Efficient Subdivision Surface Evaluation

by

Eric Hall

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Technical Report Number CS-2000-21

Waterloo, Ontario, Canada, 2001

Abstract

New algorithms and data representations are introduced for the efficient evaluation of subdivision surfaces. The first algorithm presented uses a data structure based on strips of quadrilaterals to implicitly define adjacency information for mesh faces through vertex order. This approach reduces memory requirements by a constant factor in the average case and allows for a fast evaluation scheme. The next algorithm extends this idea for use in a hardware evaluation setting. A simple API and protocol for transmitting quadstrips with adjacency information to hardware is defined. An online algorithm requiring only small constant storage for a given recursion depth is then used to subdivide each strip.

Acknowledgements

I would like to thank my supervisors Stephen Mann and Michael McCool for their advice and encouragement.

I would also like to thank Blair Conrad who was always willing to offer advice and occasional assistance as a more experienced graduate student and the Computer Graphics Lab manager.

The University of Waterloo Computer Graphics Lab has provided me with an excellent working environment. I would like to thank all those who have contributed to this environment, past and present. In particular, thanks to all of my friends from the lab and the current director Bill Cowan.

Financial support for this thesis was provided by NSERC and The University of Waterloo.

Contents

List of Figures

Chapter 1

Introduction

This thesis will address issues involved with the implementation of high-performance subdivision surface evaluation. The principal contribution will be the development of techniques that use simple geometry compression to reduce storage and bandwidth requirements when rendering subdivision surfaces. These techniques will be extended to create efficient algorithms for implementation in hardware.

In general terms, a hardware implementation must be concerned with four key issues. Firstly, the purpose of creating specialized hardware is to speed execution by performing operations in parallel. The trade off for parallel execution will be an increased gate count in the hardware. Thus, the second concern will be to minimize the complexity of the algorithm to reduce the gate count. Thirdly, we are presented with limited bandwidth to the hardware. We must therefore attempt to limit the size of data transfers. Finally, we are limited by the memory available to the hardware. An efficient implementation must therefore limit the size of internal data storage and avoid complexities such as dynamic memory allocation.

The key to creating a good hardware implementation of subdivision surface evaluation is in choosing an appropriate data structure to represent the topology of the mesh. The data structure must allow the hardware to easily retrieve neighboring vertices to maximize parallelism. If multiple steps are required to retrieve neighboring vertices then the algorithm will need to wait for the previous step to complete before continuing. The data structure must also be compact so as to allow communication bandwidth to the hardware and memory usage to be minimized.

In this thesis, mesh geometry will be represented by a simple list of vertices with associated coordinate values and possibly normal and/or texture coordinate information. The interesting part of mesh representation will be the data structure used to represent the mesh topology.

The algorithms presented in detail in this thesis are for Catmull-Clark subdivision of quadrilateral meshes. In the conclusions, a reasonably obvious extension of these ideas for Loop subdivision of triangular surfaces will be described.

Chapter 2 of this thesis will give the necessary background for understanding the rest of the document. The relevant details of the two subdivision schemes of interest, Catmull-Clark and Loop will be explained. The only known work on hardware evaluation of subdivision surfaces will then be described. A technique for analytic evaluation of subdivision surfaces will also be described in the background chapter so that the potential for implementing analytic evaluation in hardware can be compared to the new approaches to iterative subdivision presented in this thesis.

Chapter 3 will analyse data representations for mesh topology. A new representation, winged quadstrips, that are well suited to subdivision surface evaluation will be introduced. This representation will be compact and yet contain sufficient adjacency information for efficient subdivision.

Chapter 4 will introduce an algorithm for subdividing the new winged quadstrip data structure. This algorithm is designed with a hardware implementation in mind. Chapter 5 describes briefly how the new algorithm might be implemented in hardware.

Finally, the benefits of the winged quadstrip data structure and subdivision algorithm will be analysed in Chapter 6 and conclusions and suggestions for future work will be made in Chapter 7.

Chapter 2

Background

2.1 Subdivision Surfaces

Subdivision surfaces are a generalization of traditional spline surface representations. Splines are defined by a set of control vertices that are weighted by corresponding basis functions to parametrically define curves and surfaces. The control vertices, when connected, define a control hull. A process known as knot insertion inserts more control vertices into a control mesh while defining the same surface. Iterative knot insertion causes the control hull to converge towards the surface. Subdivision is a process similar to knot insertion on a regular mesh. An approximation to the limit surface can be calculated by recursive subdivision of an initial coarse mesh. The advantage of this approach is that while spline representations tend to be limited to planar topology segments of surfaces, subdivision can be used to create a smooth surface given an arbitrary coarse input mesh.

2.1.1 Catmull-Clark Subdivision Surfaces

Catmull-Clark [CC78] subdivision is a generalization of regular knot insertion on a cubic tensor product B-spline patch. An example of Catmull-Clark subdivision on a regular mesh can be seen in Figure 2.1.

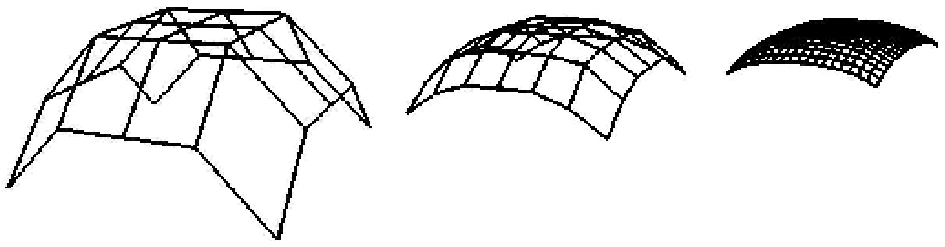

Figure 2.1: Catmull-Clark subdivision of a regular mesh

Catmull-Clark surfaces are more general than tensor product splines because they provide insertion rules for arbitrary topology surfaces as opposed to the necessarily grid-like cubic tensor product B-spline surface. A standard tensor product B-spline has a planar domain and so can only be used to model surfaces of limited topology with a single patch. For example, surfaces with sphere-like topology or genus greater than one cannot be modelled with a single B-spline patch. To represent more complex surfaces, multiple patches must be stitched together subject to constraints on boundary continuity. Satisfying these continuity constraints in an efficient and intuitive manner to give high-quality surfaces is a difficult and open problem. Subdivision eliminates this problem by defining a C^1 continuous limit surface under an arbitrary topology mesh.

Rules for Catmull-Clark subdivision can be derived by looking at the matrix representation of a tensor product B-spline surface. A patch parameterized over $u = [0,1]$ and $v = [0,1]$ with control points $P_{i,j}$ can be expressed as follows:

$$S(u,v) = \begin{bmatrix} 1 & u & u^2 & u^3 \end{bmatrix} BPB^T \begin{bmatrix} 1 \\ v \\ v^2 \\ v^3 \end{bmatrix},$$

$$B = 1/6 \begin{bmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix},$$

$$P = \begin{bmatrix} P_{0,0} & P_{0,1} & P_{0,2} & P_{0,3} \\ P_{1,0} & P_{1,1} & P_{1,2} & P_{1,3} \\ P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} \\ P_{3,0} & P_{3,1} & P_{3,2} & P_{3,3} \end{bmatrix}$$

The goal is to find an equivalent representation for the subpatch with range $u = [0, 1/2]$ and $v = [0, 1/2]$. Symmetry of B-splines allows for repetition of this construction for four regions of the range to find a new grid of 25 control vertices that define the same limit surface as the original 16 control vertices. See Figure 2.2 for a pictorial representation of a tensor product B-spline mesh represented by a finer control mesh.

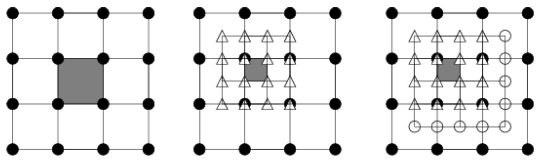

Figure 2.2: Tensor product spline subpatch

The subpatch can be expressed as follows:

$$\begin{aligned} S(u', v') &= S(\frac{u}{2}, \frac{v}{2}) \\ &= \begin{bmatrix} 1 & \frac{u}{2} & (\frac{u}{2})^2 & (\frac{u}{2})^3 \end{bmatrix} BPB^T \begin{bmatrix} 1 \\ (\frac{v}{2}) \\ (\frac{v}{2})^2 \\ (\frac{v}{2})^3 \end{bmatrix} \end{aligned}$$

$$= \begin{bmatrix} 1 & u & u^2 & u^3 \end{bmatrix} B \left(B^{-1} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{4} & 0 \\ 0 & 0 & 0 & \frac{1}{8} \end{bmatrix} B \right) P$$

$$\left(B^T \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{4} & 0 \\ 0 & 0 & 0 & \frac{1}{8} \end{bmatrix}^T B^{-1T} \right) B^T \begin{bmatrix} 1 \\ v \\ v^2 \\ v^3 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & u & u^2 & u^3 \end{bmatrix} BP'B^T \begin{bmatrix} 1 \\ v \\ v^2 \\ v^3 \end{bmatrix}$$

Solving for P' gives

$$\begin{aligned}
P'_{0,0} &= \frac{P_{0,0} + P_{1,0} + P_{0,1} + P_{1,1}}{4} \\
P'_{0,1} &= \frac{P_{0,0} + P_{1,0} + 6(P_{0,1} + P_{1,1}) + P_{0,2} + P_{1,2}}{16} \\
P'_{0,2} &= \frac{P_{0,1} + P_{1,1} + P_{0,2} + P_{1,2}}{4} \\
P'_{0,3} &= \frac{P_{0,1} + P_{1,1} + 6(P_{0,2} + P_{1,2}) + P_{0,3} + P_{1,3}}{16} \\
P'_{1,0} &= \frac{P_{0,0} + P_{0,1} + 6(P_{1,0} + P_{1,1}) + P_{2,0} + P_{2,1}}{16} \\
P'_{1,1} &= \frac{P_{0,0} + 6P_{1,0} + P_{2,0} + 6(P_{0,1} + 6P_{1,1} + P_{2,1}) + P_{0,2} + 6P_{1,2} + P_{2,2}}{64} \\
P'_{1,2} &= \frac{P_{0,1} + P_{0,2} + 6(P_{1,1} + P_{1,2}) + P_{2,1} + P_{2,2}}{16} \\
P'_{1,3} &= \frac{P_{0,1} + 6P_{1,1} + P_{2,1} + 6(P_{0,2} + 6P_{1,2} + P_{2,2}) + P_{0,3} + 6P_{1,3} + P_{2,3}}{64} \\
P'_{2,0} &= \frac{P_{1,0} + P_{2,0} + P_{1,1} + P_{2,1}}{4} \\
P'_{2,1} &= \frac{P_{1,0} + P_{2,0} + 6(P_{1,1} + P_{2,1}) + P_{1,2} + P_{2,2}}{16}
\end{aligned}$$

$$\begin{aligned}
P'_{2,2} &= \frac{P_{1,1} + P_{2,1} + P_{1,2} + P_{2,2}}{4} \\
P'_{2,3} &= \frac{P_{1,1} + P_{2,1} + 6(P_{1,2} + P_{2,2}) + P_{1,3} + P_{2,3}}{16} \\
P'_{3,0} &= \frac{P_{1,0} + P_{1,1} + 6(P_{2,0} + P_{2,1}) + P_{3,0} + P_{3,1}}{16} \\
P'_{3,1} &= \frac{P_{1,0} + 6P_{2,0} + P_{3,0} + 6(P_{1,1} + 6P_{2,1} + P_{3,1}) + P_{1,2} + 6P_{2,2} + P_{3,2}}{64} \\
P'_{3,2} &= \frac{P_{1,1} + P_{1,2} + 6(P_{2,1} + P_{2,2}) + P_{3,1} + P_{3,2}}{16} \\
P'_{3,3} &= \frac{P_{1,1} + 6P_{2,1} + P_{3,1} + 6(P_{1,2} + 6P_{2,2} + P_{3,2}) + P_{1,3} + 6P_{2,3} + P_{3,3}}{64}
\end{aligned}$$

If this construction is repeated on the four quadrants of the original domain then considerable overlap can be observed. This overlap is to be expected since the adjacent patches will be connected with C_2 continuity. As a result, a grid of twenty-five control points is created that represents the same surface as the original grid of sixteen control points.

Many conclusions can be reached by observing the regular structure of the new control points.

The observation made by Catmull and Clark was that there are three types of vertices on the subdivided mesh. They classify these vertices as either face vertices, edge vertices or replacement vertices as shown in Figure 2.3. The face vertices are an average of the four vertices on each face in the original mesh. The edge vertices are an average of two new adjacent face vertices with the original edge midpoint. The replacement vertices are the most complicated and are a weighted average of all vertices from all adjacent faces.

