Feature Adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces

Matthias Nießner
University of Erlangen-Nuremberg
and
Charles Loop
Microsoft Research
and
Mark Meyer and Tony DeRose
Pixar Animation Studios

We present a novel method for high-performance GPU based rendering of Catmull-Clark subdivision surfaces. Unlike previous methods, our algorithm computes the true limit surface up to machine precision, and is capable of rendering surfaces that conform to the full RenderMan specification for Catmull-Clark surfaces. Specifically, our algorithm can accommodate base meshes consisting of arbitrary valence vertices and faces, and the surface can contain any number and arrangement of semi-sharp creases and hierarchically defined detail. We also present a variant of the algorithm which guarantees watertight positions and normals, meaning that even displaced surfaces can be rendered in a crack-free manner. Finally, we describe a view dependent level-of-detail scheme which adapts to both the depth of subdivision and the patch tessellation density. Though considerably more general, the performance of our algorithm is comparable to the best approximating method, and is considerably faster than Stam's exact method.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Picture/Image Generation—*Display Algorithms*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Curve, surface, solid, and object representations*

General Terms: Subdivision Surfaces, GPU

Additional Key Words and Phrases: Catmull-Clark subdivision surfaces,

Matthias Nießner, email: matthias.niessner@cs.fau.de; Charles Loop, email: charles.loop@microsoft.com; Mark Meyer, Tony DeRose, {mmeyer,derose}@pixar.com

DOI 10.1145/XXXXXXX.YYYYYYY
http://doi.acm.org/10.1145/XXXXXXX.YYYYYYY

1. INTRODUCTION

Catmull-Clark subdivision surfaces [Catmull and Clark 1978] have been used extensively by the feature film industry for some time. With the advent of hardware tessellation units on GPUs, they are becoming increasingly attractive for use in games. Hardware tessellation exploits the compact representation and data independence of patch primitives to produce triangle data for immediate consumption by the rasterization stage. This means that only the vertices of a coarse model need to be animated; the GPU can amplify the coarse geometry on-the-fly with very little memory bandwidth to produce a dense tessellation of the surface. Subdivision surfaces seem ideally suited to this paradigm. However, efficiently using tessellation hardware to render Catmull-Clark subdivision surfaces has previously relied either on the construction of approximating patches [Loop and Schaefer 2008; Myles et al. 2008; Ni et al. 2008; Loop et al. 2009], or on the direct, but considerably slower Stam evaluation algorithm [Stam 1998].

Since their introduction in 1978, Catmull-Clark surfaces have been extended in a number of ways, including the treatment of boundaries [Nasri 1987], infinitely sharp creases [Hoppe et al. 1994], semi-sharp creases [DeRose et al. 1998], and hierarchically defined detail [Pixar Animation Studios 2005]. The introduction of semi-sharp creases has proven to be particularly important as they allow realistic edges to be defined while keeping memory footprints small. For example, the rounded edges of the steel frame of the *Garbage Truck* shown in Figure 9 were created with semi-sharp creases, requiring only a few bytes of tag data per creased edge. Achieving a similar shape without semi-sharp creases would require a base mesh with significantly more vertices, faces, and edges. Similarly, the use of hierarchical edits, as introduced by Forsey and Bartels [1988] and as defined in the RenderMan specification, allows detail at various resolutions to be specified much more efficiently than globally refining the mesh. An example is shown in Figure 10(b) where the sandy terrain is modeled at one scale, and the footprints are details that appear at a much finer scale.

To date there are no GPU algorithms for the real-time rendering of Catmull-Clark surfaces possessing semi-sharp creases or hierarchical detail. In this paper, we present the first such algorithm. It uses a combination of GPU compute kernels and hardware tessellation to adaptively patch Catmull-Clark surfaces. Moreover, the algorithm is exact rather than approximating (by exact we mean that the vertices of our patch tessellations lie exactly on the limit surface up to machine precision), and we present a variant that is capable of creating watertight tessellations. This variant additionally satisfies

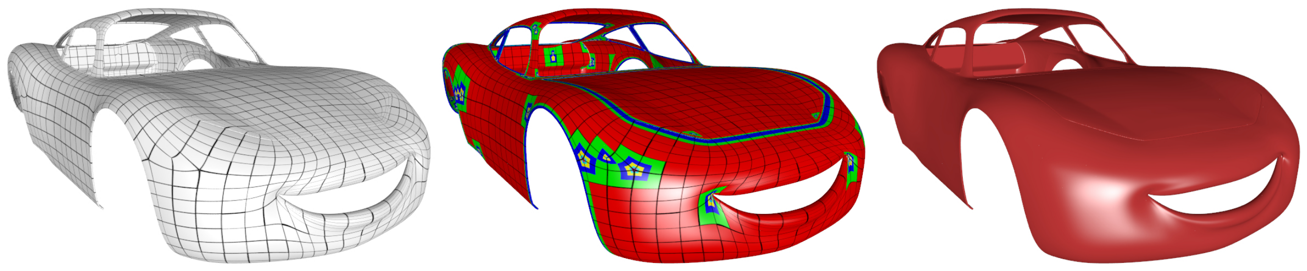


Fig. 1. Input base mesh (left), subdivision patch structure (center), and final model rendered with our method (right) © Disney/Pixar

a much stronger condition: namely, that abutting patches meet with *bitwise identical* positions and normals. With this stronger property surfaces with normal displacements are also guaranteed to be crack-free.

The algorithm is based on the idea of feature-adaptive subdivision. It has long been known that regular faces of a Catmull-Clark base mesh (that is, quad faces whose vertices have exactly four neighbors) generate a single bicubic patch in the limit of infinite subdivision, and that regions around extraordinary vertices (vertices having other than four neighbors) give rise to an infinite nesting of bicubic patches that approach a well defined limit [Doo and Sabin 1978]. A similar recursive nesting of bicubic patches also occurs near other kinds of features such as semi- or infinitely sharp creases, as well as near regions affected by hierarchical detail. Our algorithm exploits this fact to subdivide the mesh only in the vicinity of these features. At each stage of local subdivision, new bicubic patches are generated that are directly rendered using hardware tessellation. Since we only subdivide locally, our time and memory requirements are significantly less than the naive approach of globally subdividing the entire base mesh each step.

In addition to being more accurate and general than previous algorithms, the performance of our approach is competitive with an optimized implementation of the fastest approximate scheme based on Gregory Patches [Loop et al. 2009]. We also show that our approach is significantly faster than a GPU implementation of Stam's direct evaluation procedure (see Section 8). Furthermore, we also demonstrate that iteratively refining a mesh is inherently memory I/O bound, while our algorithm utilizes hardware tessellation in order to avoid this limitation.

1.1 Algorithm Overview and Contributions

Feature adaptive rendering involves a CPU preprocessing step, as well as a GPU runtime component. Input to our algorithm is a base control mesh consisting of vertices and faces, along with optional data consisting of semi-sharp crease edge tags and hierarchical details. In the CPU preprocessing stage, we use these data to construct tables containing control mesh indices that drive our feature adaptive subdivision process. Since these subdivision tables implicitly encode mesh connectivity, no auxiliary data structures are needed for this purpose. A unique table is constructed for each level of subdivision up to a prescribed maximum, as well as final patch control point index buffers as described in Section 3.2. The base mesh, subdivision tables, and patch index data are uploaded to the GPU, one time only, for subsequent runtime processing. The output of this phase depends only on the topology of the base mesh, crease edges, and hierarchical detail – it is independent of the geometric location of the control points.

