Practical Implementation Techniques for Multi-Resolution Subdivision Surfaces

David Brickhill

Introduction

Eight months ago we set out to build the first subdivision surface engine for Playstation 2®. We read many research papers and implemented the methods they described. In the process we learned about wavelets, multi-resolution analysis, inner product matrices, and numerous subdivision schemes. After two months, however, we still had not come across a paper that described a method practical for use in a console video game.

It is not that no paper addressed performance issues (see [4]), it's only that all of the work we could find relied on some form of recursive subdivision to produce the limit mesh. We knew that a recursive method had little hope of working on a platform with slow access to main RAM. To avoid recursive subdivision every frame, we initially relied on a complicated caching scheme that essentially treated subdivision as a decompression method. This proved unsatisfying because we had not really implemented subdivision in real-time, but had merely used it to decompress data into explicit meshes.

We finally arrived at an extremely fast method that avoids caching. That method is now called VALENTE™ Subdivision, and it is described in this paper.

Following this introduction, we outline the minimum design specification for a usable subdivision surface engine. Then we describe the general choices we made in regard to fundamentals, and finally go on to give the details of our implementation.

This paper assumes familiarity with subdivision surfaces and with fundamentals of linear algebra. Zorin's SIGGRAPH '98 course ([5]) is the best source we found for an introduction to subdivision.

Required Features of the Engine

In order to be of use to game developers, there is a set of features that a subdivision engine must have. Some of these features require novel insight, while others simply require a bit of clever game programming.

The primary advantage of subdivision surfaces over explicit triangle meshes is that subdivision surfaces support very dense smooth meshes without using much RAM. So rather than being a requirement, small data size is really the reason for using subdivision in the first place.

The first real requirement, and in the end the only one that matters, is speed. The engine must produce results equivalent to explicit meshes with roughly the same processing resources. Note that in this context speed is not to be measured in raw polygon or vertex counts. Given that any procedural method must create triangles before it draws them, it is difficult to imagine a procedural method outperforming an explicit mesh in vertices per second. The idea, in fact, is to reduce the primitive output while still producing visually pleasing results. So, we measure

speed by comparing processing times with explicit meshes that have roughly the same appearance on screen, and we do so in a somewhat subjective manner.

In order to maintain speed and still produce visual quality, we must have multiple levels of detail (LOD). Our LOD system must support *geo-morphing* (i.e. smooth transitions between LOD), and it must be view-dependent in such a way that part of the mesh can be at a different LOD than another part of the mesh. When different parts of the mesh are at different LOD, then we must sew up the seams created at the LOD boundaries in order to prevent visible cracking of the mesh.

Dependence on a persistent cache must be avoided. There is little point in using subdivision surfaces if it is absolutely necessary to cache all triangles in the view frustum. This increases RAM usage to the point that the performance hit taken by using subdivision is hardly warranted.

The engine must support texturing, of course. Furthermore, it must support objects that have different textures over different parts of the mesh. Texture appearances after subdivision must be true to their appearances in the base mesh.

Standard diffuse lighting must be supported in a way that takes advantage of the dense mesh produced by the subdivision. A "spot" light must light the interior of a base mesh face, even if it does not light any vertices of the base mesh.

Animating human characters are among the primary applications for a subdivision system. Therefore any practical system must support bone-driven animation.

General Techniques

In the sections that follow we present the details of our methods for satisfying each of the above requirements. Early in the development process, we made some choices about fundamental techniques. These fundamentals are not fully described in this paper, but our choices from among them are stated in this section.

After trying several subdivision schemes, including Butterfly, Catmull-Clark, and various hybrids, we chose Loop subdivision. The only disadvantage we found in the Loop scheme was that it does not interpolate the base mesh. Its advantages include small support and only a slight tendency to skew when applied to meshes with invisible edges (e.g. meshes composed of quads).

We place an upper limit of 4 on the subdivision depth. This limit permits production of 256 triangles for every triangle of the base mesh, resulting in smoothness sufficient for practical purposes.

To achieve a multi-resolution implementation, we initially used several advanced methods, including Lounsbery's biorthogonal wavelet technique (see [3]). In the end we found that the best approach was simply to use the limit positions of each vertex, regardless of the subdivision level.

Definition of Terms

Throughout our method description we use the following terms.