Notice in Figure 2.2 that columns of vertices in the subdivided mesh can be classified as one of two types. Face columns are composed entirely of face vertices and edge vertices. Vertex columns are composed entirely of edge and replacement vertices. The subdivided mesh is composed of alternating face and vertex columns. This structure will be used in the hardware subdivision algorithm.

Catmull and Clark used their classification of subdivided vertices to create a generalization for arbitrary topology meshes. An example of subdivision of an arbitrary mesh can be seen in Figure 2.4.

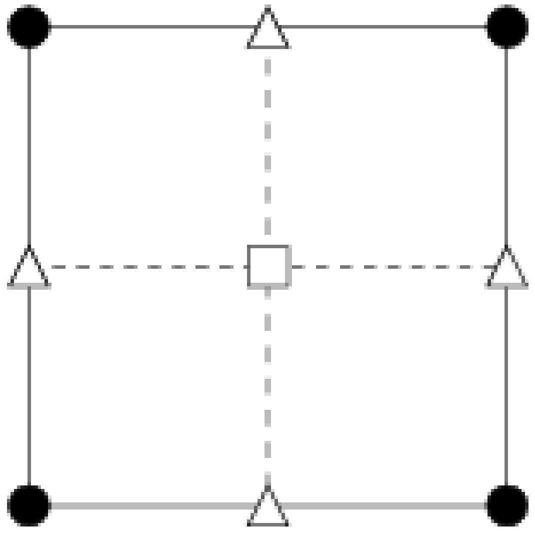

Figure 2.3: A single face subdivided using Catmull-Clark subdivision. Replacement vertices are represented with circles, edge vertices with triangles and the face vertex with a square.

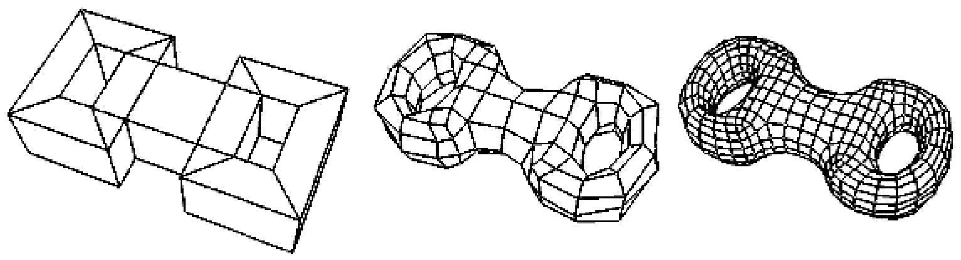

Figure 2.4: Catmull-Clark subdivision of a mesh with irregular faces and vertices

The number of vertices adjacent to a given vertex in a mesh is known as the *degree* or *valence* of that vertex. Similarly, the number of vertices surrounding a given face is known as the *degree* of that face. In a regular quadrilateral mesh, all vertices and faces have degree four. In a more general mesh, vertices and faces might have degree other than four.

To subdivide such a general mesh, new vertices are inserted on each face, edge and vertex similar to the regular case. New face vertices are inserted at the average of the surrounding vertices (possibly more than four.) New edge vertices are an average of the adjacent new face vertices with the edge midpoint, exactly as before. The rules for creating replacement vertices for vertices with degree greater than four are slightly more complicated and are designed to ensure that repeated subdivision will converge towards a smooth limit surface.

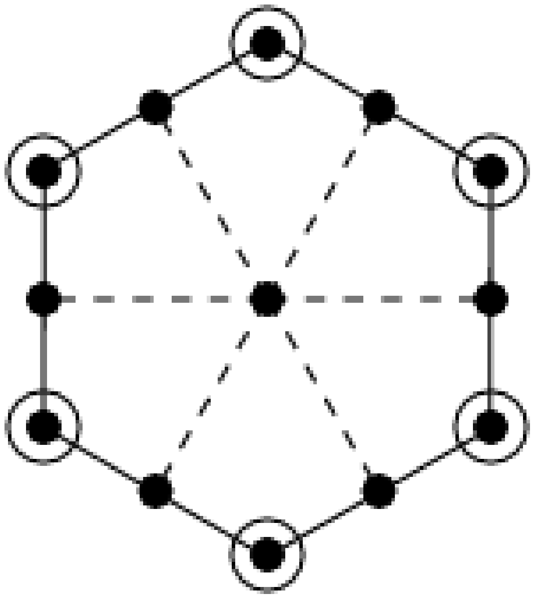

Figure 2.5: Catmull-Clark subdivision of an irregular face

On a quadrilateral mesh, it can be seen that each face is replaced by four faces after one subdivision (see Figure 2.2). Using the generalization for arbitrary topology meshes, each face is replaced by a number of faces equal to the degree of the face on the original mesh. Note that all new faces will connect a face vertex, an edge vertex, a replacement vertex and another edge vertex.

Thus, all new faces will have degree four as demonstrated in Figure 2.5. Algorithms presented later in this thesis will assume that all faces have degree four. This assumption is reasonable, since one level of subdivision on the coarsest mesh can be performed in software to reduce a mesh with irregular faces to a mesh with all faces having degree four.

Vertices with degree other than four are referred to as *extraordinary vertices*. Note that all new edge vertices will have degree four since they are connected to two face vertices and two replacement vertices. If we assume all faces are degree four then all face vertices will have degree four since they will be connected to four surrounding edge vertices. Replacement vertices will have the same degree as the vertices they are replacing. Thus, the number of extraordinary vertices in a mesh will remain fixed during subdivision. This fact is key for creation and analysis of efficient subdivision algorithms.

2.1.2 Loop's Scheme

Loop's subdivision scheme [Loo87] is a generalization of quartic Bezier patches. It is a simple scheme with only two rules for creating new vertices and a minimal subdivision kernel (i.e., the position of a vertex on the subdivided mesh is only influenced by immediate neighbors on the original mesh.) Loop's scheme operates on triangular meshes and converges towards a mesh with all vertices having degree six.

Loop's scheme is applied by subdividing each triangle of the original mesh into four new triangles. As seen in Figure 2.6 a vertex is inserted on each edge and to replace each vertex.

The new edge vertex, E, at the center of Figure 2.6 is defined by the vertices from the previous subdivision level by

$$E = (A + 3B + 3C + D)/8. \tag{2.1}$$

If the neighbors of a new vertex V at the previous subdivision level are defined as N_1, ..., N_d

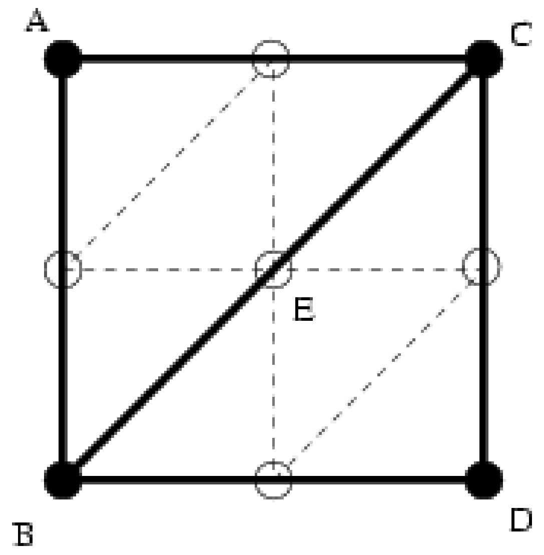

Figure 2.6: Loop subdivision

then the replacement vertex V' at the next level is defined by

$$V' = (\alpha(d)V + \sum_{i=1}^{i=d} N_i)/(d + \alpha(d)). \tag{2.2}$$

Here α is a function of vertex degree designed to give smooth limit surfaces

$$\alpha(d) = \frac{d(1-\beta(d))}{\beta(d)}, \beta(d) = \frac{5}{8} - \frac{(3 + 2 * cos(\frac{2\Pi}{d}))^2}{64}. \tag{2.3}$$

Regular vertices in Loop's subdivision scheme have degree six and regular faces have degree three. Like Catmull-Clark subdivision, all new vertices are regular and so extraordinary vertices are fixed points during iterative subdivision.

Loop's scheme is popular since it has only two simple rules and operates on triangle meshes. Most rendering hardware is designed around triangle rendering and so triangles are a desirable primitive. The disadvantage of Loop's scheme is that the quartic triangular B-splines upon which they are based are less common in surface modeling than tensor product B-splines. Another problem with Loop's scheme is that it is more difficult to create efficient array based structures to store meshes. The grid structure of Catmull-Clark surfaces can be exploited easily for efficient hardware implementation. Section 2.2 describes a technique for array based storage of triangular meshes that will be revisited in the conclusions for extending the techniques presented in this paper to Loop's scheme.

2.2 Segal and Pulli's Hardware Scheme

Segal and Pulli have proposed a scheme for fast evaluation of Loop surfaces [PS96].

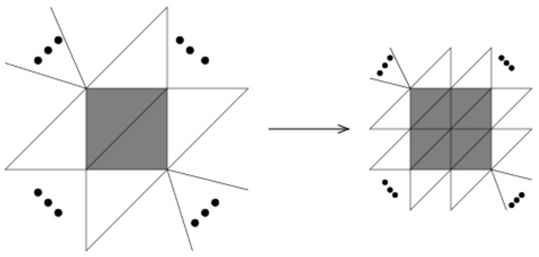

Figure 2.7: Efficient Loop subdivision by Pulli and Segal

Their scheme processes pairs of triangles in an array data structure as depicted in Figure 2.7. Some extra storage and processing is required for the potential extraordinary vertices located at the corners. Storing the vertices at all levels in this compact representation is desirable for hardware evaluation.

The most significant contribution of Pulli and Segal is their sliding window algorithm for subdivision of these arrays of triangles. To subdivide in the regular case, columns of vertices at an intermediate level of subdivision j can be classified into one of two cases. In the first case, vertices are updated and vertical edges are split. This subdivision only requires vertices from three adjacent columns at the $(j-1)$th level. In the second case, horizontal and diagonal edges are split. This subdivision only requires vertices from two columns at the $(j-1)$th level. This local control allows the arrays to be filled in from left to right with a sliding window approach. After the ith column has been calculated at the jth level, column $(i/2-1)$th at the $(j-1)$th level becomes unnecessary for further computation. It can be seen that only three columns at any level need to be stored in memory. This small constant storage is ideal for a hardware implementation.

Pulli and Segal suggest a recursive algorithm for evaluation using the sliding window approach. They note that this algorithm could easily be implemented iteratively if the number of subdivision levels is known.

A drawback of Pulli and Segal's method is that only two triangles are processed in each hardware evaluation cycle. The vertices from these triangles must be transmitted to the hardware again when adjacent pairs of triangles are subdivided. It would be desirable to process a strip of triangles and reduce the amount of overhead required in repeated transmission of vertices.

This thesis extends Pulli and Segal's method to process strips of quadrilaterals with a Catmull-Clark subdivision algorithm. The technique could also be used to process strips of triangles using Loop subdivision.

2.3 Analytic Evaluation

Most subdivision surface schemes generalize an analytic surface construction scheme to create limit surfaces for arbitrary input meshes. Two common examples have been discussed above. The Catmull-Clark scheme is a generalization of tensor product B-splines and Loop's scheme is a generalization of quartic triangular B-splines. The problem with subdivision techniques is that they must be applied iteratively and do not have obvious methods for exact evaluation of points on the limit surface. So, while subdivision provides a benefit in terms of generalization, it detracts from the precision and possibly speed of evaluation.

It is therefore useful to create an analytic representation of subdivision surfaces. Jos Stam published such a technique for Catmull-Clark surfaces [Sta98]. Before initiating his technique, it is necessary to perform two levels of subdivision on an arbitrary mesh. The first level ensures that all faces are quadrilaterals. The second level ensures that no face is adjacent to more than one extraordinary vertex. Once the mesh is in this state. it can be broken up into segments that can be analytically represented. Each segment can be thought of as one face from the initial mesh. After l levels of subdivision, each face will be subdivided into 2^l faces. Stam's work analyses the convergence of these faces to a limit surface as l approaches infinity.

Since the input mesh to Stam's technique has a maximum of one extraordinary vertex per face, the analysis is greatly simplified. The limit surface under a given face is defined by the vertices from the surrounding faces. A picture of a single (shaded) face with an arbitrary degree extraordinary vertex can be seen in Figure 2.8. The vertex order in this figure is the order used by Stam to simplify the subdivision matrices.

