Parallel View-Dependent Tessellation of Catmull-Clark Subdivision Surfaces

Anjul Patney
University of California, Davis

Mohamed S. Ebeida
Carnegie Mellon University

John D. Owens
University of California, Davis

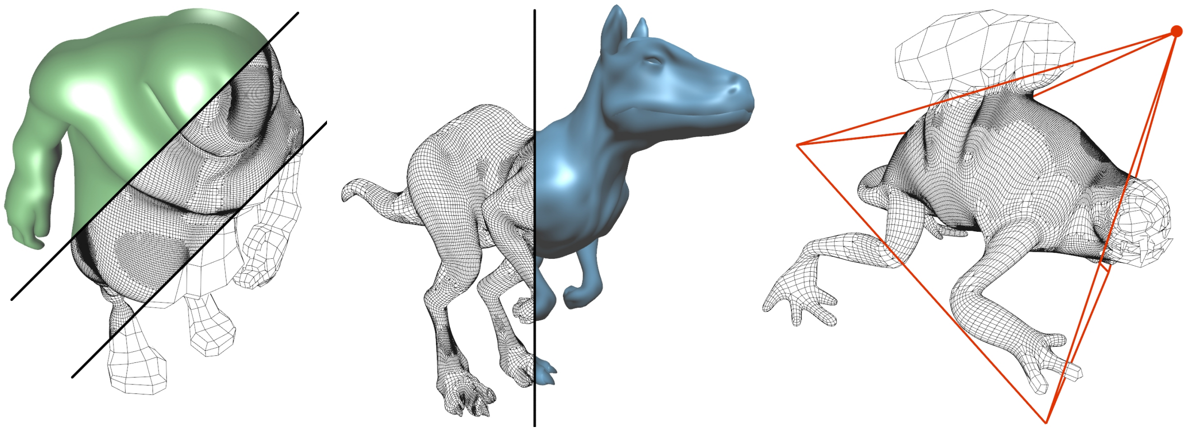


Figure 1: *We adaptively subdivide faces of a Catmull-Clark subdivision mesh until the screen-space geometric criterion is met. Using a parallel approach to the subdivision procedure, we are able to obtain interactive performance for complex real-life models. Moreover, we ensure that the subdivided mesh is free of cracks and T-junctions.*

Abstract

We present a strategy for performing view-adaptive, crack-free tessellation of Catmull-Clark subdivision surfaces entirely on programmable graphics hardware. Our scheme extends the concept of breadth-first subdivision, which up to this point has only been applied to parametric patches. While mesh representations designed for a CPU often involve pointer-based structures and irregular per-element storage, neither of these is well-suited to GPU execution. To solve this problem, we use a simple yet effective data structure for representing a subdivision mesh, and design a careful algorithm to update the mesh in a completely parallel manner. We demonstrate that in spite of the complexities of the subdivision procedure, real-time tessellation to pixel-sized primitives can be done.

Our implementation does not rely on any approximation of the limit surface, and avoids both subdivision cracks and T-junctions in the subdivided mesh. Using the approach in this paper, we are able to perform real-time subdivision for several static as well as animated models. Rendering performance is scalable for increasingly complex models.

CR Categories: I.3.7 [Computing Methodologies]: Computer Graphics—Three-Dimensional Graphics and Realism; I.3.6 [Computing Methodologies]: Computer Graphics—Methodology and Techniques

Keywords: Subdivision surfaces, Catmull-Clark, GPGPU, adaptive surface subdivision

Anjul Patney, Mohamed S. Ebeida, and John D. Owens. *Parallel View-Dependent Tessellation of Catmull-Clark Subdivision Surfaces.* In Proceedings of High Performance Graphics 2009, August 2009.

1 Introduction

Higher-order surfaces offer numerous advantages as rendering primitives, including ease of modeling and a compact representation. Of these, Catmull-Clark subdivision surfaces form one of the most popular primitives for off-line modeling and rendering. However, the difficulty of robustly subdividing such a surface in an interactive, view-dependent manner has thus far precluded its use in real-time graphics pipelines.

The programmability and peak computational capabilities of a modern graphics processor (GPU) make it an ideal candidate to perform such subdivision. However, GPUs are highly parallel processors, and thus adapting an already complex subdivision algorithm to such a parallel architecture is a challenging task. As a result, previous work in real-time adaptive subdivision has generally tackled less complex primitives such as parametric surfaces (specifically, Bézier patches and PN-triangles) [Patney and Owens 2008; Eisenacher et al. 2009; Schwarz and Stamminger 2009]. While this previous work has been successful in showing the feasibility of using higher-order surfaces in real-time graphics pipelines, we believe the advantages of Catmull-Clark surfaces over other higher-order surfaces (Section 2.1) motivate its use in future graphics pipelines as the higher-order surface of choice. In this work we describe our system for performing view-adaptive, crack-free tessellation of Catmull-Clark subdivision surfaces entirely on the GPU.

Like previous work, we use a breadth-first subdivision strategy to parallelize within a subdivision step. However, Catmull-Clark subdivision surfaces present multiple challenges over and above previous work. These challenges include maintaining a mesh-based data structure instead of a list of smaller primitives (patches), allowing the inherent dependence of each subdividing face on its 1-ring neighborhood, and avoiding cracks and T-junctions. In this paper, we address these problems.

Contribution Our main contribution in this paper is a simple yet robust framework to perform an all-quad view-adaptive tessellation for subdivision surfaces on a highly parallel graphics processor.

This is a hard problem due to the lack of a straightforward parallelization (as in the case of parametric patches), the absence of efficient GPU data structures for performing parallel surface subdivision, and the vulnerability to subdivision cracks. While previous techniques to subdivide surface meshes using GPU resources have either been highly specialized (fixed number of subdivision levels, limit surface tessellation for a few finite cases) or approximate, our approach is highly generic and completely flexible. We have carefully chosen a compact but intuitive data structure to maintain the subdivision mesh, and perform updates to this structure in a completely parallel fashion. We also present a data-parallel technique that uses templates given by Schneiders [1996] to completely avoid cracks by eliminating T-junctions in the subdivided mesh. We demonstrate the validity of our strategy by achieving real-time performance for several real-life models.

2 Background

2.1 Subdivision Surfaces

The theory behind subdivision surfaces was first described by Catmull and Clark [1978]. Subdivision surfaces are a popular class of primitives in modeling and animation of free-form surfaces. They provide a robust abstraction for smooth surfaces through the specification of a coarse piecewise linear mesh, which can be repeatedly subdivided to give an approximation of the smooth surface. Of the many different types of subdivision surface primitives, Catmull-Clark subdivision surfaces have become the standard modeling choice for many offline applications, including computer-generated motion pictures. Their use in character animation was first described by Pixar [DeRose et al. 1998].