- The ***base mesh*** is the coarse mesh that undergoes subdivision. It is sometimes called the control mesh in other literature.
- A ***base vertex*** is a vertex of the base mesh. We use the terms ***base face*** and ***base edge*** in a similar fashion.
- A ***subdivided vertex*** is a vertex of the fully subdivided mesh in its limit position.
- We use the term ***local base*** when referring to the portion of the base mesh that is relevant to producing subdivided vertices for a particular base face.
- To distinguish it from the local base, we say ***global base*** to refer to the overall base mesh.
- To refer in general to a vertex, edge, or face of the local base or global base, we use ***local base feature*** and ***global base feature***, respectively.

Achieving Speed —The VALENTE™ Subdivision Method

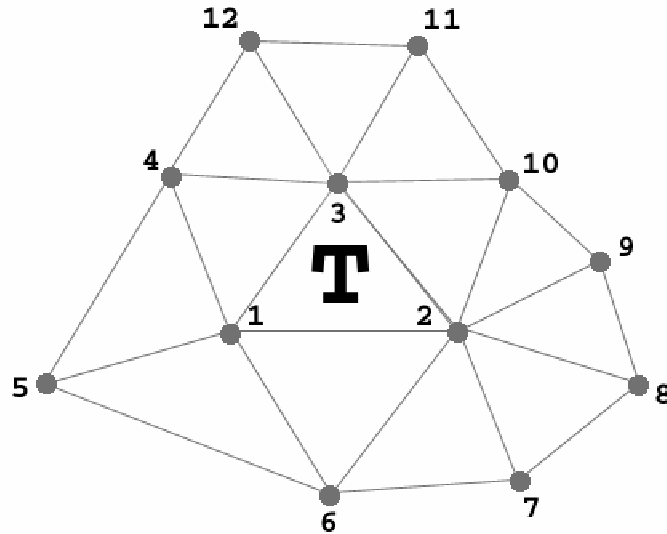
The standard approach for producing subdivision surfaces using the Loop scheme is described by Zorin and others (see [5]). The idea is to subdivide each triangle of the base mesh at the edge midpoints and then take weighted averages to compute new vertex positions. The process proceeds recursively, using the subdivided mesh as the new base mesh until the desired number of subdivision levels is reached.

Recursive methods are fairly simple to implement, except for a bit of painful bookkeeping to update the necessary adjacency information. Despite their simplicity, however, recursive methods are not practical for use in console video games. The primary shortcoming of recursive methods is that they require a great deal of random access to memory that is dispersed over an area too large to fit in a scratch pad or data cache. Even if the base mesh is so small as to fit entirely in a cache, the upkeep of adjacency information requires data structure maintenance that is prohibitively challenging to implement with the limited instruction sets of console geometry processors.

Our run-time method avoids recursion entirely by taking advantage of the fact that each vertex of the subdivided mesh can be written as a linear combination of the vertices of the base mesh, regardless of the subdivision level. Furthermore, the limited support of the Loop scheme implies that each subdivided vertex has only a few nonzero coefficients. We can therefore compute any subdivided vertex simply as a sum of scalar products of vectors, a process well suited to console game hardware. The details of our technique follow.

In a pre-process, we create a local base for each triangle T of the global base, using the Loop Subdivision scheme. Under Loop Subdivision, the local base for T will consist of all global base features in the 1-neighborhood of T. We carefully add local base vertices in a consistent order for all local bases in the following fashion: we choose the first local base vertex from among the 3 vertices of T. That first vertex is chosen such that its valence (e.g. the number of adjacent edges to the vertex) is less than or equal to the 2 subsequent vertices. If there is a tie among two choices, as in the cases of {5,5,6} or {4,3,3}, we choose the first vertex so that the

second vertex will have the lowest valence among the two choices of second vertices. Starting with the selected first vertex, we "spin" around each triangle vertex to include its 1-neighborhood vertices in counter-clockwise order around the triangle. See Figure 1 for an illustration of this step.



-Figure 1-
Numbers indicate the ordering of local control vertices.

Still in a pre-process, for each triangle T of the global base, we subdivide the corresponding local base a number of times using the traditional recursive subdivision surface process. We do not subdivide in terms of actual geometry vertices, but rather subdivide *symbolically* in terms of vertex labels. Symbolic subdivision results in an expression for each subdivided vertex in terms of local base vertices only.

