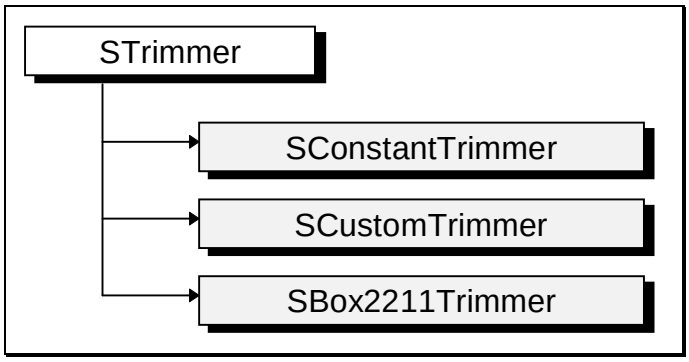


Figure 18. STrimmer Class Hierarchy.

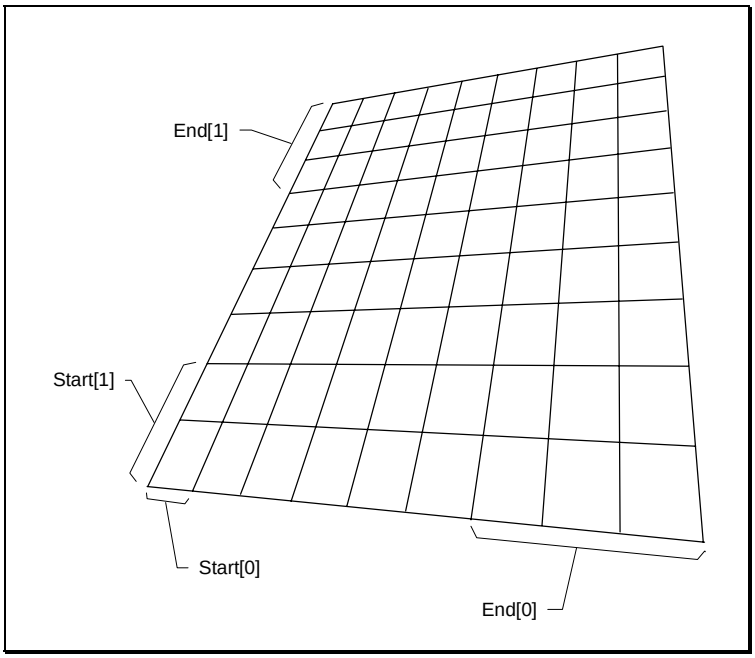


Figure 19. Trimming a control mesh on a 2D domain.

top, right, and bottom. This trimming would be represented by the vectors *Start* and *End* (both of dimension 2) where *Start*[0] is left, *Start*[1] is bottom and *End*[0] is right, *End*[1] is top. A pictorial view of this example is given in Figure 19.

One important design issue of the `STrimmer` class was the fact that it cannot be instantiated by a user of the class except using a default constructor. The regular constructor, which takes the maximum refinement level and the control mesh dimension as arguments, is a protected member function. As such, only subclasses can initialize the data. Also, the *Start* and *End* vectors for each refinement level can only be retrieved by the user at this abstract level. Only the subclass can store data into these vectors using protected member functions. Using this design, users will only be able to use the `STrimmer` class as an abstract class. They will be forced to create a Trimmer using one of the subclasses. Once such a Trimmer is created, it can be worked on at an abstract level using the `STrimmer` class.

Subclasses of `STrimmer` can specify specialized initialization for the *Start* and *End* vectors. Because these vectors are stored for each refinement level, two `IntGenMat` instances are used to store this information, one for the *Start* and one for the *End.* The dimension of these matrices is based on the maximum refinement level and the dimension of the control mesh. The header file for `STrimmer` is available in Appendix A for more information on the class.

Constant Trimmer Class

The `SConstantTrimmer` class defines a Trimmer with a constant trim value for each refinement level for each dimension. At instantiation time, it takes the maximum refinement level, the control mesh dimension and the constant trim value as arguments. With this information it initializes the matrices of the base class to the constant trim value. This value can also be set to zero to indicate no trimming needs to be performed.

Specific Box Spline Trimmer Class

This Trimmer, `SBox2211Trimmer`, is specific to the standard $M_{(2,2,1,1)}$ box spline. It is based on the observations of Section 3.3.2. It only requires the maximum refinement level as an argument at construction time. Once constructed, the *Start* and *End* vectors are set permanently in the base classes `IntGenMat`s. The maximum refinement level can be changed by calling the `setMaxRefineLevel` member function.

Custom Trimmer Class

The Custom Trimmer class, `SCustomTrimmer`, provides a way to store specific trim values for the *Start* and *End* vectors at each refinement level. This class only needs the maximum refinement level and the control mesh dimension at construction time. The *Start* and *End* vectors can be stored or retrieved from the class after construction using several member functions.

4.1.3 Refiner Classes

The core of this work is represented by the Refiner classes. These classes use all the support classes mentioned earlier to accomplish their goal. This goal is to refine of a set of control data using various refinement and subdivision algorithms.

A `Refiner` class hierarchy already existed in the spline class library before this project started. It primarily consisted of refinement algorithms that refined spline curves by performing knot insertion on the control vertices. The `Refiner` class was the base class for several such knot insertion classes. For this work, refinement was being performed on control meshes that primarily represented surfaces. I created a new templated class called `STRefiner` that encompassed the `Refiner` class hierarchy and also supported new refiner classes for this work. The class hierarchy for the `STRefiner` class is shown in Figure 20.

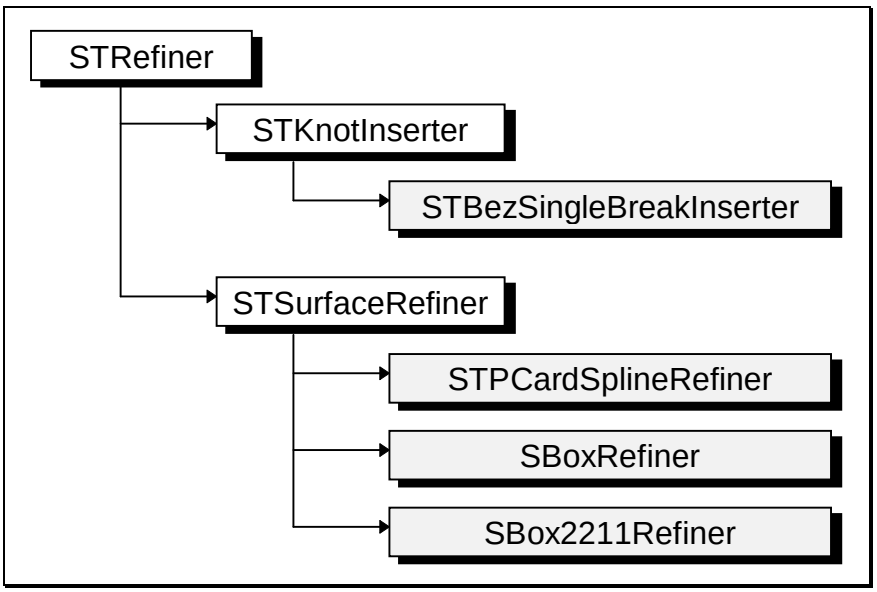


Figure 20. STRefiner Class Hierarchy.

STRefiner Class

The major difference between the `Refiner` and the `STRefiner` base classes is that the latter is now a templated class. The class hierarchy of `Refiner` was not used and all new work for this was added to a new `STRefiner` class. Most of the interface and design from `Refiner` was kept when `STRefiner` was created. A few new member functions were also added. `STRefiner` is more abstract than `Refiner`, because it is not restricted to knot insertion algorithms and can encompass any refinement algorithm based on control mesh refinement.

`STRefiner` is an abstract base class and as such does not have a constructor. This class is templated on the type of control vertices. The reason for this is that vertices can be of varied dimensions and of different storage types. When working with curves we can work with `IntVecs` or `DoubleVecs`. When working with surfaces, we can use `TupleVec` or `SIdxTuple` to represent the control vertices type.

The most important aspect of this class is the `newCVs` member function. This function allows the user to pass a set of control vertices by reference. A pointer to the refined set of control vertices is returned by this function. Two different overloaded member functions are provided for `newCVs` to allow for varied usage as shown in the following code sample.

```
newCVs( oldControlVertices, &newControlVertices);
```

```
   // pass a pointer to the newControlVertices which
   // has already been created.

pNewControlVertices = newCVs( oldControlVertices );
   // newCVs returns a pointer to new Control Vertices
```

As far as the implementation is concerned, these two member functions do nothing more than call the real work-horse function - `realNewCVs`. This function is a pure virtual function by design to ensure that each subclass implements its own version.

In certain cases, the dimension and size of the refined set of control vertices cannot be determined by the calling application because it is specific to the refiner being used. For this reason, storage management functions were required. I created two member functions to allow the calling function to allocate and free storage for the refined set of control vertices. These member functions are `allocateSpaceForNewCVs` and `freeSpaceForNewCVs` respectively. Once again these functions are also pure virtual because the storage requirements will depend on the refinement technique.

Knot Insertion Hierarchy

I created the `STKnotInserter` class to encompass all the subclasses of the old `Refiner` class hierarchy. `STKnotInserter` is the template version of the `Refiner` class with additional member functions that were required to be included as part of the `STRefiner` hierarchy. As its name applies, it works primarily by inserting knots into a knot sequence for Bézier and non uniform B-spline curves.

Only one class, `STBezSingleBreakInserter,` was ported to the `STKnotInserter` sub-hierarchy as an exercise. The other classes were not ported to the templated hierarchy because none of them were directly related to this work.

The extra member functions added to `STKnotInserter` were `allocateSpaceFor-NewCVs` and `freeSpaceForNewCVs`. These two functions were implemented in full in this abstract class. Therefore, none of the sub-classes need to be concerned with their implementation.

Surface Refiner

Refining surfaces through control mesh refinement can be performed abstractly by using the `STSurfaceRefiner` class. This abstract class is also templated on the type of the control vertices. As such it has no knowledge of the details and implementation of control vertices. As a

result the two storage management functions are still pure virtual in this class. The `realNewCVs` function is still a pure virtual also, because the details depend on the implementation of the subclass. Once several subclasses have been implemented, then the `STSurfaceRefiner` class can be used to work as a Refiner at an abstract level, not knowing which Refiner it is or the details of the implementation.

One important aspect of a surface Refiner in this hierarchy is that it has to be tied to a Trimmer class. See Section 4.1.2 for more information on the Trimmer class. In order to force each subclass to connect a Trimmer class to the Refiner, I defined a pure virtual private member function, `assignTrimmer`, for `STSurfaceRefiner`. Because this function is pure virtual, it forces each subclass to implement this function. The abstract base class, `STSurfaceRefiner`, calls this private member function in its constructor. The virtual nature of this function makes sure the correct subclasses' member function is called instead. Therefore we can force a runtime assignment of the Trimmer. The assigned Trimmer can be retrieved by calling the public `getTrimmer` member function.

The following sections discuss the details of the three refiners that I implemented for this work. Each of the three Refiners are subclassed off the `STSurfaceRefiner` abstract class. None of these subclasses are templated classes. The reason for the non-template nature of these class was mentioned earlier in this chapter regarding compiler problems.

The constructor for each of the following Refiners require the refinement level for the Refiner. This level represents the number of refinement iterations to perform on the control mesh. It can also be retrieved or modified through member functions. Another use of the refinement level is in the calculation of the total storage required for the refined control mesh.

