Evaluation of Subdivision Surfaces on Programmable Graphics Hardware

Jeff Bolz Peter Schröder
Caltech

Abstract

High-order smooth surface primitives, such as subdivision patches for example, are attractive for the modeling of free-form surfaces. In contrast to meshes they require only a few control points to specify large sections of a surface. Unfortunately, much of this bandwidth advantage is lost when such surfaces have to be tessellated on the CPU prior to transmission over the graphics bus and rendering on the graphics card. For surfaces built through linear combination of basis functions it is possible to precompute tessellations and use these to evaluate the surface at runtime in a simple computation performed entirely on a programmable graphics processor (GPU). The improved bandwidth requirements—only control points need transmission during animation, for example—coupled with the high performance of GPUs, allows us to achieve tessellation rates up to 24 million vertices per second on a 500 MHz GeForce FX.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Graphics processors;

Keywords: GPU Computing, Subdivision, Catmull-Clark, efficient, evaluation

1 Introduction

Achieving good performance in interactive graphics has until recently implied the use of low polygon count models. Smooth surface primitives, such as spline or subdivision patches, have often been limited to high precision (*e.g.*, CAD) and high quality (*e.g.*, special effects) applications at the expense of realtime performance. Among these primitives, subdivision surfaces [Zorin and Schröder 2000] are of particular interest (see Figure 1) as these are now widely used in high end modeling packages [Alias|Wavefront 2002], animation production [DeRose et al. 1998], realtime authoring kits [MacroMedia 2002], and have become part of the MPEG4 standard [2002].

Since commodity graphics hardware is tuned for triangle rendering, any high level primitive must be tessellated prior to scan conversion to take advantage of the rasterization hardware. This consumes both CPU resources for evaluation and bus resources for transmission of the resulting triangles to the GPU. Present generation graphics adapters, such as the ATI9700 or nVidia GeForce FX, have raw rasterization performance which far outstrips the bus bandwidth available to "feed" the cards [Moreton 2001]. Since the mismatch of bandwidth to compute performance is expected to continue to get worse [Semiconductor Industry Association 2002; Khailany et al. 2003], tessellation must be performed on the GPU to realize the benefit of higher rasterization throughput for smooth surface primitives. One avenue to achieve this is through the inclusion of forward difference accumulator units [Vlachos et al. 2001; Moreton 2001]. These could be used to tessellate subdivision surfaces [Peters 2000; Bischoff et al. 2000], but the required specialized hardware is not available in the ATI9700 or GeForce FX.

Traditional approaches to subdivision perform depth or breadth first subdivision on a (sub-)mesh or use direct evaluation [Stam 1998]. Depth first evaluation of pairs of Loop [1987] patches was first described by Pulli and Segal [1996], who used the microcoding support on early generation SGI rendering hardware to implement this in a stored program. Since depth first recursion requires a stack of rows of vertices, their algorithm does not easily map onto streaming oriented architectures. Exact evaluation as described by Stam could be implemented in a fragment program. Supporting all the possible cases of tags, corners, and boundaries [Biermann et al. 2000] however, would lead to a rather complex program [Zorin and Kristjansson 2002]. Instead we pursued an entirely different and much simpler approach.

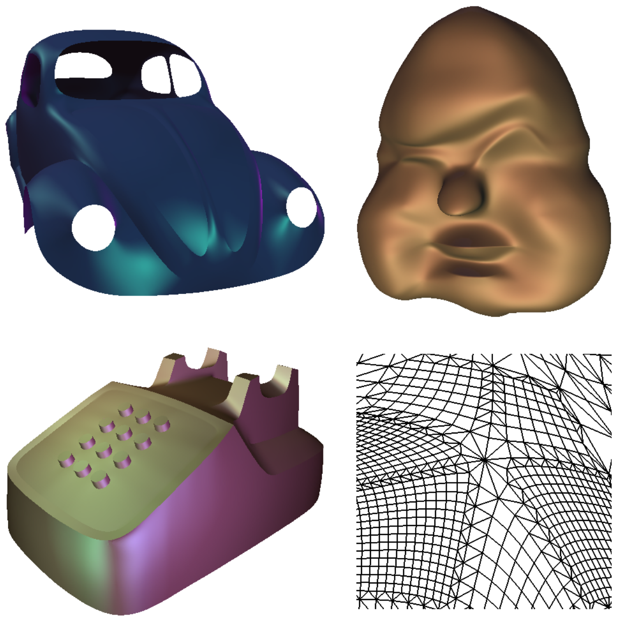
Figure 1: *Examples of Catmull-Clark subdivision surfaces tessellated with our implementation on the GeForce FX. Each consists of several hundred patches of different valence. The VW and phone have many tags in different configurations. The closeup shows the mesh structure of a watertight adaptive tessellation (subsection of the head model near corner of left eye).*

