Rapid Evaluation of Catmull-Clark Subdivision Surfaces

Jeffrey Bolz Peter Schröder

Caltech, Pasadena, CA

Abstract

Using subdivision as a basic primitive for the construction of arbitrary topology, smooth, free-form surfaces is attractive for content destined for display on devices with greatly varying rendering performance. Subdivision naturally supports level of detail rendering and powerful compression algorithms. While the underlying algorithms are conceptually simple it is difficult to implement player engines which achieve optimal performance on modern CPUs such as the Intel Pentium family.

In this paper we describe a novel table driven evaluation strategy for subdivision surfaces using as an example the scheme of Catmull and Clark. Cache conscious design and exploitation of SIMD instructions allows us to achieve nearly 100% FPU utilization in the inner loop and achieve a composite performance of 1.2 flop/cycle on the Intel PIII and 1.8 flop/cycle on the Intel P4 including all memory transfers. The algorithm supports tradeoffs between cache size and memory bus usage which we examine. A library which implements this engine is freely available from the authors.

1 Introduction

Subdivision surfaces have proven to be a useful modelling tool and are now part of all standard modelling packages (*e.g.*, 3DMax, Maya, Softimage, Mirai, Lightwave, etc.). However, their use in realtime applications such as games has been lagging because previous algorithms for their evaluation were too computationally intensive to run complex models at high frame rates with only moderate resources.

Subdivision engines are generally implemented the same way the corresponding subdivision scheme is defined, *i.e.*, as a recursive process that inserts new vertices into the mesh, refines existing point positions, and updates the connectivity [19] (see Figure 1). The associated data structures are often based on quadtrees for maximum flexibility when performing adaptive evaluation and involve many pointer indirections. Codes built on this basis do not perform as well as one might hope based on a simple flop count. Careful profiling reveals that the CPU is typically not fully utilized because it is often waiting for data to be transferred from memory due to the repeated pointer indirections.

Our approach was designed with the goal of eliminating memory latency delays and taking advantage of the CPU cache. Furthermore, the data is organized to take full advantage of Single-Instruction, Multiple-Data (SIMD) instructions [7]. As a result, our optimized engine achieves approximately 1.2 flops/cycle on a PIII and 1.8 flops/cycle on the P4.

Web3D'02, February 24-28, 2002, Tempe, Arizona, USA.

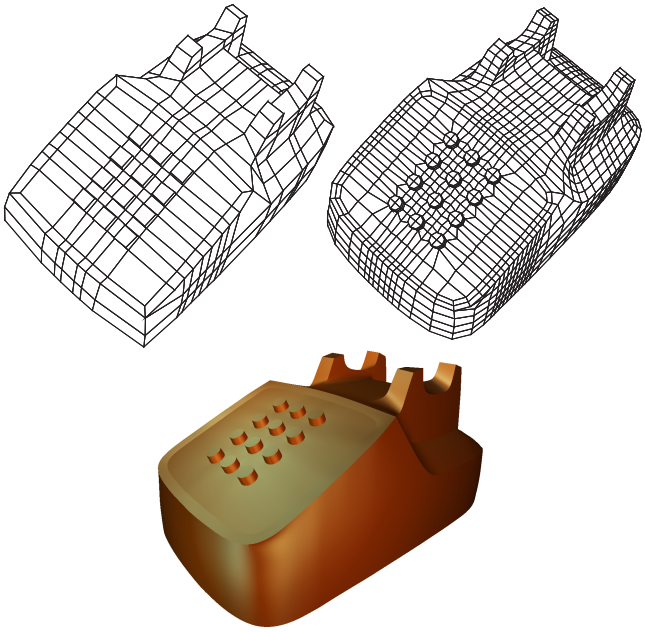

Figure 1: *A control mesh for a phone, its first subdivided level and the shaded limit surface.*

Previous work There are in essence four different approaches to the evaluation and rendering of subdivision surfaces: (I) recursive evaluation; (II) direct evaluation; (III) reduction to the regular setting; and (IV) pre-tabulated basis function composition.

Recursive evaluation based on repeated application of subdivision stencils is the most direct implementation of the standard definition of subdivision [19]. Zorin *et al.* [20] used traditional quadtrees [15] with breadth first evaluation. This setup is the most flexible. It easily supports adaptive rendering and multiresolution surfaces, but requires significant effort to achieve good performance because of the many pointer indirections. Neighbor finding in particular is expensive and hard to optimize [9]. Some of these issues can be reduced through the use of statically sized arrays at each level of the quadtree [6]. Recursive evaluation can also be implemented in a depth first fashion [13], which tends to have a much smaller memory footprint, making it attractive as a basis for hardware implementation [14].

Instead of evaluating the surface recursively one may also evaluate it directly at arbitrary parameter values. Such a strategy was first demonstrated by Stam [16] and more recently extended to piecewise smooth subdivision surfaces [18]. These evaluators are suitable for very general surface tessellation techniques and are employed in the Maya modelling package. While they have the smallest memory footprint, their performance on modern CPUs is unclear since their memory traffic has not been analyzed yet.

Subdivision schemes which are derived from splines can be evaluated by repeatedly splitting off regular sections of the surface and evaluating these with standard polynomial evaluators. The most efficient method for this purpose is based on forward differencing [11]. Pixar's Renderman uses this approach in software [5] for Catmull-Clark [4] surfaces, while Bischoff *et al.* [2] proposed

a hardware solution for Loop [12] surface rendering. While forward differencing is asymptotically the most efficient approach, recursive subdivision is required around irregular vertices with significant setup costs for each regular patch which is split off. Forward differencing also requires very careful implementation since it is numerically unstable. The fourth approach, pre-tabulated basis function composition, is explored in this paper.

