Rendering Subdivision Surfaces using Hardware Tessellation

Der Technischen Fakultät
der Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur
Erlangung des Grades Dr.-Ing.

vorgelegt von

Matthias Nießner

aus Gunzenhausen

Als Dissertation genehmigt
von der Technischen Fakultät
der Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der mündlichen Prüfung:	15.07.2013
Vorsitzender des Promotionsorgans:	Prof. Dr.-Ing. habil. Marion Merklein
Gutachter:	Prof. Dr. Günther Greiner
	Prof. Dr. Hans-Peter Seidel

Abstract

Computer-generated images are an essential part of today's life where an increasing demand for richer images requires more and more visual detail. The quality of resulting images is strongly dependent on the representation of the underlying surface geometry. This is particularly important in the movie industry where subdivision surfaces have evolved into an industry standard. While subdivision surfaces provide artists with a sophisticated level of flexibility for modeling, the corresponding image generation is computationally expensive. For this reason, movie productions perform rendering offline on large-scale render farms. In this thesis we present techniques that facilitate the use of high-quality movie content in real-time applications that run on commodity desktop computers. We utilize modern graphics hardware and use hardware tessellation to generate surface geometry on-the-fly based on patches. The key advantage of hardware tessellation is the ability to generate geometry on-chip and to rasterize obtained polygons directly, thus minimizing memory I/O and enabling cost-effective animations since only patch control points need to be updated every frame. We first convert subdivision surfaces into patch primitives that can be processed by the tessellation unit. Then patches are directly evaluated rather than by iterative subdivision. In addition, we add high-frequency surface detail on top of a base surface by using an analytic displacement function. Both displaced surface positions and corresponding normals are obtained from this function and the underlying subdivision surface. We further present techniques to speed up rendering by culling hidden patches, thus avoiding unnecessary computations. For interaction amongst objects themselves we also present a method that performs collision detection on hardware-tessellated dynamic objects. In conclusion, we provide a comprehensive solution for using subdivision surfaces in real-time applications. We believe that the next generation of games and authoring tools will benefit from our techniques in order to allow for rendering and animating highly detailed surfaces.

Acknowledgements

Writing a thesis on your own is hardly feasible. Therefore, I have to thank a lot of people for their help and support.

First of all I would like to thank my Ph.D advisor Günther Greiner who allowed me to join his amazing group; the computer graphics group at the University of Erlangen-Nuremberg. He continuously supported me, gave me unlimited academic freedom and guided me through the past years. I am very grateful for having such a great advisor and mentor.

I would also like to thank Marc Stamminger who draw my interest to the field of computer graphics and advised both my study and diploma thesis. He is always available to give thorough advice and his positive way of thinking is really unique.

A special thanks goes to Charles Loop who was my mentor during three internships at Microsoft Research. I was allowed to benefit from his deep insights into computer graphics in countless very productive discussions. Working with him was always a pleasure and he made my internships a most enjoyable experience.

I am a strong believer in the success of joint work and collaboration. Thus, I would like to particularly thank all the fantastic co-authors of my papers: Tony DeRose, Günther Greiner, Shahram Izadi, Benjamin Keinert, Nadine Kuhnert, Charles Loop, Mark Meyer, Georg Michelson, Henry Schäfer, Kai Selgrad, Christian Siegl, Marc Stamminger, Roman Sturm, Michael Zollhöfer. I am very thankful for all their contributions that helped me to successfully write this thesis.

Further, I want to thank all my colleagues at the computer graphics group of the University Erlangen-Nuremberg for such a great working environment: Maria Baroti, Frank Bauer, Elmar Dolgener, Christian Eisenacher, Günther Greiner, Roberto Grosso, Benjamin Keinert, Jan Kretschmer, Michael

Martinek, Qurin Meyer, Magdalena Prus, Nadja Ray, Henry Schäfer, Kai Selgrad, Christian Siegl, Marc Stamminger, Jochen Süßmuth, Michael Zollhöfer. In particular, I thank my office mate Michael Zollhöfer for patiently listening to all my complaints and weird research ideas, and Jochen Süßmuth for the LaTeX-template of this thesis. Another thanks goes to the computer graphics group at Microsoft Research in Redmond for all the interesting conversions at the group lunches: Mark Finch, Brian Guenter, Hugues Hoppe, Neel Joshi, Charles Loop, John Snyder.

However, the most significant contribution to my thesis is attributed to my parents. Their unconditional support and believe in me shaped not only my personality but also allowed me to exclusively focus on my studies. I never had to worry about anything beyond my education, for what I am very grateful. This eventually led to the successful completion of this thesis.

I also thank different parties for providing the test models that are being used in this thesis: Disney/Pixar (*McQueen*, *Car Body*, *Garbage Truck*, *Sandy Terrain*; thanks to Mark Meyer for making this possible), Bay Raitt (*Monster Frog*, *Big Guy*, *Sportscar*), The Blender Foundation (*Dragon Head*, *Cow Carcass*, *Chinchilla*, *Mushroom*), Headus (metamorphosis) Pty Ltd. (*Killeroo*).

--- Matthias Nießner,
Erlangen Jul. 2013

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction and Fundamentals

1.1 Motivation

Computer-generated images have become an essential part of our daily life in areas ranging from feature films to video games. Requirements regarding performance and image quality depend on the respective application. For instance, while in movies there is no tolerance for visual error, interactive applications rely on approximations in order to achieve real-time performance. Quality of generated images is significantly impacted by the underlying surface geometry. In particular, the representation of the provided geometry is crucial. Typically, real-time applications make use of polygonal object descriptions. While those can be efficiently processed by today's graphics hardware, polygonal surfaces are only linearly approximated. An alternative are tensor product surfaces where a set of regular control points defines a smooth surface. However, the requirement for regular control point grids has limited their adoption. Subdivision surfaces close the gap between polygonal and tensor product surfaces since they define a smooth surface on arbitrary topology.

Subdivision surfaces have been used extensively by the movie industry for some time. High-quality surface geometry can be obtained by defining a set of control points. Geometric properties correspond to the used subdivision scheme. Subdivision surfaces are evaluated by applying a set of subdivision rules in an iterative manner. This recursive evaluation is computationally expensive, thus requiring a significant amount of resources. In movie productions rendering of subdivision surfaces is performed in an offline process, typically on large-scale render farms. The goal of this thesis is to perform rendering of such surfaces in real-time on single desktop machines, thus enabling the use of high-quality movie content in interactive applications at minimal computational costs. For instance, the techniques we present facilitate authoring tools to provide instant and accurate feedback for artists during content creation. In addition, movie content can be used in video games without a labor-intensive and potentially error-prone conversion process.

