Fast Rendering of Subdivision Surfaces

Kari Pulli (Univ. of Washington, Seattle, WA) Mark Segal (SGI)

Abstract

Subdivision surfaces provide a curved surface representation that is useful in a number of applications, including modeling surfaces of arbitrary topological type, fitting scattered data, and geometric compression and automatic level-of-detail generation using wavelets. Subdivision surfaces also provide an attractive representation for fast rendering, since they can directly represent complex surfaces of arbitrary topology.

We present a method for subdivision surface triangulation that is fast, uses minimum memory, and is simpler in structure than a naive rendering method based on direct subdivision. These features make the algorithm amenable to implementation on both general purpose CPUs and dedicated geometry engine processors, allowing high rendering performance on appropriately equipped graphics hardware.
Key Words: subdivision surfaces, surface rendering.

1 Introduction

A subdivision surface results from iteratively refining a control mesh of arbitrary topology by adding new vertices into the mesh and modifying the locations of existing vertices. In the limit the mesh converges to a continuous surface. The control mesh usually gives a rough polygonal approximation of the resulting surface; some methods even interpolate the control mesh [4]. The first subdivision surfaces were based on quadrilateral meshes and generalize biquadratic [3] and bicubic [1] tensor product surfaces (see [5] and [9] for extensions). In addition to surface modeling, subdivision surfaces have been used for fitting scattered data [6], and geometric compression and automatic level-of-detail generation using wavelets [8].

The goal of this work was to develop an efficient algorithm to implement Loop's subdivision scheme [7] with additions by Hoppe *et al.* [6]. Loop's subdivision scheme is a generalization of C^2 quartic triangular B-splines, and is the simplest method known to lead to tangent plane smooth surfaces. The control mesh consists of triangular faces and, like subdivision surfaces in general, can have any topology. The resulting surface has the same topology as the control mesh.

Figure 1 diagrams Loop's subdivision process. Each iteration splits all the edges in the control mesh and introduces four new triangles for each original one. The location of a new vertex is obtained by a weighted average of the surrounding vertices. For example, when the edge BC is split, the vertex m is introduced at location $m = \frac{A+3B+3C+D}{8}$. Each iteration also perturbs the locations of existing vertices by taking a weighted average of the locations of the vertex and its immediate neighbors. It is also possible at any level to directly calculate the limiting location of each control vertex on the final surface, as well as the normal vector.

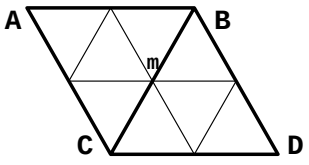


Figure 1: Triangle subdivision.

Hoppe *et al.* [6] extended Loop's scheme by introducing subdivision rules that lead to a piecewise smooth surface with features such as creases, corners, darts, and conical ver-

tices. The vertices and edges of the control mesh are typed and the weights in the subdivision rule are chosen based on the vertex and edge types (the rules are collected in Appendix).

1.1 Rendering

Figure 1 suggests a simple method for rendering a subdivision surface: for each triangle, create four new triangles, and compute the locations of the resulting (original and new) vertices, then recurse. The control mesh can be stored in a general mesh data structure which facilitates operations such as finding the neighbors for each vertex and edge. While this algorithm is straightforward, it is inappropriate for implementation in dedicated graphics hardware. Geometry engines (GEs) typically have limited memory and may be ill-suited to linked-list traversal and recursion.

We describe a novel algorithm for subdivision surface triangulation that is fast, uses little memory, and is easy to parallelize. These features make the method ideal for GE implementation. If GEs are available, then the host is freed from doing most of the work of generating triangles on the subdivision surface. In addition, instead of sending the GEs a long list of triangles, the host sends a compact representation of the surface, reducing host-GE bandwidth requirements. Even an implementation that performs all the computations in the CPU, however, can reap the speed benefits of the simple indexing, iterative processing, and small memory consumption of our algorithm.

1.2 Overview of paper

In the next section we describe the full algorithm for rendering subdivision surfaces. We describe how to do the subdivision using a simple 2D array and present a sliding window method for rendering the subdivision surface patches depth-first and in-place, thus saving memory. We then describe how we find the submeshes that are sent to GEs and describe some practical issues and limitations. Section 3 presents our results and compares subdivision surface rendering to that of NURBS surface rendering. Section 4 discusses the promise for fast rendering when other subdivision surface features are considered such as automatic LOD selection and compression. An appendix describes the actual subdivision rules.