Referring to Figure 2, let $v(s,i)$ be the ith vertex of the base face at subdivision level *s*. Note that in the figure, we have ordered the vertices of the center triangle in rows, from right to left, top to bottom, with the count starting at 0. For this example, we will subdivide only to level 2, but the process continues in a similar fashion to our maximum subdivision, level 4.

Suppose we want to compute $v(2,8)$ in the figure. Then, according to Loop Subdivision, we have

(1) $$v(2,8) = \frac{3}{8}v(1,2) + \frac{3}{8}v(1,4) + \frac{1}{8}v(1,5) + \frac{1}{8}v(1,1),$$

Note that in terms of base vertices, the expressions for the relevant level 1 subdivided vertices are

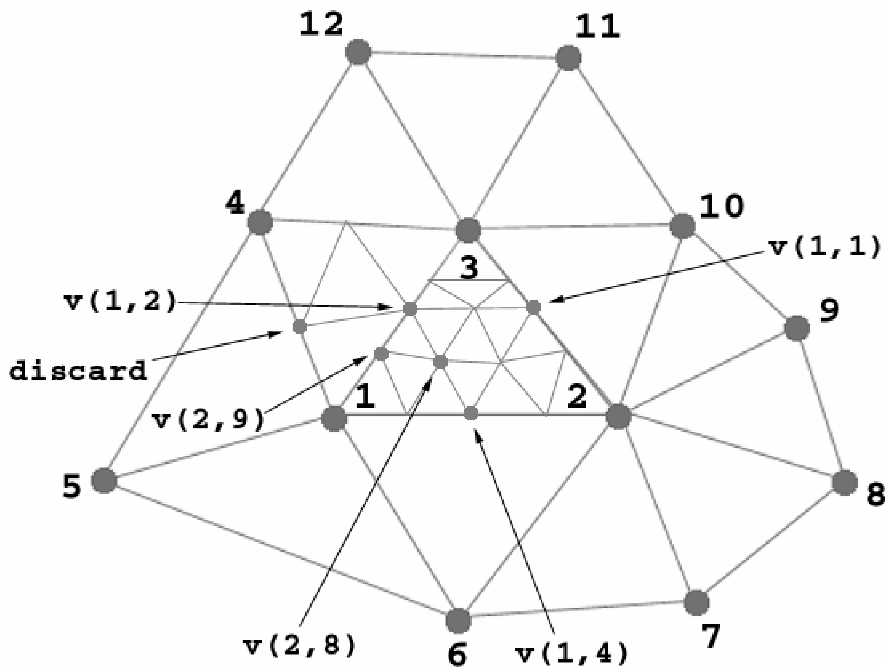
(2) $\quad v(1,2) = \frac{3}{8}C_1 + \frac{3}{8}C_3 + \frac{1}{8}C_4 + \frac{1}{8}C_2,$

(3) $\quad v(1,4) = \frac{3}{8}C_1 + \frac{3}{8}C_2 + \frac{1}{8}C_3 + \frac{1}{8}C_6,$

(4) $\quad v(1,1) = \frac{3}{8}C_3 + \frac{3}{8}C_2 + \frac{1}{8}C_{10} + \frac{1}{8}C_1,$

(5) $\quad v(1,5) = \frac{25}{40}C_1 + \frac{3}{40}C_3 + \frac{3}{40}C_4 + \frac{3}{40}C_5 + \frac{3}{40}C_6 + \frac{3}{40}C_2,$

where, for example, C_3 corresponds to the base vertex labeled "3" in the figure. By substituting (2)-(5) into (1) it is clear that have expressions for *v*(2,8) in terms of base vertex labels.



- Figure 2 -

We throw away expressions for any subdivided vertices not on the triangle T (the T label is removed from Figure 2). For example, in Figure 2, the vertex labeled "discard" is needed during the recursion to produce the symbolic expression for *v*(2,9), but it is not needed at run-time.

We use 4 subdivision recursions, which results in 256 triangles and 153 vertices for each triangle of the global base. The 153 vertices are represented as the coefficients in a linear combination of vertices of the local base.

Once all triangles of the global base have been subdivided using this process, we will have two pieces of data for each base triangle:

1. A sequence of indices of global base vertices. These global base vertices are the only ones relevant to the local base triangle.