In order to perform rendering of subdivision surfaces in real-time, we utilize modern but commodity graphics processing units (**GPUs**). In the recent years GPUs have evolved to massively-parallel computing architectures achieving multiple giga floating point operations per second [Nvi08]. We attempt to exploit those capabilities by designing our algorithms accordingly. For instance, an important requirement is to keep memory I/O low compared to performed floating point operations. Hardware tessellation [Mic09] ideally fits into this paradigm by processing patch primitives that define a smooth surface. Surface geometry is generated on-chip where obtained polygons are directly rasterized without the need of further memory access. That facilitates low-cost mesh animations by updating patch control points. With respect to subdivision surfaces the challenge is the conversion into patch primitives that can be processed by the tessellation unit. In addition, enabling high-frequency detail with displacements and fast patch processing is essential. In this thesis we present approaches (see Section 1.2) for hardware tessellation that together provide a comprehensive solution for rendering subdivision surfaces in real-time.

1.2 Contributions

In this thesis we contribute to the field of subdivision surface rendering. We present real-time rendering techniques for hardware tessellation that allow the usage of subdivision surfaces in interactive applications such as video games or authoring tools. The methods presented in this thesis range from surface evaluation approaches that facilitate high-frequency surface detail to culling and collision detection algorithms.

Our contributions include the following approaches:

- **Feature Adaptive GPU Rendering of Subdivision Surfaces** (see Chapter 4): We present a novel method for high-performance GPU based rendering of Catmull-Clark subdivision surfaces. Unlike previous methods, our algorithm computes the true limit surface up to machine precision, and is capable of rendering surfaces that conform to the full RenderMan specification for Catmull-Clark surfaces. Specifically, our algorithm can accommodate base meshes consisting of arbitrary valence vertices and faces, and the surface can contain any number and arrangement of semi-sharp creases and hierarchically defined detail. We also present a variant of the algorithm which guarantees watertight positions and normals, meaning that even displaced surfaces can be rendered in a crack-free manner. Finally, we describe a view dependent level-of-detail scheme which adapts to both the depth of subdivision and the patch tessellation density. Though considerably more general, the performance of our algorithm is comparable to the best approximating method, and is considerably faster than Stam's exact method [Sta98].
- **Efficient Evaluation of Semi-Sharp Creases in Catmull-Clark Subdivision Surfaces** (see Chapter 5): We present a novel method to evaluate semi-sharp creases in Catmull-Clark subdivision surfaces. Our algorithm supports both integer and fractional crease tags corresponding to the RenderMan (Pixar) specification. In order to perform fast and efficient surface evaluations, we obtain a polynomial surface representation given by the semi-sharp subdivision rules. We apply direct surface evaluation on regular patches and we perform adaptive subdivision around extraordinary vertices. In the end, we are able to efficiently handle high-order sharpness tags at very low cost. Compared to standard feature adaptive rendering, both render time and memory consumption are reduced from exponential to linear complexity. Furthermore, we integrate our algorithm in the hardware tessellation pipeline of modern GPUs. Our method is ideally suited to real-time applications such as games or authoring tools.

- **Analytic GPU Displacement Mapping for Subdivision Surfaces** (see Chapter 8): We present a novel method for leveraging hardware tessellation to apply a higher order displacement function to a surface at runtime. Displacement mapping is ideal for modern GPUs since it enables high-frequency geometric surface detail on models with low memory I/O. However, problems such as texture seams, normal re-computation, and under-sampling artifacts have limited its adoption. We provide a comprehensive solution to these problems by introducing a smooth analytic displacement function. Coefficients are stored in a GPU-friendly tile-based texture format, and a multi-resolution mip hierarchy of this function is formed. We propose a novel level-of-detail scheme by computing per vertex adaptive tessellation factors and select the appropriate pre-filtered mip levels of the displacement function. Our method obviates the need for a precomputed normal map since normals are directly derived from the displacements. Thus, we are able to perform authoring and rendering simultaneously without typical displacement map extraction from a dense triangle mesh. In addition, our approach facilitates high-quality shading since accurate surface normals are computed on a per pixel basis. This not only is more flexible than the traditional combination of discrete displacements and normal maps, but also provides faster runtime due to reduced memory I/O.

- **Effective Back-Patch Culling for Hardware Tessellation** (see Chapter 11): We present a novel approach to speed up rendering for hardware tessellation using back-patch culling. When rendering objects with hardware tessellation, back-facing patches should be culled as early as possible to avoid unnecessary surface evaluations, and setup costs for the tessellator and rasterizer. For dynamic objects the popular cone of normals approach is usually approximated using tangent and bitangent cones. This is faster to compute, but less effective. Instead we use the Bézier convex hull of the *parametric tangent plane*. This is much more accurate, and by operating in clip space we are able to reduce the computational cost significantly. As our algorithm vectorizes well, our culling approach can be efficiently implemented on the GPU allowing a substantial performance improvement.

- **Patch-Based Occlusion Culling for Hardware Tessellation** (see Chapter 12): We present an algorithm for performing occlusion culling on a per patch basis designed to run within the hardware tessellation pipeline. Our method dynamically generates a hierarchical Z-buffer, bounds patches on-the-fly, and tests if a patch is occluded prior to tessellation. In particular, we support displaced surfaces using tight but conservative bounds that allow effective culling. In order to deal with arbitrary scene environments, all patches can be occluders and be occluded by other patches as well as standard triangle primitives. We harness temporal coherence to maintain the visibility status of patches. These status bits are updated each frame and determine the Hi-Z map contributors. The algorithm is conservative in a sense that visible patches are never culled. Our method provides a simple, low-cost, and effective way to avoid wasting computations rendering occluded patches. It is well-suited for real-time applications such as games or authoring tools.
- **Collision Detection for Hardware Tessellation** (see Chapter 15): We present a novel method for real-time collision detection of patch-based, displacement mapped objects using hardware tessellation. Our method supports fully animated, dynamically tessellated objects and runs entirely on the GPU. In order to determine a collision between two objects, we first find the intersecting volume of the corresponding object-oriented bounding boxes. Next, patches of both objects are tested for inclusion within this volume. All possibly colliding patches are then voxelized into a uniform grid of single bit voxels. Finally, the resulting voxelization is used to detect collisions. Testing two moderately complex models containing thousands of patches can be done in less than a millisecond. This makes our approach ideally suited for real-time applications such as games that use the hardware tessellator.

1.3 Subdivision Surfaces

Subdivision surfaces [SZD*98] can be seen as a combination of tensor product patches [Far96], [HLS93] and polygonal meshes. In contrast to tensor product patches, they are defined on an arbitrary input topology but also define a smooth surface. Subdivision rules, that specify linear combinations of control points, are used to refine control points. Each step replaces a mesh given by control points and connectivity with a denser and smother mesh. The limit surface of a subdivision surface is defined after an infinite number of iterations. Subdivision rules can be written in matrix form; i.e., the *subdivision matrix*. A valid subdivision scheme must have a subdivision matrix with eigenvalues λ_i with the following property: $1 \geq |\lambda_0| \geq |\lambda_1| \geq \ldots \geq |\lambda_n|$. Otherwise the limit surface would be undefined since control points would either collapse or diverge.

There exists a variety of subdivision schemes such as Butterfly subdivision [DLG90], Loop subdivision [Loo87], Doo-Sabin subdivision [DS78], $\sqrt{3}$-subdivision [Kob00], $4-8$ subdivision [VZ01]. In this thesis we particularly focus on Catmull-Clark subdivision [CC78] due to its relevance to the movie industry. However, we would like to point out that our presented techniques can be also applied to any other subdivision scheme.