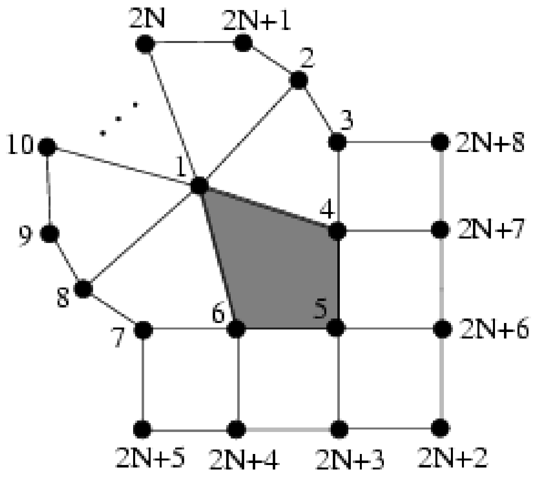

Figure 2.8: Input patch to Stam's analysis

Notice that the number of vertices in the quadrilateral fan around the extraordinary vertex is twice the degree. That is, each additional adjacent face is described by two vertices, one adjacent and one opposite.

For recursive subdivision of the central face it will be necessary to compute the four faces that result from subdividing the central face plus all faces immediately adjacent to these four new faces. Subdividing the mesh shown in Figure 2.8 results in a similar mesh with an extra layer of vertices on the regular portion of the mesh. See Figure 2.9 for the form of the subdivided mesh. Note that only vertices that are required for continued subdivision of the shaded faces are represented. The vertices are numbered to identify the region that has the same structure as in the original mesh segment. This ordering allows for recursive subdivision.

The canonical form for a mesh segment will be a single face with at most one extraordinary

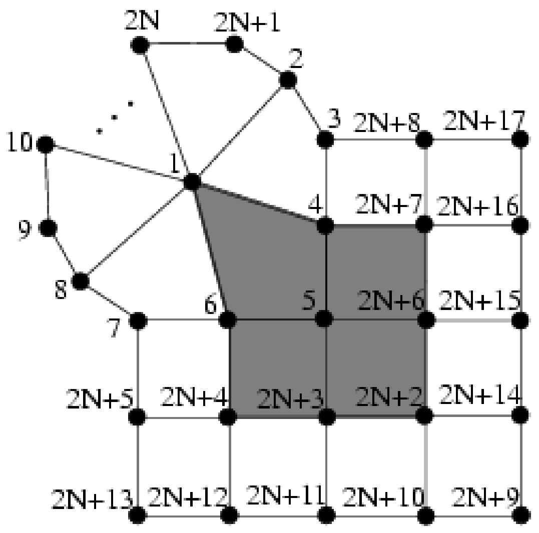

Figure 2.9: Input patch to Stam's analysis after one subdivision

vertex plus all immediately adjacent faces. After one level of subdivision, there are four mesh segments in the canonical form as depicted in Figure 2.10. That is, four patches that consist of a central face with at most one extraordinary vertex and all immediately adjacent faces.

Three of the patches in Figure 2.10 have all degree four vertices and will be called regular patches. The remaining patch has a potentially extraordinary vertex with the same degree as the extraordinary vertex on the original patch. The three regular patches can be analytically evaluated since they are equivalent to tensor product B-splines and therefore have well defined basis functions to weight the sixteen control vertices as depicted in Figure 2.10. For the irregular patch, the process can be repeated to produce four more patches, three of which can be evaluated analytically and one which can be evaluated using recursion. Stam's work provides a means for evaluation at an arbitrary parameter value by using the above procedure to find a regular patch that can be evaluated to give the surface point.

Mathematically, if the vertices in Figure 2.8 are stored in an array

$$C_0 = [v_1 ... v_{2N+8}] \tag{2.4}$$

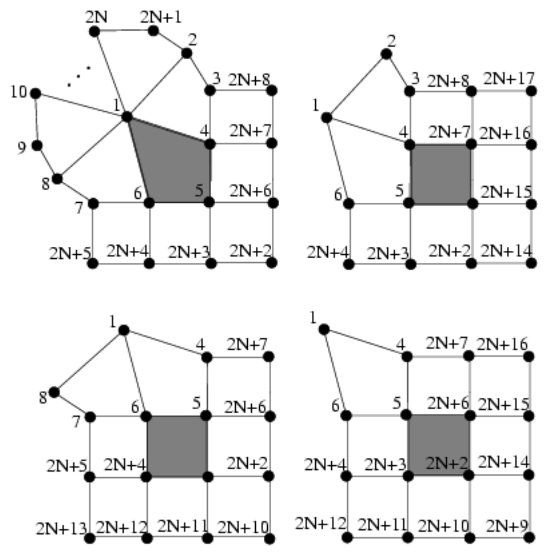

Figure 2.10: Subpatches after one level of subdivision

then a matrix A can be used to subdivide to form the irregular patch with the same connectivity at the next level:

$$C_n = AC_{n-1}. \tag{2.5}$$

The extra layer of vertices in Figure 2.9 can be generated with an extended matrix $\overline{A}$:

$$\overline{C_n} = \overline{A}C_{n-1}. \tag{2.6}$$

$\overline{C_n}$ can be generated from C_0 using this construction:

$$\overline{C_n} = \overline{A}A^{n-1}C_0. \tag{2.7}$$

To evaluate a regular portion of the mesh, a picking matrix is used to select sixteen control vertices from $\overline{C_n}$. Since there are three regular portions of the mesh, three picking matrices, P_1, P_2 and P_3, are required. Once the desired vertices are picked out of the mesh for a subpatch k at subdivision level n, a vector of B-spline basis functions can be used to evaluate the mesh as follows:

$$s_{(k,n)}(u,v) = (P_k\overline{A}A^{n-1}C_0)^T b(u,v). \tag{2.8}$$

Iteratively calculating A^{n-1} is expensive. A more efficient approach is to decompose A into eigenvalues and eigenvectors. If X is the matrix of eigenvectors of A and Λ is the diagonal matrix of eigenvalues then Equation 2.8 can be rewritten as

$$\begin{aligned} s_{(k,n)}(u,v) &= (P_k\overline{A}X\Lambda^{n-1}X^{-1}C_0)^T b(u,v) \\ &= (X^{-1}C_0)\Lambda^{n-1}(P_k\overline{A}X)^T b(u,v). \end{aligned}$$

Evaluation using Equation 2.9 can be done in three steps. First, project the control points into the eigenspace. Second, weight each of the control points according to the eigenvalues. Third, evaluate the transformed basis functions using these new control points.

To perform this evaluation, considerable storage is required. For each allowable vertex degree, the inverse of the eigenvector matrix, the eigenvalues and the transformed basis functions must be stored. From Figure 2.8 it can be seen that there are $k = 2N + 8$ vertices. The eigenvector matrix is square with dimensions equal to the number of vertices and so contains k^2 entries. The k corresponding eigenvalues must also be stored. There are k transformed basis functions each with sixteen components. Thus to evaluate patches with degree between three and N, the number of scalar values that must be stored is

$$\begin{aligned} & \sum_{d=3}^{N}((2d+8)^2 + (2d+8) + 16(2d+8)) \\ = & \sum_{d=3}^{N} 4d^2 + 67d + 200 \\ = & \frac{4N^3}{3} + \frac{71N^2}{2} + \frac{205N}{6} - 21. \end{aligned}$$

If the maximum vertex degree is restricted to ten then storage must be reserved for over five thousand values. If four byte integers are used then the required storage would be over twenty kilobytes. This storage requirement is worthy of consideration in the context of static storage on a specific hardware device. An algorithm requiring less memory would be more practical.

Any algorithm developed to perform subdivision should be compared to this analytic technique. This comparison will be performed in the analysis and conclusion chapters of this thesis.

Chapter 3

Mesh Topology Representation

In traditional graphics hardware, it is only necessary to specify how vertices are connected to form faces. This information is sufficient to render any number of faces. However, to perform subdivision, more connectivity information is required since vertices from surrounding faces must be used in the computation of vertices in the subdivided face.

This chapter will discuss and analyse traditional mesh storage techniques and extend some of these techniques to specify the additional connectivity information required for subdivision.

3.1 Mesh Analysis Tools

To analyse the space efficiency of mesh data structures, we must have some relationship between the number of vertices, V, the number of edges, E, and the number of faces, P, in a given mesh. This section will derive some results from standard planar graph theory to simplify later analysis. Warren developed a similar set of results in a course presentation at the SIGGRAPH '98 conference [War98].

The following variation on Euler's theorem for manifold meshes with boundaries will be used in our analysis.

$$V + P - E = 2 - 2G + B \tag{3.1}$$

In this equation, B, represents the number of boundaries of the mesh, i.e., the number of maximal sets of connected edges which are adjacent to only one face. G represents the genus of the surface. It is assumed that each individual polygon has one boundary (no holes).

In a closed mesh, summing all of the edges around each face will count each edge twice. Thus, the number of edges E can be related to the number of polygons P and the average number of vertices per polygon V_p by

$$E = PV_p/2. \tag{3.2}$$

Using a similar argument, summing all of the polygons around each vertex will count each edge twice for a closed mesh. Thus, the average number of polygons adjacent to each vertex P_v satisfies the relation

$$E = VP_v/2. \tag{3.3}$$

Substituting this value for E into Equation 3.1 gives

$$\begin{aligned} V + P - VP_v/2 &= 2 - 2G + B \\ V + VP_v/V_p - VP_v/2 &= 2 - 2G + B \\ 1/P_v + 1/V_p - 1/2 &= (2 - 2G + B)/(VP_v). \end{aligned}$$

The right hand side of this final equation approaches zero for large V and so we can use the following equation as an approximation for large meshes:

$$1/P_v + 1/V_p = 1/2. \tag{3.4}$$

Thus, for quadrilateral meshes, the average vertex degree is 4 and for triangular meshes the average vertex degree is 6. From Equation 3.3 we can also see that the expected number of edges is $2V$ for quad meshes and $3V$ for triangular meshes.

3.2 Winged Edge Data Structure

Perhaps the most common data structure for representing mesh topology is the winged edge data structure. This structure includes a list of faces, a list of vertices and a list of edges. Each face contains a pointer to one of its surrounding edges. Each vertex contains a pointer to one of its incident edges. The majority of the connectivity information is stored with the edges. Each edge contains a pointer to the next and previous edges on both adjacent faces. A similar approach represents each non-boundary edge as two half edges with links to its the next and previous edges on one adjacent face and a link to the symmetrical edge (if it is not a boundary.)

The winged edge data structure makes it easy to extract connectivity information but is inappropriate for hardware and perhaps for subdivision in general due to the immense amount of storage and pointer-chasing required.

Given eight pointers for each edge (two for vertices, two for faces, and four to edges) the number of pointers in the data structure will be approximately $24V$ for triangular meshes and $16V$ for quad meshes. This storage requirement can be reduced considerably with less bloated data structures as described in the following sections.

3.3 Face Based Data Structures

More recently, face based data structures have become popular for implementing subdivision. A list of faces is stored along with pointers to their adjacent faces, edges and vertices. Since each vertex on a face corresponds to an edge and an adjacent face, there are three pointers per vertex per face in our data structure. Thus, an estimate of the number of pointers in such a data structure is

$$P(3V_p) = 3P_vV. \tag{3.5}$$

For triangular meshes we have shown that P_v is approximately six on average and so the number of pointers required according to Equation 3.5 is approximately $18V$. Similarly, for quad meshes the average number of pointers is only $12V$. Thus, this approach generally requires significantly less storage and tends to be more convenient for subdivision algorithms when compared to a winged edge structure. The immediate neighbors of a face that are required for subdivision can be easily accessed with a face-based structure. Still, the storage requirement for simple face based structures can be reduced.

3.4 Quadstrip Structure

The storage requirements of the face based approach can be further reduced by defining some of the connectivity information implicitly by the ordering of vertices. OpenGL provides a scheme where faces can be transmitted to the hardware as strips of quadrilaterals or triangles. Thus, one edge (i.e., two vertices) of the current face is implicitly defined by the previous face. This optimization can reduce the size of the data by two thirds for triangular meshes or a half for quad meshes (ignoring end points). For rendering, only the vertices of each face are required. For subdivision, we can also deduce face adjacency information from a strip representation.

Unfortunately, much redundancy is still required since two adjacent strips must repeat a row of vertices. This redundancy is considered acceptable in OpenGL. To attempt to reduce the representation further would place too many restrictions upon the mesh and make determination of shared face vertices too complicated for hardware implementation.

It will be necessary to accept a similar degree of redundancy in the subdivision representation for the same reason. Redundancy can simplify the location of adjacent vertices for rendering faces or for calculating subdivided vertices.