For each frame rendered on the GPU, we use a two phase process. In the first phase, we execute a series of GPU compute kernels to perform data parallel Catmull-Clark subdivision of the base mesh. The index patterns used by the compute kernels to gather vertex data needed to perform each subdivision operation are entirely encoded in the subdivision tables. This process is repeated for each level of subdivision until a maximum depth is reached; this phase is described in Section 3.3. In the second GPU phase, we use the hardware tessellator unit to render the bicubic patches corresponding to the regular regions of the various subdivision levels computed in the first GPU phase (see Section 4). In Section 5, we develop the special treatment necessary to avoid cracks between adjacent patches, and in Section 7 we discuss how the flexibility of our algorithmic framework can be used to implement view dependent level-of-detail rendering. Since only base control point positions need to be updated on the GPU each frame, high frame rate animation of complex models is achieved. In this paper we address only the rendering of models once they have been animated. We do not address techniques for creating compelling animation.

In summary, the main contributions of our paper are:

—The first table driven data-parallel subdivision method that supports local refinement; we refer to this as *feature adaptive subdivision*.

—The first exact hardware tessellation algorithm of the true limit surface, including arbitrary valence vertices and faces, semi-sharp creases, and hierarchical details.

—A novel evaluation algorithm that guarantees bitwise identical evaluation of positions and normals of adjacent patches.

—A view dependent level-of-detail method that adapts to both subdivision level and patch density.

2. PREVIOUS WORK

The problem of GPU based rendering of Catmull-Clark surfaces has received considerable attention in recent years. We separate these approaches into three categories: global mesh refinement, direct evaluation, and approximate patching.

Global mesh refinement implements subdivision according to its standard definition. A base mesh is repeatedly (possibly adaptively) refined until the mesh achieves a sufficient density and then the resulting faces are rendered. The work of Bunnell [2005], Shiue et al. [2005], and Patney and Owens [2009] belong to this category. Global refinement schemes require significant memory I/O to stream control mesh data to and from GPU multiprocessors and global GPU memory (i.e., on- and off-chip). We demonstrate in Section 8 that performing global refinement iteratively is severely limited in performance as memory bandwidth becomes the bottleneck.

By leveraging the eigenstructure of a subdivision matrix Stam [1998] developed a method for directly evaluating subdivision surfaces at arbitrary parametric values. In order to apply this approach, a control mesh must have *isolated* extraordinary vertices. This means that two global refinement steps must be applied to an arbitrary control mesh (only one for a quad mesh) as a preprocess. Afterwards, Stam's algorithm can be easily implemented on the GPU using hardware tessellation. First, patch data must be transformed into eigenspace, making subsequent watertight boundary evaluation problematic. To do patch evaluation, one of three possible sets of precomputed eigenbasis functions must be evaluated. While this is feasible, the complexity of the algorithm and significant amount of computation limit performance, see Section 8. Furthermore, extending this approach to the evaluation of semi-sharp creases and hierarchical details remains a formidable challenge.

Another direct evaluation approach has been proposed by Bolz and Schröder [2002] originally targeting the CPU. Their approach exploits the fact that subdivision surfaces are linear functions of control point positions, meaning that a basis function can be associated with each control point. These basis functions must to be generated for all patch configurations. In order to keep basis function count within limits they require extraordinary vertices to be isolated. They indicate that approximately 5300 tables for the basis functions are required restricting vertex valence for interior patches to 12. Evaluation of surface points and derivatives is therefore reduced to dot products of the table values with control point positions. Extending their method to accommodate semi-sharp creases and hierarchical detail is problematic for several reasons. First, since crease sharpnesses can take on fractional values, there are an unbounded number of potential basis functions, meaning that the tables must be precomputed in a mesh dependent fashion, and the number of distinct tables can be very large. More fundamental is the fact that hierarchical detail coefficients are represented in local surface frames (cf. Forsey and Bartels [1988]), implying that the surface is no longer linear in the control vertices, and hence basis functions do not exist.

In anticipation of hardware tessellation, Loop and Schaefer [2008] noted the high cost of direct evaluation and the need for pre-tessellator subdivision. They 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. For instance, a quads only method was described by Myles et al. [2008] and Ni et al. [2008], a method to handle a mixture of triangles and quads was presented by Loop et al. [2009], and Myles et al. [2008] offer a method to deal with pentagonal patches as well as quads and triangles. Finally, the case of infinitely sharp creases in the context of an approximate Catmull-Clark subdivision on the GPU was handled by Kovacs et al. [2009].

It has been understood for decades that adaptive tessellation requires a specific strategy for eliminating cracks [Catmull 1974] at boundaries between distinct tessellation densities. Since a goal of our work is to create adaptive, crack-free renderings, our method is similar to that of Von Herzen and Barr [1987], where the notion of restricted quadtrees was introduced. A quadtree is said to be restricted if the subdivision levels of adjacent cells differs at most by one. With this restriction it is relatively easy to avoid cracks; fortunately the pattern of subdivision that naturally arises in our algorithm ensures this condition.

3. CATMULL-CLARK SUBDIVISION

Catmull-Clark subdivision takes a coarse base mesh as input and generates as output a new refined mesh containing more faces, edges, and vertices. By repeating this process on the sequence of new meshes, a smooth limit surface is obtained. The algorithm is defined by a simple set of rules, called the smooth rules, which are used to create new face points (f_j), edge points (e_j) and vertex points (v_j) as a weighted average of points of the previous level. Special rules are used for handling features such as boundaries and creases.

3.1 Subdivision Rules

The Catmull-Clark smooth subdivision rules for face, edge, and vertex points, as labeled in Figure 2, are defined as:

—Faces rule: f^{i+1} is the centroid of the vertices surrounding the face.

—Edge rule: $e_j^{i+1} = \frac{1}{4}(v^i + e_j^i + f_{j-1}^{i+1} + f_j^{i+1})$,

—Vertex rule: $v^{i+1} = \frac{n-2}{n} v^i + \frac{1}{n^2} \sum_j e_j^i + \frac{1}{n^2} \sum_j f_j^{i+1}$.

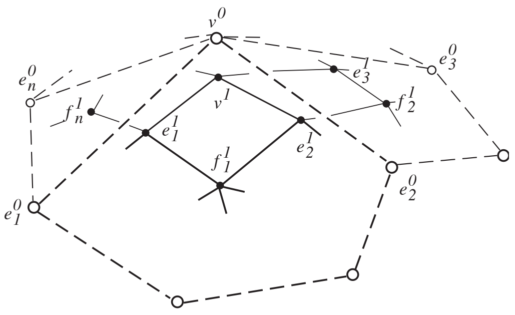


Fig. 2. Labeling of vertices of the base mesh around the vertex v^0 of valence n.

Following DeRose et al. [1998], a crease is defined by adding sharpness tags to edges. Subdividing a sharp edge creates two child edges, each of which are tagged with the sharpness value of the parent minus one. A vertex v_j containing exactly two crease edges e_j and e_k is considered to be a crease vertex. The following *sharp rules* are used for both boundaries and sharp edges (crease rules):

—$e_j^{i+1} = \frac{1}{2}(v^i + e_j^i)$

—$v_j^{i+1} = \frac{1}{8}(e_j^i + 6v^i + e_k^i)$

If a vertex is adjacent to three or more sharp edges or located on a corner then we derive it's successor by $v^{i+1} = v^i$ (corner rule).

In order to deal with fractional smoothness and propagate sharpness properly we use a slightly modified scheme of DeRose et al. [1998] where $e.s$ defines the sharpness of an edge:

—Face points are always the average of the surrounding points

—e with $e.s = 0 \rightarrow$ smooth rule

—e with $e.s \geq 1 \rightarrow$ crease rule

—e with $0 \leq e.s \leq 1 \rightarrow (1 - e.s) \cdot e_{smooth} + e.s \cdot e_{crease}$

Now, we introduce the vertex sharpness $v.s$ to handle vertices, where $v.s$ is the average of all incident edge sharpnesses and k is the number of edges around a vertex v with $e.s > 0$:

—v with $k < 2 \rightarrow$ smooth rule

—v with $k > 2 \wedge v.s \geq 1.0 \rightarrow$ corner rule
—v with $k > 2 \wedge 0 \leq v.s \leq 1 \rightarrow (1 - v.s) \cdot v_{smooth} + v.s \cdot v_{corner}$
—v with $k = 2 \wedge v.s \geq 1.0 \rightarrow$ crease rule
—v with $k = 2 \wedge 0 \leq v.s \leq 1 \rightarrow (1 - v.s) \cdot v_{smooth} + v.s \cdot v_{crease}$

Figure 3 shows the results when applying these rules on a pyramid. The edges of the pyramid's base plane are tagged as sharp.