Catmull-Clark Subdivision Catmull-Clark subdivision is applied to an arbitrary input topology and provides a quad-only mesh after one refinement step. It generalizes bicubic B-splines since the limit surfaces is C^2 everywhere except on extraordinary points (vertices of valence different of 4) where it is C^1. The algorithm is defined by a simple set of rules, which are applied to a set of control points of a subdivision level i in order to generate the control points of the next subdivision level $i+1$. At each level new face points (f_j), edge points (e_j) and vertex points (v_j) are determined as a weighted average of points of the previous level. Special rules are used for handling features such as boundaries and creases.

The Catmull-Clark subdivision rules for face, edge, and vertex points with valence n, as labeled in Figure 1.1, are defined as:

- Faces rule: f^{i+1} is the centroid of the vertices surrounding the face.
- Edge rule: $e_j^{i+1} = \frac{1}{4}(v^i + e_j^i + f_{j-1}^{i+1} + f_j^{i+1})$,
- Vertex rule: $v^{i+1} = \frac{n-2}{n} v^i + \frac{1}{n^2} \sum_j e_j^i + \frac{1}{n^2} \sum_j f_j^{i+1}$.

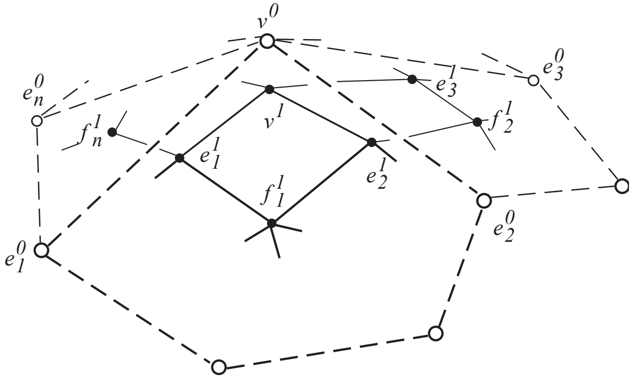


Figure 1.1: Labeling of vertices of the base mesh (subdivision level 0) around the vertex v^0 of valence n and the next subdivision level 1.

Catmull-Clark surfaces have been extended by additional subdivision rules in order to handle mesh boundaries [Nas87]. Boundary rules are also used to define infinitely sharp creases [HDD*94]; i.e., edges that interpolate the corresponding control points (crease edges). A vertex v_j containing exactly two crease edges e_j and e_k is considered to be a crease vertex. The following *sharp rules* are used for both boundaries and sharp edges (crease rules):

- $e_j^{i+1} = \frac{1}{2}(v^i + e_j^i)$
- $v_j^{i+1} = \frac{1}{8}(e_j^i + 6v^i + e_k^i)$

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

The combination of both smooth and sharp subdivision results in semi-sharp creases. Following DeRose et al. [DKT98], a semi-sharp crease (occasionally named semi-smooth in the literature) is defined by adding weighted sharpness tags to edges. Subdividing a semi-sharp edge creates two child edges, each of which are tagged with the sharpness value of the parent minus one.

In order to deal with fractional smoothness and propagate sharpness properly, we use a slightly modified scheme of DeRose et al. [DKT98]. The rules are as follows, with $e.s$ defining the sharpness of an edge:

- Face points are always the average of the surrounding points

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

The definition of the vertex sharpness $v.s$ is used to handle vertices, where $v.s$ is the average of all incident edge sharpness tags and k is the number of edges around a vertex v with $e.s > 0$:

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

Figure 1.2 shows the results when applying these rules to a pyramid. The edges of the pyramid's base plane are tagged as sharp, thus turning the quad at the bottom into a planar circular shape.

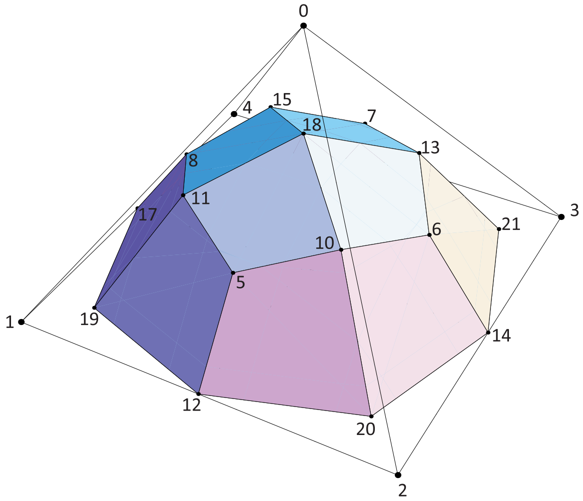


Figure 1.2: One subdivision step applied on a pyramid where the edges of the base plane are tagged sharp.

1.4 Graphics Processing Units

1.4.1 GPU Architectures

Modern graphics processers are massively parallel computation units. They are composed of several streaming multiprocessors (**SMs**) where each SM is a vector unit. Thus, SMs can process data chunks in parallel following the single-instruction-multiple-data (**SIMD**) principle. In this thesis we use an NVIDIA GTX 480 for testing which has 15 SMs and a SIMD width of 32, facilitating 480 threads to run in parallel [WKP11]. The Fermi architecture actually has 16 SMs, however, on the GTX 480 model only 15 are functional. Compared to CPUs, GPUs spend more dice area on compute rather than on caches, thus making memory I/O particularly costly. While there is a small amount of shared memory (64 KB on an NVIDIA GTX 480) available on each SM that can be used as L1 cache, most data must be obtained from global GPU memory. For instance, a SM issuing a global memory access introduction requires 4 clock cycles, however, in addition there is a memory latency of about 400 – 600 clock cycles [Nvi08]. Typically, latency is hidden by running a sufficient number of threads simultaneously; i.e., a multiple of the actual physical cores. Nevertheless, in order to achieve peak performance a sufficient amount of floating point operations per memory fetch is required [BFH04]. GPUs can be programmed for graphics purposes using the OpenGL [Shr10] or DirectX [Mic09] API (we use the latter). An overview of the corresponding graphics pipeline is shown in Section 1.4.2. In addition, modern GPUs support GPU computing using CUDA [Nvi08], OpenCL [M*09] or Direct Compute [Mic09]. A user defined amount of threads is executed in parallel whereas each thread follows a shared kernel description.

Previous to hardware tessellation, GPU computing techniques have been used to tessellate parametric surface geometry [SS09], [PEO09], [EL10], [EML09]. Some of our presented approaches also rely on GPU computing (i.e., DirectX 11 compute kernels) since data is directly processed on the GPU.

1.4.2 Graphics Pipeline and Hardware Tessellation

Figure 1.3 shows the graphics pipeline realized on current hardware, including the tessellation unit. The pipeline consists of several programmable shader stages and the configurable tessellation unit. Hardware tessellation, which our presented approaches particularly rely on, processes patch primitives in parallel. Each patch primitve is composed of a set of control points that define a corresponding patch surface. The tessellation unit uses the control points to generate a (potentially dense) polygonal surface geometry. The generated polygons are directly consumed by the rasterization units of the respective SMs without the need of further global memory access.

