Efficient GPU Rendering of Subdivision Surfaces using Adaptive Quadtrees

Wade Brainerd* Activision
Tim Foley* NVIDIA
Manuel Kraemer NVIDIA
Henry Moreton NVIDIA
Matthias Nießner Stanford University

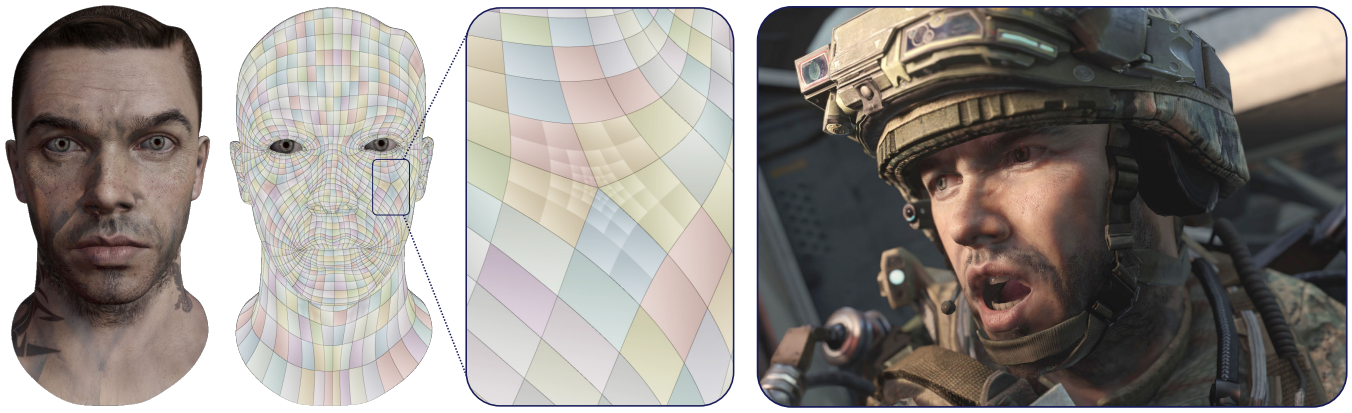


Figure 1: *In our method, a subdivision surface model (left) is rendered in a single pass, without a separate subdivision step. Each quad face is submitted as a single tessellated primitive; a per-face adaptive quadtree is used to map tessellated vertices to the appropriate subdivided face (middle). Our approach makes tessellated subdivision surfaces easy to integrate into modern video game rendering (right).* © *2014 Activision Publishing, Inc.*

Abstract

We present a novel method for real-time rendering of subdivision surfaces whose goal is to make subdivision faces as easy to render as triangles, points, or lines. Our approach uses standard GPU tessellation hardware and processes each face of a base mesh independently, thus allowing an entire model to be rendered in a single pass. The key idea of our method is to subdivide the u, v domain of each face ahead of time, generating a quadtree structure, and then submit one tessellated primitive per input face. By traversing the quadtree for each post-tessellation vertex, we are able to accurately and efficiently evaluate the limit surface. Our method yields a more uniform tessellation of the surface, and faster rendering, as fewer primitives are submitted. We evaluate our method on a variety of assets, and realize performance that can be three times faster than state-of-the-art approaches. In addition, our streaming formulation makes it easier to integrate subdivision surfaces into applications and shader code written for polygonal models. We illustrate integration of our technique into a full-featured video game engine.

Keywords: real-time rendering, subdivision surfaces

Concepts: •**Computing methodologies** → **Rendering; Rasterization;**

*Joint first authors.

SIGGRAPH '16 Technical Paper,, July 24 - 28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4279-7/16/07
DOI: http://dx.doi.org/10.1145/2897824.2925874

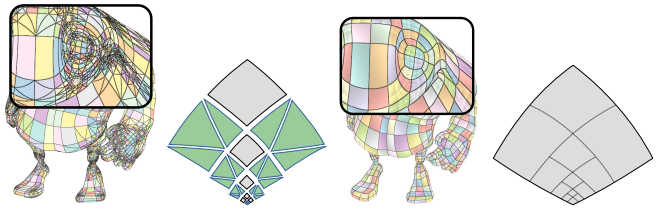
1 Introduction

Subdivision surfaces [Catmull and Clark 1978; Loop 1987; Doo and Sabin 1978] have been used in movie productions for many years. They have evolved into a *de facto* industry standard surface representation, due to the flexibility they provide in modeling. With an increasing demand for richer images with more and more visual detail, it is desirable to render such movie-quality assets in real time, enabling the use of subdivision surfaces in both content creation tools and interactive video games. Ideally, we would like subdivision surfaces to be supported as first-class rendering primitives on GPUs, akin to points, lines, and triangles.

Modern graphics hardware and APIs support tessellation of parametric surfaces, as part of the standard graphics pipeline [Microsoft Corporation 2009]. Tessellation hardware takes as input individual parametric patches of statically-known size, and uses a user-defined shader to directly evaluate their surface at arbitrary locations in each patch's domain. The evaluated vertices are connected into triangles, and processed by the subsequent graphics pipeline stages. An application can vary the sampling rate to achieve view-dependent level-of-detail. Because data expansion is done locally, memory I/O is kept low and high throughput can be achieved.

Unfortunately, hardware tessellation only supports directly-evaluable parametric surfaces, and direct evaluation of subdivision surfaces can be expensive. In the case of Catmull-Clark subdivision [1978], the limit surface for a regular face can be evaluated as a bi-cubic B-spline patch. However, the limit surface for an irregular face is defined by iterative application of the subdivision rules. Seminal work by Jos Stam [1998] has shown that the limit surface for an irregular quadrilateral face can also be directly evaluated by doing Eigenanalysis of the underlying subdivision matrix. However, Eigenanalysis-based evaluation involves many floating-point operations, making direct evaluation with Stam's method costly.

Given the complexity of direct evaluation, most fast rendering schemes use either approximation or adaptive subdivision. Approximating schemes [Loop and Schaefer 2008; Loop et al. 2009] render



(a) *FAS* **(b)** *Ours*

Figure 2: *In order to isolate an extraordinary vertex, FAS subdivides a face into multiple primitives, including transition patches (green) to stitch T-junctions. Our algorithm uses a single primitive per quad face, with a precomputed internal hierarchy.*

each face of the subdivision surface as a directly-evaluable patch that approximates the limit surface. This can yield high performance, but does not reflect the true Catmull-Clark surface. OpenSubdiv[1], a widely-used library for rendering subdivision surfaces, uses Feature-Adaptive Subdivision (FAS) [Nießner et al. 2012a]. Irregular faces, and those with special features, are subdivided to yield multiple patches that can be directly processed by tessellation hardware, while yielding an accurate surface.

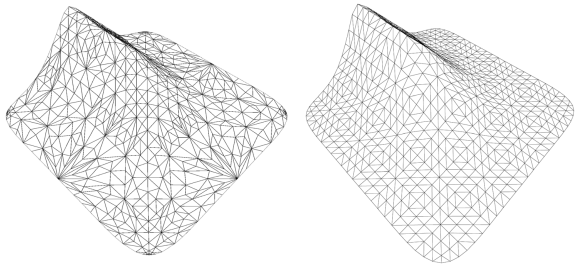
Compared to approximate schemes, FAS has several disadvantages. First, each irregular input face maps to multiple tessellator input primitives (Figure 2), the number of which correlates with the subdivision level at run time. This complicates the application logic used to assign tessellation rates and submit primitives, compared to approximation schemes that use a single primitive per input face. Second, where faces with different subdivision levels meet, T-junctions occur, which must be fixed with the introduction of *transition patches*. Third, the adaptive subdivision and patch tessellation steps are implemented as distinct compute and rendering passes, punctuated by global synchronization. Using FAS, subdivision surfaces are not integrated into the rasterization pipeline, and do not combine in an orthogonal fashion with typical vertex shader steps, such as skeletal animation. Finally, FAS, and transition patches in particular, can make the distribution of vertices on the tessellated limit surface less uniform (Figure 3), making it hard to avoid surface over- or under-sampling.

In this paper, we propose a novel end-to-end subdivision algorithm that allows us to incorporate subdivision surface tessellation into the graphics pipeline, just like other primitive types. The key insight of our method is to submit one tessellator primitive per input quadrilateral face, as in approximate schemes, but to pre-compute sufficient data to perform accurate surface evaluations equivalent to FAS. A per-face *subdivision plan*, generated ahead of time, uses a quadtree acceleration structure to record the adaptive subdivision hierarchy that arises from an input face. By traversing the quadtree, we can map the u, v location of a tessellation vertex to a directly-evaluable patch in the subdivision hierarchy. The subdivision plan also allows us to compute the subdivided control points required for given tessellation factors per-patch, within the existing graphics pipeline. Compared to the state of the art, our approach is both simpler and faster, and integrates with existing vertex and fragment shaders used for polygonal models.

