Hardware Subdivision and Tessellation of Catmull-Clark Surfaces

Charles Loop

11 May 2010

Technical Report
MSR-TR-2010-163

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Abstract

We present a table driven approach to performing Catmull-Clark subdivision in parallel utilizing one thread per output mesh vertex. We apply the procedure one or two times in order to isolate extraordinary vertices and simplify the input patch structures consumed by the hardware unit responsible for adaptive tessellation. From this simplified mesh, we evaluate the performance Stam's direct evaluation procedure, a curvature continuous patching scheme, and a tangent plane continuous approximation suitable for displacement mapping.

1 Introduction

We consider the problem of utilizing the hardware tessellator for adaptively rendering Catmull-Clark subdivision surfaces, and two related spline approximations. All of these surfaces approximate an arbitrary two-manifold control mesh (possible with boundary); all are equivalent to bicubic B-Splines over regular parts of the control mesh; and all require some amount of control mesh subdivision to regularize the inputs consumed by the hardware tessellator. Each generates a different surface corresponding to the extraordinary vertices of the control mesh, with different shape characteristics and cost. We examine these trade offs in this paper.

We first present a table driven approach to parallel Catmull-Clark subdivision on the GPU, utilizing one thread per output mesh vertex. This is done so the control mesh has *isolated extraordinary vertices*; that is, exclusively quad faces with no quad incident on more than one extraordinary vertex. A quad mesh will require one Catmull-Clark subdivision step, while one with faces other than quads will require two. This is necessary in order to implement Stam's direct evaluation procedure [18]; we show that this algorithm can be the basis of an adaptive real-time Catmull-Clark surface renderer. The advantages of doing the subdivision step on the GPU are that only the vertices of the coarsest control mesh need be transfered from CPU to GPU and skinned (animated). Our implementation utilizes device independent DirectX 11 compute shaders and hardware tessellator pipeline. We measure the performance of our approach on both the ATI Radeon 5870 and the nVidia GTX 480; the highest end consumer grade GPUs from both major vendors, at the time of this writing.

The Catmull-Clark limit surface is not curvature continuous at extraordinary vertices. This can lead to shading and reflection mapping artifacts. A possible solution to this problem would be to use curvature continuous patches [7]. This algorithm also requires isolated extraordinary vertices, and generates a single biseptic (bidegree 7) Bézier patch for each quad of the input mesh. We show that the cost of evaluating these patches using tessellation hardware is about the same as Stam's algorithm.

For applications involving displacement mapping, computing the exact limit surface or a curvature continuous spline will expend more resources than are

necessary to achieve the desired effect. That is, to generate a tangent plane continuous base surface whose tessellation can be offset in the direction of each surface normal. To this end, we verify that an approximation based on Gregory patches [8] will be significantly faster than direct evaluation of the limit surface. This is not surprising as fewer, less costly patches are used to cover the surface. None-the-less, subdividing the control mesh prior to tessellation can be advantageous even for an approximate Catmull-Clark scheme.

The approximation will be more accurate since more interpolation constraints will be satisfied over the refined mesh. The tessellation will have better adaptivity (more uniformly sized triangles), since it will be based on smaller, flatter, more uniform patches; i.e. more information will be considered in determining patch sampling rates. Subdivision increases the regular regions while decreasing the extraordinary regions. This means that for the same sampling rate, more of the samples will come from less costly regular patches and fewer will come from more costly extraordinary patches; by a ratio of 3 to 1. While there are more patches to process, the savings dues to less extraordinary evaluations will pay off for dense triangulations; our experimental results confirm this. This suggests that as traditional triangle rasterization is supplanted by micropolygon rendering, subdivision should become a preprocess to tessellation.

Regardless of the surface scheme used, extraordinary vertex isolation greatly simplifies tessellator pipeline implementation since the combinatorial complexity of extraordinary patch input is reduced from thousands of cases [8] to at most 9 (assuming a maximum valence of $n = 12$) that are identical up valence n. Each surface scheme requires its own evaluation routine that could, in principle, be compiled without valence dependent loops or branches.

2 Previous Work

Early GPU surface subdivision schemes were mostly concerned with finding ways of taking advantage of the graphics-centric programmability of vertex and pixel shaders, randomly accessing memory via texture maps; [2, 3, 17] are notable examples.

In anticipation of hardware tessellation and noting the high cost of direct evaluation and the need for a subdivision step Loop and Schaefer[7] proposed an approximation to a quads only Catmull-Clark limit surface based on bicubic Bézier patches. Several variants along these lines have appeared with various improvements to the restrictions on mesh connectivity or underlying surface algorithm; quads only [12, 14], tris and quads [9] or tris, quads, and pents [11].