In many ways, Catmull-Clark surfaces are superior to other forms of surface representations used in 3D graphics:

- Catmull-Clark subdivision rules can be applied to two-manifold meshes of arbitrary topology, without any continuity limitations. In contrast, parametric surfaces like NURBS and Bézier tensor product patches usually have restrictions on surface topology, and multiple patches need to be carefully connected to ensure piecewise continuity [DeRose et al. 1998].

- Catmull-Clark surfaces remain smooth under animation of the base mesh. Thus, vertices of the base mesh can be arbitrarily modified without producing artifacts in the final rendered surface. This is not generally true for parametric surfaces.

- Scalar fields like texture coordinates can be subdivided using the same rules as the mesh vertices. Thus, unlike polygonal meshes, static storage of texture coordinates is only needed for the base mesh.

The above advantages, in addition to the compactness of a coarse mesh representation, make subdivision surfaces attractive candidates for both off-line and interactive 3D graphics. Off-line graphics are typically implemented on a scalar architecture without real-time performance requirements, so in off-line applications, the superior features of subdivision surfaces motivate their widespread use, and performance issues relevant to real-time rendering are less important. For real-time applications, however, the difficulty of fast and parallel subdivision algorithms has thus far precluded their implementation on the GPU.

The iterative process described previously is called the refinement scheme. For a subdivision surface, this scheme defines how the subdivision takes place. Repeated application of the refinement scheme produces a sequence of finer meshes, which quickly approximate the limit surface. In practice, a few subdivision steps are usually

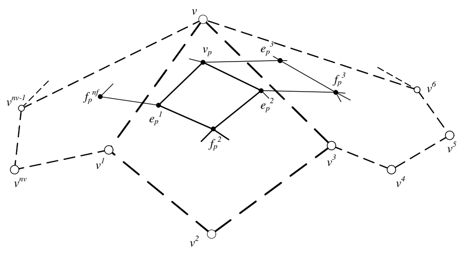


Figure 2: *Catmull-Clark refinement scheme. The refinement process generates a face point (f_p) for each face, an edge point (e_p) for each edge, and a vertex point (v_p) for each vertex. This new set of vertices is then appropriately connected to form the next level subdivision mesh. This figure shows face-points, edge-points, and vertex-points generated around a parent vertex v. For further details on the subdivision rules, please refer to Section 3.*

sufficient to provide a dense mesh for rendering purposes. The refinement scheme for Catmull-Clark surfaces is shown in Figure 2, and described in Section 3.

One possibility for avoiding the complexity of real-time subdivision is to compute the result of subdivision off-line and simply send it to the renderer (GPU) as needed. This approach does not fully exploit the power of subdivision surfaces, and in fact creates several issues. A subdivided mesh typically requires a large amount of static storage, which can be quite expensive. Moreover, it places significant pressure on the bus bandwidth to and from the GPU during rendering, which can be prohibitive for performance-sensitive applications. Off-line subdivision is also not useful for dynamic/animated surfaces, because every vertex of the dense mesh may change from frame to frame. Finally, statically subdivided meshes are not view-dependent. Thus, subdivided surfaces with an arbitrary viewpoint might display faceting artifacts or an unnecessarily detailed subdivision. Due to the these problems, static subdivision is not an effective technique for using subdivision surfaces in real-time applications. Dynamic GPU-based subdivision avoids these problems, and is thus better suited for real-time applications. Our review of prior work in this field (below) focuses on this topic.

2.2 Related Work

In recent years, several efforts have been made to dynamically subdivide Catmull-Clark meshes for rendering on a GPU platform [Shiue et al. 2005; Bunnell 2005; Bolz and Schröder 2002]. These implementations perform view-dependent subdivision on every frame of rendering, and thus do not suffer from the many of the issues described in Section 2.1. The earliest work from Bolz and Schröder [2002] first performs subdivision on the CPU, and then renders the generated polygons using a GPU. This approach suffers from both the cost of transferring the geometry to the GPU as well as a lack of viewpoint awareness.

Our work is more similar to that of Shiue et al. [2005] and Bunnell [2005], who both use GPU shaders to perform dynamic subdivision on the GPU. However, apart from the improved performance, there are two main differences from our work. First, while both these implementations use multiple textures to store the base mesh as 'patches,' we perform subdivision without modifying the original mesh structure. This makes animation a much easier task (Figure 3), since there is no need to modify a vertex at multiple locations. In general, the resulting data structures also become much easier to manage. Secondly, unlike the prior work, we perform subdivision without using precomputed lookup tables, so we have no limitations on either the number of subdivision levels, or the valence of input faces. Our implementation is thus more general in comparison.

There has also been prior work in high performance adaptive subdivision of Catmull-Clark surfaces for ray tracing [Benthin et al. 2007]. Although the approach in this paper also performs view-adaptive subdivision, it appears that the granularity of subdivision is coarse (constant for a face) and hence expected to generate a comparatively denser target mesh. Moreover, since the authors' subdivision technique is tuned for ray tracing, it is unclear how the design trade-offs will translate to a rasterizer-based rendering environment.

The closest work to our own performs software tessellation using GPU computing techniques [Patney and Owens 2008; Eisenacher et al. 2009; Schwarz and Stamminger 2009]. Research in this area has gained significant popularity due to its impressive performance for real-time applications, as well as the flexibility of a completely programmable implementation. The central idea is the reformulation of tessellation as a breadth-first task, which provides ample parallelism for acceleration using a GPU architecture. However, prior to our work, all work in this domain is restricted to parametric surfaces, which have several disadvantages compared to subdivision surfaces (Section 2.1). There is no obvious extension to subdivision surfaces, except through the approximation by Loop and Schaefer [2008]. Their approach regularizes the workload, which lends to robust rendering performance for real-time applications. However, the resulting tessellated surface is approximate and in general does not possess the same continuity as the limit surface. This is overcome by using a different approximation for tangent patches, which suppresses visual artifacts due to lack of proper continuity. We do not make these approximations, so surfaces tessellated using our technique follow properties of the limit surface.

Loop and Schaefer's approximation also forms the basis for hardware-supported Catmull-Clark subdivision in the upcoming Direct3D 11 pipeline [Microsoft Corporation 2008]. This API adds three new stages to the rendering pipeline, which allow for programmable tessellation of parametric patches, and through this approximation, Catmull-Clark surfaces as well. The tension between hardware acceleration's superior performance and efficiency vs. software's superior flexibility is broader than this paper and by no means settled, but we look forward to carefully comparing our implementation against DirectX 11 implementations when they are available.

3 Breadth-First Subdivision

In this section, we describe the core of our technique for performing real-time Catmull-Clark subdivision on a modern GPU. The original formulation of the subdivision process, as described below, is not a natural fit for a massively-parallel GPU platform. In the following subsections, we discuss modifications to this procedure that allow an efficient GPU implementation.

3.1 Catmull-Clark Refinement Scheme

Catmull-Clark subdivision begins with an arbitrary polyhedron M^0, called the control mesh. The faces of this mesh are subdivided into a collection of quadrilateral subfaces to produce the mesh M^1. The procedure is repeated for M^1 to produce the mesh M^2, then M^3, and so on. By following Catmull-Clark refinement rules, it can be shown that the mesh resulting from this recursive process successively approximates the smooth limit surface.