2 Method

Our basic idea is to divide the subdivision of an arbitrary control mesh into two tasks. The host manages the general control mesh and partitions it into small, regular patches. These patches can then be rendered independently inside a GE using an efficient algorithm. The patches are stored so that after partitioning the host need only consult a cache and send the precomputed messages to the GEs to render more frames.

We have implemented the control mesh in the host as a half-edge data structure similar to ones described in [11]. Although such a data structure is needed to describe an arbitrary control mesh, we can use a simpler data structure to describe the regular surface regions between the original control points. Our method avoids the overhead of explicitly representing concepts such as faces, edges, and neighbors, and stores only the vertices. The vertices can have arbitrary dimensions, so each vertex can have color or texture coordinates in addition to 3D space coordinates. The subdivision algorithm treats all dimensions in the same fashion.

In order to preserve continuity between patches, the patches need to overlap. Figure 2 shows two subdivisions of a patch consisting of a single control triangle. In the first subdivision, all the *support* vertices (the 1-neighborhood) of the patch are needed both to split the edges and to update the (three) old vertices in the patch. The neighboring triangles are subdivided only enough to obtain a new layer of supporting vertices and edges. In the following subdivisions this support layer shrinks even closer to the patch.

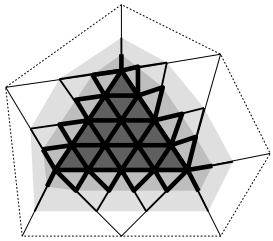


Figure 2: The immediate neighbors are needed.

We first describe how a 2D array can be used to subdivide pairs of control mesh triangles. Then we present a depth-first method for performing these calculations in a memory-efficient manner. We propose a method for finding the triangle pairs from the original control mesh, and finally describe the structure of the messages sent from host to GEs as well as some limitations caused by limited resources.

2.1 Subdivision inside a GE

Subdivision in a 2D array

A 2D array is perhaps the simplest possible data structure for describing points on a surface. However, it can store effectively only regularly tesselated surfaces of planar topology, so it is not a suitable representation for the general control mesh. Inside the control triangles things are simpler: the surface has planar topology and the vertices are connected in a very regular fashion. When we combine two triangles into a quadrangle, we obtain a natural mapping from subdivided triangle pairs to a square array. Figure 3 shows the array representation of the triangle pair after one and two subdivisions.

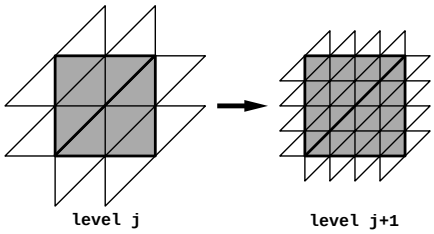


Figure 3: A 2D array stores vertices compactly.

There is no need to keep any connectivity information to find the neighbors for a vertex, since the neighbors are always in the adjacent array locations. With Hoppe's extension to Loop's subdivision scheme, each edge and vertex can have an associated type. However, we only need to store the original edge and vertex types, and the types can be deduced from the location in the 2D array and the stored original types. The extra corner support vertices are stored in an additional array.

With this scheme, it is easy to subdivide the triangles one level at a time. The size of the array at an intermediate level j is $(2^j+3)\times(2^j+3)$. At the final level we calculate the vertex coordinates and normal vectors on the final surface, store them in a $(2^j+1)\times(2^j+1)$ array, and render them as triangle strips. An exception occurs if the (diagonal) edge between the original triangles is sharp. In that case the vertices on the diagonal have two normal vectors, so we need one more element for each row for the extra normal, and each row must be rendered as two triangle strips. The number of subdivision levels is decided beforehand by the application and is a tradeoff between execution time and rendering accuracy.

The sliding window method

The simple method of subdividing a triangle pair one level at a time, even with a clever implementation, uses memory on the order of the size of the output. Since each level

of subdivision quadruples the number of triangles, we could only implement a very modest number of levels (e.g., 2) in a GE. Instead, we use a depth-first approach that incrementally calculates the vertex coordinates, renders the triangles as soon as they are calculated, and reuses memory. This technique enables us to go several levels deeper than the naive approach with the same amount of memory.