The introduction of of CUDA in the fall of 2006, allowed for general purpose programming of the GPU. Subdivision of bicubic patches using a breadth-first paradigm was pioneered by Patney and Owens[16]. Later, they applied these ideas to the problem of crack-free breadth-first parallel subdivision of Catmull-Clark surfaces [15]. The idea behind breadth-first parallel subdivision is to

perform a subdivision pass over the entire mesh, emitting the possibly adapted subdivided mesh. This is similar to our approach, but since we let the hardware tessellator handle adaptivity, we can know a priori the indices and storage requirements for the subdivided mesh. [15] use book keeping data structures and parallel scan operations with unpredictable memory requirements to manage the subdivided mesh. Furthermore, they invoke one thread per input mesh vertex, making a weighted contribution to all new mesh vertices that are influenced. Since multiple vertices may contribute to the same target location at the same time, atomic adds are need to serialize memory collisions. We use one thread per output vertex, so no such collisions can occur.

Breadth-first parallel subdivision represents a global approach, were the entire mesh is processed over multiple passes until the resulting polygons are small. Hardware tessellation on the other hand localizes the problem into patches that can be processed in a single pass [10]. To overcome the limitations on the adaptivity of current hardware tessellation algorithms, a recursive non-isoparametric patch subdivision scheme called *DiagSplit* was developed to produce uniform micropolygon output [5]. The work presented in this paper should work well with this approach.

3 Data Parallel Catmull-Clark Subdivision

Catmull-Clark subdivision [4] is an algorithm that takes as input an arbitrary topology 2-manifold control mesh and outputs a new refined mesh with more vertices, faces, and edges than the input mesh. The algorithm can be stated as a set of three simple rules for finding a new point corresponding to each face, edge, and vertex of the input mesh.

1) Face points - the average of all old points defining a face.

2) Edge points - the average of the two old vertex points and two new face points incident on the edge.

3) Vertex points - the average $\frac{n-2}{n}\mathbf{V} + \frac{\mathbf{1}}{\mathbf{n}}\mathbf{P} + \frac{\mathbf{1}}{\mathbf{n}}\mathbf{Q}$ where $\mathbf{V}$ is the old vertex point, $\mathbf{P}$ is the average of the all old vertex points adjacent to the old vertex, and $\mathbf{Q}$ is the average of the new face points of all faces incident to the old vertex.

While the refinement rules themselves are simple, the non-trivial aspect of implementing this algorithm is managing the adjacency relationships among the vertices, faces, and edges of the control mesh. A mesh is typically specified by an ordered collection of vertex coordinates, followed by a collection of faces consisting of 3 or more oriented vertex indices. From this, the adjacency relations needed to apply the Catmull-Clark refinement rules are inferred. The data structures and algorithms needed for this do not map well to the data parallel GPU environment. Fortunately, as long as control mesh connectivity remains

static, as is generally the case at runtime, then these adjacency relations can be precomputed for efficient processing on the GPU.

From rule 1) above, we see that new face points only depend on old vertices. We can therefore compute all the new face points in parallel provided the old vertex points are known. Similarly, from rule 2) we have that new edge points only depend on old vertices and new face points. So we can also compute all the new edge points in parallel once all the new face points have been computed. Finally, from rule 3) we can compute all the new vertex points in parallel as soon as the new face points have been computed.

We propose three GPU kernels (DX11 compute shaders) that execute in series, to compute in parallel all of the new face points, edge points, and vertex points respectively of the refined mesh. Each of these kernels is about as simple as the corresponding refinement rule; though each must access index tables that encode the necessary adjacency information. These tables are described next.

3.1 Subdivision Tables

The nature of our subdivision tables is best conveyed via example. Consider a simple pyramid control mesh with $V = 5$ vertices, $F = 5$ faces, and $E = 8$ edges, see Figure 1. We allocate a single vertex buffer large enough to hold both

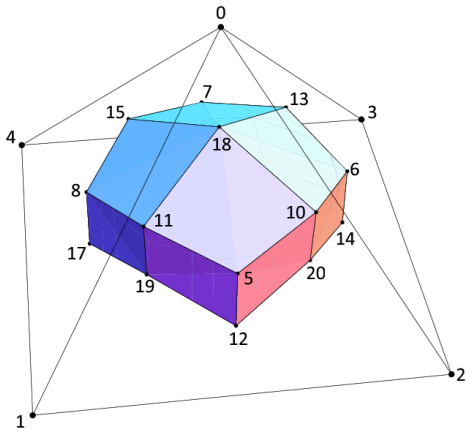

Figure 1: A pyramid control mesh (wire frame), and the result of one level of Catmull-Clark subdivision (shaded).

the old and new vertices; while not essential, this simplification lets us avoid having to specify which buffer to read from. For our pyramid mesh (assuming one level of subdivision), the vertex buffer would have $5 + 18 = 23$ entries; note that the number of new vertices computed is $F + E + V$. Column a in Figure 2 represents this vertex buffer, the entries in columns b and c illustrate the five

tables used by our subdivision kernels.