Contributions Most of the previous work either proposed hardware solutions or was primarily concerned with high level optimizations such as adaptive tessellation, view dependent rendering, or breadth first versus depth first evaluation. In contrast we focus on achieving maximum performance on standard graphics hardware together with a modern, general purpose CPU, *e.g.*, the Intel Pentium family. On such processors the key to high performance is careful attention to caching issues: memory references are quite expensive, while on-chip computation is relatively cheap. To optimally exploit this setup our method tessellates *basis functions* in an offline process using the state-of-the-art extended Catmull-Clark subdivision rules of Biermann *et al.* [1]. The given subdivision limit surface—together with associated limit tangents and texture coordinates—is then evaluated at runtime as a linear combination of these tabulated basis functions, each weighted by the appropriate control point in the input control mesh[1]. We analyze the issues involved in implementing this robustly while minimizing the number of unique basis functions which need to be tabulated. The resulting tables fit well into cache and the remaining computations execute very efficiently. The algorithm is broadly applicable and we provide a reference implementation available for download.

2 Approach

Before describing our algorithm in detail we fix some notation and give the mathematical basis for our approach. While we focus on Catmull-Clark subdivision in this paper the same ideas carry over with little change to all other subdivision methods, be they based on triangles or quadrilaterals, approximating or interpolating.

2.1 Catmull-Clark Subdivision

The input to the subdivision algorithm is a *control mesh*, which must be a topological 2-manifold possibly with boundary (for non-manifold subdivision see [17]). This mesh may consist of faces with arbitrary *degree* (number of bounding edges) and vertices with arbitrary *valence* (number of incident faces). For simplicity we assume that all faces of the mesh are quads. If this is not the case, one step of standard Catmull-Clark subdivision converts an arbitrary polygon mesh into one consisting of only quads. To allow for creases and corners, edges respectively vertices can be tagged. Corners, which are interpolating, may be convex or concave, the two cases requiring different rules.

A vertex with one incident crease edge is a "dart" vertex. A vertex with two incident creases may be either a smooth crease or a corner. A vertex with more than 2 incident creases must be a corner. At a corner, the creases partition the neighborhood of the tagged vertex into sectors which can each be tagged as convex or concave. A sector is not influenced by the topology or tags of another sector, so when discussing a corner vertex one only considers the two creases which bound the sector. Similarly, for crease vertices we may separate the two sides of the crease and treat them independently.

Subdivision proceeds by quadrisecting each face and assigning point positions to each vertex in the finer mesh. These positions are averages of the point positions in the coarser mesh and given in the form of *stencils* (see Figure 2). For more details on the rules and the reasoning behind the weights we refer the reader to the original paper by Biermann *et al.* [1]. Notice that all newly created vertices have valence four, *i.e.*, they are *regular*. Consequently, after one subdivision step all original vertices are separated by regular vertices and each face has at most one *irregular* vertex, *i.e.*, with valence other than four. We take advantage of this in our implementation to limit the number of cases we need to consider (see Section 3). The limit of repeated subdivision yields the subdivision surface (Figure 1).

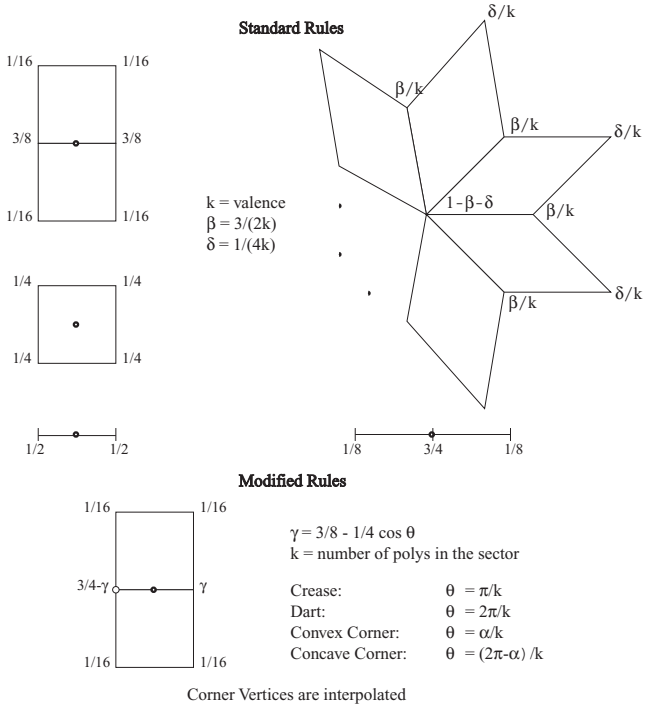

Figure 2: *Stencils for standard Catmull-Clark rules and the Biermann* et al. *rules near an irregular crease vertex (circled).*

2.2 Limit Surface Tessellation

Typically only a finite number of subdivision steps is performed and followed by application of the *limit stencils*. These are similar to the regular subdivision stencils but carry weights that move the points in one final averaging step to the limit surface. For details on limit stencils as well as limit surface tangent stencils see [1]. Often a small number of subdivision steps is sufficient for all but the most contorted models. For example, one level of subdivision to separate all irregular vertices followed by five levels of additional subdivision produces $(2^6)^2 = 4096$ quads per original face. The section of the limit surface corresponding to one face in the control mesh constitutes a *patch*.