Our main contribution is an end-to-end subdivision surface rendering pipeline, including the following key aspects:

- a novel method to render subdivision surface models (including boundaries, semi-sharp creases, and non-quadrilateral faces) in a single pass on GPU hardware
- significantly increased performance compared to state-of-the-art renderers; up to $3\times$ speedup for typical tessellation rates

[1] http://graphics.pixar.com/opensubdiv



(a) *FAS* **(b)** *Ours*

Figure 3: *Comparison of tessellation pattern quality between FAS and our approach. Our tessellation density is more uniform due to our one-to-one mapping between submitted and rendered faces.*

- extension of our method to aggregate multiple faces into a single tessellation primitive, to further accelerate rendering
- support for easily adding/removing faces and changing topology at runtime, since faces are processed independently
- proposed GPU features to further accelerate our approach

We also demonstrate integration of our approach into a production game engine.

2 Previous Work

Subdivision surfaces generalize parametric patches to arbitrary domains. Catmull-Clark surfaces [Catmull and Clark 1978], a generalization of bi-cubic B-splines, are the most prominent of subdivision schemes; however, many other schemes exist, such as Loop [Loop 1987], Doo-Sabin [Doo and Sabin 1978], and $\sqrt{3}$-subdivision [Kobbelt 2000]. Since the introduction of subdivision surfaces, several important extensions have been proposed, including the handling of boundaries [Nasri 1987], infinitely sharp creases [Hoppe et al. 1994], semi-sharp creases [DeRose et al. 1998], and hierarchically-defined detail [Forsey and Bartels 1988]. Subdivision surfaces have been the method of choice for movie production due to the flexibility they afford in modeling [Cook et al. 1987; Pixar Animation Studios 2005]. More recently, efforts have been made to render subdivision surfaces in real time on graphics hardware, for use in authoring tools and video games.

A naive way of rendering subdivision surfaces on GPUs is via global refinement; i.e., a GPU kernel iteratively applies the subdivision rules and generates a densely-refined mesh, which is then rendered in a second pass [Bunnell 2005; Shiue et al. 2005; Patney et al. 2009; Weber et al. 2015]. Unfortunately, global refinement requires significant memory bandwidth to stream mesh data between GPU multiprocessors and global memory (i.e., on- and off-chip).

Hardware tessellation can alleviate this bandwidth problem [Andrews and Baker 2006; Microsoft Corporation 2009], since tessellated geometry can be generated and consumed on-chip. Recent surveys [Schäfer et al. 2014; Nießner et al. 2015] provide a comprehensive overview of rendering techniques with the hardware tessellator. The challenge of rendering subdivision surfaces using the hardware tessellator is *patching* the base mesh: i.e., converting the faces of the base mesh into some number of directly-evaluable parametric patches. Jos Stam [1998] shows that direct evaluation of subdivision surfaces is possible; while originally presented on the CPU, a GPU implementation is straightforward. However, Stam's method requires many floating point computations (transformation to Eigenspace of the subdivision matrix) and is restricted to quad-only meshes with isolated extraordinary vertices.

Bolz and Schröder [2002] propose a direct evaluation scheme which pre-computes tabularized functions for particular topological con-

figurations of faces. Because of the large size of the required tables (linear in the number of vertices in the 1-ring, and quadratic in maximum sampling rate), they restrict their approach to a small range of topologies, and isolate extraordinary vertices using global subdivision. Our approach is similar in that we also accelerate surface evaluation by pre-computing a data structure for each face configuration. However, our quadtree data structure is more compact than Bolz and Schröder's tables, with size linear in maximum subdivision depth (logarithmic in maximum tessellation rate).

Approximate patching may be used to accelerate rendering. Loop and Schaefer [2008] propose an approximate, quad-only scheme using bi-cubic Bézier patches. Other quad-only patching schemes can be applied to rendering with hardware tessellation [Myles et al. 2008b; Ni et al. 2008]. Myles et al. [2008a] propose a method to support pentagonal patches as well as quads and triangles. The use of Gregory patches allows Loop et al. [2009] to achieve both high quality and performance for mixed quad-and-triangle meshes. Unfortunately, approximate patching can introduce distortions in the parameter domain; with additional effort these artifacts can be mitigated [He et al. 2012]. Kovacs et al. [2009; 2010] handle infinitely sharp creases in the context of an approximating scheme.

Approximate schemes are viable for some uses, but for others quality may be unsatisfactory, support for special features may be lacking, or problems may arises with displacement mapping [Nießner and Loop 2013]. Feature-adaptive subdivision (FAS) [Nießner et al. 2012a] avoids approximation and efficiently handles features such as semi-sharp creases [Nießner et al. 2012b]. It achieves similar performance to approximate schemes while producing accurate results, and is the basis for the widely-used OpenSubdiv library. FAS applies subdivision only in regions where direct evaluation is costly or infeasible, and processes regular faces as patches using the hardware tessellator. Dynamic feature-adaptive subdivision (DFAS) [Schäfer et al. 2015] extends FAS by allowing locally-adaptive subdivision depths within a single mesh, which can be important for large meshes. Our approach is similar to FAS and DFAS, but with two key differences. First, while we also compute control points for subdivided faces, we do not submit these faces to the tessellation hardware; instead, we submit a single primitive for each base face, and dynamically access the correct control points. Second, rather than process a mesh globally, in multiple passes, we process each base face independently, within the GPU rendering pipeline.

3 Overview

In this section, we outline the main steps of our approach; an overview of the algorithm is shown in Figure 4. We begin with a per-face preprocessing phase; once completed, we can render a subdivision surface in a single pass on current GPU hardware.

Input As input, we take a Catmull-Clark *base mesh*, which is defined by its topology, feature tags, and a set of *base vertices*. We support all common extensions of Catmull-Clark subdivision, including boundaries and both hard and semi-sharp crease tags on vertices and edges [Hoppe et al. 1994; DeRose et al. 1998]. We do not require that extraordinary vertices be isolated, and support meshes with arbitrary non-quad faces without additional preprocessing. We also do not require the topology of a mesh to remain fixed at runtime (including the tagging and sharpness of features). Dynamic updates are feasible on a per-patch level since each quad-face is processed independently; this is important for authoring tools, where the mesh topology may be modified frequently.

In this section, we only discuss quad faces, and defer discussion of how we handle non-quad faces to Section 7.2. In addition, we propose extensions that can aggregate multiple faces in Section 7.3.

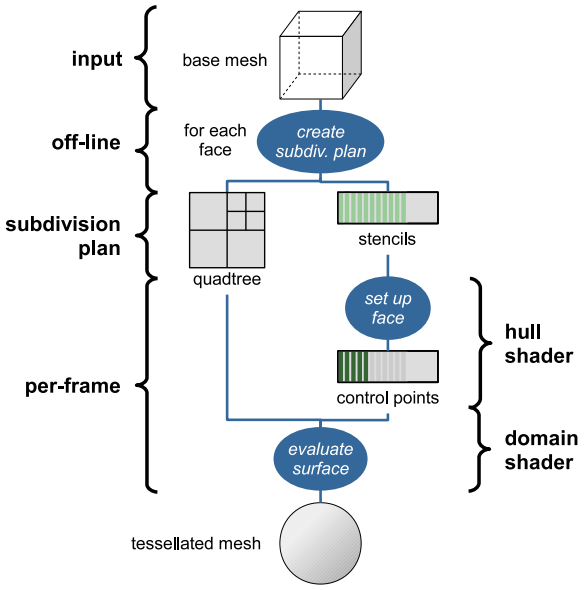


Figure 4: *Overview of our algorithm. We preprocess each face of a* base mesh *to yield a* subdivision plan*, comprising a quadtree of sub-faces, and stencils for the control points they require. At runtime, we set up the face by computing the subset of control points required for a given tessellation factor, and then use the quadtree to guide evaluation at each surface sample.*

Creating a Subdivision Plan When a base mesh is first loaded, we process each face (and its 1-ring neighborhood) to create (or reuse) a *subdivision plan*. The subdivision plan is a data structure that represents a feature-adaptive subdivision hierarchy for the face, down to some fixed maximum depth. Specifically, it comprises:

- a quadtree of the adaptive subdivision hierarchy of the face
- an ordered list of *stencils* for control points required by subdivided faces. Each stencil represents a control point as a weighted sum over base vertices in the 1-ring of the face.

Quadtree leaf nodes represent directly-evaluable sub-regions of the parameter domain, with control points in the list of stencils.

Plans can be shared by faces with equivalent topological configuration. We describe the creation, structure, and sharing of subdivision plans in more detail in Section 4.

Face Setup and Subdivision For rendering, we submit one primitive to the hardware tessellator for each quad face of the base mesh, irrespective of regularity, special features, or valence. This single primitive may represent a hierarchy of adaptively-subdivided faces, each with their own control points.