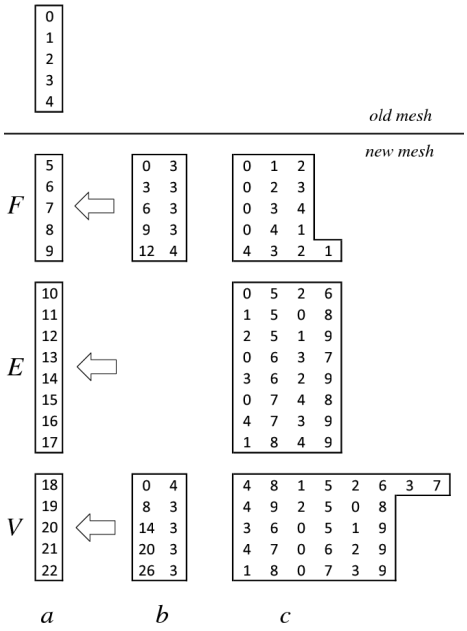

Figure 2: Catmull-Clark subdivision tables for the pyramid model.Column a represents the vertex buffer containing both old and new vertices, column b (rows F and V) shows entries in the offset-valence tables, and column c shows the entries of the vertex index tables.

In order to compute the new face points we must know the valence of each old face, as well as the indices of all incident old vertices. We encode this information in a pair of tables. One is an offset and valence table that contains two entries for each old face, see Figure 2, row F, column b. The first entry is the offset into the second table indicating the location of the first incident old vertex index. The second entry is the valence of the old face; this tells us how many indices to read from the second table. Figure 2, row F, column c shows this table for our pyramid mesh.

To compute the new edge points we only need a single index table since a new edge point is always determined by exactly four points. Each entry contains the indices of the two old vertex points and the two new face points, see Figure 2, row E, column c.

For the new vertex points we again need two tables. The first contains offset and valence pairs, see Figure 2, row V, column b. The second contains the indices of all incident old vertex points and new face points, see Figure 2, row V, column c. In this case the index table contains $2n$ entries for each old valence

n vertex. Also, note that the index of the corresponding old vertex is a constant offset of $F+E+V$ from the index of the new vertex being constructed. We store the values of F, E, and V in constant memory before executing the subdivision kernels,

Note that for a mesh containing only quads (any mesh that has been subdivided at least once), we no longer need the valence and offset table to construct new face points since all faces will have valence 4 and the offsets can be found as `4*fidx`, where `fidx` is the thread id of the new face point. In fact, for a quad only mesh we can find the new face points by running the new edge point kernel using the face index table since, for the quad only case, both kernels compute the average of 4 points. The new edge point kernel is more efficient since it requires fewer memory reads and does not contain a loop. Unfortunately, we can't combine the new quad face and edge point kernels into a single invocation due to the dependence of edge points on face points requiring intra processor synchronization; separate kernel invocations serves this purpose for us.

We handle meshes with boundary by treating boundary edges as the control polygon of a B-spline curve [13]. During table construction, when a boundary edge is encountered, the indices of the endpoints are each written twice into the new edge index table. At runtime, the new edge point will be found as the midpoint of the old edge. For boundary vertices there are two cases: the vertex is shared by two or more faces (a boundary vertex) or belongs to a single face (a corner). For a boundary vertex, we write $n = 1$ into the vertex valence and offset table, and the indices of of the two adjacent boundary vertices into the index table. At runtime, when valence $n = 1$ is encountered, we gather the boundary vertex and its two neighbors and apply the $1 : 6 : 1$ cubic B-spline subdivision rule for the new vertex point. For a corner vertex, we write $n = 0$ into the vertex valence and offset table. At runtime, we set the new vertex point to the old vertex point.

An obvious optimization for our approach would be to sort vertices and faces according to valence prior to table construction. This will mitigate the divergence created by different valences belonging to the same SIMD execution unit (a.k.a. *warp*). In practice however, we do not detect a noticeable performance gain from this optimization.