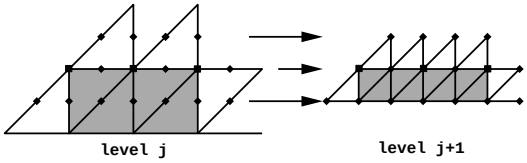


Figure 4: Initialization of the depth-first algorithm.

It turns out that it is sufficient to store only a window of three rows of the 2D array for each level, plus two rows for the final vertex coordinates and normal vectors. In the course of the algorithm we slide the window down the 2D arrays. Figure 4 illustrates the initialization phase of the algorithm. On the left side we have the three first rows of the triangle array at subdivision level j.

We obtain the first row support vertices at level $j + 1$ (the right image) by splitting the diagonal and vertical edges between the first and second row vertices at level j. Similarly, the second row in the right image is obtained by splitting the horizontal edges and updating the vertices in the left image. The third row is obtained by splitting the vertical and diagonal edges connecting the second and third rows at level j. We initialize the three-row arrays a level at time, and at the final level we calculate the final coordinates and normal vectors for one row.

After the top rows have been initialized, we begin the recursive part of the algorithm. We introduce the next row of vertices from the original message. Figure 5 (the upper arrow) shows how a new row at level j provides the neighbors needed for splitting the horizontal edges and updating the vertex coordinates in the middle row. The new and updated vertices become the new low row at level $j + 1$, and they are written over the old highest row at that level. We can now repeat the same going from level $j + 1$ to $j + 2$.

The new row at level j also enables us to split the diagonal and vertical edges connecting the level j low and middle rows, which produces yet another new row for level $j + 1$ (the lower arrow in Fig. 5). In this manner, we write new rows over old ones and descend towards the final subdivision level. There, with each new row, we calculate the final coordinates and normals for the middle row vertices and render another triangle strip.

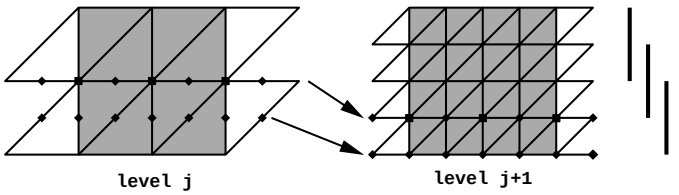


Figure 5 Introduction of the lowest row on the left enables us to calculate two new rows on the right. The bars on the right denote the extent of the window before, after the split of the horizontal edges, and after the split of the vertical and diagonal edges.

Let us calculate the space and time requirements of our method. We denote by n the number of 3D points (with normal vectors) that running the algorithm for l levels produces. The length of the rows at the finest level is approximately $\sqrt{n}$, and since the rows at a coarser level are only half as long as at the next finer level, the algorithm uses $O(\sum_{i=0}^{l} \sqrt{n}\frac{1}{2^i}) = O(\sqrt{n})$ memory. Each vertex is calculated only once, the calculation takes constant time,

and since there are only about one quarter as many vertices at a coarser level, the algorithm uses $O(\sum_{i=0}^{l} n\frac{1}{4^i}) = O(n)$ time.

```
SUBDIVIDE(level, last)
BEGIN
  IF (level != last) THEN
    split horizontal edges, update vertices
    SUBDIVIDE(level+1, last)
    split vertical and diagonal edges
    SUBDIVIDE(level+1, last)
  ELSE
    calculate final positions and normals
    render a triangle strip
  ENDIF
END
```

Here is pseudocode for the depth-first algorithm. Although the algorithm is not tail recursive, we can unroll the algorithm and execute it iteratively since we know the maximum number of subdivision levels (which is dictated by the availability of resources).

2.2 Finding triangle pairs

As we have already said, we render the control triangles in pairs. The host needs to partition the control mesh into pairs of neighboring triangles, which are then rendered in the GEs. A simple, linear time algorithm would just remove a triangle along with a neighbor, until no triangles remain. This simple approach, however, is likely to produce several triangles whose neighbors have already been paired with other triangles, which wastes resources.

Intuitively, we want to avoid running into a dead end where some of the triangles end up without a unique pair. Therefore, we want to nibble away pieces from the boundary of the mesh so it remains as compact as possible in the sense that the number of the boundary edges is minimized. We also want to avoid breaking the mesh into disconnected parts.