Tensor Product Cardinal Spline Refiner

The tensor product cardinal spline Refiner uses the subdivision algorithm presented in Section 2.2.1. The name of the class for this Refiner is `STPCardSplineRefiner`. Apart from providing definitions for the pure virtual functions declared in the `STSurfaceRefiner` class, this class performs no special operation.

By definition, the `STPCardSplineRefiner` class works with a lattice data on a two dimensional domain. Each element in the data is represented by a `Triple` because the class is not templated. It uses the `NTSIdxTuple` class to represent the control mesh type.

The Trimmer that this Refiner assigns to itself is the Constant Trimmer with a trim value of 1 for each refinement level. This Trimmer is consistent with the one discussed in Section 3.3.1 for tensor product cardinal spline surfaces.

Below is some commented sample code that gives an example on how to use the Tensor product cardinal spline refiner.

```
STPCardSplineRefiner TPCardSplineRef(5);
   // Set the refinement level to 5 iterations

IntVec sizes(2);
sizes[0] = 2; sizes[1] = 2;
TensorIndex Lat(sizes);
   // Define a 2x2 lattice with an origin at (0,0)

NTSIdxTuple oldCVs(Lat,3);
   // Use the lattice to construct the indexed control
   // vertices where each element is of dimension 3.

// Initialize the old control vertices here with data

pNewCVs = TPCardSplineRef.allocateSpaceForNewCVs( oldCVs );
   // Allocate some space for the new control vertices

TPCardSplineRef.newCVs( oldCVs, pNewCVs );
   // Call the refiner to generate the new control vertices

// Use the new control vertices here

TPCardSplineRef.freeSpaceForNewCVs( pNewCVs );
   // Free up the space for the new control vertices
```

Direction Vector Based Box Spline Refiner

This Refiner is based on box spline surfaces. The control mesh refinement algorithm used here is the generic direction vector based subdivision algorithm presented in Section 2.3.3. The name of the class for this Refiner is `SBoxRefiner`. By being the subclass of STSurfaceRefiner, this class has to implement the virtual `realNewCVs` and storage manipulation member functions.

Two new pieces of information are required for instantiating this class - the direction vectors and the magnitude of refinement. The direction vectors are represented as a `N` x `M` `TupleVec`. `N` represents the size of each vector and `M` represents the number of vectors. With regards to the implementation of the `SBoxRefiner` class, there is no dimensional restriction on the direction vectors. The class can handle domains of 1, 2, 3 or more dimensions. Practically however, for this work the direction vectors lie on a 2 dimensional domain.

The following code sample gives an example of how to use this class. Note that once the Refiner has been instantiated, it is used no differently from the Refiner defined in the code sample above. In this particular example the box spline $M_{(1,1,1)}$ is being used as a basis.

```
TupleVec DirVecs(2,3);
DirVecs(0) = Duple(1,0);
DirVecs(1) = Duple(0,1);
DirVecs(2) = Duple(1,1);
   // Setup Direction vectors for Box Spline

SBoxRefiner box(DirVecs, 2, 5);
   // Use the direction vectors to define a box spline
   // refiner with a refinement level of 5, where each
   // refinement step involves a magnitude of 2.

IntVec sizes(2);
IntVec origin(2);
sizes[0] = 1; sizes[1] = 1;
origin[0] = -1; origin[1] = -1;
TensorIndex Lat(sizes, 0, &origin);
NTSIdxTuple oldCVs(Lat,3);

// Initialize the old control vertices here with data

pNewCVs = box.allocateSpaceForNewCVs( oldCVs );
   // Allocate some space for the new control vertices

box.newCVs( oldCVs, pNewCVs );
   // Call the refiner to generate the new control vertices

// Use the new control vertices here

box.freeSpaceForNewCVs( pNewCVs );
   // Free up the space for the new control vertices
```

The Trimmer that this Refiner assigns to itself is a Constant Trimmer with a trim value of 0 along each border. The reason for this assignment is that the trimming information for this Refiner was not available as discussed in Section 3.3.2.

$M_{(2,2,1,1)}$ Box Spline Refiner

This Refiner, `SBoxRefiner2211`, is based on the specific case of the subdivision algorithm presented in Section 2.3.3. Hence, it assumes that surfaces are based on the quartic $M_{(2,2,1,1)}$ box spline. It uses the same box spline when it associates `SBox2211Trimmer` with itself.

The Refiner is instantiated using the sample code below. The refinement level is the only argument that is required at the time of instantiation. Once instantiated, using this Refiner is no different than the Refiners defined previously.

```
SBoxREfiner2211 Box2211Ref(5);
   // Set the refinement level to 5 iterations
```

4.2 Application design

To test the classes defined in the previous section, I wrote *RefEdit*, a subdivision surface editor. This editor uses several technologies, including the splines classes, to provide a prototype 3D subdivision surface manipulation environment.

There were two choices for rendering the subdivision surfaces. One was to use the preexisting rendering support classes in the CGL spline classes hierarchy. The other option was to use Open Inventor (see [Wer94]) for rendering. The latter option was chosen because Open Inventor provided a very fast 3D scene prototyping environment on top of Open GL. It provided rendering of quadrilateral meshes using standard lighting models. It also provided a mechanism for implementing several standard 3D manipulation User Interface (UI) issues with ease. What UI issues were missing were accomplished using the OSF/Motif X Toolkit (see [Mot94]).

4.2.1 RefEdit

The application, *RefEdit*, used Open Inventor for implementing the rendering and 3D manipulation environment. The X Toolkit (Xt) Intrinsics provided by Motif were used for implementing menus and dialogs. Open Inventor provides several classes for making itself work well with Xt. One such class, `SoXt`, initializes Inventor for use with the Xt Toolkit and Motif. This class also provides event handling for the application. This cooperation between the two toolkits allows for easy customizability of an application. These toolkits also allow the application to fulfill the specifications for the editor mentioned in the previous chapter.

4.2.2 Open Inventor

Open Inventor is an object-oriented 3D toolkit providing a library of objects and methods to help create interactive 3D graphics applications. It presents a programming model based on a hierarchical scene database to build 3D scenes. It provides a rich set of database primitives including shape, property, group and engine objects. These primitives help design an object in a 3D scene. Interactive manipulators, such as a trackball and a handle box, are also provided by Open

Inventor for manipulating the objects. Also provided are components, such as a material editor, a directional light editor, and various viewers to allow the user to manipulate their view of the scene.

Scene Graph

A scene graph is an ordered collection of nodes to represent objects in a 3D scene. One or more scene graphs can be stored in the scene database. The scene graph that I created for use in *RefEdit* to represent the subdivision surface scene in the Open Inventor scene data base is shown in Figure 21 and Figure 22. The scene graph is explained below using labeled references to nodes in each of the two figures. The node labeling convention used here is of the form '(a1)', where the letter represents the level in the scene graph and the number specifies the node on a specific level from left to right. The figures use the same iconic notations for nodes as presented in [Wer94]. In Open Inventor nodes are implemented as C++ classes that begin with the characters "So". Each of the nodes are briefly defined as they are encountered in the scene graph description below. Full details on nodes in Open Inventor can be found in [Wer94].

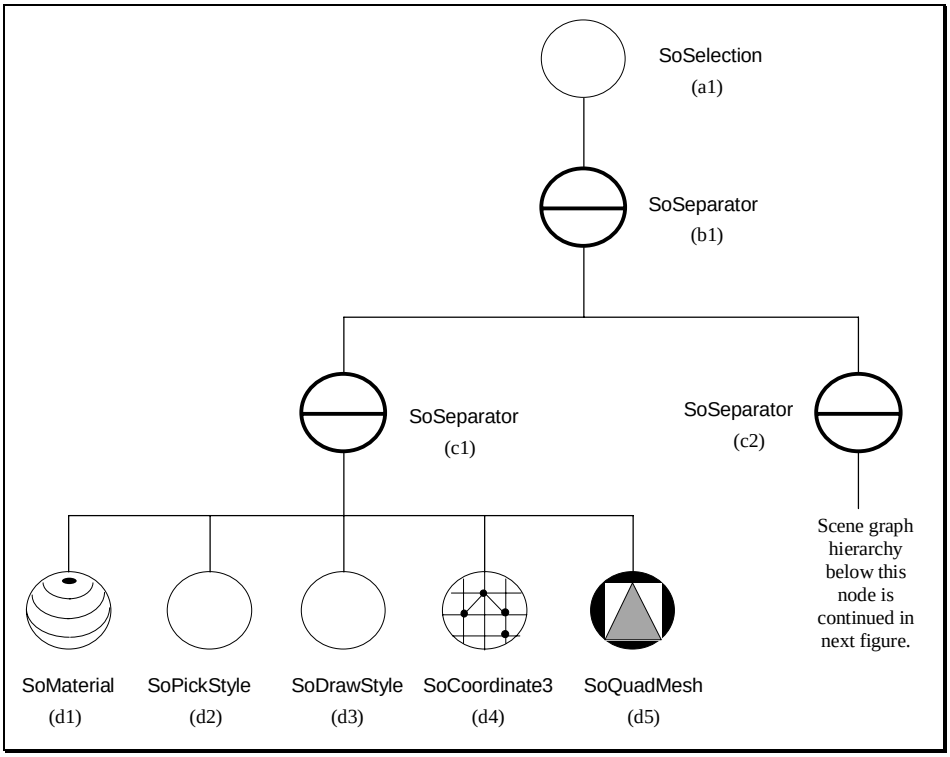


Figure 21. Scene graph for hierarchy for Control Mesh.