1.1 Linear Combination of Tessellations

The basic observation we exploit is that *the tessellation of a linear combination of basis functions is equivalent to the linear combination of tessellations of basis functions*: our algorithm is applicable to *any* surface built from linear combinations of basis functions. It also results in an embarrassingly simple fragment program. In the case of Catmull-Clark [1978] subdivision surfaces—with the full complement of tagged edges and vertices [Biermann et al. 2000]—Bolz and Schröder [2002] successfully used this approach in a carefully crafted CPU implementation and achieved 1.8 flops/cycle. Brickhill [2001] similarly used precomputed powers of the subdivision matrix for Loop surface tessellation on PS2 hardware[1].

We build on this earlier work to design an entirely GPU based algorithm. In particular we use the basis function tables provided by Bolz and Schröder (http://multires.caltech.edu/software/fastsubd/). However, in contrast to all earlier work we *guarantee* a watertight tessellation—no pixel dropouts at patch boundaries. This is achieved through consistent arithmetic on patch boundaries and a

[1]No performance numbers were reported.

novel preprocess which enforces exact bit equivalence of floating point numbers on edges under all *local symmetries* of the control mesh for all basis functions, valences, and tag statuses (smooth interior, dart, boundary, crease, convex corner, concave corner).

During animation, and after the initial download of basis function tessellation "textures," our fragment program requires only the transmission of control points. We can saturate the fragment shader hardware at marginal AGP bus bandwidth consumption. Our algorithm supports *adaptive tessellation* on a per patch basis (Figure 1) and generates vertices every 24 to 60 instructions leading to approximately 24 million vertices per second on nVidia's GeForce FX (see Section 3.1).

2 Algorithm Setup

For purposes of exposition we give here a very brief review of the approach of Bolz and Schröder and refer further details to [2002] (see also Brickhill [2001] for the Loop case). A tutorial overview of subdivision can be found in [Zorin and Schröder 2000], while details of the particular rules used are documented in [Biermann et al. 2000]. A prototype implementation of the latter is available at http://www.mrl.nyu.edu/biermann/subdivision/. This code formed the basis for the reference implementation of subdivision surfaces in MPEG4 [2002] and was used to generate the tables we use.

A Catmull-Clark subdivision surface is specified by a polyhedral 2-manifold (with boundary) mesh, possibly containing tagged vertices (dart, corner) and edges (boundary, crease). WLOG we may assume that all faces are topologic quadrilaterals. In order to separate irregular vertices—those with other than four (two on the boundary) faces incident—one step of subdivision is performed up front. Each resulting patch contains exactly one vertex of the input mesh. We will refer to this vertex as the *corner* of the patch and orient it to be at the local origin. Note that we do not assume a consistent orientation. The *control set* of a patch is made up of the control points in the 1-ring neighborhood of the given quad. The limit surface can be evaluated on an arbitrary but fixed tessellation by linearly combining tessellations of the basis functions in the control set. Of these only the section with support on the patch is required. Bolz and Schröder [2002] used a regular grid of $2^5 + 1$ samples on a side. Fewer levels of subdivision are supported through subsampling while finer subdivision is achieved through additional coarse level subdivisions on the CPU.

The number of unique basis functions is unbounded so no set of tables sufficient for all inputs can be computed ahead of time. We use the tables of limit positions and tangents provided by Bolz and Schröder in their prototype implementation. These cover smooth and dart interior vertices, boundaries and creases, as well as convex and concave corners, with valences ranging from one to twelve. For a full fledged implementation one could either store the needed tables with a given control mesh—not unlike what is done with standard texture maps—or create tables lazily upon mesh creation.

An important issue that was not addressed by earlier work is that of pixel dropouts between neighboring patches.