Because the subdivision rules take only immediate neighbors into account and depend only on the local structure of the mesh, each original control point influences a finite section of the limit surface in its vicinity. In particular, for the rules of Biermann *et al.* (assuming $s = 1$ for all concave corners; see Section 3), the *control set* of a patch are all those vertices which belong to the set of faces sharing an edge or vertex with the associated control mesh face (see Figures 3 and 5). This implies that each patch can be produced independent of all other patches if the *1-ring* of neighbors of the associated control mesh face is collected up and passed to the appropriate evaluation routine.

2.3 Basis Functions

Because of the linearity of the subdivision process the final surface can be understood as a linear combination of basis functions with

[1] A related approach was recently proposed in independent work by Brickhill [3] for Loop subdivision, however details of the implementation remain sketchy and no performance analysis is provided.

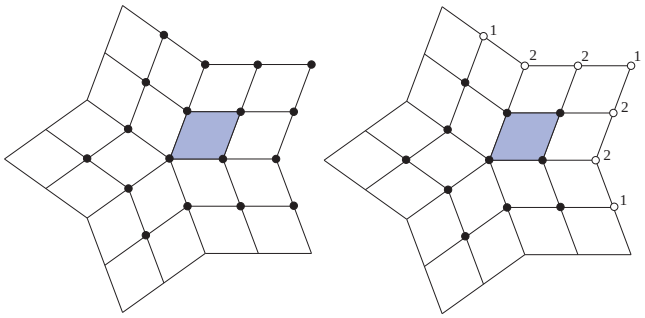

Figure 3: *Example of an interior irregular vertex of valence five with one of its incident first-level faces highlighted. The basis functions whose support overlaps the selected face are shown as black dots on the left. Due to rotational symmetry only one set of basis functions is needed for all faces incident to the irregular vertex. On the right the control set is further broken into those basis functions in the 1-ring of the irregular vertex (black dots) and the outer seven bases (white dots). Modulo symmetries there are only two distinct types among the latter (as indicated by the labels "1" and "2").*

the original control points as weights

$$s(u,v) = \sum_i B^i(u,v)p_i.$$

Here p_i are the control points, typically carrying (x, y, z) positions in world space, although they often also carry texture coordinates, colors, etc. The B^i are the basis functions, one centered at each vertex. The basis functions are defined as the result of subdividing a unit pulse. For example, construct a control mesh in the x/z-plane and move one control point to $y = 1$ to produce the associated basis function (see Figure 4). The final surface is thus a linear combination of such basis functions each weighted by the actual control point in the control mesh. The typical support of such a basis function is a 2-ring around the associated vertex.

The domain for the parameters (u, v) is the original control mesh, each quad face parameterizing its associated limit patch for $(u, v) \in [0, 1]^2$. The limit patch tessellations produce samples of the limit surfaces naturally associated with dyadic points in the domain. For example, after d levels of subdivision the limit points of the tessellation correspond to parameter values $(u_n, v_m) = (n2^{-d}, m2^{-d})$, $n, m = 0, \ldots, 2^d - 1$.

The critical observation for our algorithm is that the B^i depend only on the connectivity of the mesh and the presence of tags, but not on the actual control points. The latter only enter at runtime. Given some parameter values (u_n, v_m) associated with a particular patch the sample of the surface is found as

$$s(u_n, v_m) = \sum_i B^i(u_n, v_m)p_i.$$

The sum can be further restricted to only those vertices whose basis functions make a non-zero contribution over the selected patch, *i.e.*, the 1-ring of the associated control mesh face.

2.4 Algorithm Overview

The basic idea is to evaluate each limit patch uniformly to a user selected depth directly from the control points using precomputed arrays which contain uniform samplings of the basis functions (basis function "tables"). However, the number of distinct basis functions is unbounded since they depend, among other parameters, on the vertex valences. Even when limited by a maximum vertex valence there is still an unreasonably high number of basis functions. The problem is further compounded when permitting creases and corners in the surface.

To simplify this situation, an initial subdivision step is performed using the recursive rules, so that each first level quad has at most one irregular vertex (see Section 2.1). As a result, the basis functions with support on a given patch are a function only of the valence of the *one* irregular vertex of that patch. Additionally, this first level of subdivision provides an opportunity to apply the tangent space modifications [1] necessary for concave corners.

Production of the limit surface tessellation proceeds one patch at a time. Since evaluation of one patch has no effect on the evaluation of any other patches this could be done in parallel, though we did not yet exploit this in our implementation. For a given first level quad, collect all control points in its 1-ring. Using the basis function tables (see Section 2.2), produce a uniform tessellation of this patch of the limit surface with each point in the tessellation a weighted sum of control points, the weights being the corresponding basis function sampled at that point (see Section 2.3).

2.5 Algorithm Details

As the maximum number of subdivision levels we chose five in addition to the initial recursive subdivision step, as this appears to be more than sufficient for practical purposes. To evaluate a patch to depth five requires the basis functions to be evaluated on a grid of $(2^5 + 1)(2^5 + 1) = 33 \times 33$ uniformly spaced sample points. The tables are stored in memory as simple `float[33*33]` arrays. To subdivide to fewer levels, simply subsample these tables with a uniform grid of size $(2^d + 1)(2^d + 1)$ where d is the number of levels.

Pseudocode for the algorithm (assuming five levels of subdivision) is as follows:

```
// N = number of control points in 1-ring of face
// C = number of channels:  x, y, z, s, t, r, g, b, etc.
float sample[C][33*33];
float bases[N][33*33];
float control[N][C];

for( k = 0; k < C; ++k ) // loop over x,y,z
  for( j = 0; j < N; ++j )
    for( i = 0; i < 33*33; ++i )
      sample[k][i] += bases[j][i]*control[j][k]
```

The above code only shows computation of the surface samples. If tangent vectors are desired additional tables are required for tangents in the u and v parametric directions. These would be accumulated using the appropriate channels of the control points. Typically just (x, y, z), but some applications may also require derivatives of other channels.