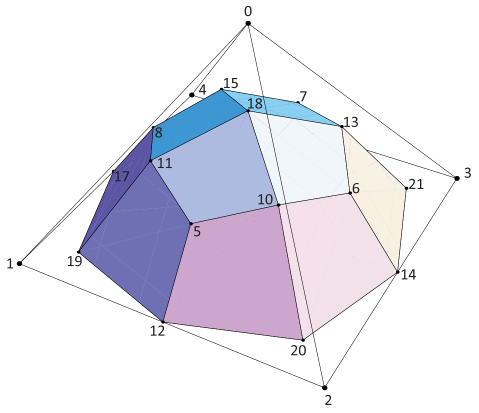
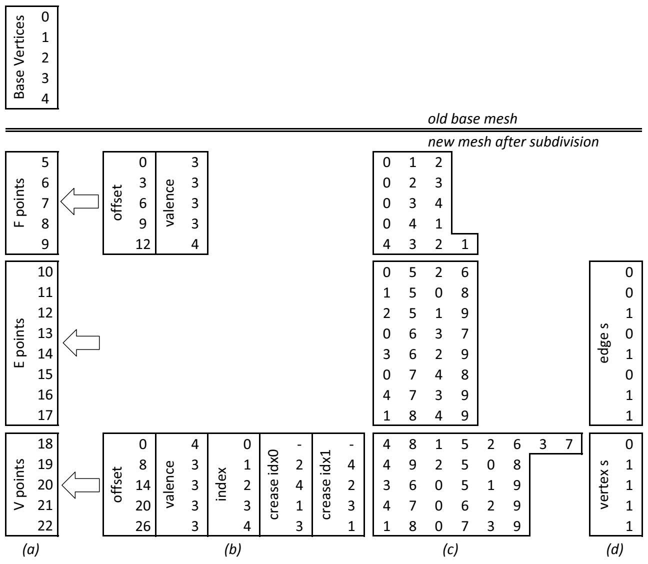


Fig. 3. One subdivision step applied on a pyramid where the edges of the base plane are tagged sharp.

3.2 Data-Parallel Subdivision

While the subdivision rules of Section 3.1 are straightforward to implement on a CPU, an efficient implementation on the GPU is non-trivial since neighborhood information is required. Fortunately, in most feature film and game applications the connectivity and sharpness tags of a mesh are typically invariant during animation, so a precomputed table driven approach is feasible. These tables are used at runtime to efficiently guide the subdivision computations. Due to the data dependency of the subdivision rules, face points must be computed first, followed by edge, and then vertex points. We use three separate compute kernels, one for each point type respectively. All kernels operate on a single vertex buffer, used for all subdivision levels. We justify this strategy as it a) simplifies our table construction, and b) optimizing this will have little impact on frame rate. The vertices of the base mesh, which may be animated at runtime, occupy a section at the beginning of this buffer. Starting from the base mesh, the subdivision kernels will compute in parallel the refined mesh for the next subdivision level.

The tables for the face, edge and vertex kernel are defined as follows. The face kernel requires two buffers: one index buffer, whose entries are the vertex buffer indices for each vertex of the face; a second buffer stores the valence of the face along with an offset into the index buffer for the first vertex of each face. Since a single (non-boundary) edge always has two incident faces and vertices, the edge kernel needs a buffer for the indices of these entities. In order to apply the edge rules, we also store the edge sharpness values $e.s$. The data for the vertex kernel is similar to the face kernel. We use an index buffer containing the indices of the incident edge and vertex points. We also need a second buffer to store the vertex valence, an index to predecessor of the vertex, the vertex sharpness $v.s$, and an offset to the starting index in the first buffer. For dealing with the case of a vertex on a crease, we must also store the indices of the edges that specify the crease (crease idx0, idx1). Figure 4 shows the subdivision tables for the pyramid in Figure 3. For meshes with boundary, the subdivision tables are adjusted according to the respective rules.



old base mesh

Base Vertices
0
1
2
3
4

new mesh after subdivision

F points	offset	valence				
5	0	3	0	1	2	
6	3	3	0	2	3	
7	6	3	0	3	4	
8	9	3	0	4	1	
9	12	4	4	3	2	1

E points					edge s
10	0	5	2	6	0
11	1	5	0	8	0
12	2	5	1	9	1
13	0	6	3	7	0
14	3	6	2	9	1
15	0	7	4	8	0
16	4	7	3	9	1
17	1	8	4	9	1

V points	offset	valence	index	crease idx0	crease idx1									vertex s
18	0	4	0	-	-	4	8	1	5	2	6	3	7	0
19	8	3	1	2	4	4	9	2	5	0	8			1
20	14	3	2	4	2	3	6	0	5	1	9			1
21	20	3	3	1	3	4	7	0	6	2	9			1
22	26	3	4	3	1	1	8	0	7	3	9			1

(a) *(b)* *(c)* *(d)*

Fig. 4. Subdivision tables for the pyramid of Figure 3: (a) is the vertex buffer, (b) contains topology information, (c) are indices which point into the vertex buffer and (d) provides the edge and vertex sharpness.

3.3 Feature Adaptive Subdivision

As mentioned in the introduction, it is well known that the limit surface defined by Catmull-Clark subdivision can be described by a collection of bicubic B-spline patches, where the set has infinitely many patches around extraordinary vertices, as illustrated in Figure 5(left). Similarly, near creases as shown in Figure 5(right), the number of limit patches grows as the crease sharpness increases.

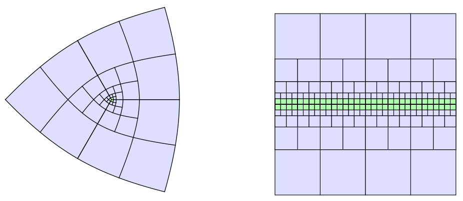


Fig. 5. The arrangement of bicubic patches (blue) around an extraordinary vertex (left), and near an infinitely sharp crease (right). Patches next to the respective feature (green) are irregular.

Feature adaptive subdivision proceeds by identifying regular faces at each stage of subdivision, rendering each of these directly as bicubic B-splines using hardware tessellation. Irregular faces are refined, and the process repeats at the next finer level. This strategy uses the same compute kernels as outlined in Section 3.2, however, the subdivision table creation is restricted to irregular faces. A face is regular only if it is a quad with all regular vertices, if none of its edges or vertices are tagged as sharp, and there are no hierarchical edits that would influence the shape of the limit patch. In all other cases the face is recognized as irregular, and subdivision tables are generated for a minimal number of subfaces. As before, all of this analysis and table generation is done on the CPU at preprocessing time.

Vertex and edge tagging is done at each level, depending on how many times the area around an irregular face should be subdivided. This might be the maximum desired subdivision depth around an extraordinary vertex, or the sharpness of a semi-smooth edge. As a result, each subdivision level will be a sequence of local control meshes that converge to the feature of interest (see Figure 6).

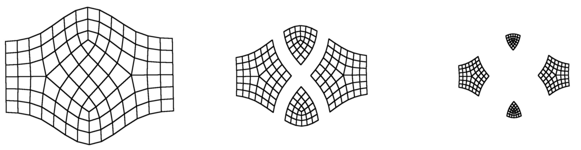


Fig. 6. Our adaptive subdivision scheme applied on a grid with four extraordinary vertices. Subdivision is only performed in areas next to extraordinary vertices.