Our algorithm proceeds as follows. Each triangle is inserted into one of four priority sets depending on how many free (unpaired) neighbors it has (see Fig. 6). Triangles with zero or one free neighbors have the highest priority (3), and the triangles with three free neighbors have the lowest priority (0). Of the remaining triangles (two free neighbors), the ones with both neighbors in set 0 have lower priority (1). The algorithm chooses a triangle from the highest nonempty set and pairs it with the neighbor of highest priority. The paired triangles are removed from the sets, and their remaining neighbors are promoted to a higher priority set (from 1 and 2 to 3, from 0 to 1). Additionally, all the triangles in 1 with a free neighbor not in 0 are promoted to set 2.

The algorithm has as many cases as we have priority sets: let's analyze each of the cases (see Fig. 7). If the highest nonempty set is 3 and the triangle under consideration does not have any free neighbors (Fig. 7(i)), we can remove it. We will, however, pair this triangle

set	number of free neighbors
3	0 or 1
2	2, one of them not in 3
1	2, both of them in 3
0	3

Figure 6 The sets for pairing triangles.

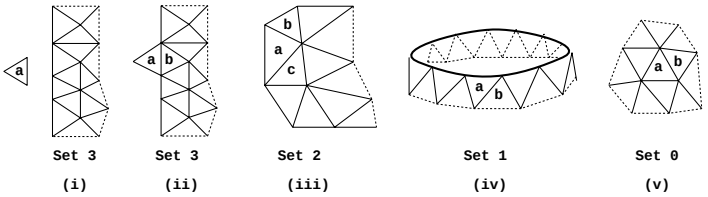


Figure 7 The different cases of the triangle pairing.

with a dummy mate and make a note that only this triangle of the pair will be actually rendered. The number of boundary edges is reduced by 3, and this choice does not break up the mesh.

If the current triangle (**a** in Fig. 7(ii)) has one neighbor we pair it with that neighbor (**b**). This choice is forced. Though it can break up the mesh, it will not produce a suboptimal triangulation: it could be that **b** was needed as a mate of some other triangle in the optimum pairing, but if we don't pair it, **a** will be left alone. The number of boundary edges may stay the same, or is reduced by up to 4 edges.

If the highest nonempty set is 2 (Fig. 7(iii)), one of the neighbors of the current triangle **a** must also belong to 2 (in this case **b**, while **c** belongs to set 0). In the example, pairing **a** with **b** just chips off a corner of the mesh. The number of boundary edges remains the same. Here we have to be careful, though, since it is possible to disconnect the mesh. Therefore, if there are any other triangles in the set 2 that do not disconnect the mesh, they should be given precedence.

If the highest nonempty set is 1 (Fig. 7(iv)), the mesh has rather regular structure at the boundary: each triangle with a boundary edge will have two neighbors that have only a boundary vertex. In the example pairing **a** with **b** would directly lead to a cascade of pair removals (the neighbor of **b** is promoted to set 3) which would peel off a layer of triangles from the mesh boundary. The number of boundary edges grows by 2, and the mesh cannot be disconnected.

The case where the highest nonempty set is 0 (Fig. 7(v)) can only occur in the beginning if the mesh forms a closed surface. The number of boundary edges increases to 4, and the mesh remains connected.

Figure 8 gives two examples of how the algorithm proceeds. In the left mesh, all the triangles with a border edge belong to set 2, while the others belong to set 0. We start by pairing two triangles marked "1", which promotes one of triangles marked "2" to set 2. They form the next pair, which now promotes one of the triangles marked "3" to set 3, and so forth. In the mesh on the right we always remove triangle pairs such that at least one of the triangles is left with only one free neighbor.

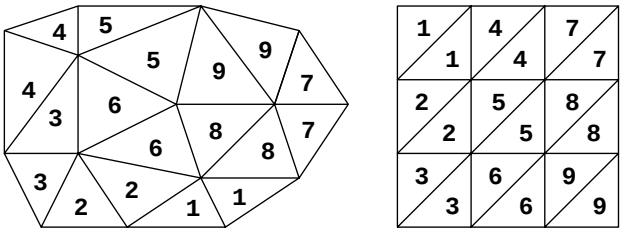


Figure 8 Pairing of triangles in a mesh.

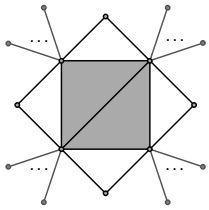


Figure 9 Two triangles and their immediate neighbors.