2.1 Watertight Surfaces

Patches of the surface are tessellated independent of each other. To avoid cracks and pixel dropouts, corresponding vertices on the boundaries of patches must match exactly. Cracks can result when two neighboring patches are tessellated at different rates. In this case the boundary is tessellated at the lesser of the rates of the two incident patches and the transition strip zippered accordingly (see Figure 1)

A more problematic issue is the required *exact* bit by bit equality of the floating point vertex coordinates which either patch produces on a joint boundary. For *any* finite precision this can only be guaranteed if the vertices which are supposed to match are computed through linear combination of the same values with the same coefficients *in the same order* (and the same initial state of the FPU). Consistent ordering can be achieved by using a total order, *e.g.*, file order, on all control points when performing linear combinations on a patch by patch basis. The coefficients (*i.e.*, control points) of these linear combinations are shared by definition. More tricky are the pre-tessellated basis functions. If the code which produces these tessellations is constructed with appropriate care one could ensure that exact bit equivalence holds. In general however, this will not be the case.

We propose a different solution which works for any set of such tables independent of how they were produced in the first place. It is based on enforcing matching floating point values on edges under the group of topologic symmetries of the (tagged) control set.

2.1.1 Symmetry Enforcement

Recall that we store for each patch type (valence, tag status) a set of tables. Each such table is associated with a basis function whose support overlaps the given patch. For two neighboring patches a given basis function will appear as a particular—in general, differently indexed—table in the control set of each patch. We need to ensure that these tables, which were computed independently, have the same entries on the shared edge.

Figure 2 illustrates a simple example of this. Two abutting patches of valence m and n respectively share a common edge. The control point highlighted in yellow contributes to the shared edge (bold). When considering each patch by itself the same basis function appears in each local picture in different locations.

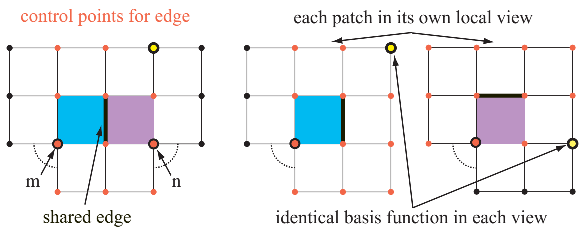
control points for edge
each patch in its own local view
m
n
shared edge
identical basis function in each view

Figure 2: *A simple example demonstrating a given basis function in different positions in the control sets of two patches.*

Bolz and Schröder [2002] discussed the issue of basis function table uniqueness. For example, many basis functions are identical (modulo symmetries) *independent* of valence. We must now enumerate the table *edges* which are identical modulo all applicable symmetries. Once this task is performed, a simple pre-process on all tables, *e.g.*, taking the minimum of all "equal" values, ensures exact bit by bit matches.

To enumerate the number of different cases we observe that of the four edges bordering a patch the *off-corner* and *on-corner* edges form disjoint sets under symmetries of the control set—an on-corner edge of one patch can never be an off-corner edge of another. Off- and on-corner edges can share values at endpoints, but this does not impact our algorithm to enforce consistency. For off-corner edges there are only four unique classes (Figure 3, left) *across all valences and all tags* (boundary, crease, concave corner, convex corner) with one exception. An edge of a patch adjacent to *any* tagged edge (Figure 3, center) has four separate classes (though they are the same independent of valence and tag status). In each class there are two, four, or eight control points corresponding to the three symmetries left/right, up/down, and mirror about the line $x = y$. Some of these symmetries may be degenerate. Figure 4 shows all eight cases in one class for a particular example. On-corner edges are influenced by the control points of the irregular

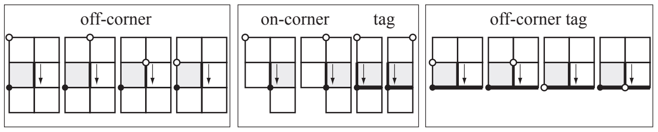

Figure 3: *Enumeration of representatives of all classes of basis functions (hollow dot) which yield the same set of values on a given directed edge (arrow) near the corner (solid dot). Edge tags are shown in bold.*

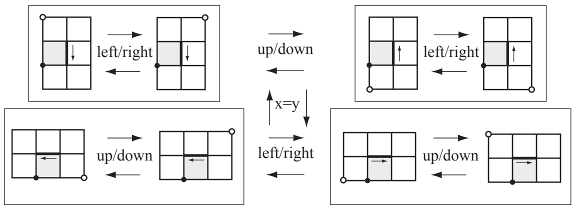

Figure 4: *Symmetries of directed off-corner edges and their control sets using a single basis function (hollow dot) as an example.*

vertex, its 1-ring, and a set of six (in general) control points from the 2-ring of the irregular vertex. The latter fall into two classes *independent* of valence and tag status—with the lone exception—as in the off-corner case–of a tagged edge (Figure 3, right) which gives rise to its own classes (again, independent of valence or type of tag).