3.5 Winged Quadstrips

A quadstrip can be represented using two vertices for the head of the strip and subsequent pairs of vertices for each face in the strip. To record adjacency information, each face in the strip must include a pointer to the faces above and below (those before and after are defined implicitly.) Also, the head of the strip must include a pointer to the face before the strip. Finally, there must be a pointer to the face after the end of the strip. Thus, our winged quadstrip data structure has the form depicted in Figure 3.1.

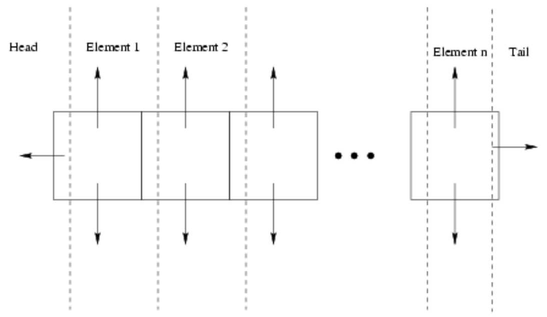

Figure 3.1: A winged quadstrip

This representation reduces the storage of connectivity information by nearly half over a standard face based representation. At each level of subdivision, each strip will be replaced by two parallel strips, each twice the length of the original strip as seen in Figure 3.2. As a result of this exponential growth in quadstrip length, not only will the estimate that connectivity storage is reduced by half become more accurate, the mesh will also be in a desirable form for efficient rendering with a simple hardware implementation. Furthermore, in a software implementation, memory access will be more coherent for vertices along the strip if such vertices are stored in arrays.

Most algorithms for creating strips from polygonal models work by breaking the model up into coarse regions with regular connectivity and then converting these regular regions into long strips [ESV96b]. Our algorithm performs a similar decomposition of regular regions and thereby produces high quality (long average length) quadstrips to be rendered. An example of these strips can be seen in Figure 3.3.

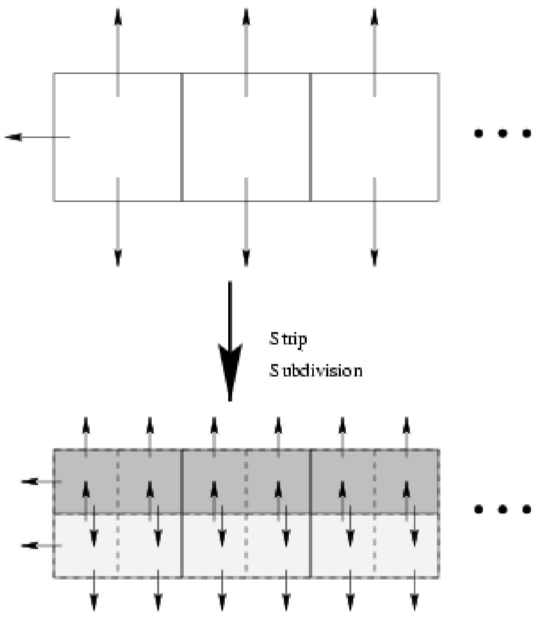

Figure 3.2: A subdivided winged quadstrip

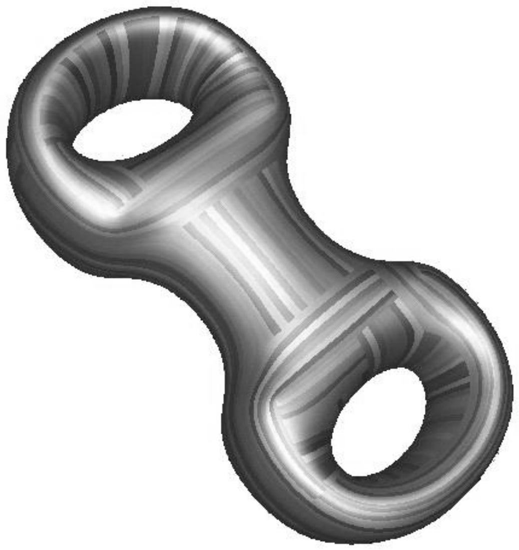

Figure 3.3: A genus two object subdivided using longitudinal strips

Chapter 4

Hardware Subdivision

For efficiency, it is desirable to process a coarse mesh in software and then send it to a graphics accelerator to be subdivided internally and rendered as a smooth surface. This approach would require considerably less memory and bandwidth when compared to subdividing the mesh in software and then transmitting the subdivided vertices. The mesh size grows exponentially with subdivision and so performing a few steps of subdivision in hardware could have a dramatic effect on bandwidth and memory usage. For example, five subdivisions will replace each face with 1024 faces. It is undesirable to store, manipulate or transmit this bulky structure.

Due to the number of polygons required to represent the limit surface, it is also undesirable to store the entire mesh in hardware after subdivision. To accommodate this restriction, an online algorithm is proposed that will process a minimal set of vertices to produce a single strip on the final mesh. More vertices will be read as required, replacing no longer relevant vertices and iteratively rendering strips from the final mesh.

The algorithm used will be similar to the sliding window algorithm proposed by Pulli and Segal for subdividing triangular meshes using Loop's scheme. Where Pulli and Segal's method subdivides pairs of triangles with Loop's scheme, the method presented here will subdivide strips of quadrilaterals with Catmull-Clark subdivision.

The only requirements for meshes subdivided with the proposed algorithm will be that all faces have degree four. This requirement can be easily satisfied using one step of Catmull-Clark subdivision in software. One step of subdivision at the coarsest level is a quick operation relative to the exponentially larger number of operations required at lower levels and so it is reasonable to perform this step in software. Quadrilaterals are also a common modeling primitive and many meshes may already be in the correct format. Note that analytic evaluation technique discussed in Section 2.3 requires two steps of subdivision to be performed on an arbitrary initial mesh and so the quadstrip subdivision algorithm compares favorably in this regard.

This chapter will introduce a hardware-compatible subdivision algorithm that renders strips of quadrilaterals with adjacency information in an online manner using constant storage given a fixed recursion depth. A simplified version of the algorithm that only handles regular grids will be described first followed by the extension to the algorithm to deal with extraordinary vertices.

4.1 Hardware Subdivision in the Regular Case

For the purposes of the strip subdivision algorithm, a strip will be represented as a series of columns of vertices as seen in Figure 4.1. Extensions to arbitrary degree vertices will be added later in Section 4.2. For now, assume that all vertices are regular (degree four) and thus each column of vertices at the coarsest level contains four vertices.

Columns of vertices at lower levels of subdivision can be classified as either face columns or vertex columns. Face columns consist of new face vertices (the average of four surrounding vertices) and "horizontal" edge vertices (the average of the two new face vertices with the edge midpoint.) Vertex columns consist of new replacement vertices (a weighted average of vertices from adjacent faces) and new "vertical" edge vertices.

Note that face columns depend on two columns from the previous level while vertex columns depend on three columns from the previous level. Thus, three columns on the coarsest level can be used to calculate three columns on the next level (two face columns and one vertex column.) This process can be repeated to the desired number of subdivision levels to initialize the algorithm. A pictorial representation of the initialization step can be seen in Figure 4.2.

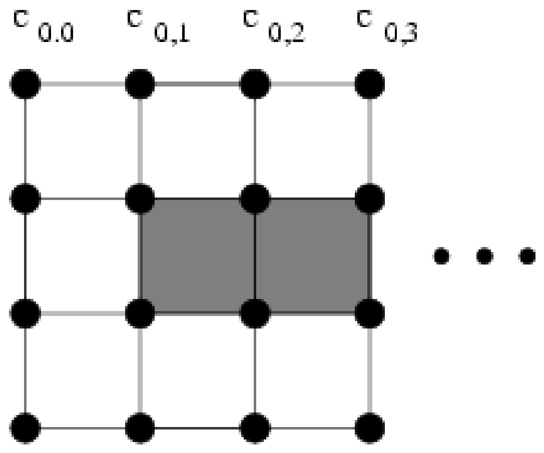

Figure 4.1: Regular quadstrip columns

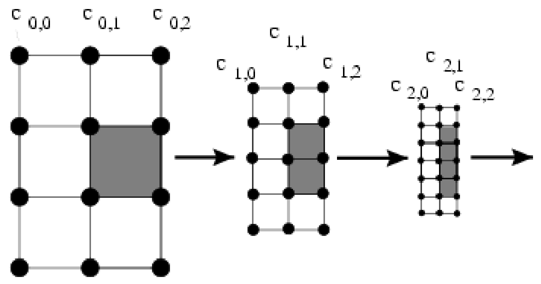

Figure 4.2: Subdivision initialization

After the initialization step, three columns of vertices are stored for each level of subdivision. At this point, one column at the finest level can be rendered and the first column at each level of subdivision is no longer required. To continue the algorithm, a new column can be read in at the coarsest level. The three latest columns at the coarsest level can then be used to calculate two new columns at the next level of subdivision. It is easy to follow this pattern to create a recursive algorithm like the one presented in Figure 4.3. This algorithm only needs to store three columns at each level of subdivision at any given time. Circular arrays are used for this purpose so that new columns always overwrite unneeded columns.

```
static void
recurseTree(int level, int column)
{
  if (level == maxLevel) {
    renderColumn(level, column);
  } else {
    calculateVertexColumn(level+1, (column-1)*2);
    recurseTree(level+1, (column-1)*2);

    calculateFaceColumn(level+1, (column-1)*2+1);
    recurseTree(level+1, (column-1)*2+1);
  }
}
```

Figure 4.3: A recursive algorithm for quadstrip subdivision

It is a simple exercise to convert this recursive algorithm to an iterative algorithm that can be more easily implemented in hardware. The only restriction is that we must fix the maximum recursion level in advance. This restriction is necessary regardless of the implementation due to fixed storage requirements.

4.2 Hardware Subdivision with Extraordinary Vertices

The simple algorithm described in the previous section becomes slightly more complicated when extraordinary vertices are taken into consideration. In the input quadstrip, any vertex may be extraordinary as shown in Figure 4.4. At lower levels, all new edge and face vertices will be regular.

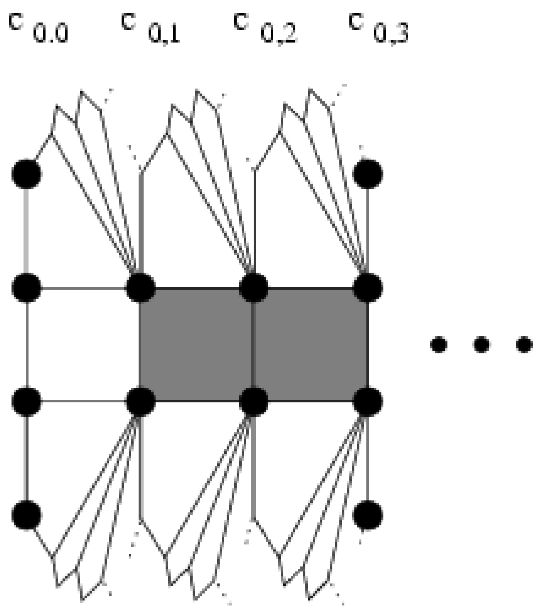

Figure 4.4: Quadstrip with extraordinary vertices (fans)

Therefore, after one level of subdivision, each column of vertices will contain at most two extraordinary vertices. It will be demonstrated that with the sliding window algorithm, three columns containing two extraordinary vertices are necessary at the coarsest level but only one column containing two extraordinary vertices are necessary for each subsequent level of subdivision. This structure allows the algorithm to handle extraordinary vertices without an unreasonable increase in storage requirements.

Extraordinary vertices will be represented in the quadstrip data structure by adding pairs of neighboring vertices for each vertex degree greater than four. These pairs of vertices will be stored in an "LNeighbor" data structure for ease of reference.

The sliding window strip subdivision algorithm can be easily modified to accommodate these new extraordinary vertices. Vertex columns with extraordinary vertices will need to do a little extra processing to create the "fan" at either end. Face columns are calculated in almost exactly the same way as before except that they might depend upon an LNeighbor vertex instead of the regular vertex from the previous level vertex column.

The remaining difficulty is with degree three vertices. Note that if a vertex has degree three, then the adjacent face column at the next level of subdivision will depend upon a vertex from the next column at the current level. This problem can be solved by temporarily filling in the absent regular array vertex with a vertex from the next column while calculating the adjacent face column. This solution complicates the algorithm and requires that an extra column be stored at the top level. Still, with this solution, we can handle all manifold meshes of quadrilaterals as desired.

The final data structure to represent the strip will contain

- A four by four array of regular vertices.
- A two by three by (MAX_DEGREE-4) array of LNeighbors.
- A two by three array of vertex degrees.

To represent the vertices at lower levels, we must add

- An array of regular vertices, 3 columns by the sum of the number of rows at each level of subdivision from 1 to MAX_SUBDIVISIONS.
- An array of LNeighbors (MAX_DEGREE-4) by MAX_SUBDIVISIONS by two.