4 Hardware Tessellation Stage

Hardware tessellation of parametric surfaces is now part of the rendering pipeline supported by commodity graphics cards. It can amplify geometry, conserving animation and CPU/GPU bus overhead, by taking a compact parametric patch description and producing a dense triangulation of the patch. The algorithm is adaptive, so that the amount of tessellation can be adjusted each frame to avoid faceting artifacts (under sampling) or saturating the rasterizer (over sampling). Within a patch, a watertight (crack-free) triangulation is guaranteed. The pro-

cess has excellent locality; that is, each patch can be processed independently, in parallel, entirely within the on-chip caches of the GPU.

On the other hand, the adaptivity, or sample rate estimation, is based solely on patch coefficients. Therefore, precise output triangle size criteria cannot be achieved. Crack-free joins between patches can be difficult to maintain unless the floating point numerics of boundary evaluations are careful considered. This problem becomes even more difficult when displacement maps are used since, in addition to bitwise consistent positions, bitwise consistent normals are required.

The hardware tessellation pipeline fits logically between vertex and pixel shading. It consists of a programmable *hull shader*, a fix function *tessellator unit*, and a programmable *domain shader*. The hull shader takes as input a fixed size array of vertices and outputs a fixed (possibly different) size array of vertices. The hull shader execution model invokes one thread per output vertex to run a user specified program. The hull shader is used for change of basis operations. A matrix multiplication of the input vertices can be used to generate the output vertices. Additional execution happens in the hull shader *patch constant function*. This executes once per patch to determine *tessellation factors*, indicating the amount to tessellate each patch edge. The fixed function tessellator unit creates a patch domain tessellation based on the tessellation factors; this allows the hardware to adaptively tessellate each patch. The domain shader executes one thread per vertex generated by the tessellator. The user specified domain shader program evaluates the patch output by the hull shader at the u, v coordinates of each tessellated vertex to generate an output vertex. The output vertex should contain a position and optionally normal, color, etc. The resulting patch triangulation is rasterized and shaded in a conventional manner.

Performing a step or two of Catmull-Clark subdivision on the GPU will isolate extraordinary vertices, simplifying the input types consumed by the hull shader. This input can be characterized by the valence n of the single extraordinary vertex in an *extraordinary net*, or *regular net* when $n = 4$, see Figure 3. These nets contain $2n + 8$ vertices.

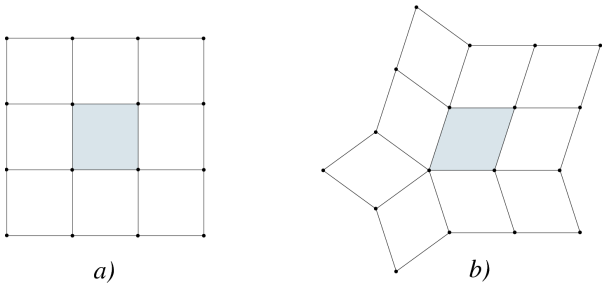

Figure 3: a) regular net, b) extraordinary net

The upper limit on hull shader input is currently primitives with 32 points. This means we can support valence 3 up to 12 (solving $2n + 8 = 32$); there are exactly 9 such extraordinary nets. These can be batched by valence and, in principle, processed without valence dependent loops or branches; though in practice, loop unrolling does not always produce faster code.

4.1 Regular Patches

Over regions of the mesh corresponding to each regular net, the Catmull-Clark limit surface is equivalent to a bicubic B-spline patch. These 16 point primitives are batched in a single draw call to be processed by the graphics pipeline.

In our implementation, we use the hull shader to transform these patches into Bézier form. Rather than using a 16×16 change of basis matrix, we take advantage of the tensor product form of bicubic polynomials. If we treat the 16 B-spline control points as 4×4 matrix with vector valued (x, y, z) elements, we can multiply on the right and left by the univariate B-spline to Bézier change of basis matrix and its transpose. Each thread will compute a single output vertex using approximately 60 FLOPS (floating point operations); we general do not count additions in our estimates as these can often be fused into a single multiply-add (MADD) instruction.

In the domain shader, we implement tensor product bicubic Bézier evaluation. Note that in addition to position,we must also evaluate partial derivatives (whose normalized cross product is the surface normal). This is straightforward within the tensor product framework. Our approach requires 162 FLOPS per evaluation thread.

Note that by maintaining the B-spline basis through to the domain shader, we can avoid the cost of hull shading; since no change of basis is needed. However, evaluating the B-spline basis is slightly more expensive than Bézier (by 24 FLOPS). So we see some benefit to this idea for low tessellation rates, it does not pay off as tessellation rate increases. All of our surface implementations utilize the same code path for regular patches.