There are three hardware tessellation stages in the graphics pipeline that fit logically between vertex and geometry shading: hull shader, tessellation unit and domain shader. Together, these stages provide for hardware tessellation of 3- and 4-sided surface patches where a surface patch is a user

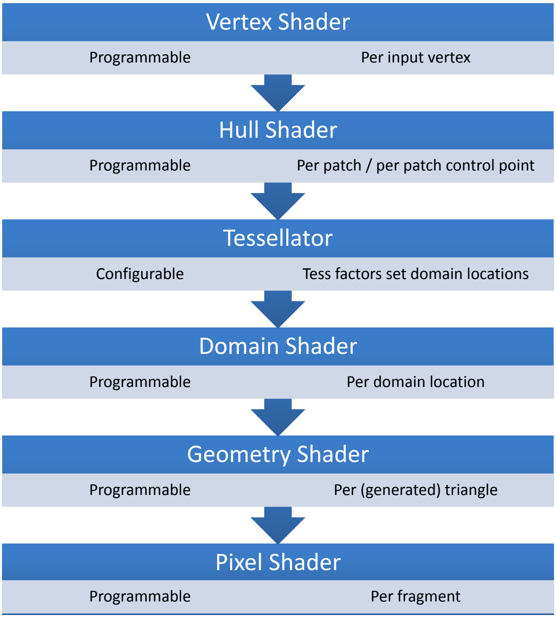
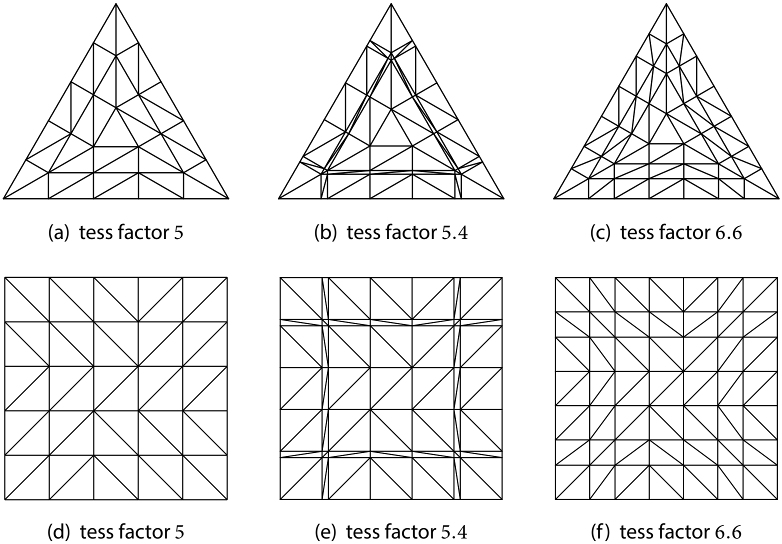


Figure 1.3: Graphics pipeline according to DirectX 11 nomenclature. For simplicity we omit the input assembly, rasterization and output merger stage which are fixed function stages.

specified collection of arbitrary dimensional input control points. The programmable hull shader stage executes both a user specified hull shader program and a hull shader constant function. The hull shader program maps the input control points to output control points by executing one thread per output vertex. There can be between 1 and 32, not necessarily equal in number, input and output control points. A hull shader program might, for example, implement a matrix-vector-multiply where each of n threads performs an m way dot product of the input control points and a row of an $m \times n$ matrix stored in constant memory. The purpose of the hull shader constant function is to determine the *tess factors* for each edge of the 3- or 4-sided patch domain as well as interior tess factors. The tess factors are used to vary the tessellation of a patch at runtime. The fixed function tessellator unit uses the tess factors to generate a tessellation of the patch domain. The tess factors do not need to be integer, and the maximum value on current hardware is 64. Non-integer, or fractional, tess factors allow the tessellator to continuously vary the tessellation rate without popping artifacts that might result from strictly integer tessellation levels. A user provided domain shader program takes as input the output control points from the hull shader and the 2D coordinates of a domain vertex generated by the tessellator unit and evaluates the patch at the corresponding domain location. In addition to position, the domain shader should output a surface normal, texture coordinate, and any other data needed for pixel shading. Since each domain vertex evaluation is independent of any other, domain shader execution is efficiently parallelizable. Figure 1.4 shows examples of different tessellation patterns (integer and fractional) created by the tessellation unit using both triangle (Figure 1.4(a-c)) and quad domains (Figure 1.4(d-f)).


(a) tess factor 5

(b) tess factor 5.4

(c) tess factor 6.6

(d) tess factor 5

(e) tess factor 5.4

(f) tess factor 6.6

Figure 1.4: Different tessellation patterns for integer and fractional tessellation factors on triangle (top) and quad domains (bottom).

PART I

Subdivision Surface Rendering

CHAPTER 2

Introduction

As discussed in Chapter 1 subdivision surfaces have been used extensively by the feature film industry for some time. In particular, Catmull-Clark subdivision surfaces [CC78] have become an industry standard. While we describe our surface rendering approaches by example of Catmull-Clark surfaces (see Chapters 4 and 5), our methods are also applicable to other subdivision schemes such as Loop subdivision [Loo87]. Since their introduction in 1978, Catmull-Clark surfaces have been extended in a number of ways, including the treatment of boundaries [Nas87], infinitely sharp creases [HDD*94], semi-sharp creases [DKT98], and hierarchically defined detail [Pix05]. The introduction of semi-sharp creases has proven to be particularly important as they allow realistic edges to be defined while keeping memory footprints small. For example, the rounded edges of the steel frame of the *Garbage Truck* shown in Figure 4.9 were created with semi-sharp creases, requiring only a few bytes of tag data per creased edge. Achieving a similar shape without semi-sharp creases would require a base mesh with significantly more vertices, faces, and edges. Similarly, the use of hierarchical edits, as introduced by Forsey and Bartels [FB88] and as defined in the RenderMan specification, allows detail at various resolutions to be specified much more efficiently than globally refining the mesh. An example is shown in Figure 4.10(b) where the sandy terrain is modeled at one scale, and the footprints are details that appear at a much finer scale. Previous to the publication of our approaches [NLMD12, NLG12] there was no GPU algorithm for the real-time rendering of Catmull-Clark surfaces possessing semi-sharp creases or hierarchical detail. Despite approximate approaches [LS08, MYP08, NYM*08, LSNC09], or the direct, but considerably slower Stam evaluation algorithm [Sta98] our approaches were also the first that allow patching of subdivision surfaces.