Putting all of the above together, the data structure that must be maintained in hardware to render strips is as follows:

```
typedef struct QuadstripTree {
  StripState state;
  bool closed;

  int maxLevel;
  int currentColumn[MAX_SUBDIVISIONS+1];

  // Degrees of (potentially) extraordinary vertices [row][column]
  int degree[2][3];

  // Coarsest level regular columns [row][column]
  StripVertex l0regular[4][4];

  // Coarsest level extra neighbors [row][column][neighbor]
  LNeighbor l0extra[2][3][MAX_DEGREE-4];

  // Regular vertices at lower levels [column][(level-1)*3-2+(1<<level)+row]
  StripVertex regular[3][(MAX_SUBDIVISIONS)*3+(1<<(MAX_SUBDIVISIONS+1))-2];

  // Extra vertices at lower levels [row][level][neighbor]
  LNeighbor extra[2][MAX_SUBDIVISIONS][MAX_DEGREE-4];

} QuadstripTree;
```

To subdivide using the data structure above, the regular grid algorithm will require some modification.

First, when calculating subdivided vertex columns, there may be extra processing required at the ends to subdivide the "quadrilateral fan." This extra processing will be rare since all new columns will be regular.

Second, degree three vertices present a problem at the initial level. To calculate three columns at the next level, it may be necessary to use a vertex from a fourth column at the initial level. An example of this situation is shown in Figure 4.5. So, for the general algorithm, four columns are stored at the initial level. If a vertex is required from the fourth column, it is stored in the gap in the third column. Processing for subsequent levels is then the same as in the regular case. Thus, only three columns are stored for lower levels. There is a small amount of extra work required to shuffle vertices when the extraordinary column is replaced but this effort is negligible.

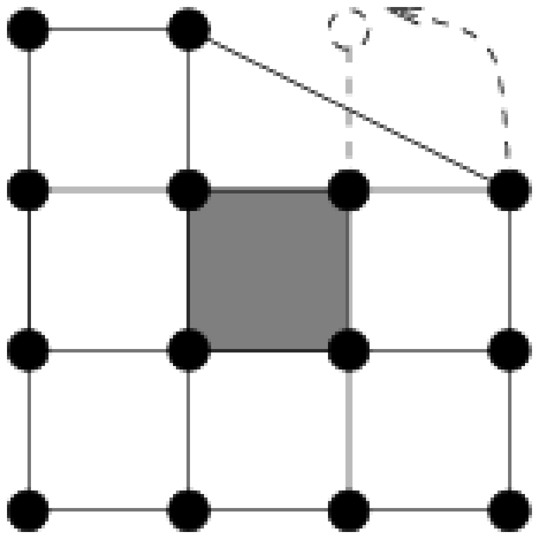

Figure 4.5: Handling degree three vertices

4.3 Boundary Conditions

The algorithm described above is an excellent tool for creating everywhere smooth surfaces from closed meshes. It remains to be resolved how to handle surfaces with cusps and boundaries.

DeRose et al. [DKT98] proposed a simple algorithm for non-uniform subdivision used in the creation of the animated short "Geri's Game." Essentially, edges are marked with a number to indicate their sharpness. Marked edges are subdivided using a special midpoint rule for a number of subdivision levels equal to their sharpness value. Thus, edges with zero sharpness value are subdivided as usual. Edges with infinite sharpness remain sharp edges or cusps in the final surface.

This simple algorithm could be easily incorporated into the quadstrip subdivision algorithm. Such an implementation would have increased hardware complexity and require a more robust API but would provide enhanced surface control. The benefit of this algorithm is that semi-sharp edges can be created.

An even simpler approach can handle boundaries and cusps while leaving the hardware algorithm exactly as described above. In traditional B-splines, boundaries and cusps can be handled by increasing the knot multiplicity of the edges. If the knot multiplicity has an integer value, this technique corresponds to repeating vertices along the edges and at cusps. So, to create an edge or cusp, the software can simply send extra strips near the boundaries.

Finally, Sederberg et al. [SZSS98] proposed an algorithm for simulating arbitrary knot multiplicity (non-integer) with subdivision surfaces. This algorithm was mathematically interesting but would require significant added complexity (particularly in a hardware algorithm) for negligible benefit.

Clearly, the simplest solution to the problem of boundaries and cusps is to repeat vertices along these edges. Future work might make use of the algorithm by DeRose et al. to provide greater sharpness control.

4.4 Calculating Normals

To properly shade polygons to give the appearance of a smooth surface, surface normals must be available. Any practical hardware evaluation scheme must therefore provide some method of calculating normals at the final level of subdivision.

The simplest solution to this problem in the case of the quadstrip subdivision algorithm discussed in this thesis is to calculate vertex normals as the average of surrounding face normals as a final step before rendering. Note that the faces may not be coplanar and so a normal defined by the cross product of the two edges adjacent to the vertex is used. This solution requires storing a fourth column of vertices in the sliding window at the finest subdivision level. It has been observed that this approach produces high quality normals, particularly when the subdivision level is high since polygons converge towards being co-planar and their normals become truly representative of the underlying surface.

Another solution would be to provide normals as input to the algorithm and to subdivide these normals using a similar method to the manner in which vertices are subdivided. There are many problems with this approach. First of all, storage requirements and computational complexity will increase dramatically. Second of all, subdividing the normals would be at least as complicated as subdividing the vertices and so the number of gates and the storage required in the system could be expected to double. This technique for subdividing extra points might be needed for texture mapping. With texture mapping, the same linear subdivision strategy used for regular vertices could be used. The difficulty with normal subdivision is that a non-linear strategy would be necessary to produce smooth looking surfaces. Implementing this non-linear subdivision would require substantially more hardware.

Thus, the recommended solution for computing normals is simply to store one extra column of vertices at the finest level and to set the normal for a given point to be the average of the surrounding face normals.

4.5 A Quadstrip API

The API for the quadstrip subdivision algorithm has been designed with ease of use in mind. Since most graphics developers are familiar with the OpenGL standard graphics API, this API will mimic its structure.

In OpenGL, quadstrips are transmitted to hardware with a glBegin statement, a sequence of vertices, and a glEnd statement.

```
glBegin(GL_QUAD_STRIP);
  glVertex(v1);
  glVertex(v2);
  glVertex(v3);
  glVertex(v4);
  glVertex(v5);
  glVertex(v6);
  ...
glEnd();
```

A picture of the strip represented by this code fragment can be seen in Figure 4.6.

In OpenGL, static structures are used to store data such as colours and normals to be associated with a vertex. In the SMASH architecture developed by Michael McCool [McC00], a stack

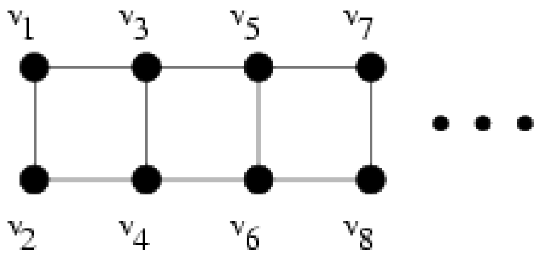

Figure 4.6: OpenGL strip format

of information to be associated with each vertex is used. The vertex call then applies all of the data on the stack to the current vertex. The quadstrip subdivision API will mimic this behaviour by pushing vertices from the surrounding fan onto the stack. Only vertices not in the strip will be pushed onto the stack for efficiency reasons. Thus, a typical sequence for transmitting a quadstrip to hardware under SMASH would be as follows:

```
smBegin(QS_QUAD_STRIP);
  smPoint(v1_n1);       // The first neighbor of v1
  ...
  smPoint(v1_nd1);     // The last neighbor of v1
  smVertex(v1);
  smPoint(v2_n1);
  ...
  smPoint(v2_nd2);
  smVertex(v2);


  ...


  smVertex(v6);
smEnd();
```

A picture of the strip with neighbors represented by this code can be seen in Figure 4.7.

To promote symmetry in the ordering, vertices on the top row will be odd and their neighbors

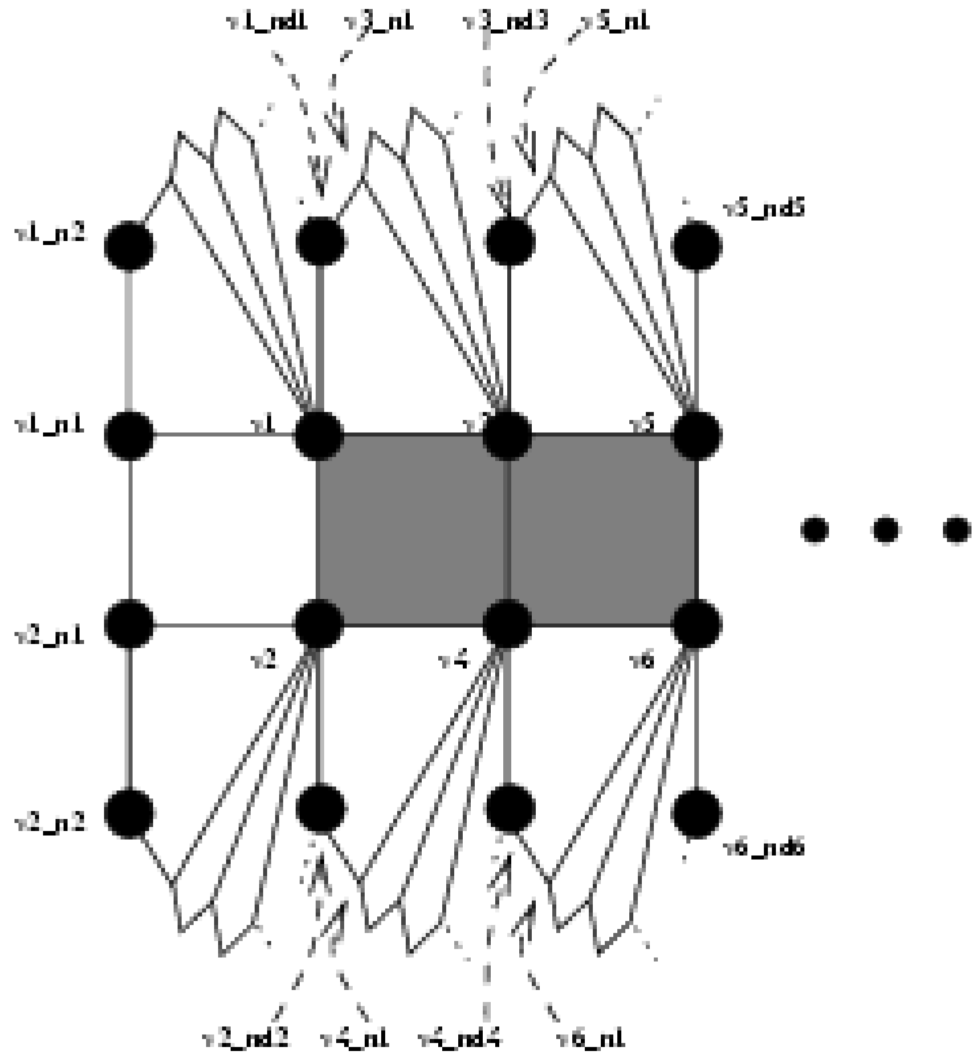

Figure 4.7: Strip API format

will be transmitted in a clockwise order, while vertices on the bottom row will be even and their neighbors will be transmitted in a counter-clockwise order. The first vertex neighbor to be transmitted will always be the first vertex adjacent to the current vertex in the order above which is not adjacent to any other strip vertex. The last vertex will be the last vertex in the sequence of surrounding fan vertices which is not adjacent to any strip vertex.

4.6 Strip Creation

Much research into geometry compression schemes is based on the creation of triangle strips. An uncompressed representation of a triangle mesh might store three vertices for each triangle. There is a substantial amount of redundancy is such a representation since most vertices are associated with multiple triangles. A better approach is to encode the mesh as a number of sets of adjacent triangles such that each set of adjacent triangles implicitly defines some of the vertices of each triangle by its association with other triangles in the set. Triangle strips are such an encoding.

An uncompressed encoding of a strip of triangles might have the form

$$[v_{1,0}, v_{1,1}, v_{1,2}, v_{2,0}, v_{2,1}, v_{2,2}, ..., v_{n,2}]$$

where each $[v_{i,0}, v_{i,1}, v_{i,2}]$ represents the vertices of a triangle in the strip. If the vertices are arranged in a strip such that $v_{i,1} = v_{i+1,0}$ and $v_{i,2} = v_{i+1,1}$ then a compressed encoding of the same strip would have the form $[v_{0,0}, v_{0,1}, v_{1,2}, v_{2,2}, ..., v_{n,2}]$. This encoding requires only $(n + 2)$ vertices while the initial encoding required $3n$ vertices.