At each step, a face with k edges is subdivided into k quadrilaterals, which results in three classes of new vertices. A *face point* f_p is positioned at the centroid of the vertices corresponding to a face. An *edge point* e_p for every edge is defined as the centroid of four vertices: the two vertices that connect to make up the edge, and the face points for the two adjoining faces. Finally, a *vertex point* v_p, for a vertex connected to n edges (valence n), is defined using the following relation

$$v_p = \left(\frac{n-2}{n}\right) v + \frac{1}{n^2} \sum_{j}^{n-1} v^j + \frac{1}{n^2} \sum_{j}^{n-1} f_p^j, \qquad (1)$$

where v is the original vertex, and v^j is the j^{th} neighboring vertex. f_p^j is the face point of the j^{th} adjoining face.

An example of a single step of Catmull-Clark subdivision is shown in Figure 2. Note that each face in the newly created mesh is a quadrilateral, and after the first subdivision step, no subsequent step can produce extraordinary vertices (i.e. those with valence $! = n$).

3.2 Overview of Approach

Our goal is to efficiently accelerate the above subdivision procedure using the highly parallel resources available on a GPU. Our algorithm must be able to take advantage of the large number of SIMD cores capable of general-purpose execution. Conceptually, a single step of subdivision consists of three tasks: generation of face points for all faces, edge points for all edges, and vertex points for all vertices.

For any realistic input, each of these steps presents a significantly parallel execution workload. However, this is not sufficient for obtaining high performance, especially on the target architecture. Traditionally, polygonal meshes have been represented using a variety of data structures, and at the onset, it is unclear which of these is most suited to performing parallel subdivision on the GPU. Apart from evaluating face points, edge points and vertex points, mesh structure must also be kept updated. Performing such updates in parallel can be tricky. Maintaining a reasonable size for the static data structure can also be equally challenging. Clearly, the choice of data structure is critical in order to to achieve high subdivision performance.

To maintain visual fidelity without rendering too many primitives, subdivision should be view-dependent. In our approach, we only subdivide faces that are inside the view frustum. Also, the depth of subdivision for each face is dependent on its contribution to the screen pixels.

Finally, since adjacent faces of the mesh may be subdivided by separate execution units to potentially different subdivision depths, we must ensure that the final mesh is free of cracks. To achieve this, we use an efficient data-parallel technique for removing T-junctions in an all-quad mesh. This technique is discussed in detail in Section 3.3.

Given the above constraints, we perform breadth-first subdivision broadly as a sequence of the following steps, exploiting parallelism within each step of subdivision across faces of the mesh. Note that we partition the execution roughly based on the tasks outlined above.

For each step of subdivision, (1) we apply the subdivision criterion to each mesh face in parallel. (2) If all faces satisfy the criterion, we display the current mesh and exit. (3) Otherwise, we generate face points for faces that need subdivision, and update the mesh structure as necessary. (4) Then we generate edge points for relevant edges, and update the mesh. (5) Finally, for all vertices that require updates, we evaluate vertex points and move old vertices to new ones.

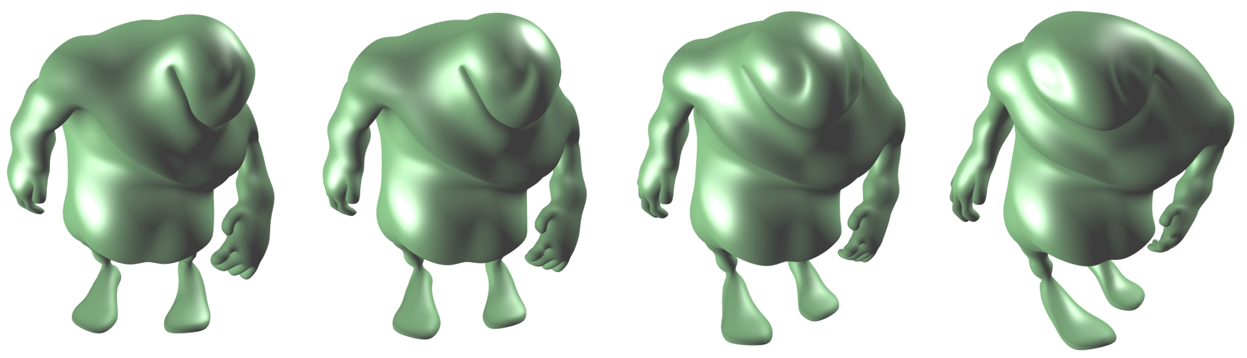
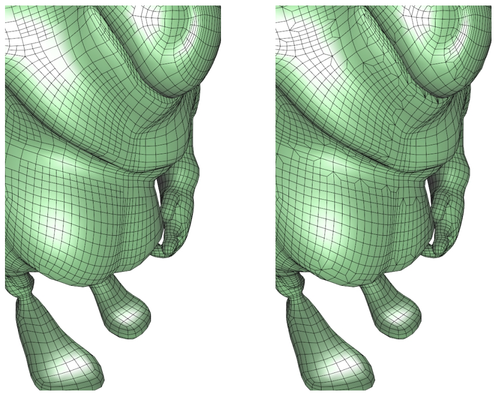


Figure 3: *Four frames of an animated subdivision surface. Note that the rendered surface remains smooth even though only vertices of the base mesh are modified. Since subdivision is performed every frame, there is no animation overhead.*



(a) Naive subdivision (b) T-junction-free subdivision

Figure 4: *Comparison of subdivision using a naive approach and our technique. Our algorithm is able to robustly remove all hanging nodes from an adaptively subdivided mesh. (For illustration purposes, the subdivision criterion has been relaxed.)*

This procedure continues until the subdivision criterion is satisfied for all faces, after which the mesh can be rendered. This is a simple extension to the conceptual formulation of Catmull-Clark subdivision that exploits the parallelism offered by the three original tasks. We discuss the details of our crack-avoidance algorithm, then our chosen data structure, and finally, our runtime algorithm that performs view-dependent subdivision.