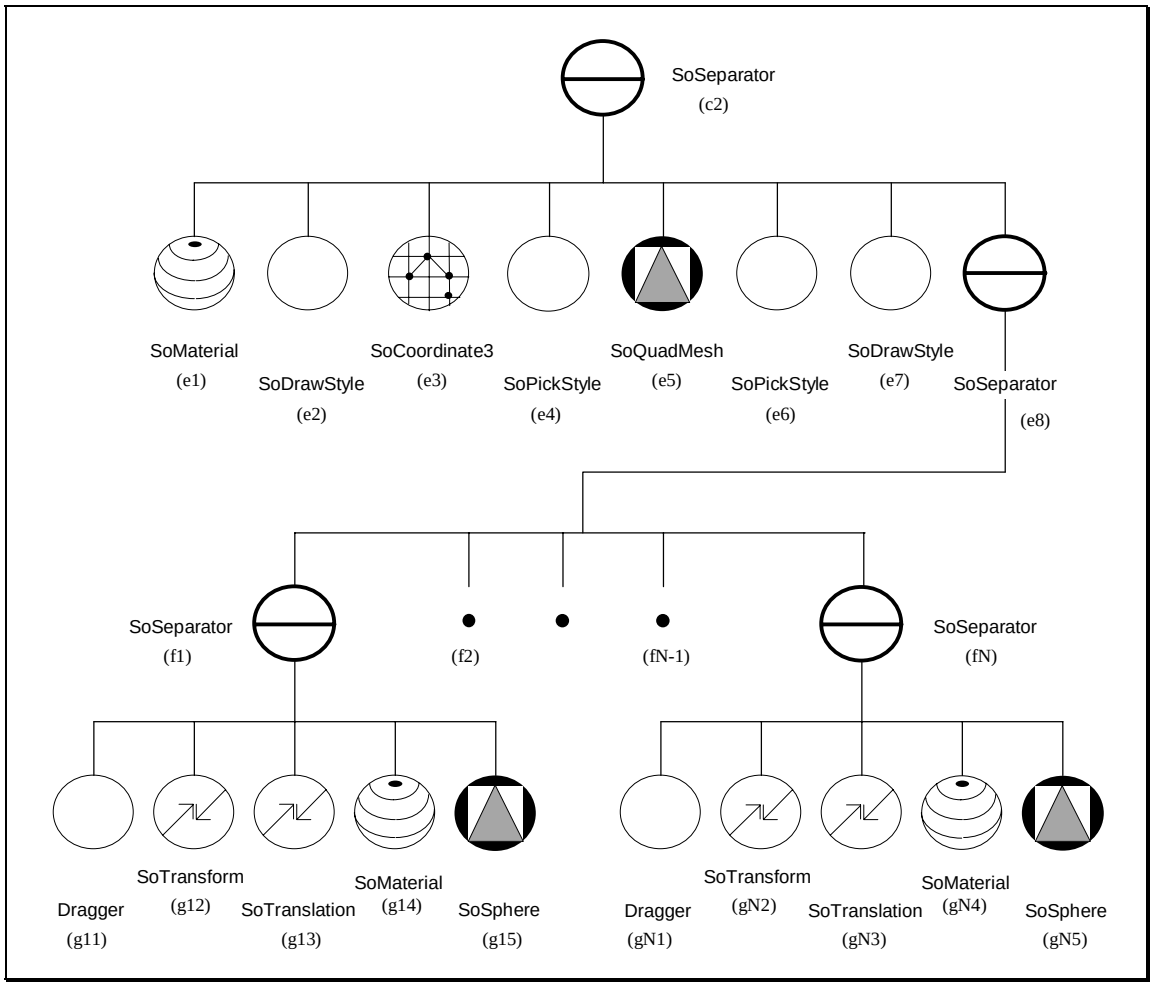


Figure 22. Scene hierarchy for rendered surface.

The two figures represent one scene graph that was split to fit the page. Figure 22 shows the node sub-hierarchy for Node (c2) from Figure 21. The traversal of the screen graph is based on standard hierarchical 3D scene traversals in computer graphics. In Open Inventor, it is traversed left to right. Some nodes (parent) can have child nodes, in which case each child is traversed before going on to the parent's sibling. Tree traversal is performed during rendering, picking and other actions. During rendering some nodes can change the current graphic state. When objects are encountered, they are rendered using the current graphic state.

The scene graph starts at a SoSelection node (a1). This node is required to manage and perform selections of objects in Open Inventor. The SoSeparator node is an example of a parent node used

to collect several nodes together as its child nodes. It also saves the graphic state during tree traversal so that graphic state set at the time the SoSeparator node is encountered is restored when all the child nodes have been traversed.

The SoSeparator node (c1) is the root of the scene graph describing the subdivision surface. Nodes (d1) to (d4) change the graphic state for the quadrilateral mesh shape node - SoQuadMesh (d5). It is this node that represents the subdivision surface in the scene graph. Quadrilateral Meshes in Open Inventor allow the user to render a surface based on quadrilaterals arranged on a rectangular grid. Nodes (d1) to (d4) are described in the table below.

Node	Type	Description
d1	SoMaterial	Changes how a shapes surface will respond to light. The lighting model is a standard graphics model which includes diffuse, ambient, specular and emissive colors. This SoMaterial node for the subdivision surface is modifiable using the material editor.
d2	SoPickStyle	A style specifying whether a shape node is pickable or not. The subdivision surface is not pickable by design.
d3	SoDrawStyle	A style specifying how to render the shape node. Options include invisible, wireframe and full shaded. This style is user selectable in the *RefEdit* through the application menus.
d4	SoCoordinate3	Defines coordinate vertices or control points for the subdivision surface's quadrilateral mesh. The coordinates are set to the refined control mesh from the output of a Refiner.

The SoSeparator node (c2) represents the root of the scene describing the control mesh. This hierarchy is a little more complex than the subdivision surface hierarchy. The control mesh is also implemented as a wireframe SoQuadMesh (e5). The table below describes nodes (e1) to (e4) that modify the graphic state for the control mesh.

Node	Type	Description
e1	SoMaterial	The material for a control mesh is light gray in color.
e2	SoDrawStyle	The control mesh is rendered in wireframe all the time so that it appears as a mesh.
e3	SoCoordinate3	Defines coordinate vertices or control points for the control mesh's quadrilateral mesh.
e4	SoPickStyle	The control mesh itself is not pickable by design.

In Open Inventor an individual control vertex of a SoQuadMesh cannot be picked (and therefore cannot be selected and manipulated). Instead the entire quadrilateral mesh is picked. As a result tiny spheres were added to the scene graph to take place of the control vertices, which can be picked, selected and manipulated. As a result, the SoQuadMesh representing the control mesh itself is not pickable (as indicated by the SoPickStyle in the previous table). At each vertex of the control mesh, SoSphere nodes ((g14), (g24), …,(gN4)) are attached. There is one sphere for every control vertex in the mesh for a total of N spheres. Nodes (e6) and (e7) indicate the SoPickStyle (pickable) and SoDrawStyle (full shaded) for the control vertices (the spheres) respectively. There is one SoSeparator for every control vertex represented by nodes (f1) to (fN) with sub-trees underneath them. Underneath each SoSeparator, each control vertex is defined identically using the following table.

Node (I=1 to N)	Type	Description
gi1	Dragger	A SoDragPointDragger is a dragger node that allows translation of an object in 3D space.
gi2	SoTransform	A scale transformation is defined to make the sphere tiny.
gi3	SoTranslation	A translation is needed position the sphere at the right 3D coordinate of the control vertex.
gi4	SoMaterial	A separate material is used for every sphere because the individual control vertex can be selected in which case the color of the sphere changes. When not selected the color is white and when selected it is red.

Control Vertex Manipulation

When a control vertex is selected (i.e., the SoSphere is selected) the application performs two operations. It turns the SoMaterial for the corresponding control vertex to the color red to indicate the control vertex has been selected. It also adds a SoDragPointDragger to the beginning of control vertex's sub-tree. This dragger provides feedback to the application every time the shape associated with it is moved. In Open Inventor a dragger has its own predefined shape. For my purposes, the shape of the dragger was replaced by the corresponding sphere so that the dragger itself is not visible and the sphere appears to move. If multiple spheres are selected, each one is moved by an appropriate delta amount. The coordinates of the control mesh (e3) are also modified to reflect the manipulation of the control vertices. When the mouse button is released, the subdivision surface is regenerated and rendered to also reflect the change in control vertices. When

a control vertex is deselected, it's material is reset to white and the dragger is removed from the sub-tree.

Components

I used Three Xt components from Open Inventor in the RefEdit application. One of these components was the examiner viewer which is used using the `SoXtExaminerViewer` class. This viewer, seen in Figure 23, uses a virtual trackball to view the objects in the scene. A camera node is implicitly added to the scene graph when the examiner viewer is created. When the user interacts with the viewer, it does nothing more than modify the camera's values such as position, size, and focus. The white area in the center of the viewer is the render area where objects are rendered, selected and manipulated. The buttons along the right are labeled in the figure. The first two buttons, Pick and View, are of primary importance. The former allows the user to pick and select objects inside the render area. In this case the viewpoint is fixed and mouse interactions

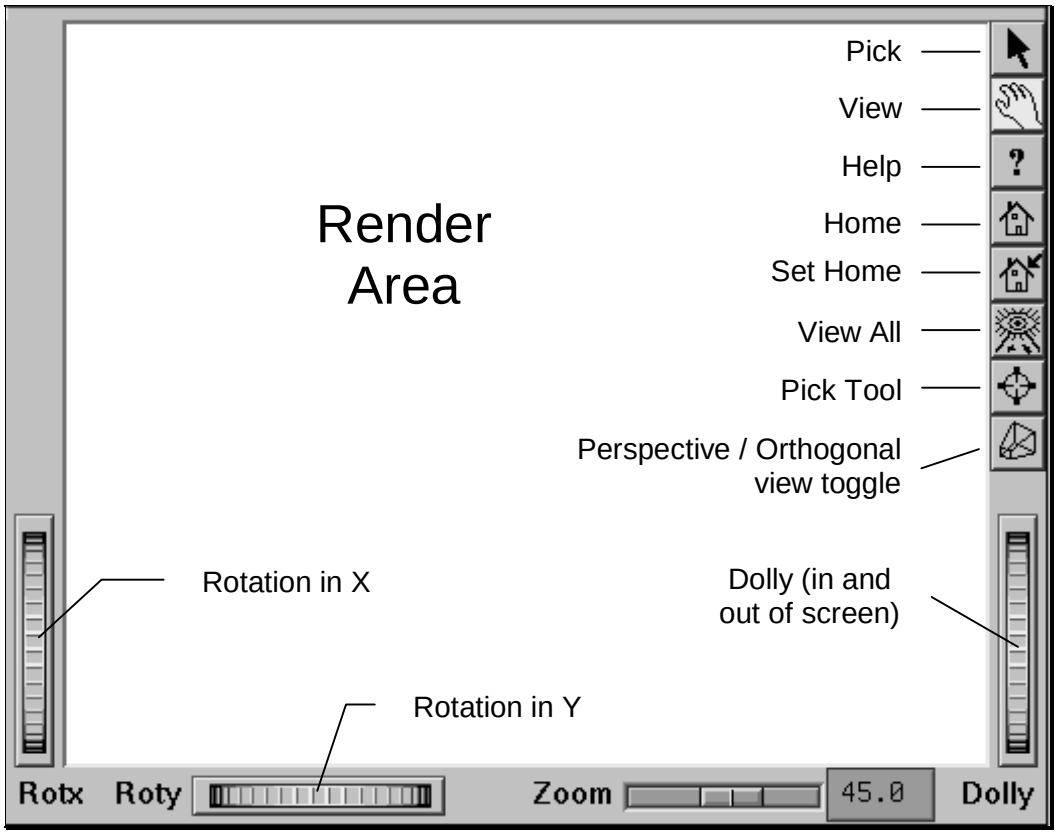


Figure 23. Examiner Viewer from Open Inventor.

indicate picking and selection of objects. The latter is use to enter view mode where interaction with the render area allows the user to change the viewpoint in the scene. The Help button brings up online help describing in detail all buttons and how the examiner viewer works.

Another component used in *RefEdit* was the material editor. This editor is a predefined component that can be plugged into an application to edit the material of an object. In the RefEdit application, it was used to modify the material of the subdivision surface. This is accomplished by attaching the material editor to the SoMaterial node (d1) of the subdivision surface. The third component used was the directional light editor. This editor allows the user to control the direction of the light falling on the scene. In *RefEdit*, the directional light editor modifies the light source of the examiner viewer. Using these components the user can modify and customize the view and objects in the scene.

4.2.3 Motif

Open Inventor is not a complete User Interface building kit. It provides the user with a base set of 3D viewers, objects, etc., but not all aspects of UI. Therefore, there was a need to define some of the missing UI functionality using another approach. I chose the Motif Xt Intrinsics Application Programmer Interface (API) because the remaining UI requirements for this work were not demanding. The requirements included a menu bar with corresponding menus at the top of the application workspace and the implementation of two dialog boxes.