The difficulty in encoding an arbitrary mesh using triangle strips is in effectively grouping the mesh triangles into strips. The goal is to reduce the total number of vertices that must be stored for the entire mesh. If S strips are used to encode n triangles then the storage required is $(n + 2S)$ vertices. To minimize this storage, the number of strips must be minimized. Minimizing the number of strips is clearly related to maximizing the length of each strip.

Much of the research into creating strips is concerned with *generalized triangle sequences* as opposed to triangle strips. Notice that in the strip encoding above, it was necessary to alternate between creating each successive triangle on the left edge and the right edge of the preceding triangle. In a generalized triangle sequence, an extra bit is used to encode the direction in which to create the next triangle. This enhancement allows for any path through the dual graph of the mesh that does not repeat vertices to be encoded.

Generalized triangle sequences can provide better compression but are not universally supported. In particular, generalized triangle sequences are not explicitly supported in OpenGL though they can be simulated by repeating a vertex in the encoding to create a degenerate triangle that is not rendered [ESV96b]. The generalized strips are useful for analysis since they correspond to paths in the dual graph of the mesh. It has been demonstrated in [ESV96a] that finding an optimal set of triangle sequences is an NP-complete problem.

In the best case, the dual graph might contain a *Hamiltonian* path. Such a path visits every vertex in the graph exactly once. In this case, the entire mesh can be encoded with a single generalized triangle sequence. In [VdFG99] a coarse mesh with a Hamiltonian cycle is refined while maintaining the property that there is always a Hamiltonian path through the dual graph. This approach generates triangulations with optimal strip representations.

In [ESV96a] a fine resolution triangle mesh is searched for large regular quadrilateral patches. These coarse regions are then encoded into long triangle strips.

While the literature focuses on generalized triangle sequences for triangular meshes, the algorithms used in this thesis focus on creating quadrilateral strips from quadrilateral meshes. A quadrilateral strip (or *quadstrip*) can be written $[v_{0,1}, v_{0,2}, v_{1,1}, v_{1,2}, v_{2,1}, v_{2,2}, ..., v_{n,1}, v_{n,2}]$. Here, $[v_{i-1,1}, v_{i-1,2}, v_{i,2}, v_{i,1}]$, represent mesh quadrilaterals.

The techniques in [VdFG99] and [ESV96a] for creating high quality triangle sequences by first examining a coarse resolution mesh are similar to the techniques discussed in this thesis. The quadstrips created after subdivision using the software quadstrip subdivision algorithm are long because they take advantage of structure in the initial coarse mesh to produce quadrilaterals in the final mesh.

The high quality quad strips produced by the software algorithm are not currently produced by the hardware algorithm. The software algorithm breaks up one strip into two longer strips at each level of subdivision. This structure will henceforth be referred to as *longitudinal*. The strips produced by the hardware algorithm are perpendicular to the input strip and are thus *latitudinal*. The latitudinal strips have fixed length depending upon the number of levels of subdivision.

It would be desirable to produce longitudinal strips using the hardware algorithm. If rasterization hardware capable of simultaneously processing 2^l strips, where l is the maximum number of levels of subdivision, was available, then longitudinal strips could be output by the hardware algorithm. If such parallel rasterization units are not available, then the longitudinal strips being produced in parallel could be buffered by the subdivision hardware and transmitted after termination of the strip or some threshold of length is reached. Using either of these techniques, use of the rendering hardware could be enhanced by transmitting higher quality strips.

The question still remains of how to produce the initial quadstrips on the coarse mesh before subdivision. My simple algorithm picks a random quad and steps across the mesh until a boundary or already processed face is encountered. This algorithm could be slightly improved by starting with a quad with a minimum number of neighbors and stepping in a direction that includes the adjacent quad with a minimum number of neighbors. This approach mimics a similar technique used for triangle strips in a toolkit developed by SGI and referenced in [VdFG99].

There is certainly potential for creating a better algorithm for producing high quality quad strips on the coarse level using the research discussed for triangle strips as a basis. This research is left as future work. It has been shown that when producing longitudinal strips from the subdivision algorithm, strip length doubles with each iteration. Thus, the strips passed to the rendering hardware will have long average length and so efficient use will be made of the rendering hardware.

Chapter 5

Architecture

5.1 Data Flow

The goal of a surface rendering system is to read a model from disk and efficiently render it to the screen. This process can be broken up into four phases as depicted in Figure 5.1. First, the model must be read from a high storage capacity medium like a disk into system memory. Second, the model must be transmitted as strips from system memory to subdivision hardware. Finally, the results from the subdivision hardware must be transmitted to rendering hardware.

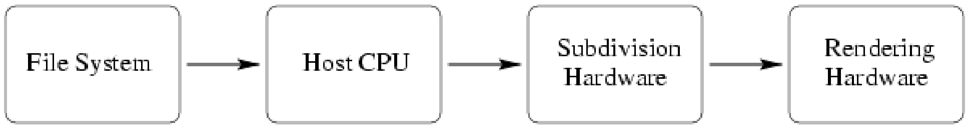

Figure 5.1: Surface rendering dataflow

Data stored on disk is typically in a format consisting of a list of vertices and a list of connectivity information. Here is an example of a box stored using SGI's Inventor [Wer94] format:

```
Coordinate3 {
  point [
  -1.0 -1.0  1.0,
     1.0 -1.0  1.0,
    -1.0  1.0  1.0,
     1.0  1.0  1.0,
    -1.0 -1.0 -1.0,
     1.0 -1.0 -1.0,
    -1.0  1.0 -1.0,
     1.0  1.0 -1.0,
    ]
  }
  IndexedFaceSet {
    materialIndex   -1
    normalIndex   -1
    textureCoordIndex   -1
    coordIndex      [
     0, 1, 3, 2, -1,
     2, 3, 7, 6, -1,
     6, 7, 5, 4, -1,
     4, 5, 1, 0, -1,
     1, 5, 7, 3, -1,
     2, 6, 4, 0, -1,
    ]
  }
```

Formats exist to store data on disk that is prearranged into strips. For efficient subdivision, it would be preferable to have the data stored on disk prearranged into strips with adjacency information. Otherwise, data will need to be read in to a robust data structure such as a half edge structure in system memory. It is then possible to process this half edge structure to produce strips with adjacency information as is required for strip subdivision. The computation cost for creating strips with adjacency information is minimal since it is performed only once on the coarse mesh.

In a realtime system, we need only be concerned about processing the data after it is already in strip format in CPU memory.

The software executing on the CPU can be broken down into two parts, an application program and an API. The application program will store the mesh and decide when it needs to be rendered. The API code will translate API calls described previously in Section 4.5 into data that will be physically transmitted via the system bus to the subdivision hardware. The API is written to be easily understood by the developers of application programs. The interface to the hardware is written more with efficiency in mind.

The data transmitted to the hardware is used to fill in the top level arrays in the subdivision tree and to trigger iterative calculation of the lower levels of subdivision. It is desirable to buffer the transmission for filling in the vertex arrays until a calculation step is required and then to send a block of vertices in a single transmission. This approach will minimize the use of system bus bandwidth which is known to be a bottleneck in rendering systems.

The hardware portion of the rendering pipeline could have a variety of formats. For a practical implementation, the subdivision hardware and the rendering hardware should be collocated on a single Application Specific Integrated Circuit (ASIC) card. For prototyping, an architecture with specialized subdivision hardware synthesised on a Field Programmable Gate Array (FPGA) card has been considered. It would then be necessary to either transmit the resulting subdivided quadstrips to a traditional graphics card or to implement a rasterization algorithm on the FPGA. This implementation is not ideal since the system bus must be used to transmit the fine resolution mesh, thus defeating one of the biggest efficiency gains of the system. However, the FPGA is an excellent prototyping tool and can be used to test feasibility and estimate gate count.

If the subdivision hardware and the rendering hardware are collocated as desired, then usage of the system bus can be minimized. The strips produced by the subdivision hardware could be efficiently transmitted to the rendering hardware without making use of the system bus.

Using this system an application program could read a quadstrip data structure with adjacency information from disk and make API calls for rendering. The API calls could then be translated into commands to the subdivision hardware, which include filling in top level arrays and initiating subdivision. The subdivision hardware can then pass the resulting fine detail quadstrips to the rendering hardware on the same card without making use of the system bus. This transmission is not only fast but also makes the system bus available for other purposes. A pictorial representation of this architecture is presented in Figure 5.2.

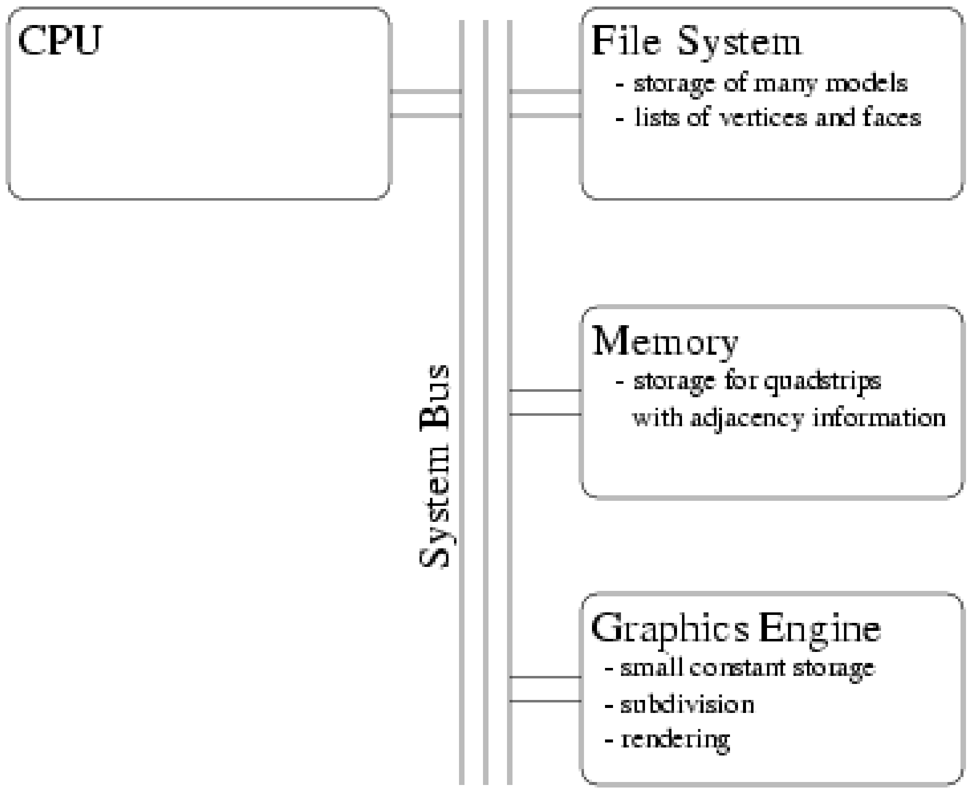

Figure 5.2: Surface rendering hardware architecture

5.2 Hardware Organization

The subdivision hardware must be designed with parallelism in mind. The two key operations are calculating face columns and vertex columns. An example implementation for calculating a face column can be seen in Figure 5.3. A clever implementation of the additions and divisions using hardware similar to a carry save adder but modified to consider the shifts could perform the calculation of an entire face column in a single clock cycle. Specialized hardware clock cycles are generally slower than CPU clock cycles but the fine-grained parallelism clearly demonstrated by Figure 5.3 will still allow the hardware to be significantly faster.

Calculation of vertex columns is more complicated since two of the vertices might be extraordinary. The extra neighbors could be stored in additional registers as depicted in Figure 5.4.

There is too much detail involved in the calculation of vertex columns to draw a simple coherent wiring diagram but the feasibility of such an implementation should be clear. An example of hardware that might be used to compute a single regular replacement vertex can be seen in Figure 5.5. If a complete hardware implementation were to be designed in practice, it could be created with a hardware description language such as VHDL.

The calculation of vertex columns requires computation of edge vertices and replacement vertices. Intermediate results used for both of these computations are the surrounding face vertices. These two columns of face vertices plus a fan of face vertices at the top and bottom of the column are calculated in the same cycle. The edge vertices can then be calculated using the face vertices as demonstrated in Figure 5.3. Calculation of replacement vertices can be expressed as

$$\frac{\sum F_i}{d^2} + \frac{\sum A_i}{d^2} + \frac{(d-2)S}{d}, \tag{5.1}$$

where d is the degree, F_i are the surrounding face vertices, A_i are the surrounding adjacent vertices and S is the vertex to be replaced.