3.3 Avoiding Cracks and T-Junctions

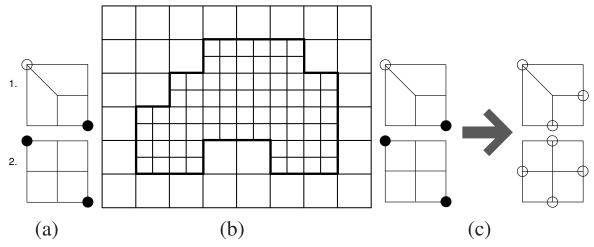
Adaptive refinement of quadrilateral meshes is a challenging problem, because straightforward algorithms result in faces with T-junctions, or hanging nodes. A hanging node is a vertex that lies on an edge without being one of its endpoints, and is usually formed at a subdivision transition. Hanging nodes can lead to cracks and pinholes on the output mesh, and are also undesirable from the perspective of mesh quality—a mesh with hanging nodes is not well suited for further processing. Schneiders [1996] proposed two different approaches based on refinement templates to solve this problem. His 3-refinement templates, which in some cases subdivide a face into 9 sub-faces, tend to generate a large number of small faces. However, his 2-refinement templates (both shown in Figure 5(a)) can remove hanging nodes from an all-quad mesh by only subdividing a face into either four (4-subdivision) or three (3-subdivision) faces. Unfortunately, he did not describe a concrete algorithm to efficiently implement such refinement, and we were also unable to find one in our literature survey. Thus, robustly performing 2-refinement is a difficult (and we believe previously unsolved) problem. In our work, we have used his ideas to develop a completely parallel crack-free refinement scheme.

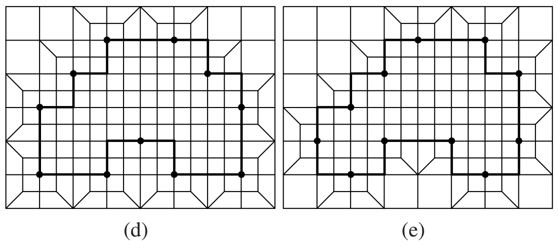
It is easy to show that a quadrilateral mesh always has an even number of vertices along its boundary. Thus, the number of vertices along a sequence of transition edges, i.e. those between two levels of subdivision, will also be even (see Figure 5(b)). We utilize this fact in eliminating hanging nodes from a mesh with adjacent faces subdivided to different levels. Since we perform elimination after every step of subdivision, the only hanging nodes that we encounter are those where the level of subdivision differs by 1.

Schneiders's approach works along this transition edge sequence, by tagging every alternate transition vertex as active. In Figures 5(d) and 5(e), active vertices are represented as solid circles. A refinement template operates on a face by subdividing it based on which of its vertices are active. By simply looking at the number of active vertices adjacent to a face, one can choose either one of the 2-refinement templates from Figure 5(a). As a result of applying these templates, all T-junctions present in the subdivided mesh will be removed. Examples of the resulting watertight mesh can be seen in Figures 5(d) and 5(e). While assigning templates to faces is trivially parallel, computation of active vertices is not.

We begin by showing how to locate active vertices on a transition edge using a propagation-based approach. Note that for every edge in the mesh, at most one of its vertices can ever be active. We start by randomly choosing a transition vertex and making it active. Starting from this vertex, we then move along transition edges, activating alternate vertices such that no two neighboring vertices are active. Examples of active vertices can be seen as solid circles in in Figures 5(d) and 5(e). Once this procedure terminates (i.e. each transition vertex is known to be either active or not), we can choose templates for transition faces in parallel. However, this propagation-based approach is serial, so it is best suited for offline computation.

We now demonstrate an efficient approach to assign active vertices along a transition edge sequence. Our contribution is demonstrating algorithms for computing an initial set of potentially active vertices for the mesh and incrementally updating this vertex set dynamically on each subdivision step. When the base mesh is initially generated, we use the same propagation-based approach as above to tag alternate vertices along all edges of the mesh as potentially active. Figure 5(f) shows an example of such a static set. Note that no two potentially active vertices share an edge. Also, any sequence of connected vertices—and thus any transition sequence—will have





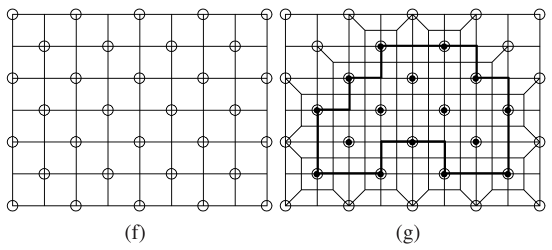


Figure 5: *Eliminating hanging nodes during adaptive subdivision. (a): The 2-refinement templates (3-subdivision and 4-subdivision) by Schneiders [1996]. (b): Mesh showing subdivision transitions. (c): Updates to the potentially-active set depend on the template used. (d), (e): Hanging nodes may be eliminated to give two different meshes, based on the selection of the active vertices (solid circles). Note every alternate vertex on the transition edge-sequence is active. (f): Our parallel technique maintains a set of potentially active vertices. (g): When these vertices intersect a transition, they are turned active. The templates can then be applied.*

alternating potentially active vertices.

We then incrementally update the set of potentially active vertices with each subdivision step, in parallel and at interactive rates. Beginning with the initial set of potentially active nodes from the base mesh that we computed offline, for each subdivision step, we must compute a new and accurate set of potentially active vertices on the current mesh, and then select subdivision templates for transition faces.

To update the set of potentially active vertices, we first remove each vertex that was active during subdivision from the active set. Then, we add all new edge points to the set. This rule translates to the individual templates as the update shown in Figure 5(c). This is also easily parallelized, and can be shown to maintain the properties of potentially active vertices.

To select subdivision templates for transition faces, we activate each potentially active vertex that lies on a transition. Next, each face counts the number of its active vertices, independently and in parallel. This number can only be 0, 1, or 2. No template is applied to a face with 0 active vertices. For 1 active vertex, the first template (3-subdivision) is used, and if 2 vertices are active, the second template (4-subdivision) is used. In Figure 5(g), we can see an example application of this procedure.

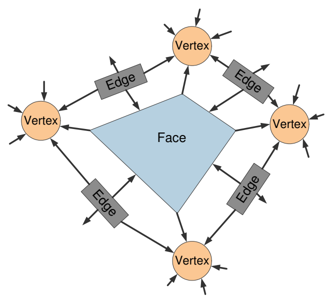


Figure 6: *General structure of the topological information stored in our mesh structure. Note that the number of arrows exiting an element is always fixed, yielding fixed-size data structures.*

ARRAY	ELEMENT	SIZE
Mesh Structure		
Vertex{In,Out}Buffer	Position: $\{v\}$	$2 \times n_v$
	Normal: $\{n\}$	
	Valence: $\{v_c\}$	
Index{In,Out}Buffer	Face: $\{v_0, v_1, v_2, v_3\}$	$2 \times n_f$
Edge{In,Out}Buffer	Edge: $\{v_0, v_1, f_0, f_1\}$	$2 \times n_e$
Temporary Data		
SubdivFlags	$\{a_p, a, v_s, e_s, f_s^3, f_s^4\}$	$\max(n_f, n_e, n_v)$
Face3Scanned	$\sum_0^i \{f_s^3\}$	n_f
Face4Scanned	$\sum_0^i \{f_s^4\}$	n_f
EdgeScanned	$\sum_0^i \{e_s\}$	n_e
Base Mesh		
VBase	Position: $\{v\}$	n_v^{init}
	Normal: $\{n\}$	
	Valence: $\{v_c\}$	
FBase	Face: $\{v_0, v_1, v_2, v_3\}$	n_f^{init}
EBase	Edge: $\{v_0, v_1, f_0, f_1\}$	n_e^{init}
SDBase	Active flag: $\{a_p\}$	n_v^{init}

Table 1: *Data structures used for managing parallel subdivision of Catmull-Clark meshes. The dynamic mesh structure is used to perform multiple steps of subdivision every frame. Temporary data structures assist these operations by providing indices for data updates. The base mesh data structure statically holds the base mesh, and is usually small in size. Please see Section 3.4 for details.*

The result is a mesh with no hanging nodes or T-junctions. Measured over a variety of models and varying subdivision conditions, our procedure adds an average of 4.46% total faces to a subdivided mesh without this procedure. Figure 4 shows before and after mesh visualizations.