Menus

A standard menu bar was implemented at the top of the application. There are 4 menus arranged from left to right in the following order: File, Surface, Refiner, and Options. Each entry in the menus are individually described in the tables below. Each menu entry is one of various types provided by Xt. Separators between menu entries for grouping several menu entries together are indicated by thick lines between the entries in the tables. More information about menus and menu types can be found in [Mot94].

Menu : File

Menu Entry	Type	Description
New	PushButton	Brings forth the “New” dialog box to create a new scene.
Open	PushButton	Loads a pre-existing scene.
Save	PushButton	Saves the currently opened scene.
Save As	PushButton	Saves the currently opened scene to a new filename. The new filename becomes the current scene’s name.
Quit	PushButton	Exits the Application.

Menu : Surface

Menu Entry	Type	Description
Invisible	ToggleButton : One Of Many	These three menu entries modify the subdivision surfaces SoDrawStyle node (d3) Surface is invisible.
Wireframe	ToggleButton : One Of Many	Surface is rendered in wireframe.
Shaded	ToggleButton : One Of Many	Surface is rendered as a shaded entity.
Edit Material	PushButton	Brings forth the Material Editor for the subdivision surface.

Menu : Refiner

Menu Entry	Type	Description
Level = 1	ToggleButton : One Of Many	These five menu entries change the level of refinement for the refiner. Set refinement level to 1.
Level = 2	ToggleButton : One Of Many	Set refinement level to 2
Level = 3	ToggleButton : One Of Many	Set refinement level to 3
Level = 4	ToggleButton : One Of Many	Set refinement level to 4
Level = 5	ToggleButton : One Of Many	Set refinement level to 5
Tensor Product Cardinal Spline Refiner	ToggleButton : One Of Many	These 3 menu entries select the Refiner. Render the surface using the cubic tensor product cardinal spline refiner.

Quartic Box Spline Refiner : 2 2 1 1	ToggleButton : One Of Many	Render the surface using the quartic box spline refiner.
Box Spline Refiner : 1 1 1 1	ToggleButton : One Of Many	Render the surface using the generic box spline refiner.
Trim Edges	ToggleButton	Toggle to turn on/off trimming of surface edges.
Use Custom Trimmer	ToggleButton	Toggle to turn on/off use of a custom trimmer on the surface.
Edit Custom Trimmer	PushButton	Brings forth the “Custom Trimmer” dialog box.

Menu : Options

Menu Entry	Type	Description
Invisible Control Mesh	ToggleButton	Toggle to make the control mesh invisible.
Edit Light	PushButton	Brings forth the Directional Light Editor for the scene.

Dialogs

The two dialog boxes implemented in RefEdit were mentioned in the tables above. The first is the “New” dialog box which allows the user to start a new scene from scratch. This dialog box, shown in Figure 24, allows the user to specify the number of rows and columns in the new control mesh. The second dialog box is the “Edit Custom Trimmer” dialog shown in Figure 25. It allows the user to define a custom trimmer by specifying the left, right, top, bottom trimming values of the Custom Trimmer.

Both dialog boxes are very similar in implementation. They use push-buttons, forms, text entry

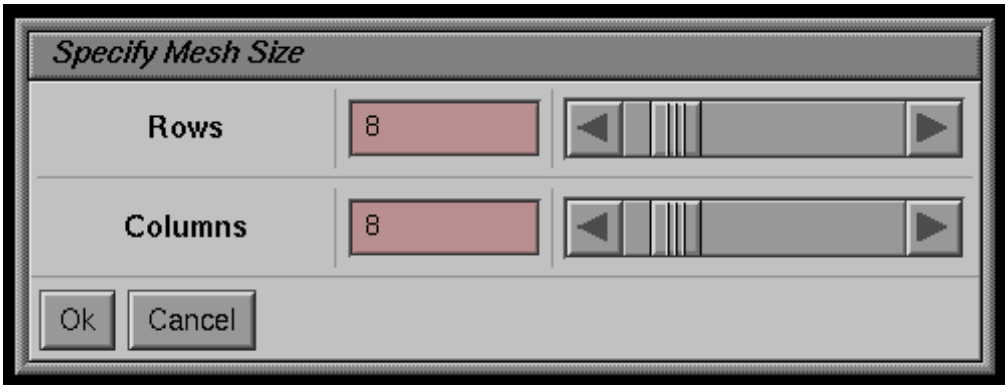


Figure 24. New dialog box.

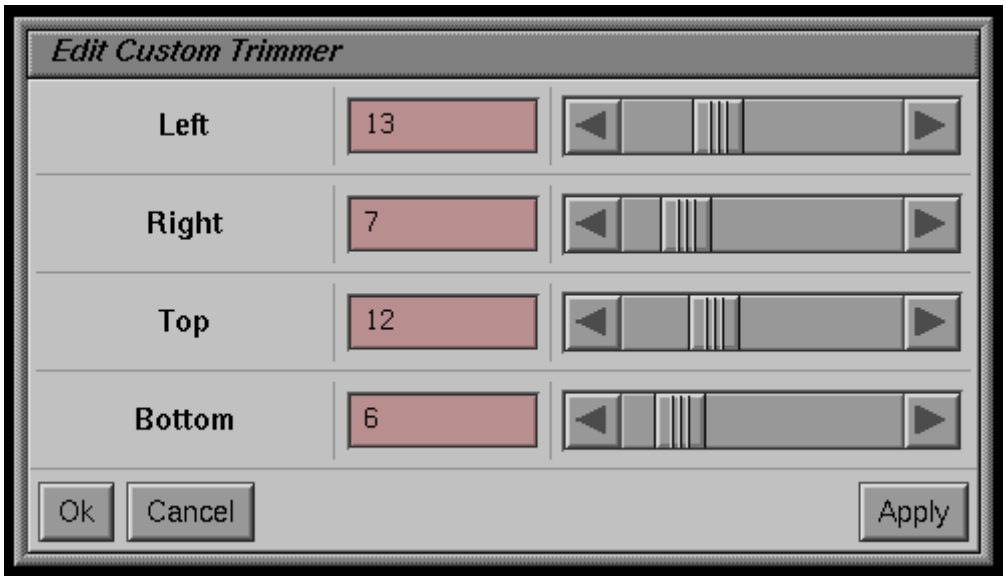


Figure 25. Edit Custom Refiner dialog box.

areas and scroll bars widgets from Motif and Xt in order to create a dialog box that retrieves numeric information from the user.

4.2.4 File I/O

The RefEdit application can read and write data files to load and save the current scene. This is accomplished by writing out enough logical data to reconstruct the entire scene when the data is read back into the application. The data written out is composed of editor environment settings and C++ class information. The former includes such items as the current directional light vector, the camera position and orientation, surface material information, and the current status of toggle buttons in the menus. The C++ class information is composed of current Trimmer, Refiner and the `NTSIdxTuple` data representing the original control mesh.

One important characteristic of all the spline classes is that they support serialization from input and output streams. This serialization allow the application to read and write class specific data from input and output streams. The refined control mesh representing the subdivision surface is not written out to the data file. Based on the current Refiner and Trimmer it is possible to reconstruct the refined control mesh data when a data file is read. Also, Open Inventor provides a mechanism to store the entire scene database to a file. However, once again, this scene database can be reconstructed from the other information in the data file and does not need to be saved.

4.3 Adding your own Refiner

One of the requirements of this work was to allow for easy integration of a new subdivision surface defined by a subdivision algorithm. Given a control mesh, if a refined mesh is achievable by a subdivision algorithm, then it is straightforward to create a Refiner class based on the subdivision algorithm. Once the Refiner class has been created and tested stand-alone, then it can be hooked into the Refiner Editor, *RefEdit*.

4.3.1 Creating a Refiner

A boiler plate for a new Refiner has been added to the `STRefiner` class directory. It consists of two files for a new Refiner class based on `STRefiner`: `SNewRefiner.cc`, the source code file and `SNewRefiner.h`, the header file for the class. These files can be modified to add the new Refiner as a subclass to the `STRefiner` class hierarchy. It is recommended the files be copied instead of being modified directly so that they are available for the addition of other Refiners in the future.

The first change required to the files would be to replace the string "`SNewRefiner`" with the class name of the new refiner. Also, both of the files have comments at important sections of the code that begin with the keyword '`NEW:`'. It is mandatory to follow the instructions of the comment starting with this keyword. The other Refiners can be examined for sample code for the important sections.

Once a Refiner is completed it can be added to the Imakefile. Once the `STRefiner` library has been built, it is important to test the new Refiner in the 'test' sub-directory with a simple application that tests the basic functionality of the Refiner. Sample input files and the corresponding output files need to be provided to for the test to ensure that future modifications to the class can verify that the changes have not broken the class. Once tested, the `STRefiner` library can be installed, ready to be used by a stand alone application.

4.3.2 Connecting to the Editor

Once a Refiner class has been defined, some changes need to made to the Editor's code to connect the Refiner to the Editor. The Editor was designed so that all the changes required would be localized to two files in the *RefEdit* application directory. `Refiners.cc` is the source code file and `Refiners.h` is the header file that require changes.

Similar to the last section, both of the files have comments at important sections that begin with the keyword '`NEW:`'. All that is required to connect the Refiner to the Editor is to follow the instructions of the comment starting with this keyword. The instructions involve adding code to support the new Refiner. The following code extracted from `Refiners.cc` shows an occurrence of the '`NEW:`' keyword in a comment.

```
////////////////////////////////////////////////////////
// REFreadRefinerFromInput()
// Reads the refiner's data from the input stream
////////////////////////////////////////////////////////
SbBool REFreadRefinerFromInput(
   RefEditData *pRE, ifstream& in )
{
  switch( pRE->nRefinerType )
  {
    case BOXREFINER:
      SBoxRefiner* pBox = new SBoxRefiner;
      in >> *pBox;
      pRE->pRefiner = pBox;
      break;

    case TPCARDREFINER:
      STPCardSplineRefiner* pTPCS = new STPCardSplineRefiner;
      in >> *pTPCS;
      pRE->pRefiner = pTPCS;
      break;

    case BOXREFINER2211:
      SBoxRefiner2211* p2211 = new SBoxRefiner2211;
      in >> *p2211;
      pRE->pRefiner = p2211;
      break;
// NEW: add an entry here for new refiner
  }
  return TRUE;
}
```

A new `case` statement can be added after the comment for a new Refiner as follows.

```
case NEWREFINER:
      SNewRefiner* pNew = new SNewRefiner;
      in >> *pNew;
      pRE->pRefiner = pNew;
      break;
```

4.4 Improvements

With any prototype application, there are always enhancements that can be made to improve the quality of the application. One important improvement necessary is the speed of the application.

The refinement is a slow process in the current implementation, although it does not need to be from a theoretical point of view. The slowdown is caused from inefficient implementation of some of the classes.

Another point related to application speed is Open Inventor. When the size of the control mesh is increased, resulting in lots of objects in the scene database, the rendering speed of Open Inventor decreases. The application was written with version 2.0 of Open Inventor. The 2.1 version of the toolkit provides several speed improvements. Also, the scene graph can be restructured and optimized for speed. For example, the scene graph for the control mesh can be redesigned so that multiple definitions of the sphere and material nodes can be removed.

Several modifications can be made to the Refiners. For example, a small box spline direction vector editor can be implemented to interactively specify other direction vectors for the Box Spline Refiner, `SBoxRefiner`. At the moment the application only instantiates the $M_{(1,1,1,1)}$ box spline. A new menu entry can be added to bring up a dialog box to specify direction vectors for a box spline. Then the `SBoxRefiner` can be instantiated with the specified direction vectors. This modification allows the user to study various box splines.