The memory required to globally subdivide a mesh to level k is proportional to 4^kF, where F is the number of faces in the original mesh. Feature adaptive subdivision generally requires far less memory as the size of the subdivision tables is proportional to the total number of irregular faces at each subdivision level. The exact storage requirements depend on the number and arrangement of irregular faces, the sharpness of creases, and so on. However, an asymptotic upper bound can be obtained by making the worst case assumption that every irregular edge (an edge is irregular if it is adjacent to an extraordinary vertex, if it is tagged as a crease, or if a hierarchical edit influences the shape of one of the patches adjacent to the edge) is subdivided into two irregular edges. If there are e edge tags in the original mesh, the storage requirements for subdividing k levels is proportional to 2^ke. Since e is typically far smaller than the total number of edges in the original mesh, and since 2^k grows far less quickly than 4^k, we achieve a significant reduction in memory use. Subdivision around extraordinary vertices behaves even better, since the number of irregular faces grows linearly with respect to the extraordinary vertex count (v) per subdivision step ($\approx 12kv$). Actual memory requirements for various models are given in Section 8.4.

4. PATCH CONSTRUCTION

Once the subdivision stage is complete, we use the hardware tessellator to adaptively triangulate the resulting patches (see Direct3D features [Microsoft Corporation 2009]). The number and location of sample points on patch edges is determined by user provided *tess factors*. For each subdivision level we define two kinds of patches: full patches and transition patches.

4.1 Full Patches

Full patches (FPs) are patches that only share edges with patches of the same subdivision level. Regular FPs are passed through the hardware tessellation pipeline and rendered as bicubic B-splines. We ensure by feature adaptive subdivision that irregular FPs are only evaluated at patch corners. This means that for a given $tessfactor$ we must perform $\lceil \log_2 tessfactor \rceil$ adaptive subdivision steps. Since current hardware supports a maximum $tessfactor$ of 64 ($= 2^6$), no more than 6 adaptive subdivision levels are required. In order to obtain the limit positions and tangents of patch corners of irregular FPs we use a special vertex shader. Using this approach, our surface representation is exact; we show in Section 8 that this is significantly faster than direct evaluation as proposed by Stam [1998].

4.2 Transition Patches

Note that the arrangement of bicubic patches created by adaptive subdivision ensures that adjacent patches correspond either to the same subdivision level, or their subdivision levels differ by one. Patches that are adjacent to a patch from the next subdivision level are called *transition patches* (TPs). We additionally require that TPs are always regular. We enforce this constraint during the subdivision preprocess by marking for subdivision all irregular patches that might become TPs. This constraint significantly simplifies the algorithm at the expense of only a small number of additional patches.

To obtain crack-free renderings, the hardware tessellator must evaluate adjacent patches at corresponding domain locations along shared boundaries. Setting the tess factors of shared edges to the same value will ensure this. However, TPs by definition share edges with neighboring patches at a different subdivision level. One solution to this problem would be using compatible power of two tess factors so that the tessellations will line up. However, allowing only power of two tess factors is a severe limitation that reduces the available flexibility provided by the tessellation unit.

In order to avoid this limitation, we split each TP into several subpatches using a case analysis of the arrangement of the adjacent patches. Since each patch boundary can either belong to the current or to the next subdivision level, there are only 5 distinct cases as shown in Figure 7.

Each subdomain corresponds to a logically separate subpatch, though each shares the same bicubic control points with its TP siblings. Evaluating a subpatch involves a linear remapping of canonical patch coordinates (e.g., a triangular barycentric) to the corresponding TP subdomain, followed by a tensor product evaluation of the patch. This means that each subdomain type will be handled by draw calls requiring different constant hull and domain shaders; though we batch these according to subpatch type. However, since the control points within a TP are shared for all subpatches, the vertex and index buffers are the same. The overhead of multiple draw calls with different shaders but the same buffers becomes negligible for a larger number of patches.

By rendering TPs as several logically separate patches, we eliminate all T-junctions in the patch structure of a surface. The TP structure around an extraordinary vertex is illustrated to the right. This means that as long we assign consistent tess factors to shared edges, in principle a crack-free rendered surface is obtained. In practice however, due to behavior of floating point numerics, additional care is required as discussed in Section 5.

The assignment of tess factors for patch edges should take into account the subdivision level i that generated the patch. We discuss various strategies for adapting the subdivision level and the assignment of tess factors in Section 7.

5. WATERTIGHT EVALUATION

Adjacent patches sharing identical tess factors is a necessary but not sufficient condition to guarantee watertight rendering. Since floating point multiplication is neither associative nor distributive, evaluating adjacent patches at the same parameter of a shared boundary may not produce bitwise identical results. These minor discrepancies may result in visible cracks (holes) between adjacent patches, particularly when patches belong to different subdivision levels.

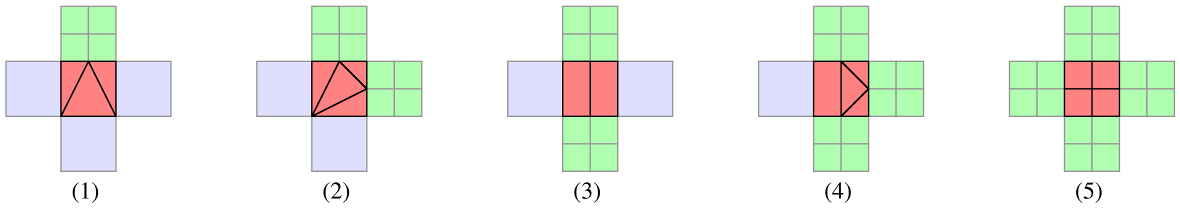


Fig. 7. There are five possible constellations for TP. While TP are colored red, the current level of subdivision is colored blue and the next level is green. The domain split of a TP into several subpatches allows full tessellation control on all edges, since shared edges always have the same length.

In this section, we present a method for the watertight evaluation of the patches generated by our feature adaptive subdivision approach. Moreover, we show that this method results in positions and normals that are bitwise identical along shared boundaries. We first describe a procedure for patches at the same subdivision level, then we describe one for patches that belong to different levels.

5.1 Same Subdivision Level

Castaño et al. [2008] propose an approach for watertight evaluation of Bézier patches that matches the order of computations performed with respect to either side of a shared boundary. However, they do not address the conversion to the Bézier basis, required for subdivision surfaces; or how to deal with irregular patches. Furthermore, their use of ownership assignments for normals at patch corners has a large memory footprint.

In contrast, we exploit the B-spline basis where the same input data is used for computing both positions and normals on either side of a shared patch boundary. Our approach requires that floating point addition and multiplication be commutative; fortunately, enabling the IEEE floating point strictness as a flag for the HLSL compiler will guarantee this.

We evaluate positions and derivatives of bicubic B-spline patches by appealing to their tensor product form:

$$\begin{aligned} S(u,v) &= N(u) \cdot P \cdot N(v), \\ \tfrac{\partial S}{\partial u}(u,v) &= \tfrac{dN}{du}(u) \cdot P \cdot N(v), \\ \tfrac{\partial S}{\partial v}(u,v) &= N(u) \cdot P \cdot \tfrac{dN}{dv}(v), \end{aligned}$$

where $N(t) = [N_0(t), \ldots, N_3(t)]$ are the univariate cubic B-spline basis functions, and $P = (P_{i,j})$ is the 4×4 array of patch control points. We perform the computation of $S(u,v)$ using repeated evaluation of univariate B-spline curves as follows. We first compute a point parameterized by u on each of the 4 curves defined by the columns of P. That is, we compute the row vector of 4 points $S(u) = [s_0(u), s_1(u), s_2(u), s_3(u)] = N(u) \cdot P$; we then compute the surface point $S(u,v)$ by univariate evaluation of $S(u) \cdot N(v)$. Derivatives are computed similarly using univariate evaluation. In the domain shader we check if $u = 0$ or $u = 1$ is true, or if $v = 0$ or $v = 1$ is true, then the evaluation must be on a domain boundary; if these are both true, then the evaluation must be on a domain corner. These cases are handled separately as follows.