3.4 Data Structure

Our goals in choosing a data structure for subdivision are twofold. First, we must be able to update the data structure in parallel during subdivision with high performance. Second, the chosen data structure must not occupy a large amount of static storage.

To meet these goals of performance and convenience, we first note that at the end of every frame, the resulting mesh needs to be rendered. It is thus sensible to maintain a vertex buffer (VB) and an index buffer (IB) as a part of the data structure, which enables rasterization with a single GPU call. A subdivided mesh can often be huge in size, so a runtime translation to these structures is undesirable.

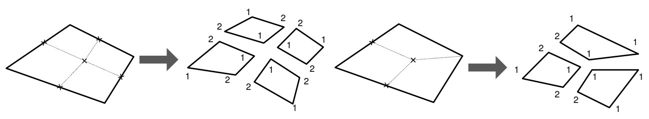


Figure 7: *Data structure updates are performed by two main tasks of our runtime algorithm: face point update (1), and edge point update (2). These updates are performed in parts by the two kernels, based on the available information and access indices. See Section 3.5 for details.*

We maintain a *VertexBuffer* of vertices that allows us to perform updates at the level of parallelism of a vertex. Similarly, our *IndexBuffer* enables updates at the level of parallelism of a face. Thus, we can evaluate vertex points as well as face points in parallel. We maintain two copies of each buffer (*VertexInBuffer*, *VertexOutBuffer*, *EdgeInBuffer*, and *EdgeOutBuffer*), in order to stream data from one to the other while performing subdivision. After every step, the pointers to the two buffers are swapped, so the next step streams data in the opposite direction.

To be able to update edges in parallel, we also maintain *Edge*{*In,Out*}*Buffer*s, which contain a sequential list of edges. An edge can be defined either by its two end vertices, or by the two faces that connect at the edge. Since evaluation of an edge point requires connecting vertices as well as centroids of adjacent faces (see Figure 2), we chose to keep both as a part of our *EdgeBuffer*. Thus, each element of the *EdgeBuffer* contains four indices: two for each vertex, and two for adjacent faces. A visual depiction of our mesh representation can be seen in Figure 6.

Unlike many mesh representations, the elements of each of the above buffers have fixed size. These buffers are dynamic and are updated on every subdivision step. Also, due to the nature of our adaptive subdivision technique, the *IndexBuffer* always stores a collection of quads. This adds to the convenience of the rendering operation.

The data structure described above is also known as a Render Dynamic Mesh [Wikipedia 2009]. A similar data structure has also been used in the past for subdivision in a serial context [Tobler and Maierhofer 2006]. Details of a Render Dynamic Mesh as used in our application are shown in Table 1.

View adaptivity implies that the number of child faces for any parent face will depend on the current viewpoint and the subdivision criterion. Thus, to ensure contiguity, updates made to the above data structure require a global pass to evaluate target indices for child faces and edges. We achieve this by using storage-conserving variations of the GPU *scan* primitive [Sengupta et al. 2007]. This adds three temporary arrays to our data structure, also shown in Table 1. The elements of the array *SubdivFlags* store the following bit-flags: a_p, a (whether vertex is potentially active, active, or both), v_s (whether vertex needs update), e_s (whether edge needs subdivision), f_s^3/f_s^4 (whether face needs 3-subdivision or 4-subdivision), and v_s (whether vertex needs update). The outputs of *scan* operations for e_s, f_s^3 and f_s^4 are placed in the *EdgeScanned*, *Face3Scanned* and *Face4Scanned* arrays, respectively. Flags that are not scanned are used during the subdivision process, and for removing T-junctions.

In order to reset the subdivision procedure at the beginning of every frame, we also retain the base mesh and the set of active vertices for it. The size of these static arrays is usually negligible in comparison to the other arrays used in subdivision.

3.5 Runtime Algorithm

Listing 1 Parallel Catmull-Clark Subdivision (Precomputation)

```
procedure Load mesh
  1: Load the subdivision mesh from the disk
  2: Compute a set of potentially active vertices by repeatedly
     choosing a new vertex and walking along edges to mark alternate vertices.
```

We repeatedly perform view-adaptive subdivision on the base mesh until the subdivision criterion is satisfied. Each step of subdivision begins with an evaluation of this criterion over the partially subdivided mesh. Once edges that need subdivision have been identified, we update the faces and edges by generating face points and edge points in parallel. Finally, we update vertices to their new positions by evaluating vertex points. As shown in Figure 7, connectivity updates for child faces and edges are performed as a part of the above three tasks, without affecting parallelism. Pseudocode for the runtime procedure can be found in Listings 1–5. Their descriptions follow.

Subdivision Test Every step of the subdivision loop begins by clearing all subdivision flags (except a_p) from previous instances of the loop. The target vertex array, *VertexOutBuffer*, is also reset to the values of *VertexInBuffer*. This simplifies calculations of vertex points later in the execution.

Our subdivision test is based on screen-space geometric extent and view frustum visibility. (Though we do not incorporate backface culling, it would be straightforward to incorporate in the function TestSubdivision.) For each edge of the input face, we compute the subdivision metric in parallel. Currently, if any edge lies on-screen and is bigger than a pre-determined threshold, its potentially active endpoints are turned active. This flag is then used to tag nearby faces and edges for subdivision. Every face that touches two active vertices is tagged for 4-subdivision, and every face that touches one active vertex is tagged for 3-subdivision around that vertex (see Figure 5(a)). Every edge touching an active vertex is also tagged for subdivision. Once all flags have been propagated, we perform a parallel prefix-sum (scan) operation over flags for faces as well as edges [Sengupta et al. 2007]. Using the prefix-sums, it is possible for each face/edge to individually deduce the indices of its children. This reduces the need for global communication when these structures are being updated.

GenFacePoints Generating face points is a straightforward parallel operation over all faces in the *FaceInBuffer*. Each face evaluates the centroid of its four corners, and updates it in the *VertexOutBuffer*. To maintain connectivity in the target subdivided mesh and simplify later tasks, we also perform some additional operations here. First, we partially update the corner indices for child faces. Since each child will usually have the face point and one parent corner as two of its corners, this update can be easily performed. Second, each face also adds its face point coordinates to the vertices corresponding to its corners in *VertexOutBuffer*. This update has a potential race condition, because multiple faces could be simultaneously competing to perform an addition to a memory location. We use atomic operations to achieve deterministic behavior in this update. Since floating-point atomic operations are not natively supported, we observe a slight performance penalty. For further discussion of this issue, please refer to Section 3.6.