Vectorization The innermost loop is easily vectorized, either manually or by a modern compiler[2], to take full advantage of the Intel Streaming SIMD Extensions (SSE). For this reason we chose to make the loop over the coordinates the outermost loop instead of the innermost. The loop through the tables vectorizes more efficiently and it is now simple to add more coordinates to the vertices. Since there are eight XMM[3] registers, the loop over the control points can be unrolled four times, using four registers for basis function data and four registers for control point coordinates. Using this arrangement of loops, control points can stay in registers throughout the execution of the innermost loop. The four registers containing basis function data each contain four consecutive entries from a different basis function table. The four registers containing control point data each contain a single coordinate of a control point repeated four times. Control mesh faces at the first level of subdivision are sorted based on the valence of their (only) irregular vertex and any tags, to ensure that faces with the same basis functions will be subdivided sequentially. Hopefully, the basis function tables can

[2]The Intel C++ Compiler 5.0.1 performs best on our implementation at the time of writing.

[3]XMM registers are the 128-bit registers used for SSE. Each register can hold four 32-bit floats.

stay in the L2 cache between calls to the above function, effectively eliminating any load time for the tables. This also means that speed should be relatively independent of the complexity of the mesh. That is, meshes with many different valence vertices and tags can be evaluated at roughly the same rate as meshes with mostly regular topology because tables will rarely by loaded from memory in either case. Experimental results have confirmed this.

3 Basis Function Table Generation

The basis functions were precomputed by generating base meshes that include only one basis function in a certain coordinate and subdividing those base meshes using an existing recursive implementation. Figure 4 shows a typical base mesh with the entire mesh in the x/z-plane except one control point which has $y = 1$ on the left and the result after three levels of subdivision and limit stencil evaluation on the right. The y-values of this patch are the basis function evaluated on a 9×9 grid.

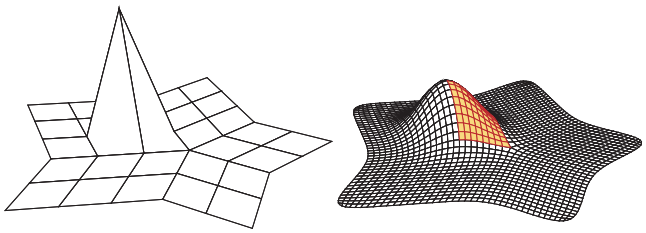

Figure 4: *A base mesh used to generate one of the basis functions for an irregular vertex with valence five (left) and the resulting basis function evaluated at level three with the center patch highlighted (right). The y-values of this patch are the basis function evaluated on a 9×9 grid.*

Basis functions were generated for valences 3-12 for interior points; 1-6 for crease vertices and convex corners; 2-6 for concave corners; and 3-7 for dart vertices[4]. For all these cases, limit positions and partial derivatives in the two parametric directions were sampled on a 33×33 grid. To simplify our code we did not take advantage of all the available symmetries. This resulted in approximately 5300 tables total for values and derivatives. In applications in which the total table size has to be kept tight the number of available symmetries can reduce the necessary tables significantly. The tables were generated with Subdivide 2.0 by Biermann and Zorin [1].

3.1 Counting Basis Functions

We now turn to some detailed issues during table generation. For purposes of this discussion we always have a distinguished vertex. This is the single vertex in a given first level face which also exists in the base mesh. In general this is an irregular vertex, but its valence may be four, making it in fact regular. We ignore this distinction below and for simplicity will always speak of the distinguished vertex as the "irregular vertex."

Smooth interior patches have control sets which consist of all the vertices in the 1-ring of the irregular vertex as well as seven additional basis functions not in the 1-ring (Figure 3). There is a line of symmetry on the diagonal of such a patch and of the seven basis functions not in the vertex 1-ring, only two are distinct (named "1" and "2" in Figure 3), and these are the same regardless of valence and will not be counted here. For an irregular vertex of valence k, the number of distinct basis functions is $k + 2$.

Dart, crease, and corner patches are those for which the irregular vertex has one (dart) or two (crease, convex corner, concave corner) incident tagged edges. For such patches, the basis functions are dependent on the location of the patch with respect to the tag(s).

[4]There are no valence 1 or 2 dart vertices or valence 1 concave corners.

Figure 5 (top) shows an example of a dart vertex, while Figure 5 (bottom) shows the arrangement for a crease or corner (convex or concave) vertex. There are $\lceil k/2 \rceil$ distinct patches due to symmetry. Each distinct patch has a distinct basis function for each vertex in the 1-ring of the irregular vertex, plus one for the irregular vertex itself: $2k + 1$ for dart vertices, $2(k + 1)$ for crease and corner (convex/concave) vertices. The total count of distinct basis functions at a tagged vertex with k incident faces is approximately $4(k + 1)^2$. Outside the 1-ring are four additional basis functions which are always the same regardless of the patch and tag locations *and valence*. Two of these are the same as in the interior case (Figure 3).