5.1.0.1 *Domain Boundaries.* In this case, two patches will be evaluated independently at corresponding domain locations. Since there is no globally consistent u, v parameterization for a subdivision surface, there are a variety of cases to consider at a boundary shared by two patches A and B. It could be, for instance, that the boundary corresponds to $u_A = 1$ and $u_B = 0$, in which case $v_A = v_B$ along the boundary since both patches are right handed. Watertightness in this case is particularly easy to obtain since the v parameter values match on either side of the boundary.

The more challenging case is when the two patches have the shared boundary parametrized in opposite directions, for instance when $v_A = 1 - v_B$ along the shared boundary. As mentioned above, our surface evaluation method reduces to repeated evaluation of B-spline curves. We therefore require that our method produces bitwise identical results when reversing the parametric direction of a curve. To be more precise, consider a cubic B-spline curve $C(u)$ defined by control points $X = [X_0, X_1, X_2, X_3]^T$; that is $C(u) := N(u) \cdot X$. Now consider the curve $C^r(u_r)$ that is parametrized in the reverse direction: $C^r(u_r) := N(u) \cdot X^r$, where $X^r := [X_3, X_2, X_1, X_0]^T$. We require that our evaluation method is *reversal invariant* in that it satisfies

$$C(u) = C^r(1-u) \qquad \text{and} \qquad \frac{dC}{du}(u) = -\frac{dC^r}{du}(1-u) \quad (1)$$

where equality means bitwise identical results.

The first step is to evaluate the B-spline basis functions so that $N(u) = N^r(1-u)$, where $N^r(u) := [N_3(u), N_2(u), N_1(u), N_0(u)]$ and $\frac{dN}{du}(u) = -\frac{dN^r}{du}(1-u)$. The following, relying only on commutativity of floating point addition, is such a procedure that computes the homogeneous form of the basis functions and derivatives:

```
void EvalCubicBSpline(in float u,
                      out float N[4], out float NU[4]) {
   float T = u;
   float S = 1.0 - u;

   N[0] = S*S*S;
   N[1] = (4.0*S*S*S + T*T*T) + (12.0*S*T*S + 6.0*T*S*T);
   N[2] = (4.0*T*T*T + S*S*S) + (12.0*T*S*T + 6.0*S*T*S);
   N[3] = T*T*T;

   NU[0] = -S*S;
   NU[1] = -T*T - 4.0*T*S;
   NU[2] =  S*S + 4.0*S*T;
   NU[3] =  T*T;
}
```

Note that replacing u by $1-u$ in `EvalCubicBSpline` interchanges the values of S and T, leading to the reversal of both basis function values and derivatives. The inhomogeneous values of the basis functions and derivatives are computed by dividing by constant factors in a post division step that is the same on both sides of the shared boundary.

The final step is to perform the dot product of X and $N(u)$ to guarantee Equation 1. We do so as follows:

$$\begin{aligned} C(u) &= (X_0N_0(u) + X_2N_2(u)) + (X_1N_1(u) + X_3N_3(u)) \\ &= (X_0N_3(1-u) + X_2N_1(1-u)) + \\ &\quad (X_1N_2(1-u) + X_3N_0(1-u)) \\ &= \quad C^r(1-u) \end{aligned}$$

A similar derivation establishes the necessary constraint on the derivatives of C and C^r. Normal vectors are computed using the usual cross product formula. Guaranteeing that they are bitwise identical on either side of a shared boundary requires commutativity of both addition and multiplication. Again, these commutativity requirements are satisfied when using IEEE floating point strictness.

5.1.0.2 *Domain Corners.* In the domain corner case, bi-reversal invariant evaluation is required. For the corner $u = v = 0$ we structure the computation as

$$\begin{aligned} S(0,0) = \; & [(P_{0,0} \cdot N_0(0) \cdot N_0(0) + P_{2,2} \cdot N_2(0) \cdot N_2(0)) + \\ & (P_{2,0} \cdot N_2(0) \cdot N_0(0) + P_{0,2} \cdot N_0(0) \cdot N_2(0))] + \\ & [(P_{1,0} \cdot N_1(0) \cdot N_0(0) + P_{1,2} \cdot N_1(0) \cdot N_2(0)) + \\ & (P_{0,1} \cdot N_0(0) \cdot N_1(0) + P_{2,1} \cdot N_2(0) \cdot N_1(0))] + \\ & P_{1,1} \cdot N_1(0) \cdot N_1(0). \end{aligned}$$

The computation of partial derivatives as well as the values of S at the other corners, is handled similarly.

For irregular patches, we handle watertightness using a special vertex shader that computes the limit surface (see Section 4.1). By definition there are only corner points. Points that are adjacent to regular patches (those have a valence of 4) are evaluated bi-reversal invariantly as described above. The remaining points are all extraordinary vertices, having only irregular patches in common. Since these are evaluated on a per vertex level, the results are shared so no special treatment is required.

5.2 Between Subdivision Levels

Special care must be taken for patches, generated by different subdivision levels, that also share a boundary. As before, evaluations on either side of a boundary must use the same input control point data. In order to insure this, we define patches that share evaluations with a finer subdivision level as *watertight critical patches* (WCP). Note that a WCP can be either a FP or a TP. Referring to Figure 7, for a WCP that is also a TP it must be case 1 (domain boundary) or case 2 (domain corner). In this situation, we augment the 16 control points for the WCP with the 16 control points of its coarser level parent patch. These 32 total control points can still be handled efficiently by the tessellation unit. In the domain shader, if a point is on a boundary between the current and the previous subdivision level, the control points of the parent patch are used for computations. The evaluation itself is done as proposed in Section 5.1.

We verified our watertightness procedure empirically by streaming domain shader output to CPU memory and comparing the results of corresponding patch evaluations along shared boundaries. These results confirm the computations are indeed bitwise identical.

Our approach to watertight evaluation necessarily involves code branches that treat domain boundaries and corners differently than domain interiors. Depending on tessellation densities, this can result in a slowdown of as much as $2x$ (see Section 8), due to the SIMD nature of GPU code execution. In our implementations, we treat watertight rendering as an optional time versus image quality trade-off that may not be necessary for all applications, e.g., authoring versus game runtime.

6. EXAMPLES

In this section we present a number of examples to illustrate the range of modeling features that can be accommodated with our algorithm.

6.1 Extraordinary Vertices

The most common reason to use subdivision surfaces instead of bicubic B-splines is to deal with extraordinary vertices. Like previous algorithms, our method is capable of rendering meshes containing extraordinary vertices, as shown in Figure 11, where different colors denote different levels of adaptive subdivision. Notice how subdivision is only used in the neighborhood of extraordinary vertices, whereas regular regions are directly tessellated using bicubic B-splines.

Subdivision around extraordinary vertices reduces the area and thus the number of evaluations needed within irregular patches. By dividing the tess factor by two after each subdivision level (see Section 4.2), after $\lceil \log_2 \textit{tessfactor} \rceil$ subdivision levels the tess factor will become 1.0. Using adaptive subdivision allows us to reduce the evaluations within irregular patches until only the corners of the patch domain need to be evaluated. The limit surface positions and normals at domain corners are computed with a special vertex shader that implements limit masks as described in Halstead et al. [1993]. This way we achieve exact evaluation at all tessellation points, even in regions around extraordinary vertices.

6.2 Semi-sharp Creases

The generalization of Catmull-Clark surfaces to capture creases, both infinitely sharp and semi-sharp, has proven to be extremely useful in real-world applications such as geometric modeling [Hoppe et al. 1994], feature film production [DeRose et al. 1998], and video games [Kovacs et al. 2009].