In this part of the thesis we focus on rendering subdivision surfaces with corresponding features; an elaborate displacement approach is proposed in Chapter 8. For rendering we use hardware tessellation (see Section 1.4.2), that exploits the compact representation and data independence of patch primitives to produce triangle data for immediate consumption by the rasterization stage. The challenge is to convert the Catmull-Clark control mesh into a set of patches that can be processed by the tessellation unit. For regular faces those are bicubic B-spline patches that are composed of 16 control points each. However, for irregular faces (faces that share a vertex with valence different from 4) the surface is only described by iterative subdivision rules (see Section 1.3) rather than a patching scheme. The same applies to features such as semi-sharp creases or hierarchical edits. An advantage of using subdivision surfaces is that only the control points need to be updated for animation. The GPU can amplify the coarse geometry on-the-fly with very little memory bandwidth to produce a dense tessellation of the surface.

To sum up, we propose two approaches for subdivision surface rendering. The first is more general and renders the Catmull-Clark surface by applying adaptive subdivision at features (see Chapter 4, [NLMD12]). The second approach extends the first by focusing on meshes with semi-sharp creases (see Chapter 5, [NLG12]).

CHAPTER 3

Previous Work

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

Global mesh refinement:
Global mesh refinement implements subdivision according to its standard definition; i.e., the subdivision rules. A base mesh is repeatedly (possibly adaptively) refined until the mesh achieves a sufficient density and then the resulting faces are rendered. The work of Bunnell [Bun05], Shiue et al. [SJP05], and Patney and Owens [PEO09] belong to this category. Global refinement schemes require significant memory I/O to stream control mesh data to and from GPU multiprocessors and global GPU memory (i.e., on- and off-chip). We demonstrate in Section 4.8 that performing global refinement iteratively is severely limited in performance as memory bandwidth becomes the bottleneck.

Direct evaluation methods:
By leveraging the eigenstructure of a subdivision matrix Stam [Sta98] developed a method for directly evaluating subdivision surfaces at arbitrary parametric values. In order to apply this approach, a control mesh must have *isolated* extraordinary vertices. This means that two global refinement steps must be applied to an arbitrary control mesh (only one for a quad mesh) as a preprocess. Afterwards, Stam's algorithm can be easily implemented on the GPU using hardware tessellation. Therefore, patch data must be transformed into eigenspace, making subsequent watertight boundary evaluation problematic. To do patch evaluation, one of three possible sets of precomputed eigenbasis functions must be evaluated. While this is feasi-

ble, the complexity of the algorithm and significant amount of computation limit performance (see Section 4.8). Furthermore, extending this approach to the evaluation of semi-sharp creases and hierarchical details remains a formidable challenge. Our approach presented in Chapter 5 is also based on the eigenvalue decomposition of the subdivision matrix. However, in contrast to Stam's approach we consider patches with semi-sharp creases.

Another direct evaluation approach has been proposed by Bolz and Schröder originally targeting the CPU [BS02]. Their approach exploits the fact that subdivision surfaces are linear functions of control point positions, meaning that a basis function can be associated with each control point. These basis functions must be generated for all patch configurations. In order to keep basis function count within limits, they require extraordinary vertices to be isolated. They indicate that approximately 5300 tables for the basis functions are required restricting vertex valence for interior patches to 12. Evaluation of surface points and derivatives is therefore reduced to dot products of the table values with control point positions. Extending their method to accommodate semi-sharp creases and hierarchical detail is problematic for several reasons. First, since crease sharpnesses can take on fractional values, there are an unbounded number of potential basis functions, meaning that the tables must be precomputed in a mesh-dependent fashion, and the number of distinct tables can be very large. More fundamental is the fact that hierarchical detail coefficients are represented in local surface frames (see Forsey and Bartels [FB88]), implying that the surface is no longer linear in the control vertices, and hence basis functions do not exist.

Approximation methods:
Due to the limitations of global refinement and current direct evaluation methods, faster but approximate methods have been proposed. In anticipation of hardware tessellation, Loop and Schaefer [LS08] noted the high cost of direct evaluation and the need for pre-tessellator subdivision. They proposed an approximation to a quads only Catmull-Clark limit surface based on bicubic Bézier patches. Several variants along these lines have appeared with various improvements to the restrictions on mesh connectivity or underlying surface algorithm. For instance, a quads only method was de-

scribed by Myles et al. [MYP08] and Ni et al. [NYM*08], a method to handle a mixture of triangles and quads was presented by Loop et al. [LSNC09], and Myles et al. [MNP08] offer a method to deal with pentagonal patches as well as quads and triangles. Finally, the case of infinitely sharp creases in the context of an approximate Catmull-Clark subdivision on the GPU was handled by Kovacs et al. [KMDZ09].

It has been understood for decades that adaptive tessellation requires a specific strategy for eliminating cracks [Cat74] at boundaries between distinct tessellation densities. Since a goal of our work is to create adaptive but crack-free renderings, our method is similar to that of Von Herzen and Barr [VHB87], where the notion of restricted quadtrees was introduced. A quadtree is said to be restricted if the subdivision levels of adjacent cells differ at most by one. With this restriction it is relatively easy to avoid cracks. Fortunately, the pattern of subdivision that naturally arises in our algorithm presented in Chapter 4 ensures this condition.

CHAPTER 4

Feature Adaptive GPU Rendering of Subdivision Surfaces

4.1 Introduction and Algorithm Overview

In this chapter we present an GPU approach for the real-time rendering of Catmull-Clark surfaces that also processes features such as semi-sharp creases or hierarchical detail. It uses a combination of GPU compute kernels and hardware tessellation to adaptively patch Catmull-Clark surfaces. The algorithm is exact rather than approximating (by exact we mean that the vertices of our patch tessellations lie exactly on the limit surface up to machine precision), and we present a variant that is capable of creating watertight tessellations. This variant additionally satisfies a much stronger condition: namely, that abutting patches meet with *bitwise identical* positions and normals. With this stronger property surfaces with normal displacements are also guaranteed to be crack-free. That becomes particularly important for our displacement mapping approach depicted in Chapter 8.

The algorithm presented in this chapter is based on the idea of feature-adaptive subdivision. It has long been known that regular faces of a Catmull-Clark base mesh (that is, quad faces whose vertices have exactly four neighbors) generate a single bicubic patch in the limit of infinite subdivision, and that regions around extraordinary vertices (vertices having other than four neighbors) give rise to an infinite nesting of bicubic patches that approach a well-defined limit [DS78]. A similar recursive nesting of bicubic patches also occurs near other kinds of features such as semi- or infinitely sharp creases, as well as near regions affected by hierarchical detail. Our algorithm exploits this fact to subdivide the mesh only in the vicinity of these

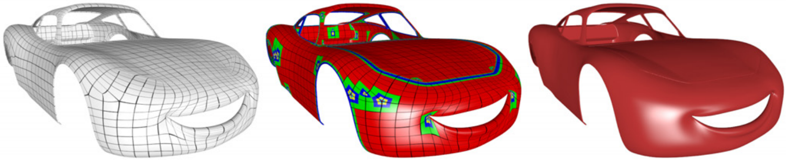


Figure 4.1: Input base mesh (left), subdivision patch structure (center), and final model rendered with our method (right). © Disney/Pixar

features. At each stage of local subdivision, new bicubic patches are generated that are directly rendered using hardware tessellation. Since we only subdivide locally, our time and memory requirements are significantly less than the naive approach of globally subdividing the entire base mesh each step.