GenEdgePoints Analogous to GenFacePoints, the kernel GenEdgePoints primarily generates edge points. By taking the centroid of its neighboring face points and vertices, an edge can perform this evaluation in a simple and regular fashion. However,

Listing 2 Parallel Catmull-Clark Subdivision (Subdivision Test)

```
procedure Initialize data structures
1: VertexOutBuffer ⇐ VertexInBuffer {copy}
2: Clear bits of SubdivFlags (except $a_p$)
3: need_subdiv ⇐ false

function TestSubdivision
1: for all faces in FaceInBuffer do {in parallel}
2:     Transform end points $v_0 - v_3$ to screen space
3:     if all four edges lie outside the view frustum then
4:         return
5:     end if
6:     Calculate the screen space extent of each edge, $|v_i v_{i+1}|$.
7:     if $|v_i v_{i+1}| \geq$ subdivision criterion then
8:         Set $a = a_p$ for all four corners. (turn potentially active
           vertices to active)
9:         Set $v_s = 1$ for all four corners.
10:        need_subdiv ⇐ true
11:    end if
12: end for

If need_subdiv is false, stop.

procedure UpdateFaces
1: for all faces in FaceInBuffer do {in parallel}
2:     Count the number of active corners for the face
3:     if 2 corners are active then
4:         Set $f_s^4$ for face to 1 and update SubdivFlags
5:     else if 1 corner is active then
6:         Set $f_s^3$ for face to 1 and update SubdivFlags
7:     end if
8: end for

procedure UpdateEdges
1: for all edges in EdgeInBuffer do {in parallel}
2:     if either of the two endpoints is active then
3:         Set $e_s$ for edge to 1 and update SubdivFlags
4:     else
5:         Add 2 or 4 to $e_s$ if any (or both) adjacent faces are tagged
           for 3-subdivision.
6:     end if
7: end for

scan SubdivFlags by masking $f_s^3$ values, output to Face3Scanned.
scan SubdivFlags by masking $f_s^4$ values, output to Face4Scanned.
scan SubdivFlags by masking $e_s$ values, output to EdgeScanned.
```

as before, this kernel also performs extra tasks to ensure connectivity and maintain efficiency in subsequent subdivision operations. Remember that partial connectivity updates were performed as a part of GenFacePoints. Since new edge points have been generated in this phase, we can now provide appropriate indices to complete those updates. Each edge that was tagged for subdivision also generates and updates its child edges, by computing their end points and adjacent faces. Finally, each edge also adds *twice* the coordinates of its midpoint to *VertexOutArray*, giving rise to a similar race condition as in *GenFacePoints*. Again, we use atomic operations to obtain deterministic results, with a minor performance hit (see Section 3.6). Note that our technique ensures that both neighbors of any subdividing edge will necessarily subdivide into either 3 or 4 children. This way we are guaranteed that no T-junctions will arise.

GenVertexPoints As a final subdivision step, we need to update the positions of old vertices to provide a closer approximation of the limit surface. To do this, we generate vertex points as detailed in Section 3.1. Remember that all adjacent face points as well as edge midpoints have already been aggregated in *VertexOutBuffer*. We use the following equation to evaluate the updated position for a vertex:

Listing 3 Parallel Catmull-Clark Subdivision (Face points)

```
procedure GenFacePoints
1: for all faces in FaceInBuffer do {in parallel}
2:     if face is tagged for subdivision then
3:         Calculate the face point $f_p$ as the centroid of the four end-
           points. Add to VertexOutBuffer.
4:         For each corner point, atomically add $f_p$ to the coordi-
           nates already in VertexOutBuffer
5:         if 4-subdivision is needed then
6:             Create four child faces in FaceOutBuffer
7:             for all Child faces do
8:                 Set first corner to be the corresponding parent corner
9:                 Set third corner to be the face point
10:                Set the other two corners to -1
11:            end for
12:        else if 3-subdivision is needed then
13:            Create three child faces in FaceOutBuffer
14:            for all Child faces do
15:                Set first corner to be the corresponding parent corner
16:                Set third corner to be the face point
17:                Set one corner to -1 (edge point)
18:                Set the last corner to be the remaining parent corner
19:            end for
20:            Set $v_s = 0$ for all corners.
21:        end if
22:    else
23:        Copy face to FaceOutBuffer while translating indices
24:    end if
25: end for
```

$$v_{out} \Leftarrow \frac{v_{in} \times (v_c - 3) + \frac{(v_{out} - v_{in})}{v_c}}{v_c} \tag{2}$$

Here v_{out} is obtained from *VertexOutBuffer*, and v_c is the valence of the input vertex. It is easy to see that equation (2) is equivalent to (1), since v_{out} initially contains the sum of the original vertex, all face points as well as twice the midpoints of all edges. Moreover, as before, this step is easily parallelized. Only vertices of faces tagged for subdivision and not lying on a subdivision transition are updated by this method.

The above sequence of subdivision steps is repeated till the subdivision criterion is met, i.e. while *need_subdiv* is true. For most models, a few levels of subdivision are sufficient for this purpose. In our tests, we have observed subdivision depths of up to 8.

3.6 Atomic Accesses

An important part of the above subdivision steps is the use of atomic updates to accumulate scattered coordinates, primarily for the computation of vertex points. We believe this is a reasonable cost for two reasons: first, atomics allow maintaining a highly regular mesh data structure, with completely parallel updates, and second, since most vertices tagged for subdivision have a small valence (≤ 5), the amount of contention is negligible. However, atomic memory accesses on current generation GPUs are relatively slow. Moreover, floating-point atomic operations are not natively supported, and thus had to be emulated (using an integer lock). Thus, although in theory there shouldn't be any impact due to atomic operations, in practice we do observe a performance hit of a few ms per frame.