4.2 Exact Evaluation of the Limit Surface

The Catmull-Clark limit surface is represented by an infinite collection of bicubic patches that become infinitesimally small near extraordinary vertices. For any point of the control mesh, a correspond patch and domain location can be found to evaluate this surface. Stam's algorithm reduces the complexity of this approach to constant time. The idea is to find the eigen decomposition of the subdivision matrix (a matrix whose entries correspond the refinement rules), and convert k iterations of subdivision into raising the eigen values to the k^{th} power. For a complete description, see [18].

The hull shader is used to project the extraordinary net to eigen space. Each hull shader thread takes the dot product of a row of the precomputed inverse

eigen value matrix with the extraordinary net. For vector valued (x, y, z) control point, each of $2n + 8$ threads must perform $6n + 24$ FLOPS. One drawback of Stam's direct evaluation procedure is that the eigen space transformation does not maintain the equivalence of adjacent patch boundary curve coefficients. Due to floating point inaccuracies, it is not possible produce watertight joins between adjacent patches.

In the domain shader, given a u, v coordinate, the value of k is determined, the $2n + 8$ precomputed eigen values are raised to the k^{th} power, and one of three sets of $2n + 8$ bicubic eigen basis functions are evaluated after a linear remapping of u, v. Finally, the sum of products of the scaled eigen values, the value of the eigen basis functions, and the eigen space extraordinary net is computed. The partial derivatives are similarly computed using the partial derivatives of the eigen basis functions in order to compute a surface normal. Each evaluation thread will do about $108n+534$ FLOPS; this a is gross estimate ignoring some fine details, intended to give a sense of the expected performance of the algorithm.

4.3 Curvature Continuous Patching

Similar to direct evaluation of Catmull-Clark subdivision surfaces using Stam's algorithm, the curvature continuous patching scheme of [8] also requires a mesh with isolated extraordinary vertices. This algorithm generates a single polynomial patch for each extraordinary net. This patch is bidegree 7 represented by an 8×8 Bézier control net.

The surface is defined by a set of precomputed basis functions encoded in a $64 \times (2n + 8)$ matrix for each valence n. Ideally, the hull shader would compute the product of this matrix and the extraordinary net vertices generating 64 output patch vertices. However, since the hull shader output can be at most 32 vertices, we must pack two control points into each output vertex, and then separate these in the domain shader; while trivial from a coding standpoint, this limitation may adversely impact performance. Each of 32 hull shader threads performs $12n + 48$ FLOPS.

Since the hull shader output is uniformly a biseptic patch independent of valence, the same domain shader can be shared by all input patch types. We leverage the tensor product structure of the biseptic patch, similar to the regular case. We estimate each evaluation thread will execute 568 FLOPS.

In joins between biseptic and bicubic patches, the shared boundary control points will not coincide; preventing a watertight join. A possible solution to this problem would be to use a Hermite basis where the extra coefficients of the degree 7 boundary would vanish, while the position and derivative coefficients at the endpoints would be identical. We leave this engineering detail to future work.

4.4 Gregory Patch Approximation

We consider two scenarios based on the Gregory patch approximation scheme of [9]. Gregory patches are used to increase the degrees of freedom necessary to form a tangent plane smooth surface with collection of patches surrounding an extraordinary vertex. A bicubic Gregory patch has 20 control points that interpolate the corner control points and form cubic Bézier boundaries. There are 8 additional interior control points that are rationally blended according to the u, v domain parameters. Once the blend has been computed, the patch can be evaluated as a bicubic Bézier patch.

In the first scenario, we assume quad faces, but not isolated extraordinary vertices. The scheme is general enough to handle triangular faces as well, but this complicates the implementation significantly. For the quad only case, a patch type is characterized by the valences of the four vertices. Excluding rotational symmetries, for a maximum valence of 12, there are 449 patch types. Allowing triangular faces would greatly increase this number. Precomputing these change of basis matrices, is non-trivial, greatly reducing the flexibility of this approach. In our implementation, we generate each $20 \times (2n+8)$ change of basis matrix on demand. Our second implementation presumes isolated extraordinary vertices. This means that we have exactly 9 change of basis matrices for all cases (up to valence 12) that we precompute. In either case, each of 20 hull shader threads will execute $6n + 24$ FLOPS.

The Gregory patch domain shader is nearly identical to the bicubic Bézier domain shader. The different is the rational blend step driven by conditional assignments based on the u, v coordinates of the current domain location. This adds 8 additional multiplies and 4 divides, for a total of 174 FLOPS.