Another modification that can be made to Refiners is the ability to handle magnitude of refinements larger than 2, i.e. $w > 2$ in the equations of Chapter 2. Once again this provides the user with an opportunity to compare refined control meshes using various magnitude of refinements.

At the moment adding a new Refiner to the application requires modification to the source of the application and then re-compiling and re-linking of the application. Ideally, it would be beneficial to provide a Plug-and-Play approach to adding Refiners. What is required is a scheme that would work without requiring modification to the application. This should be possible by defining external resources for the application that could be modified every time a Refiner is added. More research is need to define the external resources.

The current editor only provides support for the plane primitive. From this primitive, the control mesh can be manipulated as desired. An improvement to the editor can involve support for simple primitives provided by other editors such as spheres, tori, cylinders, cones. The user can then manipulate the control mesh of these primitives as desired.

Yet another important improvement that can be made is adding support for control meshes on a non-rectangular grid or irregular topology. In order to do so, new data structures for a control mesh needs to be defined which would be based on specifying a set of vertices and the connectivity

of these vertices among themselves. In practice, most irregular control meshes have large regions which are regular. This observation can be exploited to construct compact and efficient control mesh data structures. To comply with such a change, the current Refiners would have to be modified to handle irregular meshes. Adding support for irregular meshes would require significant rewrite of the code in both the spline classes and the application.

Some more improvements to the editor itself include adding support for multiple surfaces and for adding support for more than one viewpoint of the scene. These are common place in most commercial editors. These improvements aid in designing the scene and ideas from other 3D editors can be borrowed.

The issues discussed here still need to be researched in detail to determine the complete specifications required for the change and the amount of work required to implement these changes. Some of these improvements are related to moving the application from a prototype to a production level. However, as a prototype the application holds its ground in several aspects and fulfills its specification for a subdivision surface editor.

4.5 Summary

In this work I have presented an abstraction for subdivision surfaces based on the control mesh and the subdivision rule of subdivision surfaces. I used this abstraction to build a subdivision surface editor to help in the designing of subdivision surfaces through control mesh refinement. I presented the theory behind two subdivision surfaces, tensor product cardinal spline and box spline surfaces. From this theory, I generated three subdivision algorithms that were used to create three Refiners and corresponding Trimmers.

In this chapter I detailed the implementation behind the Refiner and Trimmer spline classes. These classes were created with the aid of support classes from the Splines classes repository. I wrote the *RefEdit*, a generic subdivision surface editor using these classes to implement an environment for studying subdivision surfaces based on regular mesh. The implementation for the editor was based on the spline classes, Open Inventor and Motif. The purpose behind the editor was to test the abstraction of subdivision surfaces that can be used to render these surfaces.

The result of the refinement process to render subdivision surfaces was shown in Figures 12, 13, 14, 15 and 17. These figures are screen shots from the editor using one of the three Refiners and

different levels of refinement. The tensor product cardinal spline surfaces are an example of surfaces well studied in spline theory. The ability of the editor to render these surfaces based on the refinement abstraction demonstrates the editor's strength. In some situations, the Refiner based on the $M_{(2,2,1,1)}$ box spline proved to render the surface represented by the control mesh better than the tensor product cardinal spline Refiner. However, screen shots from these results were not presented in this work.

Rendering subdivision surfaces through control mesh refinement is not an original idea presented in this thesis. Others have used control mesh refinement for the purpose of rendering and other applications. The subdivision surface abstraction and the generic subdivision surface editor is however unique to this work. Also, the ability to switch Refiners interactively in the editor is unique to this thesis. Although the control mesh used in this work were limited to regular, 4-connected meshes, the abstraction can apply to irregular meshes also. Support for irregular meshes is one of several improvements that can be made to the editor.

Appendix A

Man pages for C++ Classes

The following pages contain man pages of the following C++ classes used in this work.

- STRefiner**,**
- STPCardSplineRefiner,
- SBoxRefiner,
- SBoxRefiner2211,
- STrimmer,
- SConstantTrimmer,
- SCustomTrimmer,
- SBox2211Trimmer**,**
- Lattice,
- TensorIndex,
- NTSIdxTuple.
- TupleVec

NAME

STRefiner - Templated class for Control Vertex refinement.

DESCRIPTION

This class is a rewrite of the Refiner class, but is now templated. Its purpose is to take a set of control vertices for curves or surfaces and provide a refined set of control vertices. All knot insertion methods are derived from this class. All knot insertion classes are then used on basis, curves and surfaces. Also all surface refiners are derived from this abstract class.

PREREQUISITES

rwtool, STools.

SEE ALSO

STKnotInserter, STSurfaceRefiner, STBezSingleBreakInserter, SBoxRefiner, SBoxRefiner2211, STPCardSplineRefiner.

AUTHOR(S)

Haroon Sheikh, hsasheik@watcgl.uwaterloo.ca

COPYRIGHT

TEMPLATE CLASS STRefiner

Template argument(s)

class IndexType

Base class(es)

public virtual SError

Friends

ostream& operator<<(ostream& os, const STRefiner<IndexType>& inst);
istream& operator>>(istream& is,STRefiner<IndexType>& inst);

Public members

STRefiner(void);
Default constructor.

STRefiner(const STRefiner& rhs);
Copy Constructor.

virtual ~STRefiner(void);
Destructor.

IndexType* newCVs(const IndexType& oldVert) const;
Returns : pointer to new set of control vertices.
It is callers responsibility to free memory of new CVs.

void newCVs(const IndexType& oldVert, IndexType* newVert) const;
oldVert : reference to new set of control vertices.
newVert : pointer to new set of control vertices.
It is callers responsibility to allocate and free memory of new CVs.

virtual IndexType* allocateSpaceForNewCVs(const IndexType& oldVert) const = 0;
Allocates storage for new control vertices.
Returns: pointer to allocated storage for new control vertices.

virtual void freeSpaceForNewCVs(IndexType* newVert) const = 0;
frees the storage for new control vertices.

virtual RWString className(void) const = 0;
Class identifier.

Protected members

const static RWString classname;
Class identifier.

void printOn(ostream& os) const;
void scanFrom(istream& is);
void verify(void) const;

Private members

virtual void realNewCVs(const IndexType& oldVert, IndexType* newVert) const = 0;
oldVert : reference to old set of control vertices.
newVert : pointer to new set of control vertices.

INCLUDED FILES

STools/SError.h
rw/rwstring.h

NAME

STKnotInserter - A templated Knot Insertion class

DESCRIPTION

This is an abstract, templated knot insertion class. Provides a common base class for STBez*BreakInserter and STNUB*KnotInserter

PREREQUISITES

STRefiner, NumberSequence.

SEE ALSO

STKnotInserter, STBezSingleBreakInserter.

AUTHOR(S)

Haroon Sheikh, hsasheik@watcgl.uwaterloo.ca

COPYRIGHT

TEMPLATE CLASS STKnotInserter

Template argument(s)

class IndexType

Base class(es)

public STRefiner<IndexType>

Friends

ostream& operator<<(ostream& os, const STRefiner<IndexType>& inst);
istream& operator>>(istream& is, STRefiner<IndexType>& inst);

Public members

STKnotInserter(void);
Default Constructor.

STKnotInserter(int inDimCVs, int OutDimCVs);
Constructor.

STKnotInserter(const STKnotInserter& rhs);
Copy Constructor.

virtual ~STKnotInserter(void);

int DimCVsIn(void) const;
int DimsCVsOut(void) const;

virtual IndexType* allocateSpaceForNewCVs(const IndexType& oldCVs) const;
Allocates storage for new control vertices.
Returns: pointer to allocated storage for new control vertices.

virtual void freeSpaceForNewCVs(IndexType* newCVs) const;
frees the storage for new control vertices.

virtual void orderError(int order, char* where) const;
virtual void sizeError(int a, int b, char* where, char* msg) const;

virtual RWString className(void) const = 0;

Class identifier.

Protected members

virtual void realNewCVs(const IndexType& oldVert, IndexType* newVert) const = 0;
newVert : pointer to new set of control vertices.

int theDimCVsIn;
the dimensions of the input control vertices.
int theDimCVsOut;
the dimensions of the output control vertices.

const static RWString classname;
Class identifier.

void printOn(ostream& os) const;
void scanFrom(istream& is);
void verify(void) const;

Private members

OPERATORS, FREE FUNCTIONS, CODE, ETC.

NumberSequence padOnto(NumberSequence&, int n, double miN, double maX);
DoubleVec padOnto(DoubleVec&, int n);
This will put n of miN-1 at the beginning and n of maX+1 at the end of the NumberSequence. The second version does the same, but pads with zeros.

NumberSequence unPad(NumberSequence&, int n);
DoubleVec unPad(DoubleVec&, int n);
Remove first order and last n elements in both cases.

INCLUDED FILES

NumberSequence/NumberSequence.h
STRefiner.h

NAME

STSurfaceRefiner - Abstract templated class for Surface refinement

DESCRIPTION

Its purpose is to take a set of control vertices for surfaces and provide a refined set of control vertices. It forms the base class for several surface refiners.

PREREQUISITES

rwtool, STools

SEE ALSO

STRefiner, SBoxRefiner, SBoxRefiner2211, STPCardSplineRefiner

AUTHOR(S)

Haroon Sheikh, hsasheik@watcgl.uwaterloo.ca

COPYRIGHT

TEMPLATE CLASS STSurfaceRefiner

Template argument(s)

class IndexType

Base class(es)

public STRefiner<IndexType>

Friends

ostream& operator<<(ostream& os, const STSurfaceRefiner<IndexType>& inst);
istream& operator>>(istream& is, STSurfaceRefiner<IndexType>& inst);

Public members

STSurfaceRefiner(int RefineLevel);
Contructor.

STSurfaceRefiner(void);
Default constructor.

STSurfaceRefiner(const STSurfaceRefiner& rhs);
Copy constructor.

virtual ~STSurfaceRefiner(void);

virtual IndexType* allocateSpaceForNewCVs(const IndexType& oldVert) const = 0;
Allocates storage for new control vertices.
Returns: pointer to allocated storage for new control vertices.

virtual void freeSpaceForNewCVs(IndexType* newVert) const = 0;
Frees the storage for new control vertices.

int getRefineLevel(void) const;
Selector.

void setRefineLevel(const int& newRefineLevel);
Modifier.

const STrimmer& getTrimmer(void);
Selector.

virtual RWString className(void) const = 0;
Class identifier.

Protected members

int theRefineLevel;
# of refinement iterations to perform.

STrimmer *pTrimmer;
pointer to Refiner's trimmer object.

const static RWString classname;
Class identifier.

void printOn(ostream& os) const;
void scanFrom(istream& is);
void verify(void) const;

Private members

virtual void realNewCVs(const IndexType& oldVert, IndexType* newVert) const = 0;
oldVert : reference to old set of control vertices.
newVert : pointer to new set of control vertices.

virtual void assignTrimmer(void) = 0;
Derived classes should assign the specific trimmer here.

INCLUDED FILES

rw/rwstring.h
STRefiner.h
STrimmer/STrimmer.h

NAME

STPCardSplineRefiner - Tensor Product Cardinal Spline refiner.