In addition to being more accurate and general than previous algorithms, the performance of our approach is competitive with an optimized implementation of the fastest approximate scheme by Loop et al. [LSNC09] that is based on *Gregory Patches* [Gre74]. We also show that our approach is significantly faster than a GPU implementation of Stam's direct evaluation procedure (see Section 4.8). Furthermore, we also demonstrate that iteratively refining a mesh is inherently memory I/O bound, while our algorithm utilizes hardware tessellation in order to avoid this limitation.

Feature adaptive rendering involves a CPU preprocessing step, as well as a GPU runtime component. Input to our algorithm is a base control mesh consisting of vertices and faces, along with optional data consisting of semi-sharp crease edge tags and hierarchical details. In the CPU preprocessing stage, we use these data to construct tables containing control mesh indices that drive our feature adaptive subdivision process. Since these subdivision tables implicitly encode mesh connectivity, no auxiliary data structures are needed for this purpose. A unique table is constructed for each level of subdivision up to a prescribed maximum, as well as final patch control point index buffers as described in Section 4.2. The base mesh, subdivision tables, and patch index data are uploaded to the GPU, one time only, for subsequent runtime processing. The output of this phase depends only on the

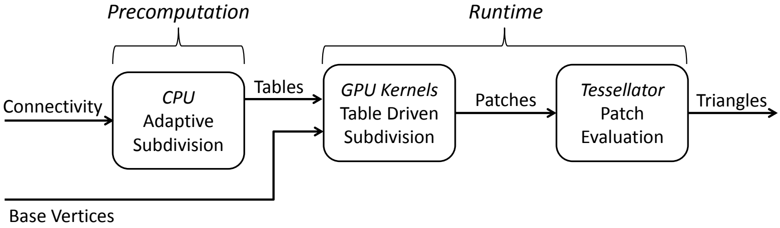


Figure 4.2: Flowchart of our feature adaptive subdivision rendering approach.

topology of the base mesh, crease edges, and hierarchical detail; i.e., it is independent of the geometric location of the control points.

For each frame rendered on the GPU, we use a two phase process. In the first phase, we execute a series of GPU compute kernels to perform data parallel Catmull-Clark subdivision of the base mesh. The index patterns used by the compute kernels to gather vertex data needed to perform each subdivision operation are entirely encoded in the subdivision tables. This process is repeated for each level of subdivision until a maximum depth is reached; this phase is described in Section 4.3. In the second GPU phase, we use the hardware tessellator unit to render the bicubic patches corresponding to the regular regions of the various subdivision levels computed in the first GPU phase (see Section 4.4). In Section 4.5, we develop the special treatment necessary to avoid cracks between adjacent patches, and in Section 4.7 we discuss how the flexibility of our algorithmic framework can be used to implement view dependent level-of-detail rendering. Since only base control point positions need to be updated on the GPU each frame, high-frame-rate animation of complex models is achieved. We address only the rendering of models once they have been animated. We do not address techniques for creating compelling animation. An overview of our approach is shown in Figure 4.2.

In summary, the main contributions of our approach are:

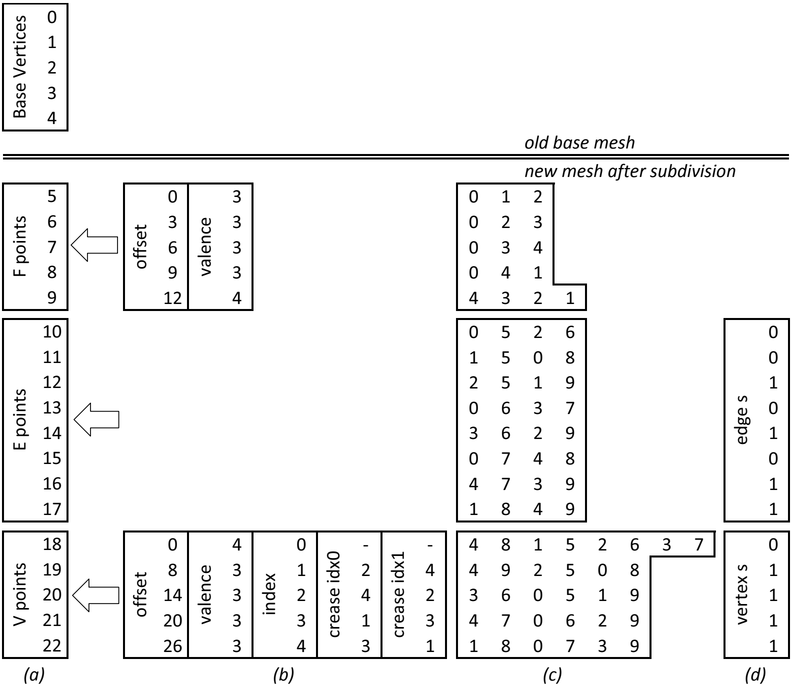
- The first table driven data-parallel subdivision method that supports local refinement; we refer to this as *feature adaptive subdivision*.

- The first exact hardware tessellation algorithm of the true Catmull-Clark limit surface, including arbitrary valence vertices and faces, semi-sharp creases, and hierarchical details.
- A novel evaluation algorithm that guarantees bitwise identical evaluation of positions and normals of adjacent patches.
- A view dependent level-of-detail method that adapts to both subdivision level and patch tessellation density.

4.2 Table Driven Subdivision on the GPU

While the rules of Catmull-Clark subdivision (see Section 1.3) are straightforward to implement on a CPU, an efficient implementation on the GPU is non-trivial since neighborhood information is required. Fortunately, in most feature film and game applications the connectivity and sharpness tags of a mesh are typically invariant during animation, so a precomputed table driven approach is feasible. These tables are used at runtime to efficiently guide the subdivision computations. Due to the data dependency of the Catmull-Clark subdivision rules, face points must be computed first, followed by edge, and then vertex points. We use three separate compute kernels, one for each point type respectively. All kernels operate on a single vertex buffer, used for all subdivision levels. We justify this strategy as it a) simplifies our table construction, and b) optimizing this will have little impact on frame rate. The vertices of the base mesh, which may be animated at runtime, occupy a section at the beginning of this buffer. Starting from the base mesh, the subdivision kernels will compute in parallel the refined mesh for the next subdivision level.

The tables for the face, edge and vertex kernel are defined as follows. The face kernel requires two buffers: one index buffer, whose entries are the vertex buffer indices for each vertex of the face; a second buffer stores the valence of the face along with an offset into the index buffer for the first vertex of each face. Since a single (non-boundary) edge always has two incident faces and vertices, the edge kernel needs a buffer for the indices of these entities. In order to apply the edge rules, we also store the edge sharpness values *e.s*. The data for the vertex kernel is similar to the face kernel. We use an index buffer containing the indices of the incident edge and vertex points. We also need a second buffer to store the vertex valence, an index to predecessor of the vertex, the vertex sharpness *v.s*, and an offset to the starting index in the first buffer. For dealing with the case of a vertex on a crease, we must also store the indices of the edges that specify the crease (*crease idx0*, *crease idx1*). Figure 4.3 shows the subdivision tables for the pyramid in Figure 1.2. For meshes with boundary, the subdivision tables are adjusted according to the respective rules.