5 Results

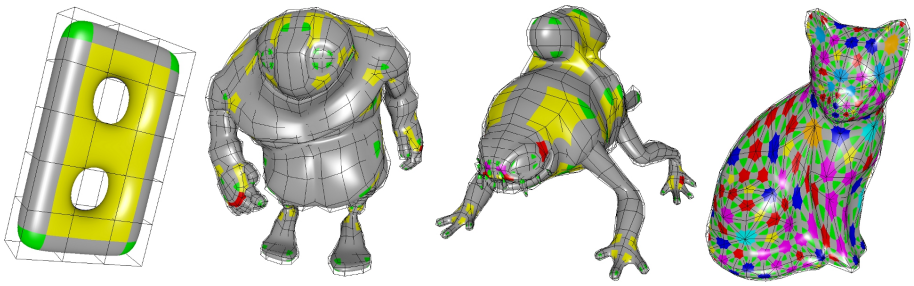

Figure 4: Models used in our timing measurements, from left to right: Twohole, Bigguy, Monsterfrog, and Cat. Regular regions of the surfaces are color gray. Extraordinary regions are colored by valence n.

We implement our algorithms using DirectX 11 compute shaders and tessellation pipeline. A compute shader is a GPGPU programming construct much like a CUDA kernel; but it is more tightly coupled to the graphics pipeline (using the same device context), it is coded in hlsl (High Level Shader Language), and is device independent. This allows us to run experiments on the latest high end GPUs from ATI (Radeon 5870) and nVidia (GTX 480). All test were run on a Dell Studio XPS 435T with a Core i7-920 processor.

We test our implementations on the four models depicted in Figure 4. Three of the models, Twohole, Bigguy, and Monsterfrog are quad meshes; the Cat is a triangle model. While the quad meshes only require one level of subdivision to isolate extraordinary vertices, we can apply a Gregory patch approximation without subdivision. For the Cat, being a triangulation, requires two levels of subdivision to isolate extraordinary vertices. While a triangulation may not be an ideal candidate for Catmull-Clark subdivision, we include this model since it has a large number and variety of extraordinary vertices.

Due to the lack of performance analysis tools (at the time of this writing) for the GTX 480, we confine ourselves to measuring frame rates with and without subdivision turned on. Differences in these measurements indicate the performance of the subdivision step(s), given in milliseconds in Table 1.

	Twohole	Bigguy	Monsterfrog	Cat
Radeon 5870	0.30	0.31	0.31	0.35
GTX 480	0.33	0.36	0.37	0.48

Table 1: Timings in milliseconds for the subdivision stage.

We can see that the Radeon 5870 slightly outperforms GTX 480 for running the subdivision kernels. In both cases, these measurements indicate that the subdivision step represents a very small fraction of overall frame time.

For the tessellation stage, we run our algorithms over these models at various tessellation rates. These algorithms are: Gregory approximation with no subdivision (Gregory1), Gregory approximation with subdivision (Gregory2), Stam's direct evaluation procedure (Stam), and curvature continuous biseptic patching (Biseptic). Our tessellation factors are uniformly integer powers of 2, going from 1 to 32 on the horizontal axis in our timing graphics (see Figure 5). We don't take advantage of adaptive tessellation here because each scheme would generate different tessellation patterns that would impact the cost of evaluation as well as rasterization, biasing our comparisons. On the vertical axis we show frames per second with log base 10 scaling.

Clearly, the GTX 480 performs much better for dense tessellations, realizing performance increases of 5 to 10x over Radeon 5870. GTX 480 has multiple rasterization and tessellation units, so this performance difference is not unexpected.

As expected, we see that Gregory1 (no subdivision) is the best performer at

low tessellation rates (less than 4 division per edge). On Radeon 5870, we see that Gregory2 (one level of subdivision, 4x the patches) actually outperforms Gregory1 at tessellation level 4 and above for Bigguy and Monsterfrog; but this is not the case on GTX 480. However, above tessellation level 16 on GTX 480, the performance for Bigguy and Monsterfrog is roughly the same for Gregory1 and Gregory2. We attribute this to the fact that Gregory2 will perform more regular evaluations and fewer extraordinary evaluations for the same sampling rate.

Also as expected, we see that Gregory1 and Gregory2 always outperform Stam and Biseptic. Somewhat unexpected however, is the extremely poor performance of Biseptic on Radeon 5870 at high tessellation rates. We suspect that this is an anomalous case outside the design parameters of ATI's tessellator implementation. Given that Biseptic requires fewer FLOPS than Stam, the performance similarity between Stam and Biseptic on GTX 480 is also somewhat puzzling. Our expectation was that for low tessellation levels, Stam would outperform Biseptic due to the lower cost of hull shading. As tessellation rates go up, we expected the lower cost of Biseptic evaluation would give it better performance. We see the former, but not the latter.