The setup and subdivision stage is responsible for computing the control points required at runtime; this stage maps well to the hardware *hull* shader. First, an application-specific metric (screen-space or other) is used to compute desired tessellation factors, which determine the level of adaptive subdivision to be applied. Then the control points required at the selected subdivision level are computed by applying the corresponding stencils.

We describe face setup and subdivision further in Section 5.

Surface Evaluation The hardware tessellator produces a set of *tessellation vertices*, each associated with a parametric location within the u, v domain of the face. Our surface evaluation step computes position and tangents of the limit surface at each of these locations. Surface evaluation maps to the hardware *domain* shader.

Evaluation traverses the quadtree in the subdivision plan to find the directly-evaluable sub-face in which the u, v location lands. In the common case, this is a regular sub-face, which is evaluated as a B-spline surface, using control points output by the subdivision stage.

We describe the surface evaluation step further in Section 6.

4 Creating a Subdivision Plan

Central to our algorithm is the subdivision plan data structure which we create for base mesh faces. Creating a subdivision plan for a face begins with adaptively subdividing the face, in a manner similar to FAS. Subdivision yields a hierarchy of *sub-faces*, encoded as a quadtree, and a list of stencils for control points.

Adaptive subdivision terminates at faces that can be directly evaluated (e.g., regular faces), or upon reaching a predefined limit on subdivision depth. This limit is a function of the maximum tessellation factor. Current GPUs support factors up to 64 (six levels).

4.1 Constructing an Adaptive Subdivision Quadtree

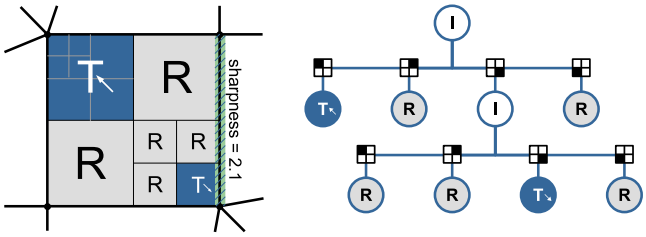


Figure 5: *Quadtree for a face with two extraordinary vertices, and one semi-sharp edge. The quadtree comprises internal (I), regular (R), and terminal nodes (T). Our terminal nodes do not support direct semi-sharp crease evaluation, so the bottom-right quadrant is recursively subdivided until the sharp feature is eliminated.*

Our quadtree (Figure 5) directly reflects an adaptive subdivision of the face. Internal nodes correspond to recursive subdivision steps, and leaf nodes correspond to sub-domains that can be efficiently evaluated. We briefly enumerate the different kinds of leaf nodes.

Regular and Boundary Nodes If recursive subdivision yields a regular face, then stencils for each of its 16 control points are added to the stencil list, and a regular node is generated, referencing these stencils. Faces with boundaries or hard creases, that are otherwise regular, are handled by generating stencils for extrapolated control points (as described by Kobbelt [1996]), after which they are handled as regular nodes.

Crease Nodes Semi-sharp features can be eliminated by recursive subdivision. However, otherwise regular sub-faces with a single semi-sharp crease can be directly evaluated with good efficiency [Nießner et al. 2012b]. Such faces are stored as crease nodes, which reference 16 control points just like regular nodes, along with a flag to indicate the creased edge, and a floating-point sharpness; we discuss the evaluation of crease nodes in Section 6.2. Using crease nodes can greatly reduce the size of quadtrees; we evaluate this effect in Section 9.5.

Extraordinary Nodes If we reach our limit on subdivision depth, and have not reached a directly-evaluable sub-face, then we create an *extraordinary node*. Such a node corresponds to a corner of the base-mesh face which is an extraordinary vertex (EV), so we compute the limit position and two tangent stencils at that corner, and add them to the stencil list. The extraordinary node references these three stencils.

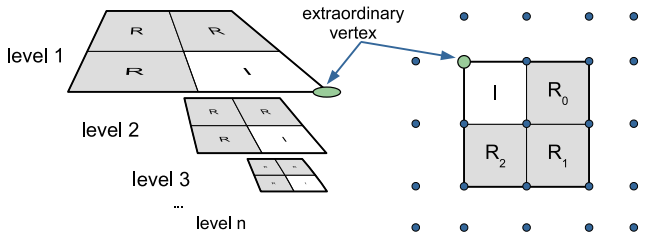


Figure 6: *A terminal node captures the repeating pattern of adaptive subdivision at an extraordinary vertex (left). For each subdivision level, the node references 24 control points shared by three regular sub-domains (right).*

Terminal Nodes In the common case (in the absence of semi-sharp feature tags), there is a single quadtree structure that arises around an EV (see Figure 6). At each subdivision level the quadtree will have three regular nodes and one internal node, terminated by an extraordinary node at the final level.

Rather than store this structure explicitly, we introduce a new kind of node: a *terminal node*, which collapses n levels of the hierarchy by standing in for $3n$ regular (or boundary) nodes and one extraordinary node. In our implementation, an extraordinary node is a special case of terminal node with $n = 0$.

The three regular nodes at each level will always share many control points. Rather than reference 48 control point stencils per level (16 each for three regular faces), a terminal node stores only 24 control points per level, laid out on a 5×5 grid . For each terminal node, we also store a rotation value, specifying which corner of the parametric domain corresponds to the extraordinary vertex.

4.2 Computing Stencils for Control Points

The quadtree in a subdivision plan is used to map a u, v location to a directly-evaluable sub-domain: e.g., a regular node. A leaf node in the quadtree indirectly references the required control point stencils (16 for a regular node). Each stencil encodes a control point as a weighted sum over 1-ring vertices. Representing stencils as weights over base vertices, rather than previous subdivision levels, ensures that each stencil can be applied independently, with no synchronization between levels.

Because stencils only depend on 1-ring vertices, we store each stencil as a dense array of weights, one for each vertex in the 1-ring, rather than pairs of weights and indices. Each face that uses the subdivision plan stores its own list of 1-ring vertices (e.g., as entries in a hardware index buffer).

Dense weight arrays reduce the amount of indirection-induced latency when computing the value of a control point. However, because floating-point addition is not associative, these computations are sensitive to the order of 1-ring vertices, which may not match between neighboring faces. When water-tight evaluation is needed, a per-face permutation must be applied to the weights to align them with a consistently-ordered list of vertices (e.g., sorted by index).

In addition to the control points of regular faces, the subdivison plan stores stencils for limit positions and tangents at each extraordinary vertex, and at any parametric locations (e.g., face corners) needed by the tessellation metric. We also refer to these additional positions and tangents as "control points."

For each control point, we calculate the minimum subdivision level where it is required. Control point stencils are sorted by increasing level, and we generate a small array indicating the number of stencils required at or below each subdivision level, up to our maximum level. Stencils required for extraordinary vertices or the tessellation metric are always computed, so we set their required level to zero.

When computing the required level for control-point stencils, we aggressively minimize the number of stencils applied at runtime, by exploiting two important properties. First, our quadtree traversal favors the patch interior when a parametric coordinates lies on a split plane. Second, we assume that for any tessellation factor f, the hardware tessellator never produces a vertex with a parametric u or v coordinate within $1/\lceil f \rceil$ of a patch edge (except for points *on* the corresponding edge or corner). Under these conditions, if a sub-domain intersects the interval $(1/2^n, 1 - 1/2^n)$, in either u or v, its control points are needed at a level $\leq n$. The first property is guaranteed by our implementation, as described in Section 6.1. The second is guaranteed on compliant implementations of Direct3D 11 *integer* and *fractional-even* tessellation modes.

4.3 Caching Plans in a Configuration Database

The plan for a face depends only on the *configuration* of elements that can exert an influence on the limit surface within its local domain. This includes the topology of the face and its 1-ring, sharpness tags for incident edges and vertices, and boundary rules.

These properties are extracted when preprocessing a face, and used as a key in a persistent *configuration database*. If a particular configuration has been encountered before, the existing plan for that configuration can be associated with the new face; otherwise, a new plan is built and recorded in the database.

The configuration database may be shared across models, and persisted across application runs. If faces are added, removed, or have their local topology or tags changed at run time, we need only repeat processing for the subset of faces that are affected.

5 Runtime Face Setup and Subdivision

At runtime, the face setup and subdivision stage is applied to each base mesh face, using a hull shader. This shader computes the edge and interior tessellation factors and the corresponding control points needed for surface evaluation. We currently implement both uniform tessellation and an adaptive metric based on that used by Call of Duty: Ghosts [Brainerd 2016]. The required subdivision level is computed as $n = \lceil log_2 \lceil f \rceil \rceil$, where f is the maximum tessellation factor for the face. Then the subdivision step applies the stencils for all level-n required control points, as described in Section 4.2. The subdivision plan is organized such that stencils can be applied in a simple loop over a flat array. Note that the subdivision step only computes control points; the subdivided face topology is encoded in the quadtree.