2. A set of 153 coefficient sequences expressing the vertices of the subdivided triangle as linear combinations of the indexed base vertices.

At run-time, we can render our surface without any recursive subdivision, using hardware instructions designed to perform fast scalar products of vectors, because all vertices of our subdivided mesh are stored as coefficients of a small set of base vertices. Each subdivided vertex is simply the result of the application of a base vertex matrix to a coefficient vector.

$$\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \dots & \dots & \dots \\ x_{12} & y_{12} & z_{12} \end{bmatrix} \bullet \begin{bmatrix} c_{i,1} \\ c_{i,2} \\ \dots \\ c_{i,12} \end{bmatrix} = v_i .$$

Each vertex $v_i, 0 <= i <= 153$, from triangle T in can be expressed as a linear combination of the 12 local base vertices.

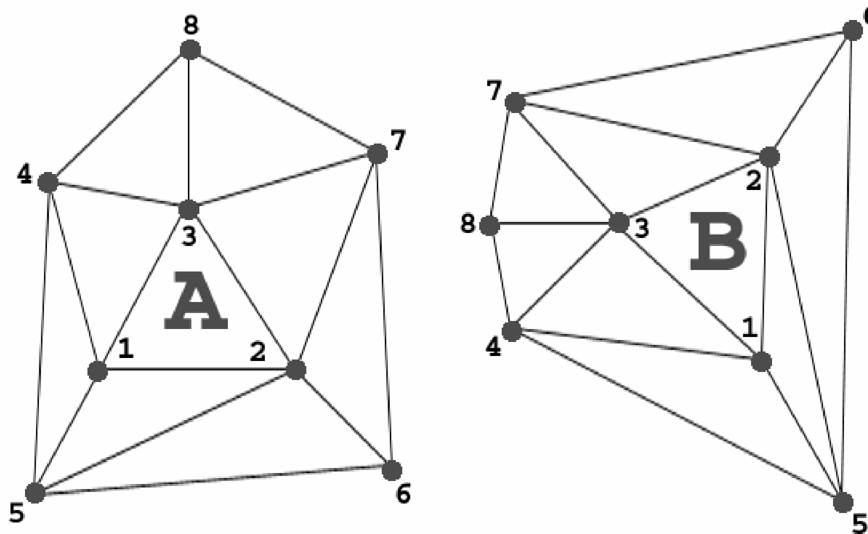
Without further pre-processing the space required for our representation would be prohibitively large, in fact far exceeding the data size of just explicitly storing the fully subdivided mesh using a naïve mesh representation. The remedy follows.

Recall that we have ordered the local base vertices in a consistent way. We can therefore keep a small data size by realizing that regardless of the spatial *geometry* given by the actual positions of the local base vertices, the global base triangles can be classified by *topology* of their local bases. Triangles whose local bases have the same topology will also have the same coefficient sequences. See Figure 3. This is the crux of our method. Two global base triangles have different topology only if one of the following is true:

1. The valences of any corresponding pair of triangle vertices differ.

2. The numbers of 1-neighborhood vertices belonging to any corresponding pair of edges differ.

3. The boundary status' (whether or not it is a boundary) of any corresponding pair of edges differ.



-Figure 3-
In spite of different vertex positions,
triangles A and B have the same classification.

So, to end the pre-process and construct the data file, we create a table of unique coefficient sequences. Then, for each triangle of the base mesh, we store only the indices of the relevant global base vertices, along with a single index indicating which unique coefficient sequence belongs to the triangle. In practice, the number of unique coefficient sequences is small (fewer than 10), even for massive meshes, because regular structure usually dominates. The data size can be further compressed by storing the coefficients for only the edges of each uniquely classified triangle, as the vertices on the interior are always linear combinations of vertices on the edges.

Our method allows us to do subdivision face-by-face over the base mesh. This implies that no global knowledge is needed, eliminating random data access over a large area. Moreover, the core loops are extremely simple, merely performing sums of scalar products. We have implemented our technique on the Playstation 2® with results fast enough for production use.

It should be noted that our method, VALENTE™ Subdivision, is a patent-pending technique of The Golden Gate Game Company.