6 Conclusion

We have presented a method of preforming Catmull-Clark subdivision in parallel on programmable graphics hardware. We perform only one or two levels of subdivision in order to isolate extraordinary vertices. This simplified input is feed to the hardware tessellation pipeline where we analyze the performance of exact Catmull-Clark evaluation utilizing Stam's algorithm, a curvature continuous patching scheme based on biseptic Bézier patches, and a Gregory patch based approximation scheme.

Our results confirm that a Gregory patch approximation is always significantly faster than exact evaluation. This indicates that for displacement mapped surfaces, such as characters in games, an approximation to the limit surface will give better performance. Reasons not to use an approximation are limited to accuracy and faithfulness to the artist intent. While valid concerns, using the approximation within the tools chain would solve the problem.

For applications where the exact limit surface is a requirement, our results show that Stam's direct evaluation procedure performs well and can absolutely be used for adaptive real-time rendering. While widely used in games and computer generated animations, the lack of curvature continuity limits the use of Catmull-Clark surfaces in other engineering disciplines. One possible solution to this problem is the use of curvature continuous biseptic patches. We have demonstrated that theses patches can be evaluated with roughly the same performance as direct evaluation. Clearly, biseptic patches at high tessellation rates challenge Radeon 5870. We speculate that this may be due to limited register

or shared memory availability, forcing off-chip memory accesses that severely harm performance.

Finally, we note that for dense tessellation rates, performing subdivision prior to tessellation has negligible impact on performance. This suggest that for micropolygon rendering, pre-tessellator subdivision may be useful. Generating more, smaller, flatter, more uniform patches will lead to better sample rate estimates and more uniformly size triangles.

While we handle meshes with boundary rather simply in our subdivision stage, we would like to incorporate state-of-the-art boundary and crease rules [1, 6] into both our subdivision and tessellation stages. We expect that the simplification created by extraordinary vertex isolation will greatly ease this task.

References

[1] Henning Biermann, Adi Levin, and Denis Zorin. Piecewise smooth subdivision surfaces with normal control. In *proceedings of SIGGRAPH*, pages 113–120, 2000.

[2] Jeffrey Bolz and Peter Schröder. Rapid evaluation of catmull-clark subdivision surfaces. In *Proceeding of the International Conference on 3D Web Technology*, pages 11–17, 2002.

[3] Michael Bunnell. Adaptive tessellation of subdivision surfaces with displacement mapping. In *GPU Gems 2*, pages 109–122. 2005.

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

[5] Matthew Fisher, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, William R. Mark, and Pat Hanrahan. Diagsplit: parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics*, 28(5):150:1–150:10, 2009.

[6] Denis Kovacs, Jason Mitchell, Shanon Drone, and Denis Zorin. Real-time creased approximate subdivision surfaces. In *Proceedings of the symposium on Interactive 3D graphics*, pages 155–160, 2009.

[7] Charles Loop and Scott Schaefer. Approximating catmull-clark subdivision surfaces with bicubic patches. *ACM Trans. Graph.*, 27(1):8:1–8:11, 2008.

[8] Charles Loop and Scott Schaefer. G^2 Tensor Product Spline Surfaces over Extraordinary Vertices. *Computer Graphics Forum*, 27(5):1373–1382, 2008. Proceedings of SGP 2008.

[9] Charles Loop, Scott Schaefer, Tianyun Ni, and Ignacio Casta no. Approximating subdivision surfaces with gregory patches for tessellation hardware. *Transactions on Graphics*, 28(5):151:1–151:9, 2009.

[10] Henry Moreton. Watertight tessellation using forward differencing. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 25–32, New York, NY, USA, 2001. ACM.

[11] Ashish Myles, Tianyun Ni, and Jörg Peters. Fast parallel construction of smooth surfaces from meshes with tri/quad/pent facets. *Computer Graphics Forum*, 27(5):1365–1372, 2008.

[12] Ashish Myles, Young In Yeo, and Jörg Peters. Gpu conversion of quad meshes to smooth surfaces. In *SPM '08: ACM symposium on Solid and physical modeling*, pages 321–326, 2008.

[13] Ahmad H. Nasri. Polyhedral subdivision methods for free-form surfaces. *ACM Trans. Graph.*, 6(1):29–73, 1987.