Previous algorithms have been capable of accurately rendering infinitely sharp creases, but ours is the first that is capable of interactively rendering semi-sharp creases. A simple example of a model using semi-sharp creasing is shown in Figure 8, where different colors denote different levels of adaptive subdivision. Note that Figure 8(c) uses a fractional sharpness crease. Sharpnesses and fractional values are entirely encoded in precomputed subdivision tables. At runtime the tables are used to fill the vertex buffer with vertex positions for each adaptively subdivided patch at each level of subdivision. These patches are then sent to the tessellation unit for evaluation.

A more realistic example of the use of semi-sharp creases is the *Garbage Truck* shown in Figure 9, which appeared in a recent feature film. The base mesh consists of 5448 patches, and 3439 creased edges, with sharpnesses ranging from 1.0 to 3.0, including some fractional sharpnesses of 1.5 and 2.3. Achieving the tight radii of curvature without semi-sharp creases would significantly increase the memory footprint of the model because the base mesh would need to be significantly more dense in most regions. Another example of semi-sharp creasing is the *Car Body* shown in Figure 1. Semi-sharp creases were particularly helpful near the boundary of the hood to achieve a tight radius of curvature there.

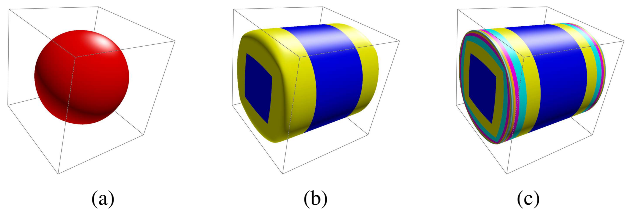


Fig. 8. All images are derived from a cube used as the base mesh. (a) no edges tagged; (b) front and backface edges have a sharpness of 3; (c) front and backface edges have a fractional sharpness of 7.8. Each color represents a distinct level of subdivision.

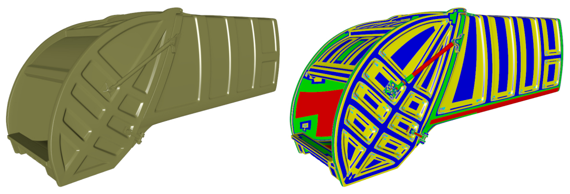


Fig. 9. Garbage Truck with semi-sharp creases. Adaptive subdivision levels indicated by color (right). © Disney/Pixar

6.3 Hierarchical Detail

Multiresolution modeling has received considerable attention in the graphics community, and can be traced back at least to the early work of Forsey and Bartels [1988]. When applied to subdivision surfaces, the idea is to represent a shape using a relatively sparse base mesh that captures the coarse features of the shape, together with hierarchical edits that are applied during subdivision to describe variation at finer scales. The scheme used by RenderMan is typical of these methods [Pixar Animation Studios 2005]. Each vertex generated through subdivision is assigned a unique index. A hierarchical edit consists of such an index together with a vector displacement. When the vertex with that index is created during subdivision, its position is offset by the vector displacement prior to subsequent subdivision. An example of this process is shown in Figure 10(b), where the base mesh captures the overall shape of the terrain, while the hierarchical edits are used to describe the footprints which occur at much finer scales.

In our method, adaptive subdivision is used in regions that are affected by hierarchical edits, and are encoded into subdivision tables as a precomputation on the CPU. During runtime, after each subdivision level, hierarchical edits applicable to that level are used to reposition vertices, then subdivision proceeds to finer levels.

Similar to hierarchical edits, T-splines [Sederberg et al. 2004] enable the representation of multiple resolutions of mesh data within a compact framework. While we have not validated our algorithm on T-spline meshes, we expect the feature adaptive principles underlying our work to be complementary to the T-spline paradigm.

6.4 Displacement Mapping

Displacement mapping involves offsetting a surface point in the normal direction by a scalar value stored in a texture map. In the context of hardware tessellation, it is crucial that both the position and normal of a surface on shared patch boundaries are bitwise identical; otherwise, cracks may appear in surface. Fortunately, a variant of our algorithm guarantees this, as illustrated in Figure 10(a).

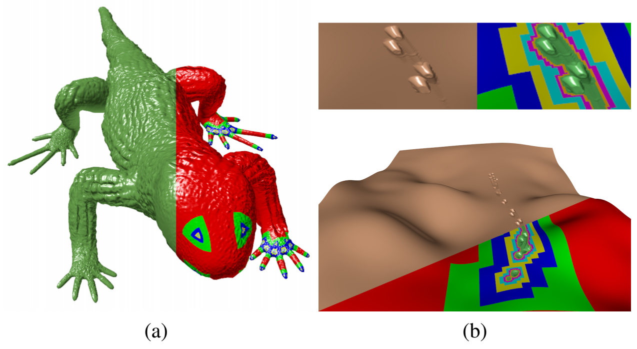


Fig. 10. Displacements (a) and hierarchical edits (b).

7. ADAPTIVE LEVEL OF DETAIL

The flexibility of our algorithmic framework can be used in a variety of ways to achieve adaptive level-of-detail control. There are two independent factors that can be used for LOD control in our framework: tess factor assignment and the depth at which subdivision is terminated. We examine each of these separately, then make specific recommendations for two important applications; namely, the rendering of characters and terrain.

Tess factor assignment: An important benefit of using hardware tessellation is the ability to vary the tessellation density of an object at runtime. This means that both the cost of over-tessellation and the poor image quality of under-tessellation can be avoided.

The hardware tessellator varies tessellation density of patches at runtime through user provided edge and interior tess factors. Tess factors are assigned in the *Hull Shader Constant Function* that is executed once for each patch. Each instance of this function must compute all the tess factors for the edges and interior of the patch. Instances corresponding to adjacent patches must provide the same tess factor for a shared edge to prevent cracks.

Since we have eliminated T-junctions from the patch structure of a model, we are free to assign these tess factors arbitrarily to shared patch edges (see Section 4). Tess factor assignment can either be done locally on a per edge basis requiring tess factor computations in the Hull Shader, or globally for an entire mesh. Local tess factors can be assigned according to some local edge based metric. Doing this based on screen space length, which approaches zero near silhouettes, can cause artifacts. More sophisticated approaches could avoid this, but at higher cost. For our applications we have achieve good results by simply using the viewing distance to edge midpoints. Alternatively, we can assign tess factors globally so that all patch edges of an object get the same value. The global tess factor could for example, be computed based on the distance from the camera to the centroid of the object. Such a global tess factor assignment would result in patches from higher levels of subdivision to appear more densely tessellated than those from lower subdivision levels. To avoid this, we assign the global tess factor to the zeroth subdivision level, halving it for each level of subdivision, resulting in a more uniform tessellation density. This strategy has proven to be effective for objects with small spatial extent, such as characters.

Adaptive subdivision level: Additional LOD management is obtained by terminating feature adaptive subdivision after an adaptively determined maximum level. This level could be based on an object's camera distance, similar to global assignment of a tess factor. This approach may result in irregular patches having a tess factor greater than 1.0, which means that a surface approximation such as Loop et al. [2009] would be required. Nevertheless, this might be reasonable for models containing very sharp creases. Such a scheme would be particularly effective for objects with a large spatial extent, such as terrains.

Given these two factors for adaptive LOD control, we recommend the following alternatives for characters and terrains, respectively.

Adaptive global tess factors with adaptive subdivision level: A global view dependent tess factor is computed per object to determine the maximum subdivision depth (i.e., $\lceil \log_2 \mathit{tessfactor} \rceil$). This combination of adaptive tess factor assignment and adaptive subdivision level is potentially the most reasonable LOD strategy for character animation, since characters usually have a limited spatial extent and thus costly per edge computations become unnecessary.

Adaptive local tess factors with adaptive subdivision level: The maximum tess factor that could possibly be assigned to any edge (based on the viewing distance) of a certain object is computed taking the respective bounding geometry into account. Thus, the most distant point on the bounding geometry determines the maximum subdivision level. As a result it is possible to locally assign tess factors on a per edge basis in the Hull Shader. If extraordinary vertices have been isolated, then irregular patches will never receive a tess factor greater than 1.0.