Each control point is computed as the convolution of the weights in its stencil with the face's 1-ring vertices. In general, 1-ring vertices can be fetched directly from memory, but performance can be improved by exploiting hardware vertex pipeline caches to amortize costs across faces that share vertices. Our target GPU supports up to 32 vertices as input to the hull shader. The vertex pipeline is is used even when there are more than 32 vertices in the 1-ring; any additional vertices are fetched directly from memory, bypassing the vertex shader. Because stencils are simple arrays of weights we can, without thread divergence, segregate the stencil convolution into pipeline vertices and memory-sourced vertices. Note that because the hull shader can only read 32 pipelined vertices, applications that wish to use the vertex shader for, e.g., animation are limited to 32 vertices in the 1-ring of each face; we have not found this restriction an issue for typical models.

A key difference from typical tessellation methods is that our hull shader outputs a variable (but bounded), rather than fixed, number of control points. The control points computed for each face are streamed to memory. In many cases, the control points written by the hull shader remain in cache when read by the domain shader, so there is little cost to communicate through memory. In our current implementation, the application allocates a buffer big enough to hold the worst-case number of control points for all faces in a model; in most cases the subdivision step does not use all of the allocated memory. In Section 8.1, we discuss future approaches that could eliminate the need for this conservatively allocated buffer.

Many adaptive tessellation metrics require access to limit positions or tangents to estimate lengths or curvatures on the limit surface. In our system, these positions and tangents are encoded as stencils in the subdivision plan. These, and any control points required at every subdivision level (e.g., limit positions/tangents at EVs), are computed before evaluating the tessellation metric. We use an entire SIMT warp (32 threads) to process each patch; to best exploit these threads, the number of stencils to apply up front is rounded up to a multiple of the warp width. We have found that speculatively computing some control points this way is a modest net win.

6 Surface Evaluation

The surface evaluation stage, implemented in the domain shader, takes as input the subdivision plan, the control points written by the subdivision step, and the parametric location of a vertex provided by the hardware tessellator. Surface evaluation begins by traversing the plan's quadtree to the provided parametric location.

6.1 Quadtree Traversal

The traversal of the quadtree is done in an iterative loop, rather than recursively. Traversal handles each type of node differently; we discuss the different cases here.

Internal Nodes Traversal advances to the child node (quadrant) containing the parameteric location. The parameteric location is then localized to the domain of the child node. As stated in Section 4.2, when a parametric location falls on a split between children, we choose the interior-most child. This choice allows us to compute fewer control points for power-of-two tessellation factors.

Extraordinary Nodes When traversal reaches an extraordinary node, the limit position and tangent control points associated with the node are fetched, and used to compute the final position and normal for the vertex. Our implementation ensures that an extraordinary node is only evaluated at the parametric location of the corresponding EV, so it can be implemented as a constant function.

Regular, Boundary, and Crease Nodes On reaching a directly-evaluable node, traversal notes the location of the 16 control points (stored as a 4×4 array), and any crease data. The shader then proceeds to B-spline evaluation.

Terminal Nodes When traversal reaches a terminal node, the correct sub-face is computed using logic similar to Stam's method [1998]. The subdivision level to descend to is given by $n = \lfloor -log_2(max(u, v)) \rfloor$, and both u and v are scaled by 2^n to localize them to the correct sub-domain. As an optimization, our implementation performs this mapping using integer math on the exponent bits of u and v.

This logic relies on the extraordinary vertex being at the origin in the parametric domain; we rotate the domain as needed. The level n must also be compared to the range of levels stored in the terminal node; a parametric location that maps to a level beyond those stored in the node treats the terminal node as an extraordinary node.

Otherwise, the traversal notes the location of the 16 control points for the regular sub-face the parametric location lands in (stored as a 5×5 array), and proceeds to B-spline evaluation.

6.2 B-Spline Evaluation

Whether traversal finds a regular, boundary, crease, or terminal node, all sub-domains that are evaluable as B-spline patches are funneled through a common code path, to reduce SIMT divergence.

First, at each of u and v we compute the B-spline basis value, and its derivative. Next, if there is a semi-sharp crease in the patch, the implementation computes and applies a crease matrix M using the construction from Nießner et al. [2012b]. Whereas Nießner et al. use M to transform the control points of the surface $(\text{basis} \cdot (M \cdot \text{controlPoints}))$, we observe that it is equivalent, and more efficient in our case, to apply the matrix to the basis values $((\text{basis} \cdot M) \cdot \text{controlPoints})$. This formulation also lets us isolate runtime support for semi-sharp creases to a single place in the pipeline; nothing else in our runtime implementation needs to be aware that semi-sharp creases exist.

Finally, our implementation fetches 16 control points for the sub-domain (from the buffer previously written by the hull shader), and evaluates the B-spline surface using the (potentially modified) basis functions. The only difference between the terminal and non-terminal cases is whether the 4×4 control points come from a 2D array with a stride of 5 or 4.

7 Special Faces

The algorithm presented so far can render a quad-only mesh in a single pass, with one primitive per face submitted to the hardware tessellator. In this section, we relax some of these assumptions to improve generality and performance.

7.1 Regular Faces

Regular (non-boundary/-crease) faces produce a subdivision plan with only a single regular node, and 16 stencils that reproduce base-mesh vertices. Rather than invoking stencil calculations and traversing a single-node quadtree, it is more efficient to render these faces as standard B-spline patches using the tessellation pipeline in a conventional manner. As is typical for other GPU subdivision surface rendering approaches, our implementation can split regular faces into a distinct optimized draw pass.

With this optimization, we can no longer render an entire subdivision surface model in a single draw call. However, the multiple draw calls used are independent, with no synchronization required, unlike the dependent compute and draw passes of FAS.

7.2 Non-Quad Faces

Our implementation handles non-quad faces by performing one round of subdivision on them during preprocessing; for an n-gon face, this yields n quads. The resulting quad sub-faces are then processed as normal. Note that we do *not* perform a global subdivision step—only non-quad faces are subdivided—and this step occurs during preprocessing, rather than at runtime.

The quads resulting from n-gon faces can be submitted in the same draw call as other (irregular) quad faces, so supporting non-quad faces does not increase the number of draw calls required.

By subdividing only some faces, T-junctions are introduced where quad and non-quad faces meet. To avoid cracks, the implementation must ensure that:

- tessellated vertex locations along the shared edge(s) line up
- computed vertex attributes at those locations will be the same

The first issue is solved in our implementation by constraining the tessellation factors along any edge shared between a quad and non-quad face. Considering the edge of the quad face, we require that it be of the form $2k$ for integer tessellation, or $4k$ for fractional-even tessellation, with k an integer. The adjacent edge of the non-quad face can then be given a tessellation factor of k or $2k$, respectively. This scheme is incompatible with fractional-odd tessellation.

The second issue is no different from ensuring consistent evaluation between other neighboring faces (e.g., between regular and irregular faces); our scheme ensures that evaluations along the shared edge will be mathematically equivalent, but may not be water-tight.

7.3 Macro Faces

At low tessellation rates, off-line subdivision outperforms hardware tessellation, in part due to per-patch overheads, and the inability to share tessellation vertices across primitives. These issues can be partly mitigated by aggregating adjacent base mesh faces into *macro faces*. We present two alternative face aggregation schemes.

Rectangular Macro Faces In the first scheme, the topology of the base mesh is divided into rectangular regions by separatrix intersection (readers unfamiliar with this concept may benefit from a survey by Bommes et al. [2013]). From each edge incident to an extraordinary vertex, a separatrix span is extended through regular vertices until it terminates at an extraordinary vertex or boundary. The intersections of the separatrices form the corners of rectangular regions of faces, which in aggregate collect the entire mesh topology into *rectangular macro faces*. T-junctions between macro faces are impossible, because a vertex cannot be an intersection for a subset of incident separatrices. In the special case of toroidal homotopy, an arbitrary regular separatrix origin is chosen.

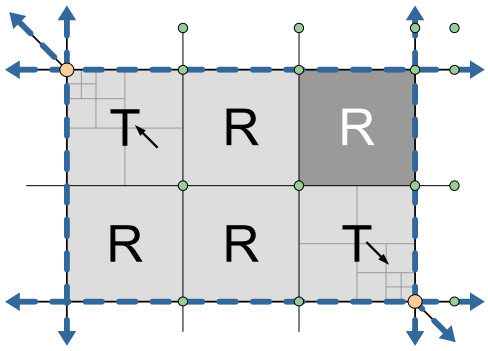


Figure 7: *A 3×2 rectangular macro face formed by the intersections of separatrices from two extraordinary vertices (orange). Aggregated irregular faces (T) reference quadtree root nodes, while regular faces (R) share a grid of B-spline control points; the control points (green) used by the shaded regular face are shown.*

The subdivision plan for a rectangular macro face uses a $W \times H$ uniform grid of per-face quadtrees (Figure 7). As a special case, regular faces do not use control point stencils, and instead share an array of $(W + 3) \times (H + 3)$ base vertices. The quadtrees and stencils for the remaining faces are merged and de-duplicated.