The divisions for the calculation of replacement vertices will be divisions by four or sixteen in the regular case and so can be implemented with shift registers. For extraordinary vertices, the divisions should be implemented as a multiplication by a pre-calculated reciprocal. Since the maximum vertex degree is fixed, storage of these reciprocals will only require a small look up table (LUT).

A complete hardware implementation has not been provided as a part of this thesis but the feasibility should be clear. It has also been demonstrated that significant gains in terms of parallelism could be achieved by calculating entire columns of subdivided vertices in a small

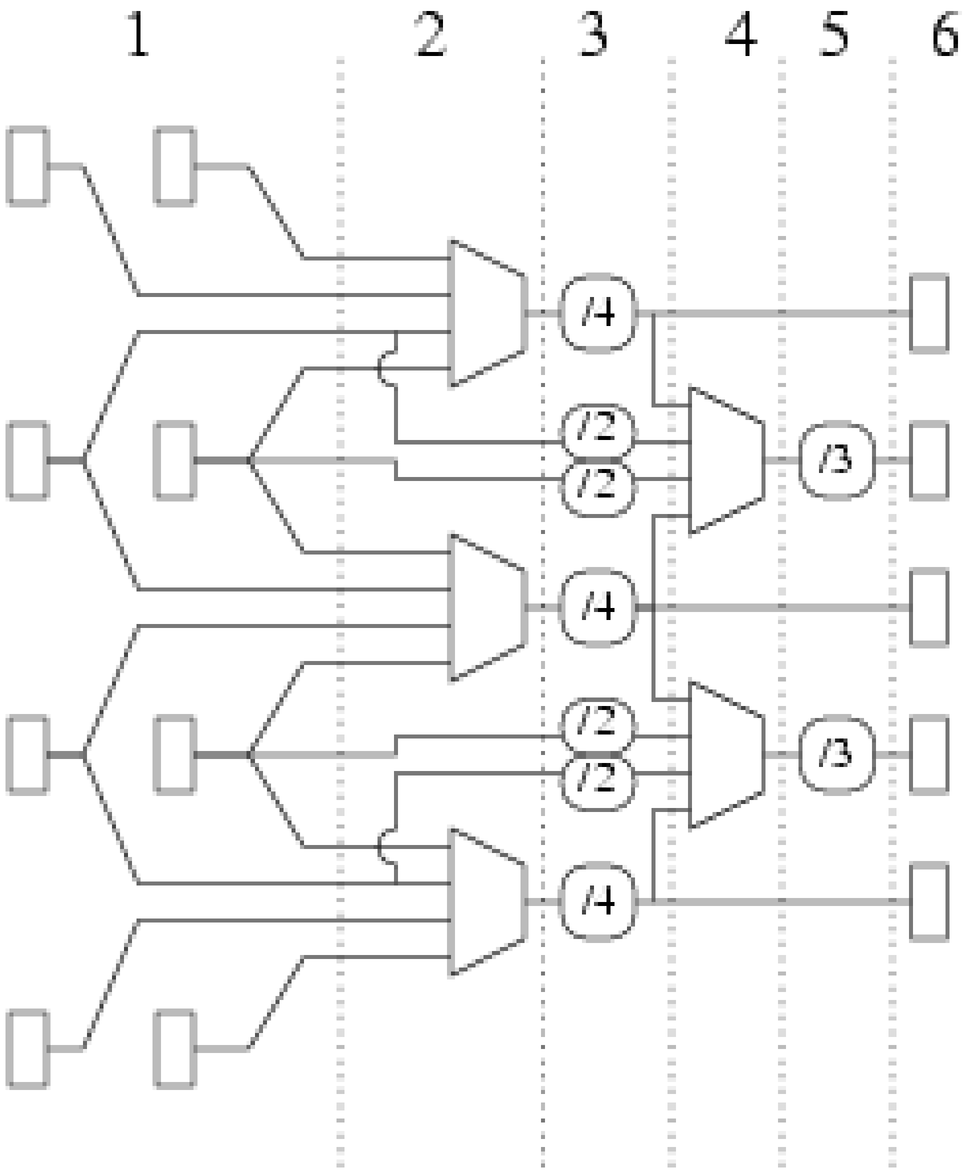

Figure 5.3: Gates for calculating a face column. 1) Registers for storing the last two columns of the previous level. 2) Adders for calculating face vertices. 3) Shift registers for division by two or four. 4) Adders for calculating edge vertices. 5) Division by three using hardwired multiplication by 1/3 6) Registers for storing the face column at the next level.

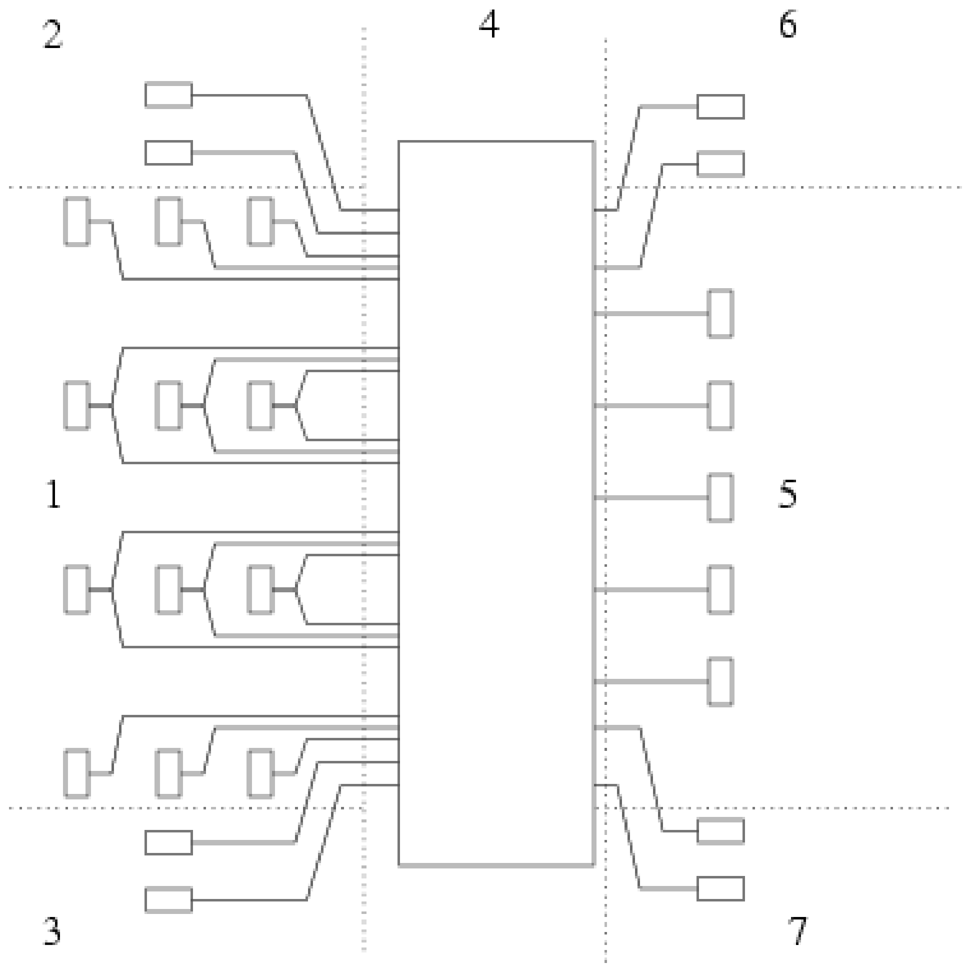

Figure 5.4: Hardware design for calculating a vertex column. 1) Registers for storing the last three columns of the previous level. 2,3) Arrays for extra neighbors. 4) Processing unit. 5) Registers for storing the face column at the next level. 6,7) Arrays for extra neighbors.

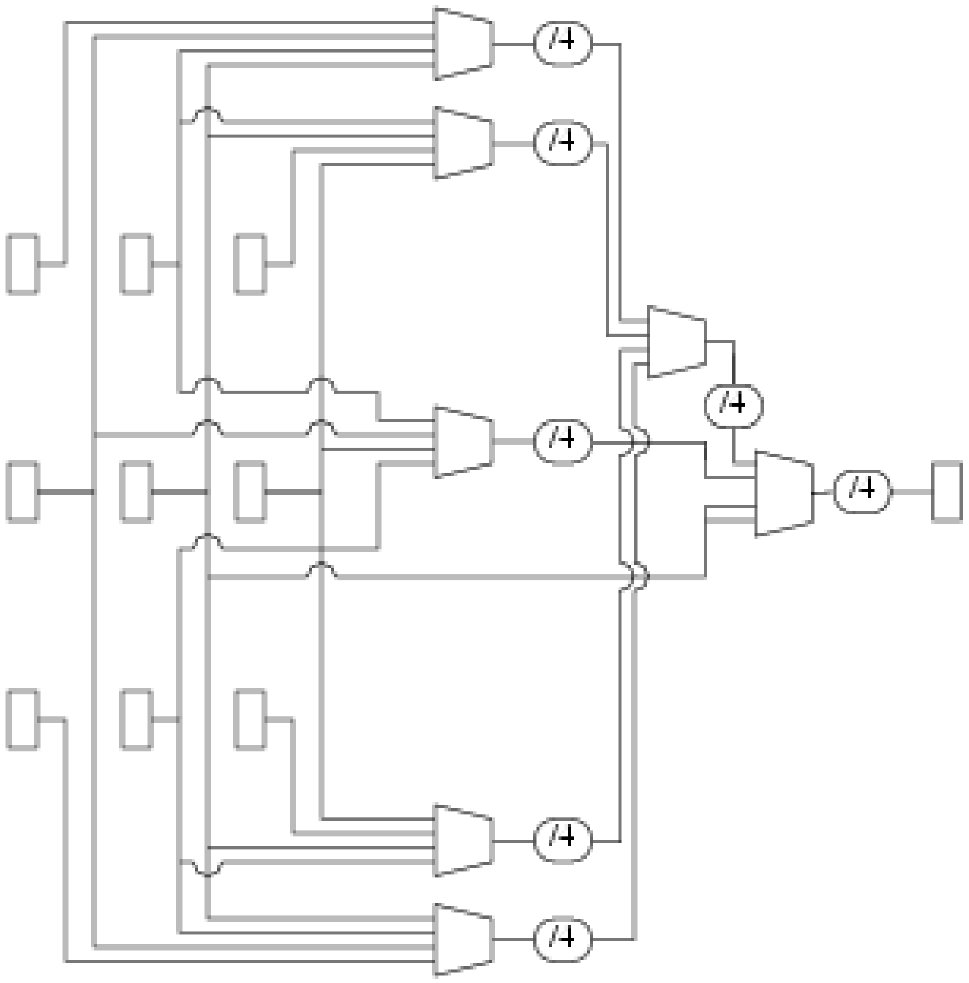

Figure 5.5: Hardware design for calculating a replacement vertex. Nine coarse level vertices are the input on the left. These vertices are processed by adders and shifters to calculate the single replacement vertex on the right.

constant number of clock cycles.

Chapter 6

Analysis

There are four things we would like to minimize in a hardware subdivision algorithm: bandwidth, storage, time complexity and hardware complexity.

Bandwidth between the host CPU memory and the on-card graphics engine memory must be reduced since communication delay is a significant bottleneck in most systems.

Memory on-card is a valuable and limited resources. Typically, there is a tradeoff between bandwidth and memory since caching more data in memory allows for data transmission to be reduced. Memory on the host should also be taken into consideration but is typically more plentiful. Access speed to host memory is not as critical and so virtual memory can be used to make this a nearly unlimited resource.

Time complexity must be considered for both the host CPU and the graphics engine. It is desirable to perform as little computation as possible on the host CPU so that it can be made available for other processing tasks. The goal on the graphics engine side is to read as many input quadstrips as possible and render subdivided strips in real time. This performance will be enhanced by maximizing parallelism.

Finally, hardware complexity must be considered. If hardware can be synthesised with a low gate count then it will be easy to include subdivision hardware in the traditional rendering pipeline found in most graphics hardware today.

These four goals have been addressed with the hardware algorithm discussed in this thesis. The following sections provide an analysis of the success of the algorithm in meeting these goals.

6.1 Bandwidth Requirements

One of the key goals of the hardware algorithm is to reduce bandwidth requirements between CPU memory and on-card graphics engine memory. It will be interesting to compare these bandwidth requirements to the traditional OpenGL API as well as to the ideal case. In the ideal case, each vertex is transmitted only once from the CPU to the graphics engine. It is difficult to conceive of an algorithm that could receive each vertex only once and yet still minimize the storage requirements. Thus, some increased bandwidth is used to allow for reduced graphics engine storage.