2.3 Some implementation issues

Host-GE interface

Figure 9 illustrates the information in the message that is sent to the GEs for rendering two triangles. The message comprises two parts: a fixed-length part and a variable-length part. The fixed-length part contains the vertex coordinates along with vertex and edge types for the triangle pair (the shaded triangles) and the neighboring triangles (the four unshaded triangles). Additional information includes the dimension of vertex coordinates and whether

the second triangle should be rendered or is a dummy. The variable-length part accommodates the variable valence of the control mesh vertices.

The overhead of subdividing the neighbors

When looking at Fig. 2 it might seem that we are doing more work subdividing the immediate neighbors than the actual patch. A closer look reveals, however, that after a few subdivisions the space and time spent on the support edges and vertices is but a small fraction of the real work.

A larger patch size with more control triangles would mean less duplicated work with the support edges. However, that would destroy the simple square array structure and the patch would not be guaranteed to be topologically planar. Also, the patch size should be constant to obtain even load balancing for maximum performance in the case of several GEs.

Limitations

Due to the limited memory, there has to be a maximum number for the levels of subdivision we can process in a GE. If the user wants more subdivisions, the first levels can be done in the host, and the results then sent to GEs.

The memory limitations also prevent us rendering meshes with arbitrarily large vertex valencies. We currently support a combined valency of 40 for the four corner vertices, though the choice is arbitrary and could easily be made somewhat larger.

3 Results

We implemented this algorithm as an experimental extension to OpenGL for the Silicon Graphics Onyx Reality Engine. The host stores the general control mesh using a half-edge data structure, chops the mesh into triangle pairs, and sends the pairs (with their neighbors) to GEs. The GE microcode (unoptimized C) subdivides the triangle pairs and renders them using the existing OpenGL code for rendering triangle strips.

3.1 Performance measurements

We present some performance measurements comparing subdivision surface and NURBS versions of the Utah teapot and a cylindrical car part with holes (see Fig. 10). In the following table, RE refers to an SGI Onyx RE2 with 12 geometry engines operating in parallel, while Indigo refers to an SGI Indigo^2 XL on which all subdivision work is done in the host.

All subdivision surface times include finding triangle pairs. The additional control points for the “car part” NURBS surface indicate trim curve points; measured times include trim region tesselation. The two sets of subdivision surface timing numbers are for 2 and 3 levels of subdivision. The triangle rates of the two examples seem typical: other complex surface models (e.g., head and distributor cap) had very similar rates.

The results show that even our unoptimized implementation was faster than the OpenGL NURBS, both for a GE microcode implementation (RE) and when the subdivision was performed completely on the host (Indigo). In addition, for a surface with complicated local geometry (see color plate) a NURBS rendering would require more triangles to obtain a comparable visual quality. The reason is that the subdivision surface representation concentrates more triangles in areas of higher detail; this effect is difficult to achieve with NURBS renderers.

The triangle rate increases with the number of subdivision levels. There seem to be three

Subdivision surfaces				
model	control points	triangles	triangles/sec (1000s)	
			RE	Indigo
teapot	157	4960	147	26
		19840	207	31
car part	126	3264	157	26
		13056	212	39
NURBS surfaces				
teapot	512	8740	141	24
car part	1143+189	5608	80	22

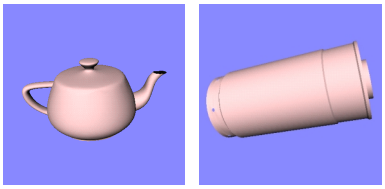


Figure 10 Subdivision surfaces: the Utah teapot and a cylindrical car part with holes.

reasons: with more subdivision levels, the setup and communication costs are amortized over a larger number of triangles, more of the computed triangles are inside the control triangles where the subdivision algorithm is most regular, and individual triangle strips become longer.

3.2 Real time editing

We implemented a simple surface editor that allows us to move the control vertices around and change the types of the control vertices and the connecting edges. The fast rendering allows interactive editing of the control mesh even for quite complicated objects (consisting of hundreds of control triangles) and gives real-time feedback by showing the resulting subdivision surface. The color plate shows a model of a head and the result of a couple of minutes' worth of editing.