Triangle Strips

If care is taken when ordering the vertices of subdivided triangles, we can easily output strips by proceeding along the “rows” of the subdivided triangle. For each strip, we advance two pointers, *top* and *bottom*. The process is most easily seen in the following code.

```
int             subdivLevel;
SUBDIV_VERT*    subdivVerts;
/* … */
int             rowCount=(1<<subdivLevel);
```

```
int           rowSize=1;
SUBDIV_VERT*  top=subdivVerts;
SUBDIV_VERT*  bottom=top+1;
int           parity=0;
while(rowCount--)
{
  int  rowVertCount=rowSize;

  StartOutputStrip();
  OutputStripVert(*(bottom++));
  while(rowSize--)
  {
    OutputStripVert(*(top++));
    OutputStripVert(*(bottom++));
  }
  rowSize+=2;
}
```

Geo-Morphing

Because we perform the subdivision on a face-by-face basis, geo-morphing is quite simple. For a vertex $v_{n,i}$ at a given subdivision level n, there is a spawning edge in subdivision level $n-1$ connecting two "parent" vertices, p_a and p_a. Suppose we want a geo-morph to subdivision level $n-w$, where $0 <= w <= 1$. Then we simply modify each vertex $v_{n,i}$ according to the following expression:

$$v_{n,i} \leftarrow (1-w)v_{n,i} + \frac{w}{2}(p_a + p_b).$$

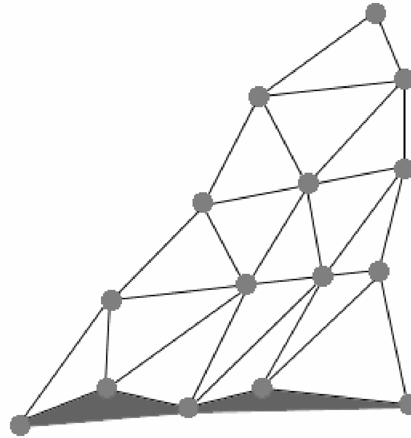
The correspondence of parent vertices to subdivided vertices is invariant from face-to-face of the base mesh.

Seams

A consequence of view-dependent level-of-detail control is that two adjacent triangles of the base mesh may be subdivided unequally. To avoid ugly cracks or "seams" in the resulting mesh, we must sew up the edges of some adjacent triangle pairs.

We make the assumption that no adjacent triangles will differ in subdivision by more than 1 level. Before processing a base mesh triangle, we decide the seam status of each edge. Then after the subdivided vertices are computed, we create a "saw-tooth" sequence of triangles for each edge that is labeled as a seam. For a given pair of triangles with unequal

subdivision, the seam is drawn along with the triangle with the higher subdivision level. The saw tooth will sew up the seam and prevent cracking. See Figure 4.



-Figure 4-
Assume that the bottom edge of the base face required a seam. It is formed in simple saw-tooth fashion by skipping odd edge vertices.

Textures

Subdivision must be applied to texture coordinates as well as geometry. DeRose's paper ([1]) on scalar fields goes into detail on this topic. DeRose proposes adding texture coordinates to the coordinates of the 3D geometry vectors and applying the subdivision scheme as usual. This process is fine when the entire base mesh has the same texture and when texture coordinates are continuous, but for games we must support cases where two adjacent triangles have continuous geometry but discontinuous texturing. See Figure 5 for an example.

Our solution is to partition the base mesh into one or more texture space meshes. Careful ordering of faces in the geometry base mesh can indicate the one-to-one correspondence between geometric faces and texture faces, so no table lookup is necessary.

We apply the same pre-process and run-time method to the texture meshes that we apply to geometric meshes. Applying the VALENTE™ method to both a texture face and a geometric face produces two sequences of vertices, one made of texture coordinates, and the other made of 3D spatial vertices. These are in direct correspondence and can easily be output as triangle strips.

-Figure 5-
Example of one base mesh that would require several texture meshes. The "T", "R", and "F" quads are each textured with a different bitmap, and the texture coordinates are not contiguous.

Normals and Lighting

Conventional techniques for subdivision surfaces include computation of surface normals as outer products of tangent vectors. We could represent the tangent vectors as a series of linear combinations of base vertices, just as we do for geometric vertices and texture coordinates. In fact we implemented this approach in an early version of the technology. The problem is that it triples both the data size and the processing time because the coefficient sequences for tangents differ from those for vertices.

We cannot ignore subdivided vertices completely and simply interpolate lighting of the base mesh vertices, because this would make it impossible to spot light the interior of a base mesh face, even at full subdivision. Our solution is to take an approach that combines simple interpolation with vertex-by-vertex lighting.