Listing 4 Parallel Catmull-Clark Subdivision (Edge Points)

```
procedure GenEdgePoints
 1: for all edges in EdgeInBuffer do {in parallel}
 2:   if edge is tagged for subdivision then
 3:     Calculate the edge midpoint
 4:     For each end point, atomically add twice the midpoint coordinates to those already in VertexOutBuffer
 5:     Calculate the edge point $e_p$ as the centroid of the two endpoints and adjacent face points. If edge lies on boundary, set edge point to edge midpoint. Add to VertexOutBuffer.
 6:     Complete indices for child faces that were previously incomplete, depending on whether each face is tagged for 3-subdivision or 4-subdivision. Write to FaceOutBuffer.
 7:     Create four child edges in EdgeOutBuffer
 8:     for first and second child edges do
 9:       Set first end point to corresponding parent end point
10:       Set second end point to the edge point
11:       Set neighboring faces to the two child faces from different parent faces
12:     end for
13:     for third and fourth child edges do
14:       Set first end point to one of the neighboring face points
15:       Set second end point to the edge point
16:       Set neighboring faces to children of the same parent
17:     end for
18:   else
19:     Copy edge to EdgeOutBuffer while translating indices. Also consider adjacent faces that might have subdivided into three children.
20:   end if
21: end for
```

Listing 5 Parallel Catmull-Clark Subdivision (Vertex Points)

```
procedure GenVertexPoints
1: for all vertices in VertexInBuffer do {in parallel}
2:   if vertex $v_{in}$ is tagged for update then
3:     Retrieve the corresponding vertex $v_{out}$ from VertexOutBuffer
4:     Retrieve the valence $v_c$ from VertexInBuffer
5:     Set $v_{out} \Leftarrow \frac{v_{in} \times (v_c - 3) + (v_{out} - v_{in})/v_c}{v_c}$
6:     Write to VertexOutBuffer
7:   end if
8: end for
```

We believe that future GPU generations will offer better support for atomic operations, resulting in improved subdivision performance.

3.7 Implementation Details

We implemented our subdivision framework using NVIDIA CUDA 2.2 on a system with an Intel Core 2 CPU with 2 GB memory and an NVIDIA GeForce GTX 280 GPU. This GPU contains 32 programmable SIMD cores, each of which can work on 32-wide vectors. Subdivision is well-suited to a highly parallel processor such as a modern GPU—even moderately complex models have a large number of faces, edges and vertices. Moreover, during execution, this number tends to grow quickly with the level of subdivision.

4 Results

The GPU implementation of our approach demonstrates the merits of parallelization—we demonstrate real-time rendering performance for most of our test models, even under animation. This is in spite of the fact that our CUDA programs have not been aggressively optimized, and suffer from several incoherent long-latency memory transactions. Looking at our performance, we believe that our parallelization technique has contributed to maintaining high throughput for real-time tessellation.

	Mesh Size (faces)		Subdivision	Rendering
Model name	Input	Output	depth	time (ms)
Bigguy	1,450	91,992	5	37.12
Monsterfrog	1,292	80,452	5	34.17
Monsterfrog2	1,292	83,750	5	35.41
Killeroo	2,894	80,277	5	31.34
Three-block	18	60,018	7	30.06
Complex	1,350	495,076	6	176.67

Table 2: *Rendering performance for the models used in our experiments. The input meshes are adaptively subdivided to meet subdivision criterion every frame. These results were taken at a resolution of 800 × 800 with a screen-space subdivision threshold (maximum edge length) of 5 pixels.*

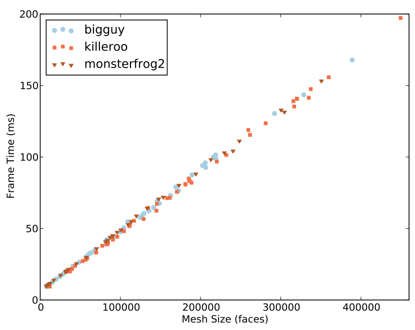
Rendering Performance Figures 1 and 10 show examples of rendered images using our approach. The subdivision performance for these cases is shown in Table 2. Figure 3 shows a subdivision surface rendered from an animated base mesh, and Figure 9 shows how meshes with sharp boundaries (following DeRose [1998]) and textures fit into our framework. In Figure 8(a), we have plotted the relationship between the number of faces in the subdivided mesh and the time it takes to perform subdivision and rendering. This relationship is unmistakably linear and is consistent across all models tested.

Performance Scalability Figures 8(b) and 8(c) graph the performance of subdivision (generated mesh size/subdivision time) with two varying parameters: the subdivision criterion and the screen size of the rendered image. Both have an impact on the complexity of the work load. From the two plots it can be seen that in general, subdivision throughput either improves or is maintained as the workload is increased. The subdivision performance of roughly 2.5–3 MFaces/s is independent of any specific model, and (as a rough measure) is about 2.5 times the performance of recursive subdivision as we previously reported on a less powerful GPU (GeForce 8800 GTX) [Patney and Owens 2008].

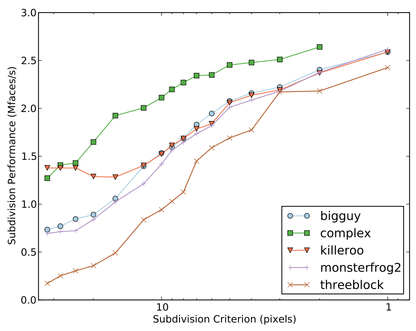
5 Discussion

From our results, our first observation is the consistently linear dependence between the size of the final mesh and the time taken to subdivide to it. This performance characteristic is similar to what is observed for parametric surfaces [Patney and Owens 2008], which is encouraging because it follows our original goal of extending breadth-first subdivision to subdivision surfaces. Also, as the complexity of the subdivision workload increases, performance improves towards its peak of approximately 2.5–3 MFaces/s (Figure 8(b) and 8(c)). This suggests that our parallel approach to subdivision scales well with hardware, and can be expected to perform better as GPU capabilities improve.

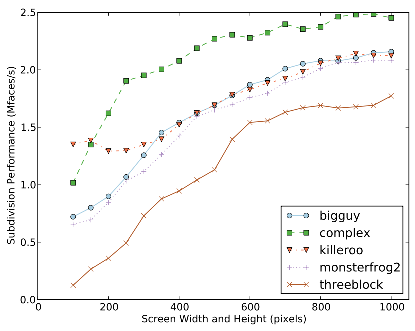
While performance achieved by our implementation could be optimized to more impressive numbers, this is not our primary contribution. We have provided a data structure and an associated algorithm to efficiently perform breadth-first subdivision in a completely parallel environment. It has several advantages over related implementations found in literature:



(a) Render times with varying final mesh size.



(b) Varying the subdivision criterion (threshold edge length).



(c) Varying the screen size of the surface rendering.

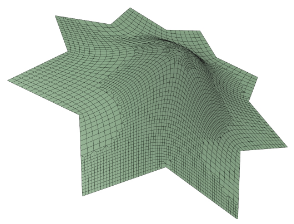
Figure 8: *Performance results of breadth-first subdivision. These graphs plot subdivision performance for varying parameters of the system. In (a), rendering times for various models are graphed as a function of the size of the final subdivided mesh, showing a near-linear relationship. Graphs (b) and (c) plot the rendering performance with varying subdivision criterion as well as with the screen size of the target mesh. As the complexity of the workload is increased, subdivision performance (throughput) generally improves and approaches a peak of about 2.5–3 MFaces/s.*

- In our approach, we do not need to maintain unreasonable amounts of extra storage for the mesh: only three main data structures are sufficient. Moreover, our data structure does not rely on variable-size neighbor lists, and hence has a constant element size. At no point during runtime do we need to loop through any list of unknown size.
- We always maintain the mesh as a set of quads, even after performing view-adaptive updates. This keeps the subdivision process uniform and permits drawing the entire mesh in a single draw call. An all-quad mesh is also well-suited to Reyes-like applications, where the subdivided mesh is evaluated further (diced) before rendering. Having non-quads in the mesh makes dicing a much harder problem.
- Using our technique, we can obtain completely watertight subdivision – there are no subdivision cracks or T-junctions. Moreover, elimination of cracks also happens completely in parallel, and is thus well-suited to data parallel systems.
- Our data structure allows generalized subdivision. Little preprocessing is needed, and there are no assumptions or approximations on the input. We expect it to be straightforward to extend our technique to other forms of refinement, such as Loop and Doo-Sabin.