Finally, the control points in the 1-ring are subject to a symmetry under rotation by $2\pi/k$ and a mirror symmetry about the edge under consideration (Figure 5). Except when the symmetry is degenerate each class has four members. All these classes are specific to a particular valence and tag status. This completes the enumeration

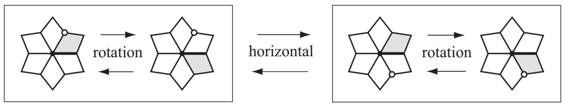

Figure 5: *The symmetries of the on-corner basis function edges (hollow dot denotes basis function). There is a unique set of classes for each vertex and valence. If there is a tagged edge, then the classes are also dependent on the location of the edge with respect to the tag,* i.e.*, the presence of a tag breaks some of the symmetries.*

of all classes. The proof of this fact follows from a simple argument. Fix any edge shared by two patches in the control mesh. That edge *must* have the same values from the point of view of either side. This holds no matter what the valence of either patch, or whether there is a tag present (a tag on a shared edge, must of course be shared, *etc.*)

We implemented a simple program which enforced these symmetries in a pre-process performed on the tables of Bolz and Schröder. It reads all tables, maps edges onto their symmetry classes, and for each edge writes back the minimum of all values found in a given slot. For tangent (derivative) tables the same symmetries hold *with the proviso* that d/dx maps to d/dy under the $x = y$ symmetry.

3 Implementation

Each patch is evaluated by rendering a 2^n+1 by 2^n+1 quadrilateral ($n \leq 5$) to a vertex buffer. Each pixel corresponds to a vertex of the tessellated patch. The fragment program computes the weighted sum of the control points passed to the program in constant registers with the weights fetched from the appropriate tessellation textures. Normals can be computed by using two additional tessellation textures for the associated partial derivatives. Among other uses, normals can be employed to apply a displacement map to the surface tessellation [Lee et al. 2000].

Control points are passed into the fragment program in the order induced by the total ordering. The associated basis function values are looked up in a texture using a constant offset—passed in constant registers—and an interpolated offset appropriate for the given fragment (Figure 6). Each patch type, *i.e.*, valence and tag status,

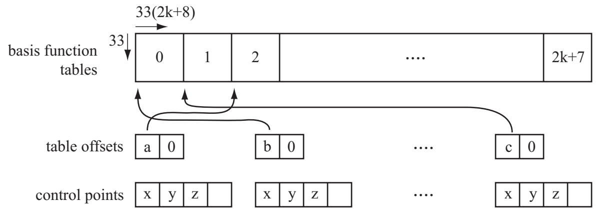

Figure 6: *Layout of basis function tessellations within a texture. Control points map to the right basis function through an associated address.*

has its own texture containing the $2k + 8$ tables associated with its control set. The unrolled loop is optimized over two iterations, since they can share an addition

```
R1.xyzw = OffSet01.xyzw + TEXCOORD0.xyxy;
R1.x = TEX0[R1.xy]; R0 += R1.x * cp0
R1.x = TEX0[R1.zw]; R0 += R1.x * cp1
```

where `cp0` and `cp1` are two control points; `OffSet01` contains the constant texture offset into `TEX0` in `xy` and `zw` respectively; and `TEXCOORD0` is the interpolated texture coordinate for this patch.

In a second pass this buffer is rendered as a standard vertex buffer triangle mesh. Since current driver releases do not yet expose the “render to vertex buffer” functionality our implementation *currently* requires a round trip to the CPU to render the vertex buffer.

If global ordering of the bases is not required—giving up on one of the requirements for bit consistency—the above code can be further optimized by simply adding control points in the order their associated tables are stored within the texture. Additionally the basis functions can be packed into a 4-channel float texture which reduces the number of instructions even further. With this four iterations of the loop can be written as

```
R0 = 33*i + TEXCOORD0; R1  = TEX0[R0];
R0 += R1.x * cp0; R0 += R1.y * cp1;
R0 += R1.z * cp2; R0 += R1.w * cp3;
```

This reduces the number of instructions per control point by 40%. However, a simple instruction count does not fully determine performance. Texture instructions may stall due to bandwidth limitations, and some instructions may inherently require more than one cycle to complete.