3.3 Comparison with NURBS

Subdivision surfaces possess several potential advantages over NURBS patches. The most obvious advantage is that a single subdivision surface can model an object of any topology, whereas objects with complicated shapes usually need to be covered by several NURBS patches. The color plate shows such a surface. With a single continuous surface there is no need to stitch patches together to avoid cracks or to trim patches to avoid surface interpenetration. Further, the control points of a NURBS patch need to be in a topologically regular 2D lattice, whereas the subdivision surface control mesh gives more freedom about the connectivity of the control points.

4 Discussion

4.1 Compression in communication

Rendering a (piecewise) smooth surface accurately by polygonal approximation requires a large number of polygons. Our method not only relieves the host from the task of tesselating the surface into a set of triangles, but it also reduces communication between host and GEs. Let's assume that the user wants to subdivide the control mesh four times to achieve a very high geometrical accuracy. Every triangle in the control mesh is subdivided four times, yielding $4^4 = 256$ triangles. When these triangles are rendered, for example as triangle strips, we still have to send each vertex twice (on average) to the graphics hardware. A single subdivision message, however, sends typically only 10-15 vertices (two triangles and their neighbors), along with some type information, but produces 512 triangles.

Subdivision surfaces combined with wavelet analysis may yield even greater compression in communication [2]. A multiresolution representation of the control mesh would allow sending larger submeshes compactly.

4.2 Level-of-detail control

In order to obtain interactive rendering speeds, automatic level-of-detail (LOD) control is essential. With subdivision surfaces, there is no need to store or create alternate descriptions of geometry at different levels of detail. The LOD control is built in: one parameter, the number of subdivision levels, dictates how detailed the rendering is. The color plate shows a teapot at four different levels of detail.

It is quite easy to obtain a continuous LOD control by interpolating between subdivision levels. This is important for eliminating "jumping" of the surface, e.g., when one zooms in to an object to display more detail. Suppose we want to display the object at subdivision level 3.5. Let us call the vertices introduced at level j even and vertices at level $j + 1$ odd. We would then calculate the surface points at level 4, use the average of the parent vertices (even) for each odd vertex as the starting point, and the actual location as the ending point. The non-integer subdivision levels between 3 and 4 can now be obtained by interpolating between the starting and ending points.

It is also possible to render different control mesh triangle pairs at different levels of subdivision. A naive implementation would leave cracks between patches, but with a bit of extra information, the cracks can easily be filled. Let's assume that while rendering the current patch we know that the patch on the left was rendered with one fewer levels of subdivision. We know that each leftmost vertex of the even rows corresponds to a vertex on the neighboring patch. Introducing a new triangle consisting of the even, odd, and the next even vertex will fill the crack. In general, if there are more levels between the patches, the crack filling polygons consist of the vertices shared by the patches and of all the finer level vertices between them.

4.3 Conclusions

We have presented an algorithm for rendering Loop's subdivision surfaces with Hoppe's extensions that uses very little memory and is efficient to execute. A control mesh of arbitrary topology is efficiently divided into patches consisting of triangle pairs and their immediate neighboring vertices. The patches can be rendered separately and they are about the same size, which makes the algorithm attractive for parallel implementation if several GEs are available. Our algorithm works also for any other subdivision scheme with the same support of the subdivision masks and can be extended in a straightforward manner to triangle-based schemes with larger support.

Subdivision surfaces have several advantages over traditional surface descriptions, such as NURBS. Arbitrarily complicated objects can be represented by a single surface rather than a collection of patches; the control mesh is quite natural to edit by designers and provides both smooth surfaces and sharp features; level-of-detail control is supported in a natural way; and our implementation outperformed the OpenGL NURBS versions of comparable surface models.

Acknowledgements

We would like to thank Tony DeRose, Hugues Hoppe, and Paul Beame for useful discussions, and Ziv Gigus, David Blythe, Ben Zhu, Bob Drebin, Erik Lindholm, and David Immel for their help in implementing this method for Reality Engine.

Appendix

The edges and vertices in the control mesh can have the following types. An edge can be either smooth (continuous surface normals) or a crease (discontinuous surface normals). A vertex can be either smooth, dart, crease,

or corner, depending whether it has zero, one, two, or more incident crease edges, respectively. A crease vertex is regular if there is exactly two smooth edges between the two creases, otherwise it is nonregular. Additionally, any vertex can be tagged as a conical vertex.