Base Vertices
0
1
2
3
4

old base mesh

new mesh after subdivision

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

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

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

Figure 4.3: Subdivision tables for the pyramid of Figure 1.2: (a) is the vertex buffer, (b) contains topology information, (c) are indices which point into the vertex buffer and (d) provides the edge and vertex sharpness.

4.3 Feature Adaptive Subdivision

It is well known that the limit surface defined by Catmull-Clark subdivision can be described by a collection of bicubic B-spline patches, where the set has infinitely many patches around extraordinary vertices, as illustrated in Figure 4.4(left). Similarly, near creases as shown in Figure 4.4(right), the number of limit patches grows as the crease sharpness increases.

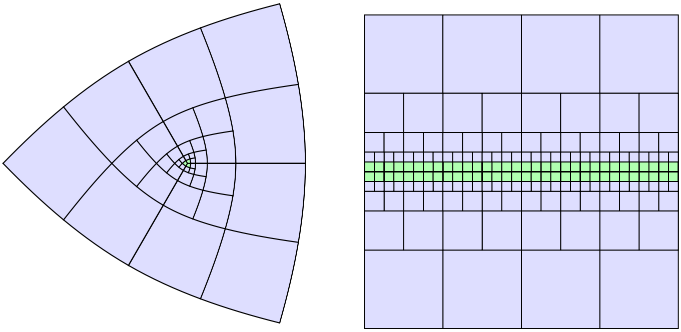


Figure 4.4: The arrangement of bicubic patches (blue) around an extraordinary vertex (left), and near an infinitely sharp crease (right). Patches next to the respective feature (green) are irregular.

Feature adaptive subdivision proceeds by identifying regular faces at each stage of subdivision, rendering each of these directly as bicubic B-splines using hardware tessellation (see Section 4.4). Irregular faces are refined, and the process repeats at the next finer level. This strategy uses the same compute kernels as outlined in Section 4.2, however, the subdivision table creation is restricted to irregular faces. A face is regular only if it is a quad with all regular vertices, if none of its edges or vertices are tagged as sharp, and there are no hierarchical edits that would influence the shape of the limit patch. In all other cases the face is recognized as irregular, and subdivision tables are generated for a minimal number of subfaces. As before, all of this analysis and table generation is done on the CPU at preprocessing time.

Vertex and edge tagging is done at each level, depending on how many times the area around an irregular face should be subdivided. This might be the maximum desired subdivision depth around an extraordinary vertex, or the sharpness of a semi-sharp edge. As a result, each subdivision level will be a sequence of local control meshes that converge to the feature of interest (see Figure 4.5).

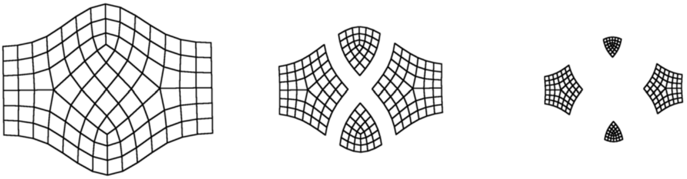


Figure 4.5: Our adaptive subdivision scheme applied on a grid with four extraordinary vertices. Subdivision is only performed in areas next to extraordinary vertices.

The memory required to globally subdivide a mesh to level k is proportional to $4^k F$, where F is the number of faces in the original mesh. Feature adaptive subdivision generally requires far less memory as the size of the subdivision tables is proportional to the total number of irregular faces at each subdivision level. The exact storage requirements depend on the number and arrangement of irregular faces, the sharpness of creases, and so on. However, an asymptotic upper bound can be obtained by making the worst case assumption that every irregular edge (an edge is irregular if it is adjacent to an extraordinary vertex, if it is tagged as a crease, or if a hierarchical edit influences the shape of one of the patches adjacent to the edge) is subdivided into two irregular edges. If there are e edge tags in the original mesh, the storage requirements for subdividing k levels is proportional to $2^k e$. Since e is typically far smaller than the total number of edges in the original mesh, and since 2^k grows far less quickly than 4^k, we achieve a significant reduction in memory use. Subdivision around extraordinary vertices behaves even better, since the number of irregular faces grows linearly with respect to the extraordinary vertex count (v) per subdivision step ($\approx 12kv$). Actual memory requirements for various models are given in Section 4.8.4. Also

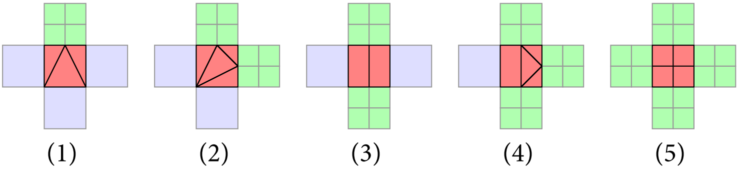


Figure 4.6: There are five possible constellations for *transition patches* (**TPs**). While TPs are colored red, the current level of subdivision is colored blue and the next level is green. The domain split of a TP into several subpatches allows full tessellation control on all edges, since shared edges always have the same length.

note that our approach presented in Chapter 5 further reduces the memory requirement for models with semi-sharp creases.

4.4 Patch Construction

Once the adaptive subdivision stage is complete (see Section 4.3), we use the hardware tessellator to adaptively triangulate the resulting patches (see Section 1.4.2). The number and location of sample points on patch edges is determined by user provided *tess factors*. For each subdivision level we define two kinds of patches: full patches and transition patches.

4.4.1 Full Patches

Full patches (**FPs**) are patches that only share edges with patches of the same subdivision level; those are either regular or irregular. Regular FPs are passed through the hardware tessellation pipeline and rendered as bicubic B-splines. We ensure by feature adaptive subdivision that irregular FPs are only evaluated at patch corners. This means that for a given *tessfactor* we must perform $\lceil \log_2 \mathit{tessfactor} \rceil$ adaptive subdivision steps. Since current hardware supports a maximum *tessfactor* of 64 ($= 2^6$), no more than 6 adaptive subdivision levels are required. In order to obtain the limit positions and tangents of patch corners of irregular FPs, we use a special vertex shader that applies the corresponding Catmull-Clark limit stencils (the stenscils are provided in [HKD93], [LS08]). Using this approach, our surface representation is exact; we show in Section 4.8 that this is significantly faster than direct evaluation as proposed by Stam [Sta98].

4.4.2 Transition Patches

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

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

In order to avoid this limitation, we split each TP into several subpatches using a case analysis of the arrangement of the adjacent patches. Since each patch boundary either belongs to the current or to the next subdivision level, there are only 5 distinct cases as shown in Figure 4.6.