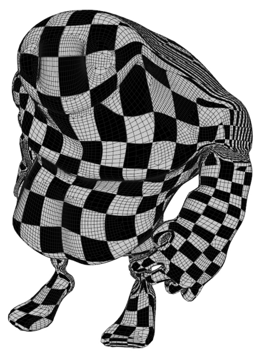
Limitations Inefficient memory accesses are our biggest performance limitation. Many accesses in our code are uncoalesced due to non-deterministic mesh layout, and thus hurt performance. Also, since our algorithm needs to perform atomic accesses for data structure updates, we incur a penalty in rendering performance. We believe that performing faster memory accesses is the biggest optimization challenge for our approach. To address this, we are investigating the feasibility of a software-managed geometry cache.

6 Conclusion

We have presented a simple and generic data structure and algorithm that helps to perform recursive subdivision of Catmull-Clark meshes entirely on a GPU. Our scheme uses fairly regular data structures and carefully manages updates to ensure maximal parallelism. Subdivision is divided into a sequence of four broad steps, which link directly to the conceptual definition of subdivision surfaces. We are able to use this approach to demonstrate real-time rendering performance for several complex models.



(a) A non-closed mesh with edges tagged as 'sharp'.

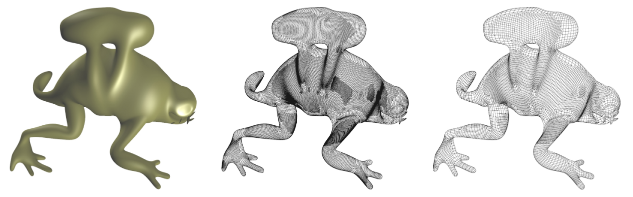


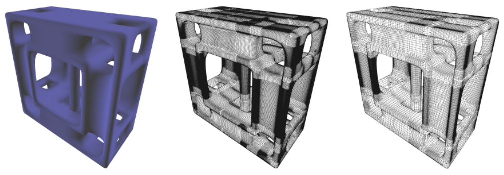
(b) Applying a texture to the big guy model.

Figure 9: *Our technique can effectively support non-closed meshes (using edge rules from DeRose [1998]) as well as textures. In the above figures, these features have been demonstrated. No performance issues were observed in these experiments.*

We have also generalized the idea of performing programmable surface tessellation using breadth-first subdivision to include Catmull-Clark meshes, thereby extending its scope to a richer domain of 3D modeling. We have also shown how the crack prevention operation can be effectively parallelized. Our work presents an alternative to fixed-function hardware tessellation; one potential outcome of our work could be the adoption of programmable tessellation as a common operation in real-time graphics.

Finally, we see our work as part of a larger effort to study efficient data structures for programmable graphics on a GPU. Research in this area is still in infancy, and we hope that our ideas like ours will encourage further development of techniques to efficiently handle the irregular execution and data management tasks often encountered in graphics, and map them to a massively parallel GPU architecture.


(a) Monster Frog


(b) Complex

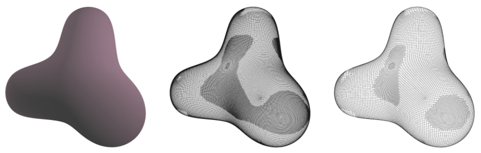

(c) Three-block

Figure 10: *More examples of our test models. Each rendered image is accompanied by a high-detail and a low-detail mesh representation. We control detail by varying the subdivision criterion.*

Acknowledgments We would like to thank Bay Raitt of Valve Software for the "Big Guy" and "Monster Frog" models. The "Killeroo" subdivision mesh is courtesy of Headus Inc. (`http://www.headus.com.au`). Thanks also to our funding agencies, the National Science Foundation (Award 0541448) and the SciDAC Institute for Ultrascale Visualization, and to NVIDIA for equipment donations.

Finally, thanks to Eric Lengyel and the anonymous reviewers for their comments and valuable suggestions that improved the work.

References

BENTHIN, C., BOULOS, S., LACEWELL, D., AND WALD, I. 2007. Packet-based ray tracing of Catmull-Clark subdivision surfaces. Tech. Rep. UUSCI-2007-011, SCI Institute, University of Utah.

BOLZ, J., AND SCHRÖDER, P. 2002. Rapid evaluation of Catmull-Clark subdivision surfaces. In *Web3D 2002: Proceedings of the Seventh International Conference on 3D Web Technology*, 11–17.

BUNNELL, M. 2005. Adaptive tessellation of subdivision surfaces with displacement mapping. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, Mar., ch. 7, 109–122.

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

DEROSE, T. D., KASS, M., AND TRUONG, T. 1998. Subdivision surfaces in character animation. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, 85–94.

EISENACHER, C., MEYER, Q., AND LOOP, C. 2009. Real-time view-dependent rendering of parametric surfaces. In *I3D '09: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, 137–143.

LOOP, C., AND SCHAEFER, S. 2008. Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Transactions on Graphics 27*, 1 (Mar.), 8:1–8:11.

MICROSOFT CORPORATION. 2008. Introduction to the Direct3D 11 graphics pipeline. `http://www.microsoft.com/downloads/details.aspx?familyid=E410716F-12BF-4E8F-AC41-97B4440C3B90`.

PATNEY, A., AND OWENS, J. D. 2008. Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics 27*, 5 (Dec.), 143:1–143:8.

SCHNEIDERS, R. 1996. Refining quadrilateral and hexahedral element meshes. In *Proceedings of the Fifth International Conference on Numerical Grid Generation in Computational Field Simulations*, 679–689.

SCHWARZ, M., AND STAMMINGER, M. 2009. Fast GPU-based adaptive tessellation with CUDA. *Computer Graphics Forum 28*, 2 (Mar.), 365–374.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for GPU computing. In *Graphics Hardware 2007*, 97–106.

SHIUE, L.-J., JONES, I., AND PETERS, J. 2005. A realtime GPU subdivision kernel. *ACM Transactions on Graphics 24*, 3 (Aug.), 1010–1015.

TOBLER, R. F., AND MAIERHOFER, S. 2006. A mesh data structure for rendering and subdivision. In *Proceedings of WSCG (International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision)*, 157–162.

WIKIPEDIA, 2009. Polygon mesh — Wikipedia, the free encyclopedia. [Online; accessed 28-April-2009].