Figure 11 presents the masks used for both subdividing the control mesh and calculating the final positions and normal vectors at each vertex location. A new value is obtained by an affine combination of a vertex and its neighbors, and the mask gives the appropriate weights. For example, a new vertex position can be obtained by $v_0' = \sum_{i=0}^{n} c_i v_i / \sum_{i=0}^{n} c_i$, where v_0 is the old vertex position and the other v_i's are the neighbors. The following table gives the masks for subdivision (S) and final locations (F)[1]. In the formulas δ_{0i} is 1 if the edge c_0-c_i is a crease, 0 otherwise, $a(n) = 5/8 - (3 + 2\cos(2\pi/n))^2/64$ and $b(n) = (3 + 2\cos(2\pi/n)/8$.

	vertex type	c_0	c_i
S	smooth or dart	$n/a(n) - n$	1
S	crease	6	δ_{0i}
S	corner	1	0
S	conical	b(n)	1
F	smooth or dart	3n/(8a(n))	1
F	regular crease	4	δ_{0i}
F	nonregular crease	3	δ_{0i}

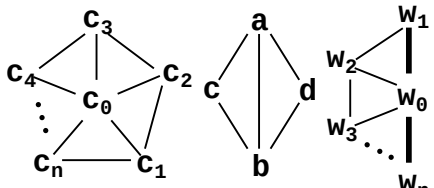


Figure 11 The subdivision masks.

The center mask is used when subdividing edges (between a and b). If the edge is smooth or if either of the vertices (a or b) is a dart, the weights {a,b,c,d} are {3,3,1,1}. If the edge is a crease and one of the edge vertices (a) is a regular crease while the other (b) is either non-regular crease or corner, the weights are {5,3,0,0}. In the remaining cases the weights are {1,1,0,0}.

The normal vectors can be calculated by taking the cross product of two surface tangent vectors u_1 and u_2. For non-crease vertices, we use the left mask. The center weight c_0 is 0 for both u_1 and u_2, while $c_i = \cos(2\pi i/n)$ for u_1 and $c_i = \cos(2\pi(i-1)/n)$ for u_2. For crease vertices we use the right mask (the fat line denotes the crease), where u_1 goes along and u_2 across the crease. The u_1 weights are $w_1 = 1$, $w_n = -1$, other w_i's are 0. For a regular crease vertex, u_2 weights $\{w_0, ..., w_4\}$ are $\{-2, -1, 2, 2, -1\}$. For a non-regular crease vertex, u_2 weights are, for $n \geq 4$, $w_0 = 0$, $w_1 = w_n = \sin\theta$, and $w_i = (2\cos\theta - 2)(\sin(i-1)\theta)$ for $1 < i < n$, where $\theta = \pi/(n-1)$; for $n = 3$, $\{w_0, ..., w_3\}$ are $\{-1, 0, 1, 0\}$; for $n = 2$, $\{w_0, w_1, w_2\}$ are $\{-2, 1, 1\}$.

References

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

[2] M. Deering. Geometry compression. In *SIGGRAPH '95*, pages 13–20, August 1995.

[3] D. Doo and M. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10(6):356–360, September 1978.

[4] N. Dyn, D. Levin, and J. Gregory. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Transactions on Graphics*, 9(2):160–169, April 1990.

[5] M. Halstead, M. Kass, and T. DeRose. Efficient, fair interpolation using catmull-clark surfaces. In *SIGGRAPH '93*, August 1993.

[6] H. Hoppe, T. DeRose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer, and W. Stuetzle. Piecewise smooth surface reconstruction. In *SIGGRAPH '94*, July 1994.

[7] C. Loop. Smooth subdivision surfaces based on triangles. Master's thesis, Department of Mathematics, Univ. of Utah, 1987.