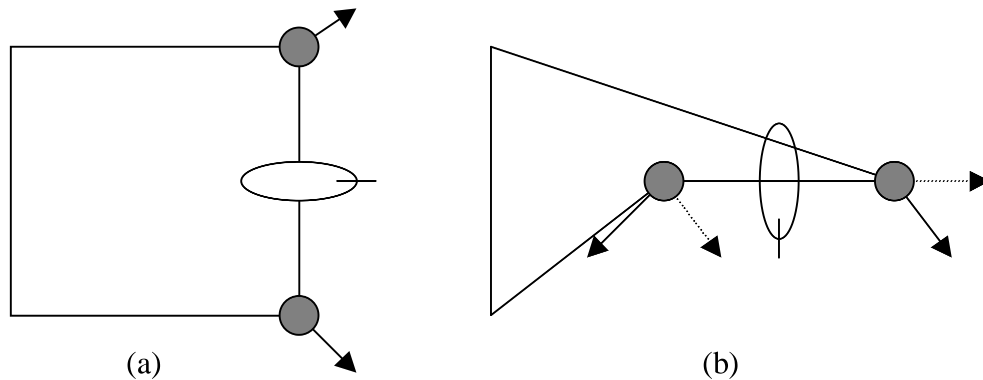
Along with geometric position, we store the surface normal for each base mesh vertex. At run-time, for each light source and for each base mesh vertex, we compute the diffuse light intensity using the standard dot product approach. For each subdivided face, we interpolate these dot products linearly. Then we apply spot lighting and distance attenuation to each vertex of the subdivided face, and combine those results from the interpolated surface normal values. While perhaps not mathematically correct, the images produced are visually pleasing and do not require tangent computation or storage for subdivided vertices.

Character Animation

To animate a character, we simply apply traditional bone control to the base mesh vertices. Along with transforming each base vertex, we also rotate the corresponding normals. The

additional processing required for weighted multi-bone control is negligible in the case of subdivision surfaces, because it need only be done for base vertices.

There are some degenerate cases to be aware of. As stated above, we rotate the normals of the base mesh and use the results for lighting. There are cases when this method will produce unexpected results. See Figure 6.



-Figure 6-

The filled circles represent base mesh vertices, and the ellipse represents a bone that controls both vertices. In (a), the normals are correct for both vertices. In (b), after a rotation of 90 degrees about the bone, both normals are incorrect. The dotted lines represent the correct normals after rotation. We have no clean way around this problem, but production artwork seldom includes animations calling for rotations that cause such extreme skewing of the mesh.

Conclusion

We have presented a novel technique for high speed rendering of subdivision surfaces. This technique has been implemented on the Playstation 2® has proven practical for use in a console game. The core algorithms are simple enough that they could be implemented in future hardware.

Simple core algorithms, however, do not necessarily imply easy engine development. We found that most of our time was spent writing 3DS Max plug-ins just to produce the data. In fact, our future work will certainly include more plug-in development, especially relating to converting an arbitrary dense mesh to a form compatible with VALENTE™ Subdivision.

References

[1] DeRose, T., Kass, M., Truong, T. Subdivision Surfaces in Character Animation. *Proceedings of the 25th annual conference on Computer Graphics*, 1998, Pages 85 – 94. (**PAT. # 6,037,949, assignee is Pixar).**

[2] Lee, A., Moreton, H., and Hoppe, H. Displaced Subdivision Surfaces. *Computer Graphics Proceedings, SIGGRAPH 2000*, 2000.

[3] Lounsbery, M., DeRose, T., Warren, J. Multiresolution Analysis for Surfaces of Arbitrary Topological Type. *ACM Transactions on Graphics*, Vol. 16, No. 1, January 1997, Pages 34-73.

[4] Pulli, K., and Segal, M. Fast Rendering of Subdivision Surfaces. *The art and interdisciplinary programs of SIGGRAPH '96 on SIGGRAPH '96 visual proceedings*, 1996, Page 144. (**PAT. # 6,078,331, assignee is SGI).**

[5] Zorin, D. Subdivision for Modeling and Animation. *SIGGRAPH ’98 Course Notes*, 1998. Located on the web at www.mrl.nyu.edu/~dzorin/sig98course/part2/.