DESCRIPTION

A tensor product cardinal spline refiner of control vertices. The refiner works on a lattice indexed set of control vertices. It returns a new set of refined control vertices refined a specified number of times.

```
STPCardSplineRefiner TPCardSplineRef(5);
IntVec sizes(2);
IntVec origin(2);
sizes[0] = 1; sizes[1] = 1;
origin[0] = -1; origin[1] = -1;
TensorIndex Lat(sizes, 0, &origin);
NTSIdxTuple oldCVs(Lat,3);
.
.  // initialize oldCVs here
.
pNewCVs = TPCardSplineRef.allocateSpaceForNewCVs( oldCVs );
TPCardSplineRef.newCVs( oldCVs, pNewCVs );
TPCardSplineRef.freeSpaceForNewCVs( pNewCVs );
```

PREREQUISITES

rwtool, STools, STRefiner, STSurfaceRefiner, NTSIdxTuple.

SEE ALSO

SBoxRefiner, SBoxRefiner2211.

AUTHOR(S)

Haroon Sheikh, hsasheik@watcgl.uwaterloo.ca

COPYRIGHT

CLASS STPCardSplineRefiner

Base class(es)

public STSurfaceRefiner<NTSIdxTuple>

Friends

ostream& operator<<(ostream& os, STPCardSplineRefiner& inst);
istream& operator>>(istream& is, STPCardSplineRefiner& inst);

Public members

STPCardSplineRefiner(void);
Default Constructor.

STPCardSplineRefiner(const STPCardSplineRefiner& rhs);
Copy Contructor.

STPCardSplineRefiner(int RefineLevel);
Contructor.
RefineLevel - # of iterations to perform. RefineLevel is > 0.

~STPCardSplineRefiner(void);
Destructor.

virtual RWString className(void) const;
Selector function to report the class identifying name.

NTSIdxTuple* allocateSpaceForNewCVs(const NTSIdxTuple& oldCVs) const;
void freeSpaceForNewCVs(NTSIdxTuple* newCVs) const;
Storage allocation/deallocation for new control vertices.

Protected members

const static RWString classname;

void printOn(ostream& os) const;
void scanFrom(istream& is);
void verify(void) const;

Private members

virtual void realNewCVs(const NTSIdxTuple& oldCVs, NTSIdxTuple* newVert) const;
Refine old control vertices to new control vertices.

NTSIdxTuple* allocateSpaceForNextCVs(const NTSIdxTuple& oldCVs) const;
void freeSpaceForNextCVs(NTSIdxTuple* newCVs) const;
Allocate/deallocate space for the next set of c.v.s.

virtual void assignTrimmer(void);
Assign the specific trimmer.

INCLUDED FILES

STools/SError.h
Tuple/TupleVec.h
NTSIndexedVec/NTSIdxTuple.h
rw/rwstring.h
STSurfaceRefiner.h
iostream.h
assert.h

NAME

SBoxRefiner - Box Spline Control Vertices Refiner

DESCRIPTION

A box spline refiner of control vertices. The refiner works on a lattice indexed set of control vertices. It returns a new set of refined control vertices according to the arguments passed to the contructor. It is recommended to perform multiple refinements using a small magnitude of refinement than to refine once with a high magnitude. For example, refine 5 times using a magnitude of 2 rather than refine once with a magnitude of 32.

```
TupleVec DirVecs(2,3);
DirVecs(0) = Duple(1,0);
DirVecs(1) = Duple(0,1);
DirVecs(2) = Duple(1,1);
   // Setup Direction vectors for Box Spline
SBoxRefiner box(DirVecs, 2, 5);
IntVec sizes(2);
IntVec origin(2);
sizes[0] = 1; sizes[1] = 1;
origin[0] = -1; origin[1] = -1;
TensorIndex Lat(sizes, 0, &origin);
NTSIdxTuple oldCVs(Lat,3);
.
.  // initialize oldCVs here
.
pNewCVs = box.allocateSpaceForNewCVs( oldCVs );
box.newCVs( oldCVs, pNewCVs );
box.freeSpaceForNewCVs( pNewCVs );
```

PREREQUISITES

rwtool, STools, STRefiner, STSurfaceRefiner.

SEE ALSO

STPCardSplineRefiner, SBoxRefiner2211.

AUTHOR(S)

Haroon Sheikh, hsasheik@watcgl.uwaterloo.ca

COPYRIGHT

CLASS SBoxRefiner

Base class(es)

public STSurfaceRefiner<NTSIdxTuple>

Friends

ostream& operator<<(ostream& os, SBoxRefiner& inst);
istream& operator>>(istream& is, SBoxRefiner& inst);

Public members

SBoxRefiner(void);
Default Constructor.

SBoxRefiner(const SBoxRefiner& rhs);
Copy Contructor.

SBoxRefiner(const TupleVec& DirVecs, int Mag, int RefineLevel);
Contructor
DirVecs - Set of direction vectors defining the box spline.
Mag - magnitude of refinement at each iteration. Mag is > 0.
RefineLevel - # of iterations to perform. RefineLevel is > 0.

~SBoxRefiner(void);
Destructor.

TupleVec getDirVecs(void) const;
int getMag(void) const;
Selectors.

virtual RWString className(void) const;
Selector function to report the class identifying name.

void setDirVecs(const TupleVec& newDirVecs);
void setMag(const int& newMag);
Modifiers.

NTSIdxTuple* allocateSpaceForNewCVs(const NTSIdxTuple& oldCVs) const;
void freeSpaceForNewCVs(NTSIdxTuple* newCVs) const;
Storage allocation/deallocation for new control vertices.

Protected members

const static RWString classname;

void printOn(ostream& os) const;
void scanFrom(istream& is);
void verify(void) const;

Private members

TupleVec theDirVecs;
Direction vectors of box spline.
int theMag;
Magnitude of refinement at each iteration.

virtual void realNewCVs(const NTSIdxTuple& oldCVs, NTSIdxTuple* newVert) const;
Refine old control vertices to new control vertices.

NTSIdxTuple* allocateSpaceForNextCVs(const NTSIdxTuple& oldCVs) const;
void freeSpaceForNextCVs(NTSIdxTuple* newCVs) const;
Allocate/deallocate space for the next set of c.v.s.

virtual void assignTrimmer(void);
Assign the box spline trimmer here.

INCLUDED FILES

STools/SError.h
Tuple/TupleVec.h
NTSIndexedVec/NTSIdxTuple.h
rw/rwstring.h
STSurfaceRefiner.h
iostream.h
assert.h

NAME

SBoxRefiner2211 - Box Spline Vertices Refiner for 2,2,1,1 Box Spline

DESCRIPTION

The refiner works on a lattice indexed set of control vertices. It returns a new set of refined control vertices according to the arguments passed to the contructor. This box spline refiner is based on the 2,2,1,1 box spline, ie. the box spline with the following direction vectors: [1, 0], [0, 1], [1, 1], [1, -1], [1, 0], [0, 1]. It is recommended to perform multiple refinements using a small magnitude of refinement than to refine once with a high magnitude. For example, refine 5 times using a magnitude of 2 rather than refine once with a magnitude of 32.

```
SBoxRefiner2211 boxSpline(3);
IntVec sizes(2);
IntVec origin(2);
sizes[0] = 1; sizes[1] = 1;
origin[0] = -1; origin[1] = -1;
TensorIndex Lat(sizes, 0, &origin);
NTSIdxTuple oldCVs(Lat,3);
.
.  // initialize oldCVs here
.
pNewCVs = box.allocateSpaceForNewCVs( oldCVs );
box.newCVs( oldCVs, pNewCVs );

// Use pNewCVs

box.freeSpaceForNewCVs( pNewCVs );
```

PREREQUISITES

rwtool, STools, STRefiner, STSurfaceRefiner.

SEE ALSO

STPCardSplineRefiner, SBoxRefiner.

AUTHOR(S)

Haroon Sheikh, hsasheik@watcgl.uwaterloo.ca

COPYRIGHT

CLASS SBoxRefiner2211

Base class(es)

public STSurfaceRefiner<NTSIdxTuple>

Friends

ostream& operator<<(ostream& os, SBoxRefiner2211& inst);
istream& operator>>(istream& is, SBoxRefiner2211& inst);

Public members

SBoxRefiner2211(void);
Default Constructor.

SBoxRefiner2211(const SBoxRefiner2211& rhs);
Copy Constructor.

SBoxRefiner2211(int RefineLevel);
Constructor.
RefineLevel - # of iterations to perform. RefineLevel is > 0

~SBoxRefiner2211(void);
Destructor

virtual RWString className(void) const;
Selector function to report the class identifying name.

NTSIdxTuple* allocateSpaceForNewCVs(const NTSIdxTuple& oldCVs) const;
void freeSpaceForNewCVs(NTSIdxTuple* newCVs) const;
Storage allocation/deallocation for new control vertices.

Protected members

const static RWString classname;

void printOn(ostream& os) const;
void scanFrom(istream& is);
void verify(void) const;

Private members

virtual void realNewCVs(const NTSIdxTuple& oldCVs, NTSIdxTuple* newVert) const;
Refine old control vertices to new control vertices.

NTSIdxTuple* allocateSpaceForNextCVs(const NTSIdxTuple& oldCVs) const;
void freeSpaceForNextCVs(NTSIdxTuple* newCVs) const;
Allocate/deallocate space for the next set of c.v.s.

virtual void assignTrimmer(void);
Assign the specific trimmer.

INCLUDED FILES

STools/SError.h
Tuple/TupleVec.h
NTSIndexedVec/NTSIdxTuple.h
rw/rwstring.h
STSurfaceRefiner.h
iostream.h
assert.h

NAME

STrimmer - Surface Trimmer

DESCRIPTION

This is a abstract surface trimmer class. It defines the trimming of a surface for each extent of every surface dimension for every refinement Level. This extent is represent by a Start vector and an End vector for every refinement level. for example, on a two dimensional domain space, the surface can be trimmed from the left, top, right, and bottom and it would be represented by Start[2] and End[2] where Start[0] is left, Start[1] is bottom and End[0] is right, End[1] is top Sub-classes can have specialized initialization for the Start and End Values.

USAGE

```
#include <STrimmer/STrimmer.h>
STrimmer *myTrimmer = new SConstantTrimmer( 5, 2, 1 );
IntVec Start = myTrimmer->getStart( 3 );
IntVec End = myTrimmer->getEnd( 4 );
```

LIBRARIES

rwmath, rwtool

SEE ALSO

STRefiner, SConstantTrimmer, SCustomTrimmer, SBox2211Trimmer

AUTHOR(S)

Haroon Sheikh, hsasheik@cgl

COPYRIGHT

CLASS STrimmer

Base class(es)

public virtual SError

Friends

ostream& operator<<(ostream& os, const STrimmer& inst);
istream& operator>>(istream& is, STrimmer& inst);

Public members

STrimmer(void);
virtual ~STrimmer(void);

IntVec getStart(int nRefineLevel) const;
IntVec getEnd(int nRefineLevel) const;
void getExtents(int nRefineLevel, IntVec& start, IntVec& end) const;
Get the start and end value for the specified refinement level.

virtual RWString className(void) const;
Selector function to report the class identifying name.

Protected members

STrimmer(int maxRefineLevel, int Dim);
STrimmer(const STrimmer& inst);

const static RWString classname0;

virtual void printOn(ostream& os) const;
virtual void readFrom(istream& is);

void setStart(int nRefineLevel, const IntVec& newValue);
void setEnd(int nRefineLevel, const IntVec& newValue);
void setExtents(int nRefineLevel, const IntVec& newStart,
const IntVec& newEnd);
Set the start and end value for the specific refinement level.

void setDimension(const int& newDim);
Sets new dimension for start/end vectors.
Status of start/end vectors after this call is undefined.

void setMaxRefineLevel(const int& newMax);
Sets new maximum for refinement level.
Status of start/end vectors after this call is undefined.