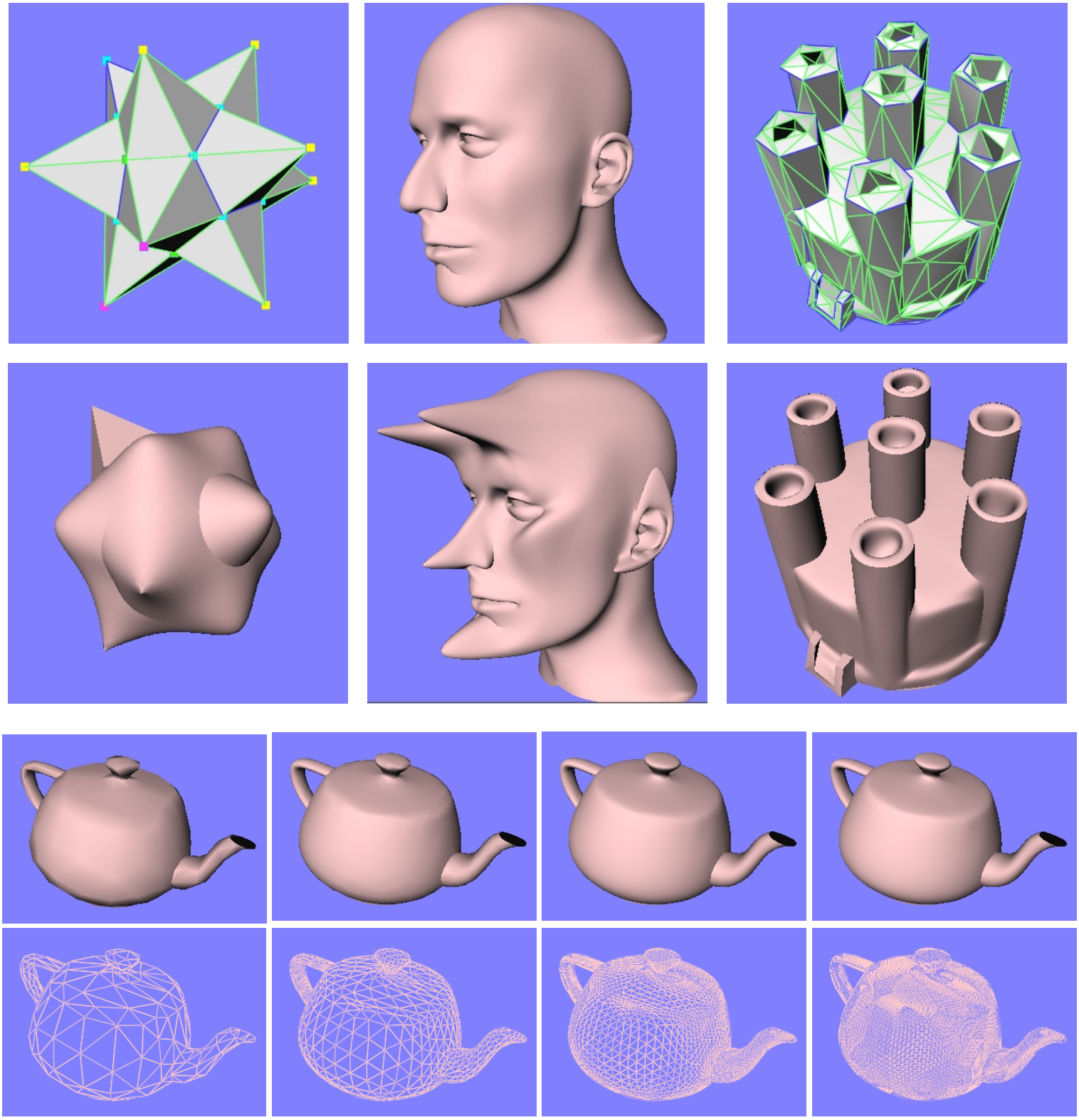
[8] M. Lounsbery. *Multiresolution analysis for surfaces of arbitrary topological type*. PhD thesis, Dept. of Computer Science and Engineering, Univ. of Washington, September 1994.

[9] A. H. Nasri. Polyhedral subdivision methods for free-form surfaces. *ACM Transactions on Graphics*, 6(1):29–73, January 1987.

[10] J. Schweitzer. *Analysis and appliction of subdivision surfaces*. PhD thesis, Department of Computer Science and Engineering, Univ. of Washington (in preparation), 1996.

[11] K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE CG&A*, 5(1):21–40, January 1985.

[1] Formulas from [6], except for conical vertices from [10].



Left top: a control mesh and the resulting subdivision surface. Different colors denote different vertex and edge types.
Center top: A head before and after some simple edits.
Right top: Another example of a complicated object with a single surface.
Bottom: A teapot with 0, 1, 2, and 3 subdivisions. Note that the visual accuracy is already very good with only 2 subdivisions.
(Pulli and Segal)