Figure 8 shows a model rendered with rectangular macro faces. The domain shader indexes and re-parameterizes the parametric location into the appropriate grid cell. Irregular cells are evaluated using their quadtree; for regular cells, 16 control points are fetched from the $(W + 3) \times (H + 3)$ array and evaluated as a B-spline patch.

Depending on base mesh topology, separatrix intersection can delineate rectangular macro faces of arbitrary size. Current hardware limits tessellation factors to 64 for the whole macro-face; as such,

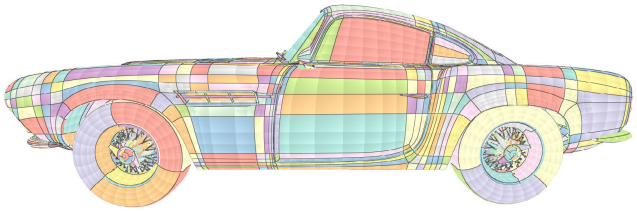


Figure 8: *The Stirling model (©Disney/Pixar) rendered with rectangular macro faces. Colors indicate different macro faces; shading shows the parameter (sub-)domains of aggregated face quadtrees. The wheels show that macro face dimensions are limited to 16, allowing a maximum per-face tessellation factor of 4.*

the effective tessellation factor for base faces is limited by the size of the macro face. Our implementation greedily adds perpendicular separatrices to spans longer than n edges, allowing aggregated faces to reach at least tessellation factor $64/n$.

In the special case of uniform integer tessellation factors, motorcycle graphs [Eppstein et al. 2008] can be employed, allowing for T-junctions between macro faces. Given a per-face tessellation factor of f, a $W \times H$ macro face would use tessellation factors of $f \cdot W$ and $f \cdot H$, ensuring a tessellated vertex is placed at each vertex of the base mesh. Adjacent faces in the original mesh will share a consistent set of tessellation vertices along their shared edge.

Regular Quartet Macro Faces In the second scheme, quartets of regular faces incident to a shared vertex are greedily extracted from the base mesh topology. A single 5×5 array of base mesh vertices can be used to evaluate any of the aggregated faces (Figure 9).

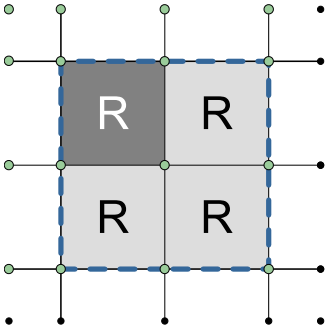


Figure 9: *A regular quartet macro face aggregates 4 regular faces. B-spline control points (green) for the shaded face are shown.*

Figure 10 shows a model rendered with separate passes for quartet, irregular, and un-aggregated regular faces. The shader used for quartets is similar to that for regular faces, but with 25 rather than 16 control points. The domain shader selects the correct sub-face, maps the location, and evaluates a B-spline patch using a subset of the control points.

When using no-uniform tessellation factors, quartets must be adjacent to either other quartets, or pairs of individual faces, to avoid cracks. Edges shared between quartets and individual faces are handled similarly to T-junctions between quad and non-quad faces.

8 Hardware Proposals

Our algorithm enables subdivision surfaces to be streamed through GPU tessellation pipelines, outperforming state-of-the-art techniques on current hardware. Nonetheless, some simple changes to GPU hardware could further improve the performance and memory efficiency of future subdivision surface rendering pipelines.

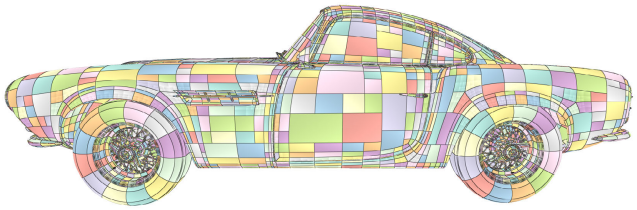


Figure 10: *The Stirling model (©Disney/Pixar) with regular, quartet, and irregular faces. As the tessellation factor is uniform, quartets are allowed to be offset relative to one another.*

8.1 Subdivision Ring Buffer

In our software implementation, the control points required for a particular adaptive subdivision level are calculated in the hull shader and used by the domain shader. Current graphics APIs require the amount of data output from each stage to be fixed at compile time. Statically allocating for the worst-case control point count would waste bandwidth, and limit occupancy. Thus, our implementation passes control points through a scratch buffer in memory; the buffer is sized for the worst case, but in practice we expect a fraction to be resident in caches. In practice, only a subset of primitives are processed at a time, so a smaller buffer should suffice.

As an example, the Armor Guy model has 8,409 irregular faces, which at maximum tessellation require 1,373,540 control points, or 22 MB of scratch storage at 16 bytes per aligned control point. On a GeForce GTX 980, the occupancy of our algorithm is limited to 12 primitives for each of 16 SMs. At an average of 192 control points per face, 1MB of storage would suffice for all primitives in flight.

We thus propose that hardware should support a *subdivision ring buffer*, written by the hull shader, and read by the domain shader. Upon determining the number of control points to write, a hull shader would allocate space on the ring buffer, and write the computed control points to that memory. The allocation step might amount to an atomic memory operation; indeed, an application can already implement simple allocation with atomic adds.

Deallocation is a harder task, benefiting more from hardware support. The space allocated by a hull shader needs to be kept live until all downstream domain shader invocations have completed; attempting to, e.g., reference-count this allocation in software would be complex and costly, but hardware pipelines already perform lifetime management for other data that flows between these stages. The total space needed for a hardware-managed ring buffer depends on the maximum number of primitives in flight, and the average control point count for each primitive.

8.2 Subdivision Cache

Our implementation processes each face independently, but in practice many control points will be shared between neighboring faces, or even across render passes (for example, when rendering to shadow maps). The Big Guy model requires a maximum of 105,648 control points for all faces to reach subdivision level 6. If stencils are de-duplicated between faces, this count decreases to 51,199.

We propose a *subdivision cache*, to enable re-use of control points within and across passes. A subdivision cache would be allocated by the application, to hold a fixed number of control point, referenced by index. The cache might be implemented as a buffer of control points, along with a buffer of flags (one per control point) to indicate validity. Before computing a control point, the hull shader would check for an existing entry in the cache at the corresponding index; on a miss it would compute the new control point and insert it into the cache.