8. RESULTS

Our implementation uses DirectX 11 running under Windows 7. We used Direct Compute for GPU subdivision and the Direct3D 11 graphics pipeline to access the hardware tessellator. All GPU code was written in HLSL, and all timing measurements were made on a NVIDIA GeForce GTX 480. Timings are provided in milliseconds and account for all runtime overhead except for display of the GUI widgets, text rendering, etc..

8.1 Comparison to Global Mesh Refinement

Table I. Timing using the Big Guy model for our scheme (feature adaptive patching) compared against our global table driven subdivision method and the previously published GPU subdivision algorithm by Shiue et al.. Note that all timings include final rendering, while we additionally break out draw time for our global subdivision scheme.

Subdivision Level	0	1	2	3	4
Feature Adaptive Patching	0.10	0.20	0.34	0.81	2.30
Shiue Subdivision	0.62	7.26	13.97	21.42	34.93
Global Table Subdivision	0.06	0.18	0.79	3.07	12.05
Draw Time (Table Subd.)	0.04	0.06	0.37	1.45	5.78

In this section we show that our patch based feature adaptive approach is faster than repeated global refinement of a mesh on the GPU. This is a direct result of the high compute to memory bandwidth ratio of modern graphics processors; that is, fetching a value from memory takes as much time as a large number of floating point operations. This number is increasing with each new GPU generation. Recall that a global refinement approach must stream old and new mesh vertices to and from off-chip GPU memory. This requires a small amount of computation, but a large amount of memory I/O. By utilizing the hardware tessellator to process the large number of regular patches that arise in Catmull-Clark subdivision, we avoid this bottleneck since patches are evaluated and rendered on-chip requiring considerable computation but little memory I/O.

To demonstrate this point, we compare our scheme to two global refinement algorithms implemented on the GPU. The first utilized our table driven subdivision approach (see Section 3) to globally refine a mesh for each subdivision level. The second algorithm is a modern GPU implementation of the subdivision kernel proposed by Shiue et al. [2005]. Shiue's algorithm requires extraordinary vertices to be isolated (that is, no edge can be adjacent to two extraordinary vertices). We achieve this by statically pre-subdividing the mesh on the CPU. Note that neither our feature adaptive patching scheme nor our table driven subdivision has this limitation.

As shown in Table I our feature adaptive patching scheme outperforms global subdivision for all but the first subdivision levels. This can be easily explained because for the first subdivision levels little refinement is required; while our feature adaptive patching must always setup patches even if evaluations are only done at patch corners. However, beyond the first subdivision levels feature adaptive patching pays off since hardware tessellation requires less memory I/O. As the subdivision level increases this difference quickly becomes large. For subdivisions levels beyond level 4 we cannot do the comparison since global subdivision runs out of memory.

Furthermore, our feature adaptive patching scheme is faster than Shiue's algorithm for all subdivision levels and our table driven subdivision outperforms Shiue's algorithm in all cases (see Table I). Our implementation of Shiue differs slightly from the original approach in that we use the compute shader in order to launch threads instead of the pixel shader. This change was made to provide more freedom to optimize thread allocation in order to achieve best performance for Shiue's algorithm on modern GPUs. Due to its fundamental design their depth-first approach requires both a significant number of compute kernel invocations and draw calls. Redundant computations within the subdivision kernel and between distinct kernel calls further harm performance. This may have been a reasonable design when launching blocks of threads had to be done by the graphics pipeline using pixel shaders. In contrast to this depth-first approach, today's more generally programmable GPUs prefer breadth-first algorithms such as our table driven subdivision. Nevertheless, all GPU subdivision algorithms contain a significant amount of memory I/O due to their iterative nature and independent rendering of the resulting triangles. Our patching scheme, however, utilizes the tessellation unit which minimizes memory I/O; once patches are set up patch evaluations and final rendering is done on-chip without additional memory transfer. This behavior is clearly demonstrated in Table I where the draw time alone for global subdivision is already larger than our entire feature adaptive subdivision scheme (including all kernel launches and patch renderings) beyond subdivision level 2.

8.2 Comparison to Direct Evaluation and Approximate Patching Algorithms

We have shown in the previous section that using hardware tessellation allows rendering with a minimum of memory I/O, making it faster than iterative mesh refinement. We now compare our feature adaptive algorithm and its water-tight variant (see Section 5) against other patching schemes. Therefore, we consider Stam's direct evaluation algorithm [1998] and the approximate Gregory patching scheme proposed by Loop et al. [2009]. Since Stam evaluation requires isolated extraordinary vertices, in the following comparison we first perform one level of global subdivision for all algorithms. We use the *Big Guy* and *Monster Frog* models since they do not possess semi-sharp creases or hierarchical detail. The respective images including the patch structure is illustrated in Figure 11 while the timing results are shown in Figure 12.

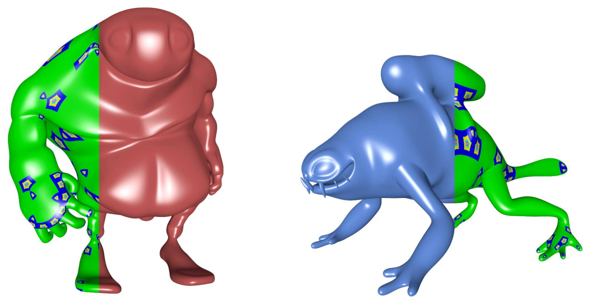


Fig. 11. Exact evaluation of subdivision surfaces using our adaptive scheme applied on the Big Guy (left) and Monster Frog (right) model. Respective timings are shown in Figure 12.

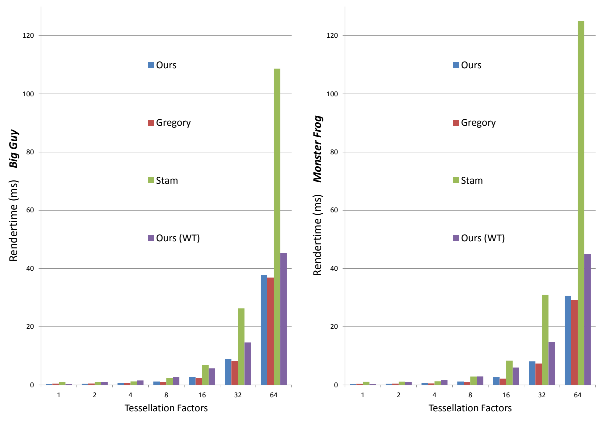


Fig. 12. Comparison of our method (w/ and w/o watertightness) against Stam evaluation (both exact) and the approximate Gregory scheme using different tess factors applied to the Big Guy (left) and Monster Frog (right) models.

Even though our algorithm is considerably more general than Stam's (because it can handle semi-sharp creases and hierarchical edits), our method runs faster on all tess factors. Moreover, the performance gap becomes larger as the tess factor increases due to the relatively expensive domain shader evaluation used in Stam's algorithm. Also note that the difference is larger on the *Monster Frog* model than for the *Big Guy* model. This is reasonable, since the *Monster Frog* contains a higher percentage of irregular patches than the *Big Guy* (764 of 1292 and 592 of 1450 irregular patches, respectively).

Compared to Gregory patches, our method is exact (at evaluation points) rather than approximating, more general, and is marginally faster when using small tess factors (due to the more expensive control point computation of the Gregory scheme). For higher tess factors our method becomes slightly slower, but still achieves a comparable performance.

Our watertight evaluation method produces bit-wise identical results, but at the cost of reduced performance caused by the required shader restructuring. The difference between normal and watertight rendering is especially noticeable when using smaller tess factors. This is due to the divergent code of the domain shader, and becomes less significant as the tess factor increases. Note that neither Gregory patches nor Stam evaluation can guarantee watertightness.