For ease of analysis, it will be assumed that all mesh vertices have degree four and that there are no boundaries.

Using a standard OpenGL quadstrip implementation, we can imagine sliding a window encompassing two rows of vertices over the entire mesh. It is clear that each vertex will be sent to the hardware twice, once as a bottom row and once as a top row. Thus, the transmission requirements will be $2V$, where V is the number of vertices.

Using the proposed algorithm, we can imagine a window encompassing four rows sliding over the mesh. Using this approach, each vertex must be transmitted four times and so the bandwidth used will be $4V$.

The amount of bandwidth required for the proposed algorithm is therefore approximately double that required for transmitting standard strips. If we consider that the proposed algorithm will render many times the number of polygons as the traditional algorithm (4^l times, where l is the number of subdivision levels) it can be seen that there is potential for significant bandwidth reduction by performing the subdivision in hardware as opposed to subdividing in software and then transmitting the entire fine mesh.

6.2 Storage Requirements

The number of vertices in a regular column at subdivision level j is

$$V_j = 3 + 2^j. \tag{6.1}$$

Assuming three columns are stored at each level of subdivision, the total vertex storage that must be reserved for l levels of subdivision on the regular portion of the mesh is

$$S_{regular}(l) = \sum_{j=0}^{l} 3(3 + 2^j) = 9l + 3(2^{l+1}) - 3 \tag{6.2}$$

To store extraordinary vertices, six sets of vertex fans must be stored at the top level but only two need to be stored at intermediate levels of subdivision. The vertex storage required for each fan is dependent upon the maximum allowable vertex degree, $V_{d_{max}}$. So, the total vertex storage required for fans is

$$S_{extra}(l) = 6(V_{d_{max}} - 4) + \sum_{j=1}^{l} 2(V_{d_{max}} - 4) = (6 + 2l)V_{d_{max}} - 24 - 8l \tag{6.3}$$

Typically, smooth looking surfaces can be achieved with five or less levels of subdivision. This judgment is qualitative and depends upon the input mesh and view position. The other decision that must be made is the limit for vertex degree. Meshes used for modeling purposes rarely have vertices with degree greater than six but we will allow for a maximum degree of ten for our analysis.

If the subdivision algorithm uses limits of five levels of subdivision and vertex degree of ten, then the total vertex storage that must be reserved is

$$S_{total} = S_{regular}(5) + S_{extra}(5) = 410 \tag{6.4}$$

Storage for 410 vertices with each vertex represented by four integer homogeneous coordinates will require less than seven kilobytes of storage. This volume of storage is not unreasonable for an implementation in specialized hardware. Also note that this volume of storage compares well with the analytic evaluation technique described in Section 2.3. The storage required for the quadstrip algorithm depends primarily on the number of levels of subdivision while the storage required for analytic evaluation depends primarily on maximum vertex degree.

6.3 Time Complexity

The hardware is designed to calculate an entire column of face vertices or replacement vertices in a single clock cycle. To initialize the algorithm, three columns are created at each level and so if l is the maximum number of subdivision levels then roughly $3l$ clock cycles are used for the initialization. Reading in a new column at the top level generates 2^d new columns at the dth level of subdivision. The total number columns computed given one new row at the coarsest level is $2^{l+1} - 2$. If the number of subdivision levels is limited to five then the computation of thirty-two columns at the lowest level requires only sixty-two clock cycles. If reasonable gate delay is assumed on card then current hardware might be expected to run at 200MHz. This would produce nearly one hundred million columns of quadrilaterals per second which is more than enough to saturate any current rendering hardware. If thirty frames are rendered per second then each frame could include nearly one hundred million individual quadrilaterals rendered at the finest level of subdivision. Note that in a 1200x1600 screen there are less than two million pixels. Thus, the capability of rendering one hundred million quadrilaterals per frame should be more than sufficient.

6.4 Hardware Complexity

An estimate of the required gate count can be calculated by examining Figures 5.3 and 5.4.

Computation of a vertex column in a single clock will require calculating two adjacent columns of face vertices. Each face vertex is calculated with a four integer adder and a two bit shift register. Assuming the same hardware is used for each level of subdivision, there must be enough adders and shifters to calculate two columns at the deepest level of subdivision. If this level is l then

$2^l + 4$ adders and two bit shifters are required. It will also be necessary to calculate the face vertices in the fan around the potentially extraordinary vertices at the top and bottom of the column. If the maximum vertex degree is d_{max} then the total number of adders and shifters for all face vertices will be $2^l + 2d_{max} - 4$.

The calculation of each edge vertex involves summing the result of two face vertex computations with the midpoint of the given edge. This computation will require two single bit shifters, a four integer adder and a division by three (performed by a multiplier). The same hardware could be used to calculate the edge vertices for vertex columns and face columns. The most edge vertices that will need to be calculated in a single cycle will be from the deepest subdivision level of a vertex column with maximum degree vertices on the top and bottom. If the maximum level is l and the maximum degree is d_{max} then the total number of adders and multipliers required for these edge vertices will be $2^l + 2d_{max} - 6$.

The calculation of replacement vertices will be similar to edge vertices in the regular case. Equation 5.1 describes the computation of replacement vertices. In the regular case, four surrounding face vertices will be added to four adjacent vertices and the result will be divided by sixteen. The result of this computation will be added to the vertex that is about to be replaced divided by two. This computation will require two four integer adders, a three integer adder, a four bit left shift and a two bit right shift. At the deepest level, there will be 2^l such sets of gates.

The calculation of replacement vertices at the top and bottom of a column will be more complicated. Account must be taken for arbitrary degree that will require larger adders and a general multiplier with a LUT. If the vertex degree is d then one LUT must store $1/d^2$ and another must store $(d-2)/d$. If the maximum degree is d_{max} then for calculation of top and bottom extraordinary vertices, there will be four d_{max} integer adders, four multipliers and four lookup tables with size $d_{max} - 4$.

All that remains to estimate the number of gates required to implement the algorithm in hardware is control logic. This logic is quite simple and is only required to trigger calculation of face and vertex columns and shifting of data from one column to the next when new columns are being calculated. It is difficult to make an estimation of the size of this hardware without actually creating it but it is reasonably clear that this portion of the hardware would be relatively small.

Thus, the gates required for implementing the entire algorithm include

- $2^{l+2} + 4d_{max} - 10$ four integer adders
- 2^l 4 bit right shifters
- $2^l + 2d_{max} - 4$ 2 bit right shifters
- $2^{l+1} + 4d_{max} - 12$ 1 bit right shifters
- 2^l 1 bit left shifters
- $2^l + 2d_{max} - 6$ multipliers by 1/3
- 4 LUTs with size $d_{max} - 4$
- 4 adders for adding d_{max} integers
- control logic

For reasonable values of maximum vertex degree and maximum numbers of subdivision levels, the numbers of gates described above should be reasonable.

Chapter 7

Conclusions

7.1 Summary

The key contribution of this thesis is the concept of using quadstrips for efficient subdivision surface evaluation schemes. Subdividing strips in software allows for reduced memory requirements and produces high quality longitudinal strips to be rendered by traditional rendering hardware. Subdividing strips in hardware can be accomplished using the sliding window idea introduced by Pulli and Segal [PS96].

A software implementation of the proposed hardware algorithm has been implemented to prove feasibility. Analysis of this implementation provides little insight since no operations are performed in parallel but it was a useful tool for developing the algorithm completely. For example, handling degree three vertices was a difficulty that was identified and solved with the software implementation.

A theoretical analysis of the hardware algorithm has been provided. This analysis demonstrates that a hardware implementation could be produced with a reasonable number of gates and small constant storage requirements that is capable of producing high detail output given coarse detail input at a rate capable of saturating current rendering hardware. If the subdivision hardware is collocated with the rendering hardware then it would be possible to render more polygons and thus smoother surfaces while using less system bus bandwidth and fewer CPU cycles. Reducing CPU usage will free up the CPU to perform other tasks related to interactive rendering. From a qualitative perspective, the end result of this implementation will be an improvement in the appearance and performance of interactive graphics applications.

7.2 Other Subdivision Schemes

Creating a hardware algorithm for subdividing triangle strips using Loop's scheme could be accomplished by combining the work in this thesis with that of Kari Pulli and Mark Segal [PS96]. While Pulli and Segal subdivide a pair of triangles using a sliding window algorithm, a strip of triangles could also be subdivided using the same sliding window to produce longitudinal or latitudinal strips with almost exactly the same structure as was used in this thesis for Catmull-Clark subdivision. The only change is to replace each single quad with a pair of triangles and extraordinary vertices only need to store immediate neighbors instead of adjacent and opposite neighbors.

Both Loop and Catmull-Clark subdivision have the properies that each face on the subdivided mesh can be uniquely associated with a face on the original mesh and each vertex on the original mesh can be uniquely associated with a vertex on the subdivided mesh. These properties make parsing of the mesh into segments for subdivision easier. Dual subdivision schemes such as Doo-Sabin [DS78] that replace vertices with faces would be difficult to implement using ideas discussed in the thesis due to the difficulties with parsing the mesh.

Peters and Reif have developed a scheme they refer to as "The Simplest Subdivision Scheme" [PR97]. This scheme does not have the property of creating faces with a common degree. Since the structure of the subdivided mesh is irregular, it would be difficult to implement in an array-based hardware implementation.

Finally, there are a number of interpolatory subdivision schemes such as the Butterfly Subdivision Scheme [DLG90]. These scheme tend to have complex insertion rules compared to the simple averaging of points required for Loop or Catmull-Clark. As a result of these difficult calculations, they are not well suited for hardware implementation.

7.3 Future Work

The obvious next step in this research is to produce a hardware implementation. An FPGA implementation is currently being considered. This sort of implementation is ideal for prototyping. Unfortunately, the subdivision hardware on the FPGA will probably not be collocated with the rendering hardware and so system bus bandwidth will still be heavily utilized. To have rendering hardware collocated with the subdivision hardware, it would need to be implemented on the FPGA simultaneously with the subdivision hardware.

The work by Velho [VdFG99] for subdividing a single triangle strip and producing a single longer triangle strip as the output is appealing in the context of the work presented in this thesis. If such a strip could be processed using a sliding window algorithm then more efficient use could be made of the rendering hardware since strips produced by the subdivision algorithm would be exponentially longer than those produced by the algorithm in this thesis.

Finally, more work could be done on the physical hardware implementation to produce greater parallelism. Subdivision could be performed at multiple levels simultaneously using pipelining. This sort of work could increase polygon throughput for the hardware algorithm.

Bibliography

[CC78] E. Catmull and J. Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350–355, September 1978.

[DKT98] Tony DeRose, Michael Kass, and Tien Truong. Subdivision surfaces for character animation. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 85–94, 1998.

[DLG90] Nira Dyn, David Levin, and John A. Gregory. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Transactions on Graphics*, 9(2):160–169, April 1990. ISSN 0730-0301.

[DS78] D. Doo and M. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10:356—360, September 1978.

[ESV96a] Francine Evans, Steven Skiena, and Amitabh Varshney. Efficiently generating triangle strips for fast rendering. Technical report, State University of New York at Stony Brook, Stony Brook, NY, March 1996.

[ESV96b] Francine Evans, Steven Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. In *Visualization '96*, October 1996.

[Loo87] Charles Teorell Loop. Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, 1987.

[McC00] Michael McCool. Smash: A next-generation api for programmable graphics accelerators. Technical report, SIGGRAPH 2000 Course on Real-Time Programmable Shading, 2000.

[PR97] Jörg Peters and Ulrich Reif. The simplest subdivision scheme for smoothing polyhedra. *ACM Transactions on Graphics*, 16(4):420–431, October 1997. ISSN 0730-0301.

[PS96] Kari Pulli and Mark Segal. Fast rendering of subdivision surfaces. In *Proceedings of 7th Eurographics Workshop on Rendering*, pages 61–70, 282, Porto, Portugal, June 1996.

[Sta98] Jos Stam. Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 395–404, 1998.

[SZSS98] Thomas W. Sederberg, Jianmin Zheng, Malcolm Sabin, and David Sewell. Non-uniform subdivision surfaces. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 387–394, 1998.

[VdFG99] Luiz Velho, Luiz Henrique de Figueiredo, and Jonas Gomes. Hierarchical generalized triangle strips. In *The Visual Computer*, volume 15(1), pages 21–35, 1999.

[War98] Joe Warren. Subdivision surfaces course. Technical report, SIGGRAPH 2000 Course on Subdivision Surfaces, 1998.

[Wer94] Josie Wernecke. *The Inventor Mentor.* Addison-Wesley Publishing Company, 1994.