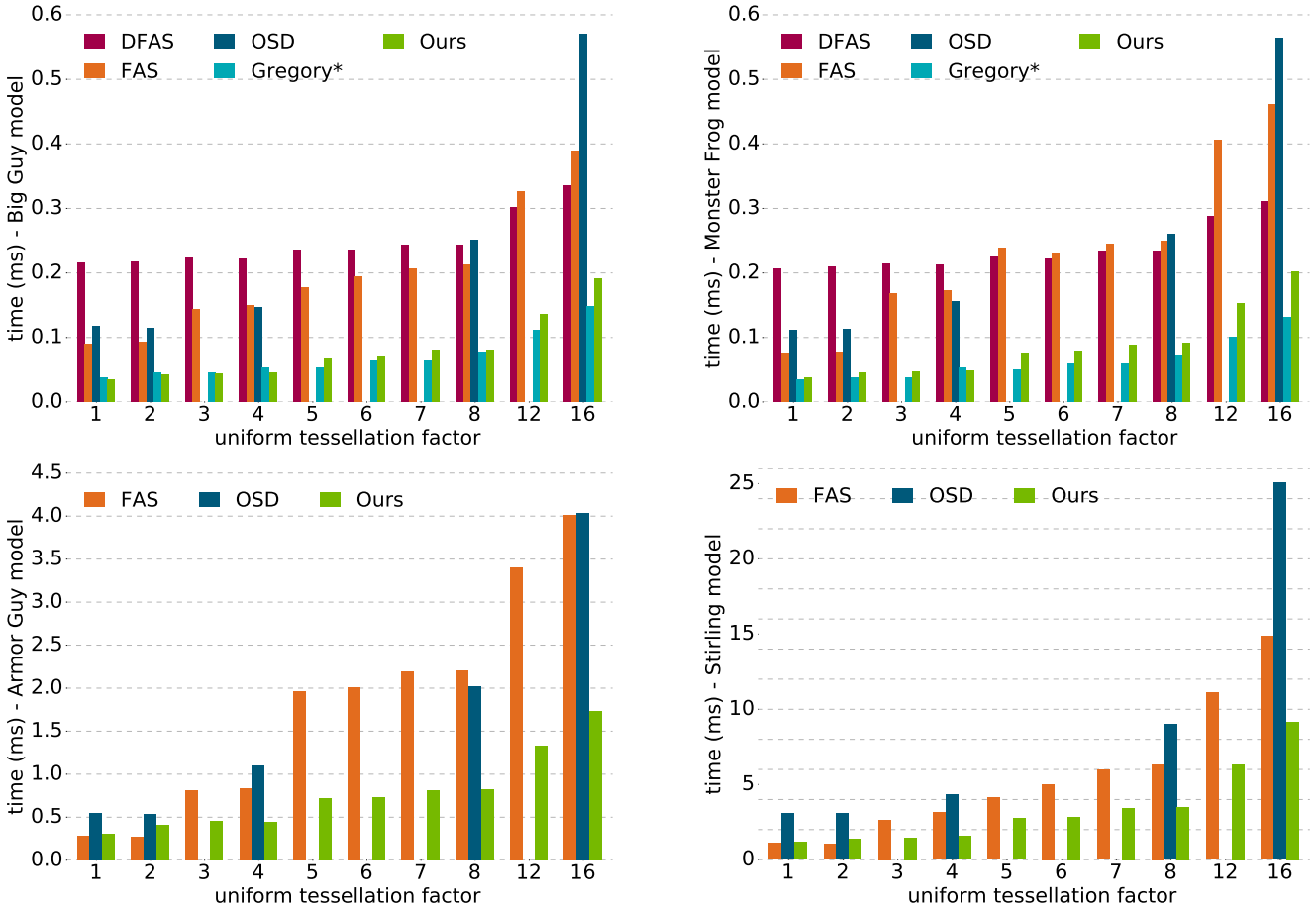


Figure 11: *Comparison of our method (green) against DFAS [Schäfer et al. 2015] (red), FAS [Nießner et al. 2012a] (orange), OpenSubdiv (blue), and the approximate Gregory scheme [Loop et al. 2009] (aqua), for each of the models in Figure 12. We outperform all existing non-approximate methods by a significant margin, and our performance is on par with the fastest approximate method that uses Gregory patches. Note that Gregory cannot handle semi-sharp creases, thus Armor Guy and Stirling cannot be rendered with this approach.*

The subdivision ring buffer and subdivision cache are mutually exclusive approaches, optimized for different use cases. The ring buffer optimizes for the convenience of single-pass streaming submission, and leverages the way our algorithm can process patches in isolation. The subdivision cache optimizes for repeated submission, and mitigates the primary source of overhead in our algorithm (redundant control point computations).

Name	Description	API
Ours	Our adaptive quadtree approach	OpenGL
OSD	OpenSubdiv 3.0	Direct3D 11
FAS	[Nießner et al. 2012a]	Direct3D 11
DFAS	[Schäfer et al. 2015]	Direct3D 11
Gregory*	[Loop et al. 2009]	Direct3D 11

Table 1: *Rendering methods evaluated. Note that Gregory only approximates the limit surface and cannot handle semi-sharp creases.*

9 Evaluation

We evaluate our method on NVIDIA GeForce GTX 980 hardware, under Windows, and compare against the methods listed in Table 1. Note that we compare against the published code for FAS and DFAS, while for Gregory, we use the optimized implementation in the FAS codebase. Figure 12 shows renderings of the models used for evaluation, and Table 2 summarizes their statistics and features.

9.1 End-to-End Rendering Performance

Figure 11 compares end-to-end time to render the models with our approach (Ours) and state-of-the-art methods. The adaptive subdivision implementations (FAS, DFAS, OSD) submit different geometry based on the desired tessellation factor. In order to provide a fair comparison, we use uniform tessellation factors, and only subdivide as required for each factor. Performance results for OSD are only shown for power-of-two tessellation factors; the OSD viewer enforces this restriction for uniform tessellation. The Gregory approximation scheme does not support creases, so results are only available for Big Guy and Monster Frog. Also, The DFAS implementation failed to load the more complicated models.

In Figure 11, our method is between 1.2 and 3.4 times faster than the other non-approximate approaches for tessellation factors of 3 and up, while FAS is sometimes faster for low factors. Our performance is competitive with the fastest approximation (Gregory); however, Gregory patching cannot handle semi-sharp creases, which are widely used (e.g., Armor Guy and Stirling).

The performance benefit of our method is primarily due to submitting fewer primitives to the tessellation hardware, thus reducing per-primitive costs. For irregular faces, FAS, DFAS, and OSD submit more patches as the tessellation rate increases, while we always submits the same, constant number of primitives. On models with many regular faces (e.g., Stirling), the benefit of our method is lessened; FAS has approximately equal performance on regular faces.

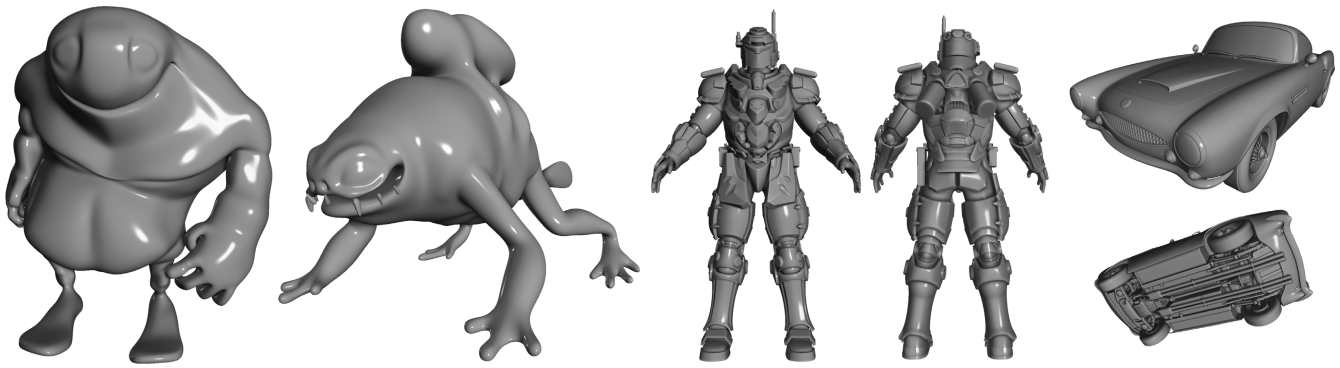


Figure 12: *Models used in our evaluation, from left to right: Big Guy, Monster Frog, Armor Guy (©2014 DigitalFish, Inc. All rights reserved), and Stirling (©Disney/Pixar).*

Model	Faces	Regular	Irreg.	Tags
Big Guy	1450	858	592	none
Monster Frog	1284	510	774	none
Armor Guy	8639	962	7677	creases
Stirling	79895	35264	44631	creases

Table 2: *Statistics for models used, including number of faces and a breakdown into regular/irregular faces. Two models make use of crease tags, and all but Big Guy include non-quad faces.*

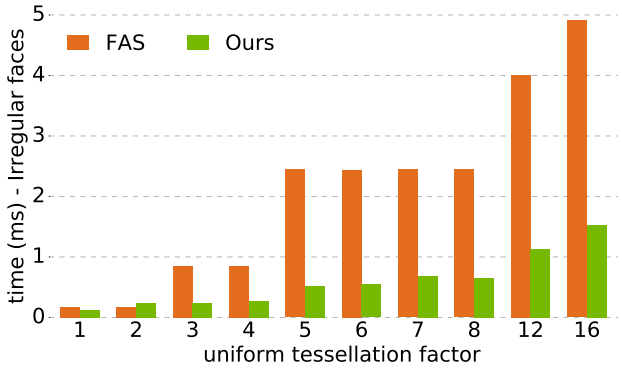


Figure 13: *The performance benefit of Ours is greatest for irregular faces; Ours can be over three times as fast as FAS for a synthetic model of only irregular faces (a* $12 \times 12 \times 10$ *grid of cubes).*

Figure 13 shows how we compare to FAS on a synthetic model with only irregular faces. In this range of tessellation factors, the performance of FAS is dominated by per-primitive costs; note how the FAS results are flat for tessellation factors 5 through 8, which all submit the same number of primitives.

FAS can be faster than our method at low tessellation rates because our single-pass implementation does not share computed control points across faces, while the multi-pass FAS implementation does. The potential benefit of sharing control points is greatest at low tessellation factors. To show this effect, we next break down the timings for FAS and our approach.

9.2 Time Breakdown

Figure 14 summarizes how much time FAS and our method take to compute control points (by applying stencils), and to draw a tessellated surface for the Armor Guy model. In the case of FAS, the time spent on stencils is clearly delineated by its own compute pass. In our case, we perform control point computation and drawing in the first frame, while subsequent frames disable control point computation in the hull shader and re-use the previously computed values.