IntGenMat theStartData;
IntGenMat theEndData;

Private members

INCLUDED FILES
iostream.h
STools/SError.h
rw/rwstring.h
rw/ivec.h
rw/igenmat.h

NAME

SConstantTrimmer - Constant Surface Trimmer

DESCRIPTION

This surface trimmer class trims a constant value along each of the extents of the surface. For example, on a two dimensional domain space, the surface will be trimmed from the left, top, right, and bottom by the same constant value.

USAGE

```
#include <SConstantTrimmer/SConstantTrimmer.h>
SConstantTrimmer myTrimmer( 5, 2, 1 );
IntVec Start = myTrimmer.getStart( 2 );
IntVec End = myTrimmer.getEnd( 4 );
myTrimmer.getExtents( 3, Start, End );
```

LIBRARIES

rwmath, rwtool.

SEE ALSO

STrimmer, STRefiner, SCustomTrimmer, SBox2211Trimmer

AUTHOR(S)

Haroon Sheikh, hsasheik@cgl

COPYRIGHT

CLASS SConstantTrimmer

Base class(es)

public STrimmer

Friends

ostream& operator<<(ostream& os, const SConstantTrimmer& inst);
istream& operator>>(istream& is, SConstantTrimmer& inst);

Public members

SConstantTrimmer(void);
SConstantTrimmer(int nMaxRefineLevel, int Dim, int TrimValue);
SConstantTrimmer(const SConstantTrimmer& inst);
virtual ~SConstantTrimmer(void);

void setConstantValue(int newValue);
Assigns a new contant trim value.

virtual RWString className(void) const;
Selector function to report the class identifying name.

Protected members

const static RWString classname1;

virtual void printOn(ostream& os) const;
virtual void readFrom(istream& is);

Private members

int theConstTrimValue;

INCLUDED FILES

iostream.h
rw/rwstring.h

rw/ivec.h
STrimmer.h

NAME

SCustomTrimmer - Custom Surface Trimmer

DESCRIPTION

This surface trimmer class trims the surface by a user specified value along each extent of the surface for a specific refinement level. For example, on a two dimensional domain space, the surface will be trimmed from the left, top, right, and bottom by user specified values for each for specific refinement level.

USAGE

```
#include <SCustomTrimmer/SCustomTrimmer.h>
IntVec initStart, initEnd;
SCustomTrimmer myTrimmer( 5, 2 );
myTrimmer.setExtents( 0, initStart, initEnd );
myTrimmer.setExtents( 1, initStart, initEnd );
myTrimmer.setExtents( 2, initStart, initEnd );
myTrimmer.setExtents( 3, initStart, initEnd );
myTrimmer.setExtents( 4, initStart, initEnd );
IntVec Start = myTrimmer.getStart( 2 );
IntVec End = myTrimmer.getEnd( 1 );
myTrimmer.getExtents( 4, Start, End );
```

LIBRARIES

rwmath

SEE ALSO

STRefiner, STrimmer, SConstantTrimmer, SBox2211Trimmer

AUTHOR(S)

Haroon Sheikh, hsasheik@cgl

COPYRIGHT

CLASS SCustomTrimmer

Base class(es)

public STrimmer

Friends

ostream& operator<<(ostream& os, const SCustomTrimmer& inst);
istream& operator>>(istream& is, SCustomTrimmer& inst);

Public members

SCustomTrimmer(void);
SCustomTrimmer(int maxRefineLevel, int Dim);
SCustomTrimmer(const SCustomTrimmer& inst);
virtual ~SCustomTrimmer(void);

IntVec getStart(int nRefineLevel) const;
IntVec getEnd(int nRefineLevel) const;

void getExtents(int nRefineLevel, IntVec& Start, IntVec& End);
Retrieves the trim values for start and end for specific refinement level.

void setStart(int nRefineLevel, const IntVec& newValue);
void setEnd(int nRefineLevel, const IntVec& newValue);

void setExtents(int nRefineLevel, const IntVec& newStart, const IntVec& newEnd);
Assigns new trim values for start and end for specific refinement level.

void setDimension(const int& newDim);
Sets new dimension for start/end vectors.

Status of start/end vectors after this call is undefined.

void setMaxRefineLevel(const int& newMax);
Sets new maximum for refinement level.
Status of start/end vectors after this call is undefined.

virtual RWString className(void) const;
Selector function to report the class identifying name.

Protected members

const static RWString classname1;

virtual void printOn(ostream& os) const;
virtual void readFrom(istream& is);

Private members

INCLUDED FILES

iostream.h
rw/rwstring.h
rw/ivec.h
STrimmer.h

NAME

SBox2211Trimmer - Surface Trimmer for a box spline surface.

DESCRIPTION

This surface trimmer class trims the surface according to the requirement for a 2,2,1,1 box spline surface.

USAGE

```
#include <SBox2211Trimmer/SBox2211Trimmer.h>
SBox2211Trimmer *myTrimmer = new SBox2211Trimmer( 5, 2 );
IntVec Start = myTrimmer->getStart( 3 );
IntVec End = myTrimmer->getEnd( 4 );
```

LIBRARIES

rwmath, rwtool

SEE ALSO

STRefiner, STrimmer, SConstantTrimmer, SCustomTrimmer

AUTHOR(S)

Haroon Sheikh, hsasheik@cgl

COPYRIGHT

CLASS SBox2211Trimmer

Base class(es)

public STrimmer

Friends

ostream& operator<<(ostream& os, const SBox2211Trimmer& inst);
istream& operator>>(istream& is, SBox2211Trimmer& inst);

Public members

SBox2211Trimmer(void);
SBox2211Trimmer(int maxRefineLevel);
SBox2211Trimmer(const SBox2211Trimmer& inst);
virtual ~SBox2211Trimmer(void);

IntVec getStart(int nRefineLevel) const;
IntVec getEnd(int nRefineLevel) const;
void getExtents(int nRefineLevel, IntVec& start, IntVec& end) const;
Get the start and end value for the specified refinement level.

void setMaxRefineLevel(const int& newMax);
Sets new maximum for refinement level.
Status of start/end vectors after this call is undefined.

virtual RWString className(void) const;
Selector function to report the class identifying name.

Protected members

const static RWString classname0;

virtual void printOn(ostream& os) const;
virtual void readFrom(istream& is);

Private members

INCLUDED FILES

iostream.h

rw/rwstring.h
rw/ivec.h
STrimmer.h

NAME

Lattice - An abstract class for multi-key to single key indexing

DESCRIPTION

A lattice allows for the creation of a map from a user defined n-dimensional space to a linear indexing scheme. The user provides an outline of the space and the Lattice generates an unique integer for each element in the space.

PREREQUISITES

rwmath, rwtool, STools

SEE ALSO

TriangleIndex, TensorIndex.

AUTHOR

Bruce Hickey, bhickey@watcgl.uwaterloo.ca

COPYRIGHT

CLASS Lattice

Base class(es)

public SError

Friends

ostream& operator<<(ostream& os, const Lattice& inst);
istream& operator>>(istream& is, Lattice& inst);

Public members

enum latEnum { BEGIN = -9, END = -99 };

Lattice(void);
A default (no arguments) constructor is included so we can create arrays of Lattice.

Lattice(unsigned n);
A n-dimensional Lattice

Lattice(const Lattice&);
Copy constructor.

virtual ~Lattice(void);
Class destructor.

virtual int operator ()(int);
virtual int operator ()(int, int);
virtual int operator ()(int, int, int);
virtual int operator ()(int, int, int, int);
virtual int operator ()(int, int, int, int, int);
virtual int operator ()(IntVec& i);
The first five opreators simply build an Intvec and call the last version. This exist simply to allow flexible access. They all update the index member.

virtual int operator ()();
Get map of current value in index.

int operator++();

int operator--();
Shorthand for next()/prev().

void deepenShallowCopy(void);
Dealias data.

virtual int getDimension( ) const =0;
virtual void putDimension(unsigned) =0;
Retrieve/change the dimension. The new dimension is returned after the change.

virtual IntVec getIndex(void) const;
virtual int getIndex(int i) const;
virtual int putIndex(int i);
virtual int putIndex(int i, int n);
virtual int putIndex(const IntVec& i);
Retrieve/change the index vector to a specific number or sequence.
PutIndex will return the map of the new index.

virtual int prev(void) =0;
virtual int next(void) =0;
Move index to next/previous position and return map of new index.

virtual int first(void) =0;
virtual int last(void) =0;
Move index to next/previous position and return map of new index.

virtual RWBoolean isEOI(void) const ;
virtual RWBoolean isBOI(void) const ;
Check for Begining/End Of Index.

virtual int size(void);
The size of the array indexed by this lattice.

Protected members

IntVec index;
The last index, used for fast repeated access.

virtual RWBoolean verify(void) const =0;
Conditions specific to derived instances of Lattice.

virtual int multiToSingle(void) const =0;
Convert from multiple to single indexes, the multiple index used is the INDEX variable above.

RWBoolean isEqual(const IntVec& iv, int n) const;
Check if all elements of iv are equal to n.

Private members

INCLUDED FILES

iostream.h
STools/SError.h
rw/ivec.h
rw/rwstring.h

NAME

TensorIndex - A rectangular lattice.

DESCRIPTION

A TensorIndex provides multiple to single indexing. This allows for the storage of an n-dimensional rectangular array, such as the control points of a tensor product. The size of the array, in each dimension, is maintained.

```
IntVec sizes(3);
//  sizes[0] = 2; sizes[1] = 3; sizes[2] = 5;
//  TensorIndex ti(sizes);  // Indexes into a 2x3x5 array
```

PREREQUISITES

rwmath, rwtool, STools

SEE ALSO

TriangleIndex

AUTHOR

Bruce Hickey, bhickey@watcgl.uwaterloo.ca

COPYRIGHT

CLASS Lattice

Base class(es)

public SError

Friends

ostream& operator<<(ostream& os, const TensorIndex& inst);
istream& operator>>(istream& is, TensorIndex& inst);

Public members

TensorIndex(void);
Default Constructor.

TensorIndex(const TensorIndex&);
Copy Constructor.

TensorIndex(const IntVec& sizes, IntVec* idx=0, IntVec* org=0);
Constructor with initialization values, you can pass a pointer to an initial index if desired.. org was added to support displaced origins.

~TensorIndex(void);
Destructor.

void deepenShallowCopy(void);
Dealias the data within this object.

TensorIndex copy(void) const;
Make of copy of this object.

TensorIndex& operator = (const TensorIndex& i);
Assignment operator.

int operator == (const TensorIndex& i);

int operator != (const TensorIndex& i);
Index comparison.

virtual int next(void);
virtual int prev(void);
Set to the prev/next index.

void reset(void);
Resets dimesion to 0, sizes vector is empty.

const IntVec& getSize(void) const;
int getSize(int) const ;
Retrieve size information about all/one dimension(s).

void putSize(int);
void putSize(IntVec&);
void putSize(int, int);
Change the size along the all/one parameter space(s).

const IntVec& getOrigin(void) const;
int getOrigin(int) const;
Retrieve origin information about all/one dimension(s).

void putOrigin(int);
void putOrigin(int, int);
void putOrigin(IntVec&);
Change the origin along the all/one parameter space(s).

int getDimension(void) const ;
void putDimension(unsigned);
Retrieve/change the dimension.

int first(void);
int last(void);
Set index to first/last.