Patches are rendered using the vertex buffer and indexed primitives, regular quad-strips in the interior and triangles to stitch patches together at the edges. All necessary indices are precomputed. The textures containing the basis functions store the finest tessellation, for five levels. To evaluate the surface to fewer levels suitably chosen texture coordinates can be used to subsample the texture: for $n < 5$ levels of subdivision appropriate coordinates (OpenGL) range from $.5 - 2^{4-n}$ to $32.5 + 2^{4-n}$.

A sophisticated adaptivity criterion for this tesselation algorithm uses upper bounds for the curvature of patches to determine the level of tesselation [Grinspun and Schröder 2001].

3.1 Performance

We have implemented the fragment programs described above and measured their performance on actual GeForce FX hardware using a number of different geometric models. They contain many different valences and tags in a variety of combinations (Figure 1). The presence of tags has no impact on performance, while the valence of a patch does influence the instruction count per tessellated vertex (for detailed flop counts of different evaluation methods see [Bolz and Schröder 2002]). As representative examples we consider valence 4 and 9.

The valence 4 (9) case requires composition of 16 (26) basis function tables. Achieving bit accurate results requires 40 (65) instructions because of the ordering constraint. If bit accurate results are not required the tighter instruction sequence results in 24 (40) instructions. Both sequences require two registers. Computing normals as cross products of tangent table evaluations adds an additional 52 (84) instructions in a second pass.

To estimate achievable performance we assume the use of four parallel fragment pipelines (*i.e.*, instruction issues per cycle), but discount these by a factor of two for the number of float registers used. For a concrete example consider 500 MHz and 1 M instruction issues per second for a theoretical peak throughput of $1000/24 \approx 41.66$ (not bit accurate; valence 4) to $1000/65 \approx 15.38$ (bit accurate; valence 9) million vertices per second. In practice these numbers cannot be achieved. For example, assume that round robin scheduling of functional units requires large groups of fragments to execute the same instruction sequence. A switch to another patch then requires a "flush" driving down utilization. For example, for five levels of subdivision each patch generates $33^2 = 1089$ vertices. If the batch size is 512 fragments this would result in only $1089/(3*512) \approx 71\%$ utilization. We determined the utilization ratio due to (a hypothesized) batch size experimentally, by varying the size of the quadrilateral rendered into the pbuffer. The number of cycles *actually* consumed was verified by rendering the same patch 1000 times into the same pbuffer. In particular this experiment verified that there are no detrimental effects due to memory latency. Assuming this utilization and the experimentally observed number of instructions per cycle, patches can be tessellated at a rate of between 8 (valence 9; bit accurate) and 24 (valence 4; not bit accurate) million vertices per second. Running tests on actual hardware, we observed a 23% performance penalty for the bit-consistent vs. non bit-consistent code. Note that this *better* than the penalty predicted by instruction sequence length differences (40%) alone.

The numbers have been established under synthetic test conditions since the current OpenGL drivers have a very high software overhead associated with pbuffer switches—only about 200 pbuffer switches per second are achievable. Each new patch requires such a switch. While the pipeline flush is of course unavoidable—and we accounted for it—the software reload of the entire OpenGL state currently performed by the driver can be entirely eliminated[2]

As a concrete example of tessellation performance consider the VW model. It has about 300 patches which can be tessellated at subdivision depth five in approximately $1/60$th of a second. Rendering time would be additional and depends of course on shading and lighting parameters.

4 Discussion and Future Work

The simplicity and high speed of table driven tessellation makes the method we described very attractive for modern GPUs. The speed is mostly due to the great regularity in the memory access and computation patterns. This is also the reason it was found to be attractive for CPUs [Bolz and Schröder 2002]. However, GPUs get to benefit relatively more since their architecture is highly tuned for algorithms of this kind.

We improved earlier work by describing a pre-process manipulation of the tables which *guarantees bit accurate results* at patch boundaries when coupled with a global composition order. The latter unfortunately comes at a performance penalty in our implementation due to addressing constraints in the current assembly language.

Our algorithm is applicable to *any* smooth surface type which is built from linear combinations of basis functions, for example, NURBS. The algorithm should scale well with increasing fragment processing power in future GPUs.