[14] Tianyun Ni, Young In Yeo, Ashish Myles, Vineet Goel, and Jörg Peters. Gpu smoothing of quad meshes. In *SMI '08: IEEE International Conference on Shape Modeling and Applications*, pages 3–9, 2008.

[15] Anjul Patney, Mohamed S. Ebeida, and John D. Owens. Parallel view-dependent tessellation of catmull-clark subdivision surfaces. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 99–108, New York, NY, USA, 2009. ACM.

[16] Anjul Patney and John D. Owens. Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia)*, 27(5), December 2008.

[17] Le-Jeng Shiue, Ian Jones, and Jörg Peters. A realtime gpu subdivision kernel. *ACM Trans. Graph.*, 24(3):1010–1015, 2005.

[18] Jos Stam. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. *Computer Graphics*, 32(Annual Conference Series):395–404, 1998.

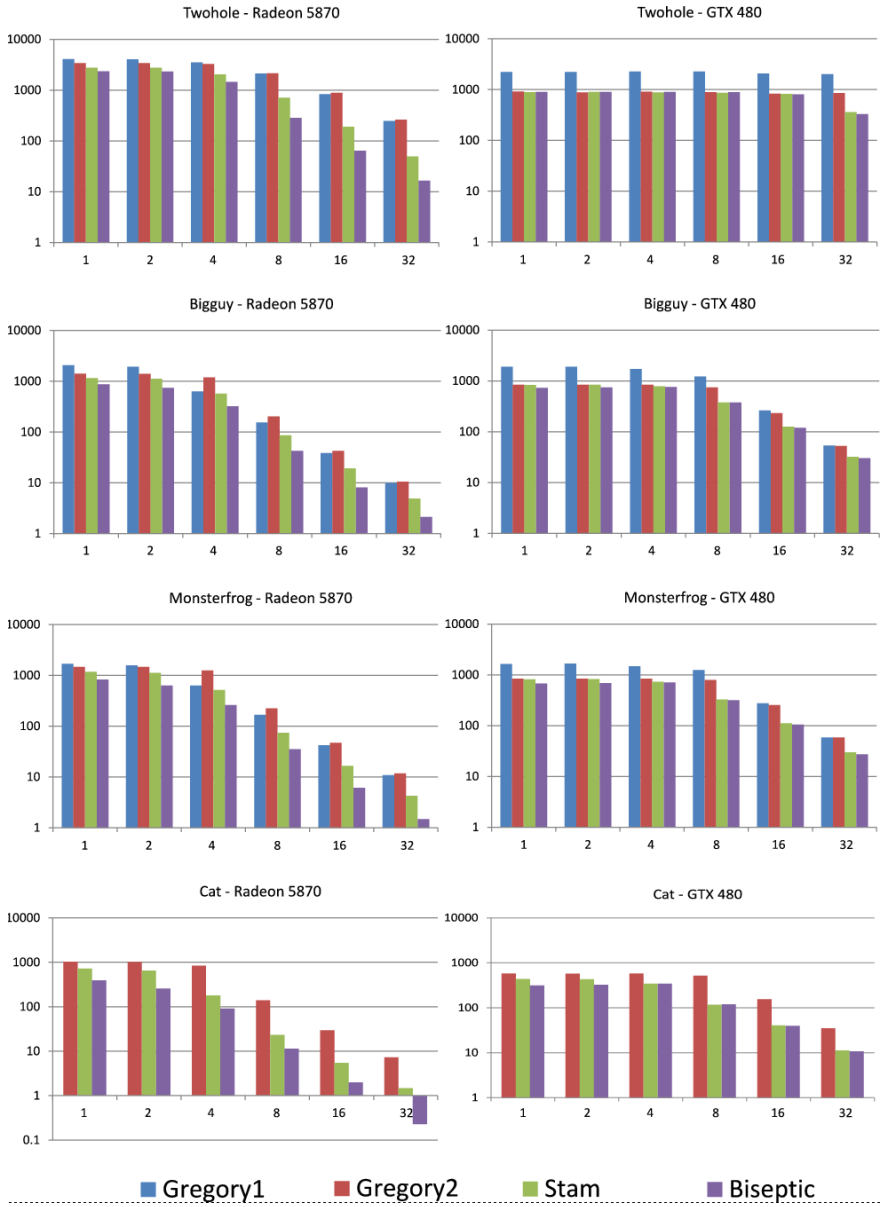

Figure 5: Timing results for our tessellator algorithms. The vertical axis measures frames per second with log base 10 scaling. The horizontal axis measures uniform edge tessellation factors with log base 2 scaling.