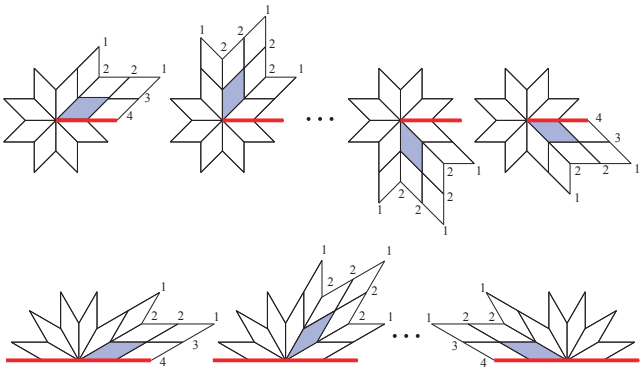

Figure 5: *Control sets for patches with an irregular vertex incident on one (top) or two (bottom) edge tags. The former is a* dart *vertex. $\lceil k/2 \rceil$ of its patches have a unique set of basis functions (all others follow from symmetry). With two incident edge tags the vertex is either* crease *(no vertex tag) or* corner *(convex or concave vertex tag). All three cases have their own set of basis functions. Note that there are very few unique basis functions in the outer ring of a given patch (modulo symmetries). This is true* across all cases.

Concave corners were already enumerated in the previous paragraph, but they require an additional projection step on vertices in the 1-ring of the corner towards the tangent plane at the corner *at each level of subdivision*. The amount that the vertex is projected is controlled by a flatness parameter $s \in [0, 1]$. Different values of s lead to different basis functions. More importantly, some basis functions have a 3-ring rather than the standard 2-ring support because of this projection step. This would greatly increase the number of tables and complicate the management of control sets. This issue is entirely avoided by restricting the flatness parameter to $s = 1$. This achieves the entire projection step within the first subdivision level, which is performed in the standard recursive manner. After this projection, the entire 1-ring is in the tangent plane. Once the 1-ring is entirely in the tangent plane, it will remain in that plane throughout the rest of subdivision. So subsequent subdivision may be performed as if $s = 0$, and hence the increased support width is avoided in the table generation.

3.2 Remarks

Gamma rules are used for darts, corners, and creases to guarantee C^1 smoothness at irregular vertices [1]. The basis functions with and without gamma rules differ in a *2-ring* of an irregular vertex. This means that patches that are not even on the crease are influenced by the special rules. To avoid this, we do not use gamma rules *on the first level* when generating the basis functions. This reduces the difference to the 1-ring. They are still used in the initial recursive subdivision and all subsequent levels. Since the gamma modifications only matter in the limit, the surfaces generated without gamma rules on the second level are still smooth.

In the original work by Biermann *et al.* [1] the shape of the surface near corners depends on α, the angle between the tagged edges at the corner. Letting α be arbitrary is impractical as this would lead

to an infinity of cases. We address this by fixing $\alpha = \pi/2$ during table generation. Input surfaces may of course have any angle between the creases.

Tangent space modifications are used in the original Biermann *et al.* [1] rules to accommodate normal constraints during subdivision. We only allow these at the first subdivision level to avoid another explosion of cases.

Arbitrary polygons in the coarsest level require two levels of subdivision before the irregular vertices are separated. Otherwise more basis functions would be required to deal with faces that have two irregular vertices.

Number of basis functions The previously mentioned total of 5300 tables does not take into account all of the repeated basis functions or symmetries. If this is done the number could be reduced to roughly 2200. Note however that this would only simplify generation and offline storage of tables. During runtime the various symmetries would need to be explicitly "unpacked" to ensure proper alignment of data for the SIMD instructions.

For a library which must handle any and all input it is not feasible to store all possible tables ahead of time. Instead one could store the tables necessary for a particular input model with the model itself or generate them during the initial load phase using a direct evaluation code [18], for example. In practice we have found our particular set of tables to be sufficient for all models encountered.

4 Analysis

In a perfect world there would be no memory access latencies and subdivision algorithms would be compared based on their operation counts alone. Unfortunately that is not the case, and programmers must take into account the limitations of their target architecture. Yet it is still important to compare the theoretical maximum speed of different subdivision surface evaluation algorithms.

For the following operation counting arguments we assume that all vertices have valence four. This is true on average because of the Euler characteristic of a 2-manifold mesh[5]. An operation (op) will be a scalar vector multiplication (mult) or vector addition (add). For our algorithm this means we will consider calculating only one coordinate, since the others have identical operations. The final numbers given will be operations per base face and need to be multiplied with the number of channels.

Table driven evaluation begins with an initial level of subdivision using the recursive rules. Computing a face center takes 3 adds and 1 mult, an edge vertex 5 adds and 2 mults, and refining a vertex 8 adds and 3 mults for a total of 29 ops per face (recall that there are exactly two edges and approximately one vertex per face on average in a quad mesh).

For a face with a valence four vertex, there are 16 first level vertices with basis functions whose support overlaps the selected face. So 16 mults and 15 adds are necessary to calculate each vertex in the tessellation. Tessellating each first level quad to a depth d creates $(2^d + 1)(2^d + 1)$ vertices. For $d > 1$ ($d = 1$ is the special case of only the first subdivision to separate the irregular vertices) the total operation count per face is:

$$\#(d) \;=\; 29 + 4 \cdot 31 \cdot (2^{d-1} + 1)(2^{d-1} + 1)$$

	d=1	d=2	d=3	d=4	d=5	d=6
#(d)	29	1145	3129	10073	35865	135065

[5]A more careful analysis reveals that the total cost per mesh is related to the *squares* of the valences. However, meshes have to become very large and very pathological for our assumption to break the counting argument in a significant way.