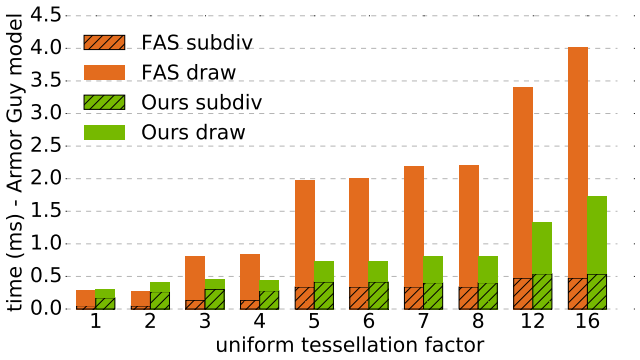


Figure 14: *Breakdown of time spent applying stencils (hatched) and doing other rendering work (solid) for FAS and Ours. FAS applies stencils in a distinct compute pass; Ours integrates this work into the hull shader, at the cost of some redundant computation.*

For our method, stencil computation constitutes between 30% and 54% of the total time, while FAS spends between 13% and 17%; in both cases the fraction of time spent on stencils decreases at higher tessellation factors. We spend more time overall on stencil application, since our implementation must independently compute shared control points per-face. Ours compensates for this with increased drawing efficiency. In scenarios where control points can be reused across passes or frames, each implementation only pays for the drawing time (solid bars) in this graph.

In the case of a tessellation factor of 1 or 2, our method has slightly better draw time than FAS; the end-to-end performance difference is primarily due to redundant control point computations. In Section 8.2, we describe how an idealized subdivision cache could avoid redundant control point computations, and how to improve the performance at low tessellation factors.

9.3 Adaptive Tessellation

When using uniform tessellation (as in Figure 11), FAS outperforms DFAS, which includes additional overhead to support non-uniform subdivision. Figure 15 compares DFAS and our method using a distance-adaptive tessellation metric on Big Guy. The performance of both approaches improves with distance, but we have lower overheads, and better performance across the distance range.

9.4 Macro Faces

We evaluated the impact of macro faces (Section 7.3) on rendering performance, using the Stirling model, as well as a synthetic model (a torus) composed entirely of regular faces. The 79,895 input faces of the Stirling model were aggregated into 12,737 rectangular macro faces. Using quartet macro faces instead yields 4,852 quartets, 46,110 irregular and 15,852 individual regular faces.

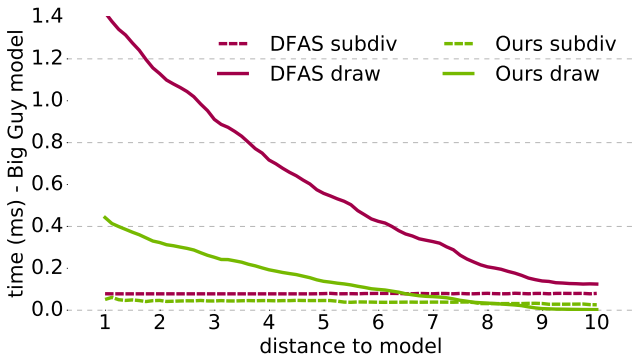


Figure 15: *Our approach outperforms DFAS when using distance-adaptive tessellation, due to lower overheads and faster drawing.*

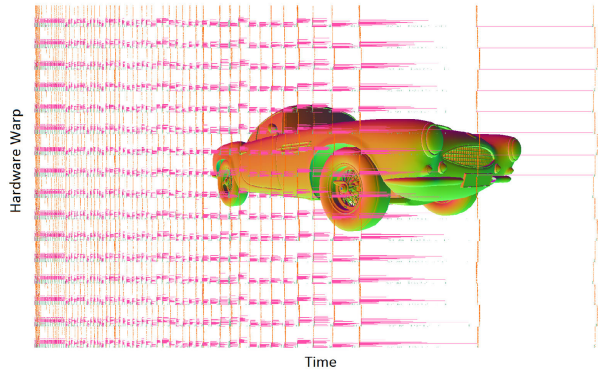


Figure 16: *Pipeline stalls limit the performance of rectangular macro-patches. In this diagnostic visualization, color boxes represent active warps. Individual long-running hull shader warps (purple) stall the tessellation pipeline, reducing overall performance. Stirling model ©Disney/Pixar.*

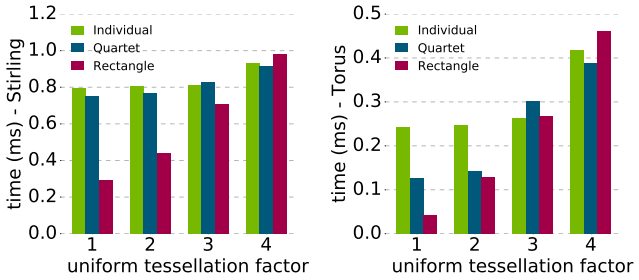


Figure 17: *Performance evaluation of using rectangular and quartet macro-faces. Stencil time is removed to isolate draw performance from occupancy limits related to the hull shader (Figure 16).*

We found that rectangular macro faces did not improve end-to-end performance for non-trivial models; upon investigation, we found that when using rectangular macro faces a small number of long-running warps in the hull shader were keeping other work from running. Figure 16 shows a diagnostic visualization of running GPU tasks; the horizontal axis is time, and the vertical axis shows different hardware warps. It appears that primitives are consumed in order by the hardware tessellator (in order to provide the ordering guarantees required by rasterization APIs), so that a long-running hull shader can stall the pipeline. Aggregating many faces (and their control points) into macro faces increases the worst-case time spent on control point computations in the hull shader, increasing the likelihood and impact of pipeline stalls.

	Nodes	Stencils	Render (ms)
Torus			
Recursive	8,634	6,200	0.084
Direct	1,338 (-84%)	1,232 (-80%)	0.052 (-38%)
Stirling			
Recursive	∼5.3M	∼4M	9.83
Direct	∼4M (-23%)	∼3.4M (-14%)	10.46 (+6%)

Table 3: *Impact of direct vs. recursive evaluation of creases on total size of subdivision plans (quadtree nodes and stencils), as well as render time, for Stirling and a creased torus (Figure 18).*

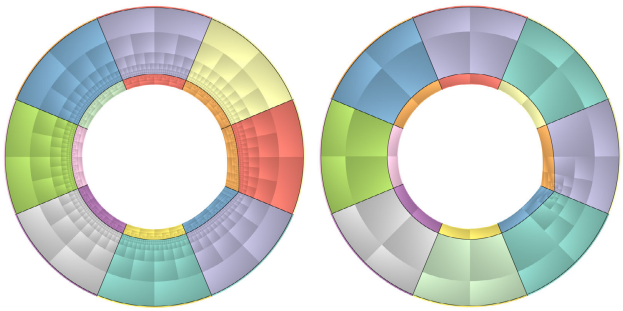


Figure 18: *A creased torus without (left) and with (right) direct evaluation of semi-sharp creases, using exponential (left) and linear (right) numbers of quadtree nodes, respectively.*

In Figure 17, we remove the influence of these pipeline stalls by comparing draw times only; control points are computed in a prepass. Under these conditions, rectangular macro faces improve performance at the lowest per-face tesellation factors, but benefit decreases as tessellation factors increase. Quartet macro faces provide more benefit for the synthetic model with only regular faces than for Stirling. In both cases, performance decreases as tessellation factor increases. The use of a more expensive domain shader makes macro faces less valuable at higher tessellation factors.

If future hardware can address the bottlenecks in Figure 16, rectangular macro faces could be useful for minification. Macro faces outperform individual faces for tessellation factors of 2 and below, and enable effective per-face tessellation factors below 1.0.

9.5 Direct Evaluation of Semi-Sharp Creases

As discussed in Section 6.2, our implementation supports direct evaluation of semi-sharp creases. In the best case, direct evaluation avoids a potentially exponential increase in the number of quadtree nodes and control points required (w.r.t. crease sharpness). However, directly handling creases adds complexity to our domain shader.

We evaluated the impact of using direct evaluation, as opposed to recursive subdivision, for semi-sharp creases; the results are summarized in Table 3. The first model is a torus with several creases of sharpness 4.7 (see Figure 18). The second is the Stirling which uses creases of sharpness 1, 2, and 3. For each model and approach, the table gives the total number of quadtree nodes and stencils across all subdivision plans, as well as end-to-end render times. In the case of the torus, direct evaluation yields a speedup of 38%, while for the Stirling we observe a slowdown of 6%.

The reason that enabling direct evaluation causes a slowdown for Stirling is that the code path that supports semi-sharp creases is always present, even for faces that don't need it. The added code increases register pressure, thus slightly lowering warp occupancy. Furthermore, only a subset of domain shader threads execute the

	Vertex	Index	Plans	Quadtree	Stencils	Scratch
Big Guy	23KB	107KB	19	21KB	346KB	1.6MB
Monster Frog	20KB	119KB	70	76KB	1.1MB	2.2MB
Armor Guy	157KB	1MB	2,099	2.7MB	25MB	21MB
Stirling	1.3MB	7MB	1,859	1.8MB	23MB	52MB

Table 4: *Memory required for models in Figure 12, including vertex and index buffers, number of subdivision plans, sizes of quadtree and stencil data, and size of scratch buffer needed at runtime.*

crease-related code, increasing SIMT divergence. Because only a small fraction of the faces on Stirling are creased (and these have very low sharpness values) the additional overhead of rendering them via recursive quadtree nodes is low.