8.3 Semi-sharp Creases and Hierarchical Edits

Models containing semi-sharp creases cannot be handled by previous algorithms. Performance measurements of our algorithm for such models (the *Car Body* and the *Garbage Truck*) are given in Table II. Note that even for high tess factors real-time frame rates are achieved. Also note that employing the tessellator allows us to generate and render nearly one billion triangles per second.

Table II. Performance of our method on the Car Body and Garbage Truck models as a function of tess factor (TF).

	Car Body		Garbage Truck	
TF	Tris	Time (ms)	Tris	Time (ms)
1	109,251	1.58	644,286	10.54
2	136,839	1.60	655,138	10.57
4	216,529	1.68	723,070	10.59
8	883,713	1.92	1,183,478	10.90
16	2,725,881	4.24	3,786,922	12.90
32	9,440,953	10.51	11,735,594	23.82
64	34,014,791	39.40	40,584,514	54.36

Our method is also the first capable of interactively rendering models containing hierarchical detail. An example is shown in Figure 10(b) which depicts a sandy terrain consisting of very few faces in the base mesh (a 12×19 grid), together with fine scale footprints that are modeled using hierarchical detail. See the accompanying video for an animated version of this example.

8.4 Memory Requirements

As mentioned in Section 3.3, memory on the GPU needs to be allocated to store subdivision tables for each feature adaptive patch, and the vertex buffer needs to be large enough to store patch vertices at all levels of subdivision. The exact memory requirements for various levels of subdivision for particular models are shown in Figure 13. Modern GPUs are typically equipped with a gigabyte or more of buffer memory, so the memory requirements of our algorithm make it possible to simultaneously handle many such models.

9. CONCLUSION

We have presented a novel method of GPU based rendering of arbitrary Catmull-Clark surfaces. In contrast to previous algorithms, our method is exact and implements the full RenderMan specification of Catmull-Clark surfaces, including arbitrary base mesh topology, semi-sharp creases, and hierarchically defined detail. We also presented a variant of the algorithm that produces watertight positions and normals, allowing for the crack-free rendering of displaced surfaces. We demonstrated the method on feature film quality models, and showed that even for such complexity we are able to generate nearly one billion triangles per second.

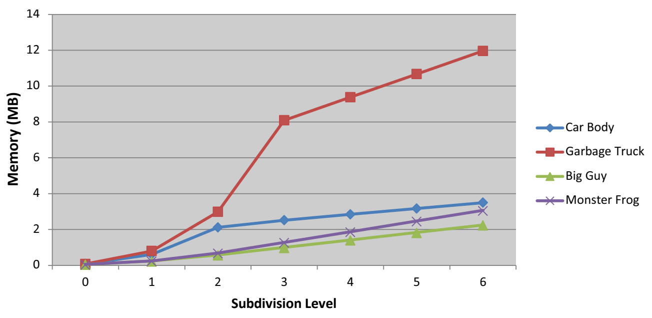


Fig. 13. Memory requirements to store vertex buffers and subdivision tables as a function of the maximum subdivision level.

Though this paper has focused on feature film applications, we believe our algorithm can be effectively used to increase the realism and cinemagraphic experience in the next generation of games.

ACKNOWLEDGMENTS

We thank Bay Raitt of Valve Software for the BigGuy and Monster-Frog models, and we thank Keith Klohn for the sand field example. The Carbody and Garbage Truck models are courtesy of Pixar Animation Studios.

REFERENCES

BOLZ, J. AND SCHRÖDER, P. 2002. Rapid evaluation of catmull-clark subdivision surfaces. In *Proceeding of the International Conference on 3D Web Technology*. 11–17.

BUNNELL, M. 2005. Adaptive tessellation of subdivision surfaces with displacement mapping. In *GPU Gems 2*. 109–122.

CASTAÑO, I. 2008. Tessellation of subdivision surfaces in DirectX 11. Gamefest 2008 presentation. http://developer.nvidia.com/object/gamefest-2008-subdiv.html.

CATMULL, E. 1974. Subdivision algorithms for the display of curved surfaces. Ph.D. thesis, The University of Utah.

CATMULL, E. AND CLARK, J. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design 10,* 6, 350–355.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes image rendering architecture. *SIGGRAPH Comput. Graph. 21,* 4, 95–102.

DEROSE, T., KASS, M., AND TRUONG, T. 1998. Subdivision surfaces in character animation. SIGGRAPH '98. 85–94.

DOO, D. AND SABIN, M. 1978. Behaviour of recursive division surfaces near extraordinary points. *Computer Aided Design 10,* 6, 356–360.

FISHER, M., FATAHALIAN, K., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. Diagsplit: parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics 28,* 5, 150:1–150:10.

FORSEY, D. R. AND BARTELS, R. H. 1988. Hierarchical b-spline refinement. *SIGGRAPH Comput. Graph. 22,* 4, 205–212.

HALSTEAD, M., KASS, M., AND DEROSE, T. 1993. Efficient, fair interpolation using catmull-clark surfaces. SIGGRAPH '93. ACM, New York, NY, USA, 35–44.

HOPPE, H., DEROSE, T., DUCHAMP, T., HALSTEAD, M., JIN, H., MCDONALD, J., SCHWEITZER, J., AND STUETZLE, W. 1994. Piecewise smooth surface reconstruction. *SIGGRAPH '94*, 295–302.

KOVACS, D., MITCHELL, J., DRONE, S., AND ZORIN, D. 2009. Real-time creased approximate subdivision surfaces. In *Proceedings of the symposium on Interactive 3D graphics*. 155–160.

LOOP, C. AND SCHAEFER, S. 2008. Approximating catmull-clark subdivision surfaces with bicubic patches. *ACM Trans. Graph. 27,* 1, 8:1–8:11.

LOOP, C., SCHAEFER, S., NI, T., AND NO, I. C. 2009. Approximating subdivision surfaces with gregory patches for tessellation hardware. *Transactions on Graphics 28,* 5, 151:1–151:9.

MICROSOFT CORPORATION. 2009. Direct3D 11 Features. http://msdn.microsoft.com/en-us/library/ff476342(VS.85).aspx.

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

MYLES, A., NI, T., AND PETERS, J. 2008. Fast parallel construction of smooth surfaces from meshes with tri/quad/pent facets. *Computer Graphics Forum 27,* 5, 1365–1372.

MYLES, A., YEO, Y. I., AND PETERS, J. 2008. Gpu conversion of quad meshes to smooth surfaces. In *SPM '08: ACM symposium on Solid and physical modeling*. 321–326.

NASRI, A. H. 1987. Polyhedral subdivision methods for free-form surfaces. *ACM Trans. Graph. 6,* 1, 29–73.

NI, T., YEO, Y. I., MYLES, A., GOEL, V., AND PETERS, J. 2008. Gpu smoothing of quad meshes. In *SMI '08: IEEE International Conference on Shape Modeling and Applications*. 3–9.

PATNEY, A., EBEIDA, M. S., AND OWENS, J. D. 2009. Parallel view-dependent tessellation of catmull-clark subdivision surfaces. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*. ACM, New York, NY, USA, 99–108.

PIXAR ANIMATION STUDIOS. 2005. The RenderMan Interface version 3.2.1. (https://renderman.pixar.com/products/rispec/index.htm).

SEDERBERG, T., CARDON, D., FINNIGAN, G., NORTH, N., ZHENG, J., AND LYCHE, T. 2004. T-spline simplification and local refinement. *SIGGRAPH Comput. Graph. 23,* 4, 276–283.

SHIUE, L.-J., JONES, I., AND PETERS, J. 2005. A realtime gpu subdivision kernel. *ACM Trans. Graph. 24,* 3, 1010–1015.

STAM, J. 1998. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. *Computer Graphics 32,* Annual Conference Series, 395–404.

VON HERZEN, B. AND BARR, A. H. 1987. Accurate triangulations of deformed, intersecting surfaces. SIGGRAPH '87. ACM, New York, NY, USA, 103–110.