Recursive subdivision which proceeds on a face by face basis simply repeats the calculations that led us to the total of 29 operations per face. Each face is split into 4 at each level of subdivision, so the total number of operations for d levels of subdivision is:

$$\#(d) \;=\; 29 \textstyle\sum_{i=0}^{d-1} 4^i \;=\; 29 \cdot (4^d - 1)/3$$

	d=1	d=2	d=3	d=4	d=5	d=6
#(d)	29	145	609	2465	9889	39585

To calculate limit positions requires an additional 8 adds and 3 multiplies for each vertex at the finest level. The formula becomes:

$$\#(d) \;=\; 29 \cdot (4^d - 1)/3 \;+\; 11 \cdot 4^d$$

	d=1	d=2	d=3	d=4	d=5	d=6
#(d)	73	321	1313	5281	21153	84641

Our algorithm has a higher operation count than that of recursive subdivision. Its advantage is that memory is accessed in a very regular, cacheable manner. Recursive subdivision does not access memory in a sequential manner and its performance is limited by the memory subsystem.

Forward differencing takes advantage of the piecewise polynomial nature of the subdivision scheme away from irregular vertices. Regular patches are evaluated with forward differencing.

There is considerable overhead in initializing the forward differences. Exact numbers would require significant analysis, so we will make a conservative estimate of 50 operations. This overhead makes it senseless to use forward differencing until the patches are tessellated to at least 4. If a quad has four irregular vertices, the first regular patches are created on the second level of subdivision, and are not tessellated to 4×4 quads until the fourth level. So this method may not be superior to recursive subdivision until the fourth level.

The cost of one step of forward differencing is 3 adds. So the cost of tessellating a patch with 4^n vertices is approximately $3 \cdot 4^n$, leading to a total consisting of

- the cost of subdivision to 3 levels near base vertices. This is roughly the same as the cost of subdividing the base mesh to 3 levels. For $d > 3$, this must be done in each face and for each vertex, so the cost is approximately four times the cost of three levels of subdivision;
- the cost of creating control points for the regular patches. This is the same as the cost of subdividing one face, 29 operations;
- the overhead of forward differencing;
- the cost of forward differencing.

$$\#(d) \;=\; 4 \cdot 609 \;+\; 3 \cdot \textstyle\sum_{i=4}^{d} (29 + 50 + 3 \cdot 4^{i-2})$$

	d=4	d=5	d=6
#(d)	2817	3630	6171

Forward differencing is clearly the most efficient way to tessellate a surface in terms of operation count, but considering the amount of recursive subdivision involved it is still subject to high memory latency issues. It would certainly be more efficient at higher levels of subdivision, but more than six levels is exceedingly expensive in most applications and very rarely required.

4.1 Implementation Issues

Some vertices in the tessellation are shared by more than one patch and are calculated more than once. Due to the imprecision of floats, their positions may differ slightly, enough to cause pixel dropouts during rendering. To avoid this problem, choose one computed position to be "correct" and copy its value to all other instances of that vertex.

Input mesh	levels	tessellation	limit pos (P3)	pos,tangents	edge write	limit pos (P4)	pos,tangents	edge write
64 quads	6	262144 quads	37ms	173ms	8ms	8.6ms	31.1ms	3.3ms
64 quads	4	16384 quads	2.5ms	7.0ms	2.5ms	1.11ms	2.68ms	0.87ms
64 quads	2	1024 quads	0.84ms	1.67ms	0.51ms	0.52ms	0.92ms	0.34ms
384 quads	6	1572864 quads	186ms	660ms	53ms	52ms	166ms	21ms
384 quads	4	98304 quads	18ms	45ms	20ms	7.8ms	17ms	7.6ms
384 quads	2	6144 quads	7.7ms	12.5ms	9.5ms	4.4ms	6.6ms	5.0ms
6144 quads	4	1572864 quads	298ms	730ms	322ms	125ms	276ms	114ms
6144 quads	2	98304 quads	137ms	213ms	170ms	66ms	100ms	74ms

Table 1: *Timing results showing size of the input mesh, number of levels of subdivision, number of quads after subdivision, timings to calculate limit positions only, to calculate limit positions and limit tangents, and to perform an edge writethrough to guarantee no pixel dropouts. The times include the time spent calculating the first level of subdivision using recursive rules. Timings were taken on a 733 MHz PIII and a 1.7 GHz P4. Timings are averages over hundreds of runs.*

Since each patch is evaluated separately from all other patches, there is no need for all patches to be sampled at the same rate. This provides simple, patch-based adaptivity. As with other adaptivity schemes, special care must be taken to ensure neighboring patches with different levels of tessellation do not lead to cracks in the surface. One solution is to render triangle fans (Figure 6) connecting one vertex in the coarser tessellation to many vertices in the finer tessellation.

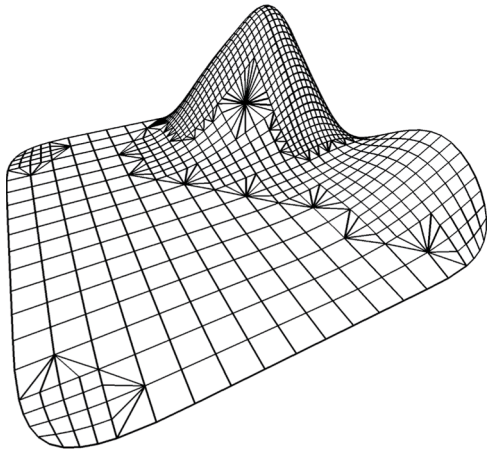

Figure 6: *An adaptively subdivided mesh. Triangle fans are used to prevent cracks at boundaries between patches evaluated at different depths. Notice the patch that is adjacent to two patches that are subdivided two levels deeper. There is no restriction that adjacent patches must be subdivided to within one level of each other.*

If two patches which are evaluated at different levels meet at a crease, their positions must match to prevent cracks but their tangents are in general different. Positions from the finer tessellation can be used at the boundary of the coarser patch, but tangents cannot. This can be addressed in one of two ways

- force opposite sides of a crease edge to be subdivided to the same number of levels. This way there are always true limit tangents everywhere, and in meshes with relatively few creases this restriction is not a problem;
- interpolate the tangents of the nearest two vertices in the coarser tessellation. The pros and cons are the opposite of the first solution. These interpolated tangents should be fairly accurate, otherwise the adaptivity criterion would have caused that patch to be subdivided further.