9.6 Memory Usage

Table 4 summarizes the memory requirements for our implementation to render each of our models. The vertex storage only includes base mesh vertex position data; this is independent of our algorithm, but included for reference. The more complex models require more plans, due to a greater variety of topological configurations and crease tags. The static storage for each model is dominated by stencils for support control points; storing this data more compactly is a possible direction for future work.

The largest runtime cost is the "scratch" memory required to stream supports from the hull shader to the domain shader. In the case of our largest model, Stirling, the application must allocate a scratch buffer of 55 MB. Hardware support for a subdivision ring buffer (Section 8.1) would reduce this to a fixed driver-managed overhead.

9.7 Limitations

Our approach relies on having a fixed limit on subdivision depth, when generating the subdivision plan. Furthermore, for performance we rely on the fact that control points at deeper adaptive subdivision levels are only needed at higher tessellation rates.

The fixed limit on subdivision depth means that our approach cannot accurately evaluate the limit surface at arbitrary parametric locations, which might be desirable, e.g., to produce high-quality normals by evaluating the surface per-fragment. Where approximate results are acceptable, our extraordinary nodes could be replaced with, e.g., Gregory patches approximating the limit surface.

Our approach also has limitations that make it difficult to use the *fractional-odd* hardware tessellation mode. First, our approach to avoiding cracking for non-quad faces and quartet macro-faces (Sections 7.2 and 7.3) relies on placing a tessellated vertex at the midpoint of edges that would otherwise create T-junctions; this is not possible with an odd tessellation factor. Second, for tessellation factors in the $(1, 2.5)$ range, tessellated vertices may have parametric coordinates arbitrarily close to patch corners, violating our assumption that control points for deeper subdivision levels are only needed at higher tessellation rates (Section 4.2).

Our current implementation does not support hierarchical edits [Forsey and Bartels 1988], but we expect that our subdivision plan structure could be extended to represent edits. However, because edits might require adaptive subdivision in the interior of a face (not just at corners/edges), our optimizations to avoid computing control points at low tessellation factors might be less effective.

As described in Section 5, when using a vertex shader (e.g., for animation), the number of vertices in the 1-ring of a face is restricted to the number of vertices that can be passed from a vertex shader to a hull shader (32 on current APIs and GPUs).

Figure 19: *Our approach has been implemented in a production game engine, and is used here to render the character's face.*

Although we developed our algorithm and implementation only for Catmull-Clark subdivision [1978], we expect the technique would generalize to other subdivision schemes such as Loop [1987].

10 Conclusion

We have introduced a novel approach for rendering subdivision surfaces, which integrates into the hardware tessellation pipeline. By submitting only a single primitive for each input quad face, we improve both the simplicity and efficiency of rendering. Our approach achieves performance up to three times that of state-of-the-art methods for typical tessellation factors.

We have integrated our approach into a production game engine (Figure 19), and hope that by demonstrating a streamlined approach for rendering full-featured subdivision surface models, we can spur further interest in, and adoption of, subdivision surfaces in game rendering. With further improvements in hardware, subdivision surfaces may soon be as easy to render as triangle meshes.

Acknowledgments

The authors would also like to thank Akimitsu Hogge for ideas and useful discussion, Christer Ericson, John Spitzer, and Aaron Lefohn for their support of this work, and the anonymous reviewers for their valuable feedback.

We would like to thank Sledgehammer Games for generously allowing us the use of assets from their game. Armor Guy model courtesy of ©2014 DigitalFish, Inc. All rights reserved. We are also grateful for the Stirling model, used with permission from Disney/Pixar. We made use of many test models from the OpenSubdiv distribution. In addition, we thank Bay Raitt of Valve Software for the Big Guy and Monster Frog models.

References

ANDREWS, J., AND BAKER, N. 2006. Xbox 360 System Architecture. *IEEE Micro 26*, 2, 25–37.

BOLZ, J., AND SCHRÖDER, P. 2002. Rapid evaluation of catmull-clark subdivision surfaces. In *Proceedings of the seventh international conference on 3D Web technology*, ACM, 11–17.

BOMMES, D., LÉVY, B., PIETRONI, N., PUPPO, E., SILVA, C., TARINI, M., AND ZORIN, D. 2013. Quad-mesh generation and processing: A survey. *Comput. Graph. Forum 32*, 6 (Sept.), 51–76.

BRAINERD, W. 2016. Catmull-clark subdivision surfaces. In *GPU Pro 7: Advanced Rendering Techniques*, W. Engel, Ed. A K Peters/CRC Press, Boca Raton, FL, USA.

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

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

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes Image Rendering Architecture. *Computer Graphics (Proceedings of SIGGRAPH) 21*, 4, 95–102.

DEROSE, T., KASS, M., AND TRUONG, T. 1998. Subdivision Surfaces in Character Animation. In *Proceedings of SIGGRAPH 98*, Annual Conference Series, ACM, 85–94.

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

EPPSTEIN, D., GOODRICH, M. T., KIM, E., AND TAMSTORF, R. 2008. Motorcycle graphs: Canonical quad mesh partitioning. In *Proceedings of the Symposium on Geometry Processing*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, SGP '08, 1477–1486.

FORSEY, D. R., AND BARTELS, R. H. 1988. Hierarchical b-spline refinement. In *ACM SIGGRAPH Computer Graphics*, vol. 22, ACM, 205–212.

HE, L., LOOP, C., AND SCHAEFER, S. 2012. Improving the parameterization of approximate subdivision surfaces. In *Computer Graphics Forum*, vol. 31, Wiley Online Library, 2127–2134.

HOPPE, H., DEROSE, T., DUCHAMP, T., HALSTEAD, M., JIN, H., MCDONALD, J., SCHWEITZER, J., AND STUETZLE, W. 1994. Piecewise Smooth Surface Reconstruction. In *Proceedings of SIGGRAPH*, ACM, 295–302.

KOBBELT, L. 1996. Interpolatory subdivision on open quadrilateral nets with arbitrary topology. In *Computer Graphics Forum*, 409–420.

KOBBELT, L. 2000. $\sqrt{3}$-subdivision. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 103–112.

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

KOVACS, D., MITCHELL, J., DRONE, S., AND ZORIN, D. 2010. Real-time creased approximate subdivision surfaces with displacements. *IEEE Transactions on Visualization & Computer Graphics*, 5, 742–751.

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

LOOP, C., SCHAEFER, S., NI, T., AND CASTAÑO, I. 2009. Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation. *ACM Trans. Graph. 28*, 151:1–151:9.

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

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

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

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

NASRI, A. 1987. Polyhedral Subdivision Methods for Free-Form Surfaces. *ACM Transactions on Graphics (TOG) 6*, 1, 29–73.

NI, T., YEO, Y., MYLES, A., GOEL, V., AND PETERS, J. 2008. Gpu smoothing of quad meshes. In *Shape Modeling and Applications, 2008. SMI 2008. IEEE International Conference on*, IEEE, 3–9.

NIESSNER, M., AND LOOP, C. 2013. Analytic displacement mapping using hardware tessellation. *ACM Transactions on Graphics (TOG) 32*, 3, 26.

NIESSNER, M., LOOP, C., MEYER, M., AND DEROSE, T. 2012. Feature-adaptive gpu rendering of catmull-clark subdivision surfaces. *ACM Transactions on Graphics (TOG) 31*, 1, 6.

NIESSNER, M., LOOP, C. T., AND GREINER, G. 2012. Efficient evaluation of semi-smooth creases in catmull-clark subdivision surfaces. In *Eurographics (Short Papers)*, 41–44.

NIESSNER, M., KEINERT, B., FISHER, M., STAMMINGER, M., LOOP, C., AND SCHÄFER, H. 2015. Real-time rendering techniques with hardware tessellation. *Computer Graphics Forum 35*, 1, 113–137.

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).

SCHÄFER, H., NIESSNER, M., KEINERT, B., STAMMINGER, M., AND LOOP, C. 2014. State of the art report on real-time rendering with hardware tessellation. In *Eurographics 2014 - State of the Art Reports*, EG, 93–117.

SCHÄFER, H., RAAB, J., KEINERT, B., MEYER, M., STAMMINGER, M., AND NIESSNER, M. 2015. Dynamic feature-adaptive subdivision. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, ACM, 31–38.

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. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, 395–404.

WEBER, T., WIMMER, M., AND OWENS, J. D. 2015. Parallel reyes-style adaptive subdivision with bounded memory usage. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*.