An interesting question for future research is whether more traditional subdivision surface construction through breadth first or depth first recursive refinement can have efficient implementations on GPUs. The advantage of this would be lower overall flop count (albeit only by a small factor [Bolz and Schröder 2002]) and the possibility of more fine-grained control over adaptive tessellation. The main challenge in this is the management of the irregular topology of the control mesh as it becomes refined.

On the hardware side "render to vertex buffer" functionality must be exposed in the drivers to reap the full benefit of our method and the unnecessary overhead associated with pbuffer switches needs to be removed when the OpenGL state does not in fact change.

Acknowledgment This work was supported in part by NSF (DMS-0220905, DMS-0138458, ACI-0219979), the DOE (W-7405-ENG-48/B341492), nVidia, the Center for Integrated Multiscale Modeling and Simulation, Alias|Wavefront, Pixar, and the Packard Foundation. Special thanks to Matt Papakipos, Nick Triantos, David Kirk, and all speakers and students in the "Hacking the GPU" class (Caltech, Fall 2002). The models are courtesy Khrysaundt Koenig (head), Leif Kobbelt (VW), and Patrick Mullen (phone).

References

ALIAS|WAVEFRONT, 2002. Maya. http://www.aliaswavefront.com/.

BIERMANN, H., LEVIN, A., AND ZORIN, D. 2000. Piecewise Smooth Subdivision Surfaces with Normal Control. In *Proceedings of SIGGRAPH*, 113–120.

BISCHOFF, S., KOBBELT, L. P., AND SEIDEL, H.-P. 2000. Towards Hardware Implementation Of Loop Subdivision. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 41–50.

BOLZ, J., AND SCHRÖDER, P. 2002. Rapid Evaluation of Catmull-Clark Subdivision Surfaces. In *Proceedings of Web3D*, 11–17. Software available for download.

BRICKHILL, D. 2001. Practical Implementation Techniques for Multi-Resolution Subdivision Surfaces. In *Game Developers Conference*.

CATMULL, E., AND CLARK, J. 1978. Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes. *Computer Aided Design 10*, 6, 350–355.

DEROSE, T., KASS, M., AND TRUONG, T. 1998. Subdivision Surfaces in Character Animation. In *Proceedings of SIGGRAPH*, 85–94.

GRINSPUN, E., AND SCHRÖDER, P. 2001. Normal Bounds for Subdivision-Surface Interference Detection. In *Proceedings of IEEE Scientific Visualization*, 333–340.

KHAILANY, B., DALLY, W. J., RIXNER, S., KAPASI, U. J., OWENS, J. D., AND TOWLES, B. 2003. Exploring the VLSI Scalability of Stream Processors. In *Proceedings of the Ninth Symposium on High Performance Computer Architecture*.

LEE, A., MORETON, H., AND HOPPE, H. 2000. Displaced Subdivision Surfaces. In *Proceedings of SIGGRAPH*, 85–94.

LOOP, C. 1987. *Smooth Subdivision Surfaces Based on Triangles*. Master's thesis, University of Utah, Department of Mathematics.

MACROMEDIA, 2002. Shockwave Player. http://www.macromedia.com/shockwave/.

MORETON, H. 2001. Watertight Tessellation Using Forward Differencing. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 25–32.

MPEG 4 COMMITTEE , 2002. MPEG 4 Standard. http://mpeg.telecomitalialab.com/standards/mpeg-4/mpeg-4.htm, March.

PETERS, J. 2000. Patching Catmull-Clark Meshes. In *Proceedings of SIGGRAPH*, 255–258.

PULLI, K., AND SEGAL, M. 1996. Fast Rendering of Subdivision Surfaces. In *Rendering Techniques '96*, Springer Wien, 61–70.

SEMICONDUCTOR INDUSTRY ASSOCIATION , 2002. International Technology Road Map for Semiconductors. http://public.itrs.net/, December.

STAM, J. 1998. Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values. In *Proceedings of SIGGRAPH*, 395–404.

VLACHOS, A., PETERS, J., BOYD, C., AND MITCHELL, J. L. 2001. Curved PN Triangles. In *Proceedings of Symposium on Interactive 3D graphics*, 159–166.

ZORIN, D., AND KRISTJANSSON, D. 2002. Evaluation of Piecewise Smooth Subdivision Surfaces. *The Visual Computer 18*, 5-6, 299–315.

ZORIN, D., AND SCHRÖDER, P., Eds. 2000. *Subdivision for Modeling and Animation*. Course Notes. ACM SIGGRAPH.

[2]Efforts to remove pbuffer switching overhead are under way.