This issue is not limited to tangents, but also applies to texture or color coordinates, or any other parameters one chooses to subdivide.

For recursive subdivision engines, adaptivity can be achieved by simply refining the surface until a local flatness criterion is satisfied. But this algorithm must determine how many times to subdivide each patch based only on the first level control points. A simple, robust solution to this is an open problem. A simple function that takes into account how much the control points deviate from the plane of the control face works adequately.

5 Results

Table 1 shows several timings of our implementation run on various input meshes. There are several things to observe in this data:

- A flops/cycle count for 384 quads subdivided to 6 levels in 52ms at 1.7 GHz, assuming 16 multiplies, 15 adds, and one store:

$$384 \text{ quads} \cdot \frac{66^2 \text{verts}}{\text{quad}} \cdot \frac{3 \text{ coords}}{\text{vert}} \cdot \frac{32 \text{ flops}}{\text{coord}} = 1.6 \times 10^8 \text{flops}$$

$$\frac{1.6 \times 10^8 \text{ flops}}{.052s} \cdot \frac{1s}{1.7 \times 10^9 \text{ cycles}} \approx \mathbf{1.8} \text{ flops/cycle}$$

- Consider the timings for 384 quads subdivided to 4 and 6 levels, with and without normals, on the P3. One would expect calculation of normals to increase the time by a factor of three. The ratio $45/18 < 3$, but this can be attributed to an overhead of about 5ms. The ratio $660/186 > 3$ is more interesting. This is caused by the tables not fitting well in the L2 cache. The tables in this case occupy about 204KB, which is uncomfortably close to the 256KB cache size. This effect only occurs when subdividing to six levels. Tables for five levels occupy 54KB, which easily fits. However, it is still cheaper to subdivide one level with recursive rules and five with tables than, say, three levels with recursive rules and three with tables (compare 384 quads to 6 levels with 6144 quads to 4 levels) because the recursive rules are not as fast as tables in the implementation.
- The P4 is 50% faster than the PIII, clock for clock. The P4 achieves 1.8 flops/cycle, whereas the PIII only achieves 1.2 flops/cycle. We attribute this to the Netburst™[8] architecture, in particular the new cache subsystem and high speed bus.
- Edge writethrough can be a serious performance hit, and should not be used except for very high quality renderings. Such writethroughs are costly because they require unaligned, non-consecutive writes to far away memory.
- The mesh with 64 quads has 70% tagged vertices, but its performance is comparable to the other results.

We can approximate the time spent in the innermost loops in the 6-level execution time by subtracting the 2-level time from the 6-level time. If we take into account the loads and stores when counting operations, then there are 48 ops/coord and calculations similar to the above give an estimate of 2.98 ops/cycle on the P4 with a theoretical limit of 4 ops/cycle. Inspecting the disassembly of the innermost loops of our implementation, there are 61 clock cycles spent in these loops compared to a theoretical lower limit of 48 (the difference being due to loop overhead). So instead of an upper limit

of 4 ops/cycle, the best we could hope for is 3.15 ops/cycle. Our implementation thus performs very efficiently on the P4, but not quite as well on the PIII. We attribute this to data transfer latencies that prevent the PIII from achieving optimal speed.

It is hard to determine timings that fairly represent adaptive performance. Suffice it to say that adaptive performance is as one would expect it to be based on the uniform subdivision timings. That is, there is no performance penalty associated with adaptivity aside from calculating the adaptivity criterion.

Since the tessellations have such a regular layout and the data structures are so simple, it is possible to render efficiently using quad strips and triangle fans for adaptive subdivision. On an nVidia Quadro2 Pro card our implementation can render 8 million unshaded quads or 3 million shaded quads per second. The Volkswagen model (Figure 7) evaluated at level four has 77312 quads, which can be rendered at 40 f/s on the Quadro2 Pro. It takes our implementation only 14 ms on the P4 (base mesh has about 300 quads, so timings are close to the timings for the 384 quad mesh) to generate this subdivided mesh with tangents, little over half the time it takes the video card to render. Using 50% of the CPU is enough to saturate the graphics card assuming that one evaluates on every frame. For static models of course, evaluation would only be required once.