RWBoolean isFirst(void);
RWBoolean isLast(void);
Is index set to first/last.

virtual RWBoolean isEOI(void) const ;
virtual RWBoolean isBOI(void) const ;
Check for Begining/End Of Index.

virtual RWString className(void) const;
Class id.

void printOn(ostream&) const;
void scanFrom(istream&);
I/O routines.

IntVec indexSlice(IntVec&);
Returns a vector containing the indices of the 1-d positions of the specified slice in the TensorIndex.
The argument IntVec& should be of size getDimension() and contain a -1 in the position that is to be sliced.
For example you can obtain the Nth col of a 2-d TensorIndex by using the IntVec [-1 N]

void reshape(const IntVec&, IntVec* org = 0);
Reshape the lattice in all dimensions

Protected members

IntVec sizes;

degree along each parameter

IntVec origin;
displacement of origin

int multiToSingle(void) const ;
Converting multiple index to single index.

RWBoolean checkEnds(void) const;
RWBoolean checkSizes(void) const;
virtual RWBoolean verify(void)const;
Tests to ensure that the index is within the valid domain.

const static RWString classname;
Class id.

Private members

INCLUDED FILES
Lattice.h

NAME

NTSIdxTuple - Non-templated Vectors of elements indexed by lattices.

DESCRIPTION

This class holds TupleVec (each vector a Triple) data that can be indexed by TensorIndex lattice. This is just a non-templated version of SIdxTuple using the following instantiation : SIdxTuple<TensorIndex, Triple, TupleVec>.

PREREQUISITES

Lattice, Tuple, STools, RWTools, VecMtx

SEE ALSO

SIdxTuple

AUTHOR

Haroon Sheikh hsasheik@cgl.uwaterloo.ca

COPYRIGHT

CLASS NTSIdxTuple

Base class(es)

public SError

Public members

NTSIdxTuple(void);

Default constructor.

NTSIdxTuple(const NTSIdxTuple&);

Copy constructor.

NTSIdxTuple(TensorIndex lat, unsigned vecElemLen=3);

Creates a TupleVec, with tuples of length vecElemLen, the size of the TupleVec is determined by the lattice. Since we typically do work in 3-space the element length defaults to this.

NTSIdxTuple(TensorIndex lat, TupleVec dat);

Creates a NTSIdxTuple based on the indexing scheme of lat on the data dat.

~NTSIdxTuple(void);

Class destructor.

inline Triple operator()(int a);
inline Triple operator()(int a, int b);
inline Triple operator()(int a, int b, int c);
inline Triple operator()(int a, int b, int c, int d);
inline Triple operator()(IntVec&);

Indexed Data access.

inline Triple getNthEntry(int n);

Direct data access

inline TensorIndex& refLattice(void);

Return a reference to the lattice. This is designed to be overloaded be derived classes to return a reference of their type.

inline IntVec LatticeSize(void) const;
Return the size vector of the lattice

inline IntVec LatticeOrigin(void) const;
Return the origin vector of the lattice

inline int vecElemSize(void) const;
Return the vector element size

NTSIdxTuple copy() const;
Returns a copy with distinct instance variables.

virtual inline RWString className(void) const;
Class id.

void scanFrom(istream&);
void printOn(ostream&);
I/O primitives

Protected members

TupleVec data;
The data is contained in the NTSIdxTuple.

TensorIndex lattice;
The indexing function, a copy is made and retained by the NTSIdxTuple.

void verify();
Input verification.

Private members
Nothing

OPERATORS, FREE FUNCTIONS, CODE, ETC.

istream& operator>>(istream& is, NTSIdxTuple&);
ostream& operator<<(ostream& os, NTSIdxTuple&);

INCLUDED FILES

iostream.h
STools/SError.h
Tuple/Tuple.h
Tuple/TupleVec.h
Lattice/TensorIndex.h

NAME

TupleVec - A vector of vectors

DESCRIPTION

The data in a TupleVec is stored like this: [[xyz...] [xyz...] ...].You don't normally need to know this, but it may come in handy some day if you have data. You can use the constructor that takes a DoubleVec to specify the data (and create an alias to it) Use the Tuple<->DoubleVec type conversion to use this with Tuples. This should allow t=vec(i) and vec(i)=t. The TupleVec class itself does not use Tuples at all in the implementation, and you don't have to use them either, if you don't want to. A TupleVec will be resized to accomodate input. The size of each Tuple must be equal though.

Prerequisites

Uses the rwmath classes.

See Also

DoubleVec

AUTHOR(S)

Al Vermeulen, ahvermen@watcgl.uwaterloo.ca

COPYRIGHT

CLASS TupleVec

Base class(es)

public SError

Friends

ostream& operator<<(ostream&, const TupleVec&);
istream& operator>>(istream&, TupleVec&);

Public members

void tupleLength(unsigned int);
void length(unsigned int);

Changes will leave the data in an unpredictable state. See insertSpace() for more controlled operations.

TupleVec(unsigned int tupleLength=0, unsigned int vectorLength=0, void* dummy=NULL);
TupleVec(unsigned int tupleLength, DoubleVec& data, void* dummy=NULL);
TupleVec(const TupleVec&);
virtual ~TupleVec(void);

The second one creates an alias to data. To make sure the TupleVec has its own copy of the data use TupleVec(tlen,data.copy()) The dummy parameter is currently ignored. dummy has been introduced for template compatibility with frame dependent classes, such as SPointVec and SVectorVec.

TupleVec& operator=(const DoubleVec&);
TupleVec& operator=(const TupleVec&);
void reference(TupleVec&);

For all 3 the length of the tuples in this and in the argument must agree. The first op= sets each tuple to the argument. The length of the argument must be TupleLength(). In the second op=, the length of the vector, and the tuple size must agree. Reference creates an alias to the data in its argument. reference can change the length of the vector. To create a reference with its own copy of the data use: a.reference(b.copy());

TupleVec& operator+=(const TupleVec& arg);

TupleVec& operator-=(const TupleVec& arg);
TupleVec& operator*=(const TupleVec& arg);
TupleVec& operator/=(const TupleVec& arg);
Vector arithmetic *this[i][j] ?= arg[i][j].

TupleVec& operator+=(const DoubleVec& arg);
TupleVec& operator-=(const DoubleVec& arg);
TupleVec& operator*=(const DoubleVec& arg);
TupleVec& operator/=(const DoubleVec& arg);
Vector arithmetic *this[i][j] ?= arg[i].

TupleVec& operator+=(double arg);
TupleVec& operator-=(double arg);
TupleVec& operator*=(double arg);
TupleVec& operator/=(double arg);
Scalar arithmetic *this[i][j] ?= arg.

TupleVec copy(void) const;
void deepenShallowCopy(void);
copy() returns a copy() with seperate data. deepenShallowCopy() dealiases the data.

unsigned int tupleLength(void) const;
unsigned int length(void) const;
Query the number or size of the tuples.

DoubleVec data(void);
DoubleVec data(void) const;
data() provides a DoubleVec that points to the same data as the vector. use data().copy() to get a DoubleVec with a copy() of the data.

DoubleVec operator()(int i);
const DoubleVec operator()(int i) const;
Element access with bounds checking.

DoubleVec componentVec(int i);
const DoubleVec componentVec(int i) const;
Accesses a vector of the ith components.

TupleVec subVec(int start, unsigned int len);
const TupleVec subVec(int start, unsigned int len) const;
Accesses a subvector.

DoubleVec vecNorm(void) const;
void vecNorm(DoubleVec*) const;
Return the vector of the norms of individual vectors comprising TupleVec.

void insertSpace(int where, int amount);
A negative amount means delete space. where will be the first new element if amount is positive, where will be the first element deleted if space is negative.

virtual void printOn(ostream&) const;
virtual void scanFrom(istream&);
I/O routines.

RWString className(void) const;
Class identifier.

Protected members

const static RWString classname;
Class identifier.

void verify(void) const;
Verification of data.

Private members

unsigned int tupleLen;

DoubleVec vec;

tupleLen is the length of each tuple, vec is the data

OPERATORS, FREE FUNCTIONS, CODE, ETC.

double dot(const TupleVec&, const TupleVec&);

CLASS DECLARATIONS

class ostream

INCLUDED FILES

rw/dvec.h

STools/SError.h

Bibliography

[Cat78] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. In *Computer-Aided Design*, Volume 10, pages 350-355, September 1978.

[Cav89] A. S. Cavaretta and C. A. Micchelli. Subdivision algorithms. In T. Lyche and L. Schumaker, editors, *Mathematical Methods in Computer Aided Geometric Design*, Academic Press, 1989.

[Cha74] G. M. Chaikin. An algorithm for high speed curve generation. In *Computer Graphics and Image Processing*, Volume 3, pages 346-349, 1974.

[Dae89] M. Daehlen. On the Evaluation of Box Splines. In T. Lyche and L. Schumaker, editors, *Mathematical Methods in Computer Aided Geometric Design*, Academic Press, 1989.

[Dae91] M. Daehlen and T. Lyche. Box Splines and Applications. In H. Hagen and D. Roller, editors, *Geometric Modeling: Methods and Applications*, Springer-Verlag, 1991.

[deB95] C. de Boor, K. Hollig, and S. Riemenschneider. *Box Splines*. Springer-Verlag, 1995.

[DeR93] T. DeRose, et al. Piecewise Smooth Surface Reconstruction. In *Proceedings of SIGGRAPH '94*, pages 295-300, 1994.

[Doo78] D. Doo and M. Sabin. Behaviour of recursive division surfaces near extraordinary points. In *Computer-Aided Design*. Volume 10, pages 356-360, September 1978.

[Far93] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, third edition, 1993.

[Hic94] B. Hickey. *Fitting Data of Arbitrary Dimension with B-splines and Applications to Colour Calibration*. Masters Thesis, University of Waterloo, Waterloo, Ontario, Canada, 1994.

[Hol89] K. Hollig. Box-Spline Surfaces. In T. Lyche and L. Schumaker, editors, *Mathematical Methods in Computer Aided Geometric Design*, Academic Press, 1989.

[Hol90] K. Hollig and Harald Mogerle. G-splines. In *Computer Aided Geometric Design*, Volume 7, pages 197-207, 1990.

[Loo94] C. Loop. Smooth Spline Surfaces over Irregular Meshes. In *Proceedings of SIGGRAPH '94*, pages 303-310, 1994.

[Mot94] *OSF/Motif 1.2 Edition for X11, Release 5, Volume Four: X Toolkit Intrinsics Programming Manual.* Silicon Graphics IRIS Insight Manual, O'Reilly & Associates Inc., 1994.

[Pet93] J. Peters. Smooth free-form surfaces over irregular meshes generalizing quadratic splines. In *Computer Aided Geometric Design*, Volume 10, pages 347-361, 1993.

[Rog91] Rogue Wave Software, *Math++ Introduction and Reference Manual*. 1991.

[Sch81] L.L. Schumaker. *Spline functions: Basic Theory*, Whiley & Sons, New York, 1981.

[Wer94] J. Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics With Open Inventor, Release 2.* Silicon Graphics IRIS Insight Manual, Addison-Wesley Publishing Company, 1994.