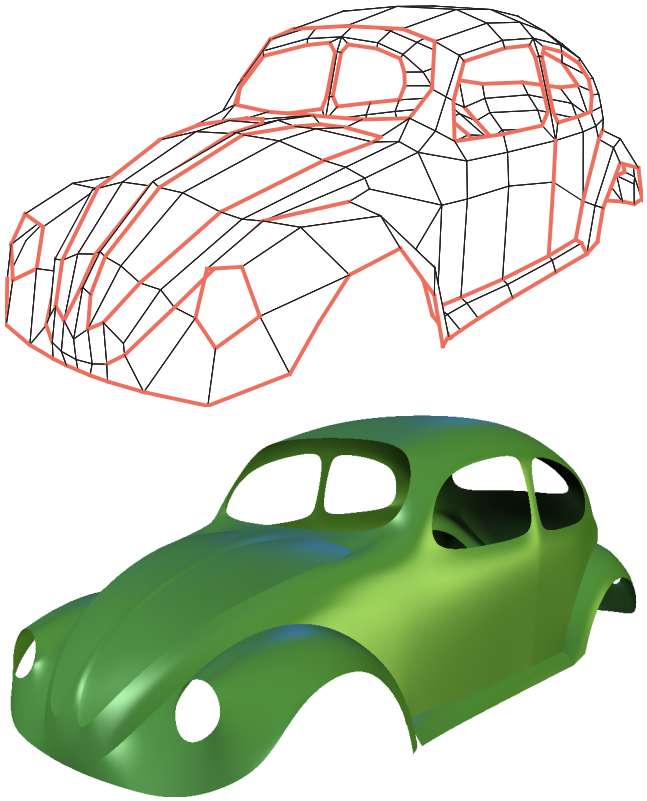

Figure 7: *A Volkswagen model control mesh with many tags and its limit surface at level four.*

6 Conclusion and Future Work

We have demonstrated an extremely efficient approach to subdivision based on precomputed tessellations of Catmull-Clark basis functions. These can be produced with any standard subdivision code and may contain crease, dart, and corner rules. The method carries over to other subdivision approaches in a straightforward fashion. The algorithm is well-suited for parallelization both at the level of SIMD operations and at the level of parallel execution units. The results should apply equally well to other modern CPUs with multiple execution units, deep pipelining, and their general sensitivity towards caching issues. The improvements in the memory architecture of the P4, in particular less bus transfer resource contention within the CPU and faster access to the cache, yield a performance improvement of 50%.

In future work we hope to perform more extensive performance comparisons between our table driven approach and depth first recursive subdivision as well as forward difference based approaches. The recursive version is of particular interest for multiresolution surfaces which add detail displacements at every subdivision level to significantly enrich the set of surfaces that can be modelled in this fashion. Such an engine would also be very useful for fast decompression of geometry [10]. Additional work should be devoted to adaptive rendering criteria which can be evaluated fast enough to amortize their cost.

Source code for a library and demo is available at:
http://multires.caltech.edu/software/fastsubd/

Acknowledgment This work was supported in part by NSF (DMS-9874082, ACI-9721349, DMS-9872890, ACI-9982273), the DOE (W-7405-ENG-48/B341492), Intel, Alias|Wavefront, Pixar, Microsoft, and the Packard Foundation. Special thanks to Stephen Junkins, Michael Rosenzweig, Michael Julier, Patrick Mullen, Pierre Alliez, Mathieu Desbrun, Andrei Khodakovsky, and Cici Koenig.

References

[1] Biermann, H., Levin, A., and Zorin, D. Piecewise Smooth Subdivision Surfaces with Normal Control. *Proceedings of SIGGRAPH 2000* (2000), 113–120.

[2] Bischoff, S., Kobbelt, L. P., and Seidel, H.-P. Towards Hardware Implementation Of Loop Subdivision. *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (2000), 41–50.

[3] Brickhill, D. Practical Implementation Techniques for Multi-Resolution Subdivision Surfaces. In *Game Developers Conference*, 2001.

[4] Catmull, E., and Clark, J. Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes. *Computer Aided Design 10*, 6 (1978), 350–355.

[5] DeRose, T., Kass, M., and Truong, T. Subdivision Surfaces in Character Animation. *Proceedings of SIGGRAPH 98* (1998), 85–94.

[6] Havemann, S. Interactive Rendering of Catmull/Clark Surfaces with Crease Edges. Tech. Rep. TUBSCG-2001-01, TU Braunschweig, 2001.

[7] Intel Corporation. *C++ Class Libraries for SIMD Operations Reference Manual*., 1997–1999.

[8] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 1997-2001.

[9] Junkins, S. Fast Triangle Neighbor Finding for Subdivision Surfaces. Tech. rep., Intel Architecture Labs, September 1999.

[10] Khodakovsky, A., Schröder, P., and Sweldens, W. Progressive Geometry Compression. *Proceedings of SIGGRAPH 00* (2000), 271–278.

[11] Lien, S.-L., Shantz, M., and Pratt, V. Adaptive Forward Differencing for Rendering Curves and Surfaces. *Computer Graphics (Proceedings of SIGGRAPH 87) 21*, 4 (1987), 111–118.

[12] Loop, C. Smooth Subdivision Surfaces Based on Triangles. Master's thesis, University of Utah, Department of Mathematics, 1987.

[13] Müller, K., and Havemann, S. Subdivision Surface Tessellation on the Fly using a versatile Mesh Data Structure. *Computer Graphics Forum 19*, 3 (2000).

[14] Pulli, K., and Segal, M. Fast Rendering of Subdivision Surfaces. In *Rendering Techniques '96*, 61–70, 1996.

[15] Samet, H. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[16] Stam, J. Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values. *Proceedings of SIGGRAPH 98* (1998), 395–404.

[17] Ying, L., and Zorin, D. Nonmanifold Subdivision. In *Proceedings of Visualization 2001*, 2001.

[18] Zorin, D., and Kristjansson, D. Evaluation of Piecewise Smooth Subdivision Surfaces. *Visual Computer* (2002).

[19] Zorin, D., and Schröder, P., Eds. *Subdivision for Modeling and Animation*. Course Notes. ACM Siggraph, 2000.

[20] Zorin, D., Schröder, P., and Sweldens, W. Interactive Multiresolution Mesh Editing. *Proceedings of SIGGRAPH 97* (1997), 259–268.