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

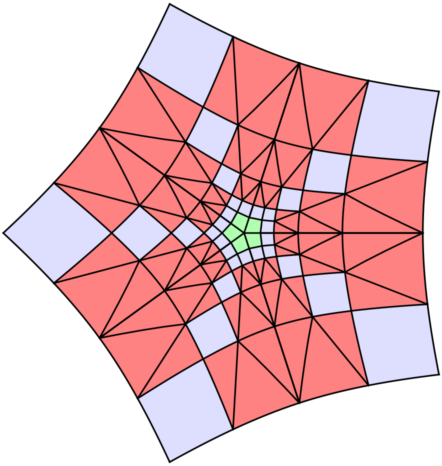


Figure 4.7: The structure of patches around an extraordinary vertex; red areas are transition patch domains, blue are regular Full Patches, green (central) indicated irregular patch domains - note that by evaluating only at the corners of these domains our scheme becomes exact.

By rendering TPs as several logically separate patches, we eliminate all T-junctions in the patch structure of a surface. The TP structure around an extraordinary vertex is illustrated in Figure 4.7. This means that as long we assign consistent tess factors to shared edges, in principle a crack-free rendered surface is obtained. In practice however, due to behavior of floating point numerics, additional care is required as discussed in Section 4.5.

The assignment of tess factors for patch edges should take into account the subdivision level *i* that generated the patch. We discuss various strategies for adapting the subdivision level and the assignment of tess factors in Section 4.7.

4.5 Watertight Evaluation

Adjacent patches sharing identical tess factors is a necessary but not sufficient condition to guarantee watertight rendering. Since floating point multiplication is neither associative nor distributive, evaluating adjacent patches at the same parameter of a shared boundary may not produce bitwise identical results. These minor discrepancies may result in visible cracks (holes) between adjacent patches. In particular, inconsistent normals at patch boundaries become an issue when applying surface displacements (see Chapter 8).

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

4.5.1 Same Subdivision Level

Castaño et al. [Cn08] propose an approach for watertight evaluation of Bézier patches that matches the order of computations performed with respect to either side of a shared boundary. However, they do not address the conversion to the Bézier basis, required for subdivision surfaces; or how to deal with irregular patches. Furthermore, their use of ownership assignments for normals at patch corners has a large memory footprint.

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

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

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

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

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

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

$$C(u) = C^r(1-u) \qquad \text{and} \qquad \frac{dC}{du}(u) = -\frac{dC^r}{du}(1-u) \tag{4.1}$$

where equality means bitwise identical results.

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

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

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

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

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

The final step is to perform the dot product of X and $N(u)$ to guarantee Equation 4.1. We do so as follows:

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

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

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

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

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

For irregular patches, we handle watertightness using a special vertex shader that computes the limit surface (see Section 4.4.1). By definition there are only corner points. Points that are adjacent to regular patches (those have a valence of 4) are evaluated bi-reversal invariantly as described above. The remaining points are all extraordinary vertices, having only irregular patches in common. Since these are evaluated on a per vertex level, the results are shared so no special treatment is required.

4.5.2 Between Subdivision Levels

Special care must be taken for patches, generated by different subdivision levels, that also share a boundary. As before, evaluations on either side of a boundary must use the same input control point data. In order to ensure this, we define patches that share evaluations with a coarser subdivision level as *watertight critical patches* (WCP). Note that a WCP can be either a FP or a TP. Referring to Figure 4.6, for a WCP that is also a TP it must be case 1 (domain boundary) or case 2 (domain corner). In this situation, we augment the 16 control points for the WCP with the 16 control points of its coarser level parent patch. These 32 total control points can still be handled efficiently by the tessellation unit. In the domain shader, if a point is on a boundary between the current and the previous subdivision level, the control points of the parent patch are used for computations. The evaluation itself is done as proposed in Section 4.5.1.

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

Our approach to watertight evaluation necessarily involves code branches that treat domain boundaries and corners differently than domain interiors. Depending on tessellation densities, this can result in a slowdown of as much as 2*x* (see Section 4.8), due to the SIMD nature of GPU code execution. In our implementations, we treat watertight rendering as an optional time versus image quality trade-off that may not be necessary for all applications, e.g., authoring versus game runtime.

4.6 Examples

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

4.6.1 Extraordinary Vertices

The most common reason to use subdivision surfaces instead of bicubic B-splines is to deal with arbitrary topology; i.e., extraordinary vertices (vertices with a valence different of 4). Like previous algorithms, our method is capable of rendering meshes containing extraordinary vertices, as shown in Figure 4.11, where different colors denote different levels of adaptive subdivision. Notice how subdivision is only used in the neighborhood of extraordinary vertices, whereas regular regions are directly tessellated using bicubic B-splines.

Subdivision around extraordinary vertices reduces the area and thus the number of evaluations needed within irregular patches. By dividing the tess factor by two after each subdivision level (see Section 4.4.2), we enforce that after $\lceil \log_2 \mathit{tessfactor} \rceil$ subdivision levels the tess factor will become 1.0. Using adaptive subdivision allows us to reduce the evaluations within irregular patches until only the corners of the patch domain need to be evaluated. The limit surface positions and normals at domain corners are computed with a special vertex shader that implements limit masks as described in Halstead et al. [HKD93]. This way we achieve exact evaluation at all tessellation points, including regions around extraordinary vertices.

4.6.2 Semi-Sharp Creases

The generalization of Catmull-Clark surfaces to capture creases, both infinitely sharp and semi-sharp, has proven to be extremely useful in real-world applications such as geometric modeling [HDD*94], feature film production [DKT98], and video games [KMDZ09].

Previous algorithms have been capable of accurately rendering infinitely sharp creases, but feature adaptive subdivision is the first that is capable of interactively rendering semi-sharp creases. A simple example of a model using semi-sharp creasing is shown in Figure 4.8, where different colors denote different levels of adaptive subdivision. Note that Figure 4.8(c) uses a fractional sharpness crease. Sharpnesses and fractional values are entirely encoded in precomputed subdivision tables. At runtime the tables are used to fill the vertex buffer with vertex positions for each adaptively subdivided patch at each level of subdivision. These patches are then sent to the tessellation unit for evaluation.

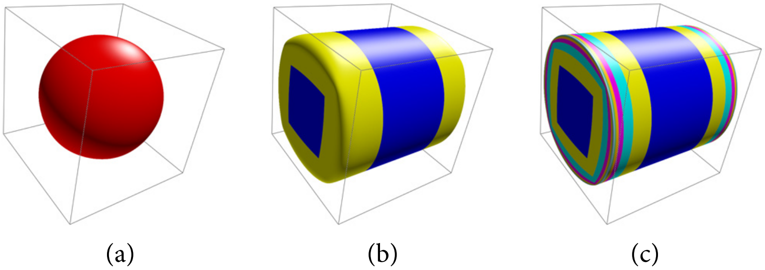


Figure 4.8: All images are derived from a cube used as the base mesh. (a) no edges tagged; (b) front and back-face edges have a sharpness of 3; (c) front and back-face edges have a fractional sharpness of 7.8. Each color represents a distinct level of subdivision.

A more realistic example of the use of semi-sharp creases is the *Garbage Truck* shown in Figure 4.9, which appeared in a recent feature film. The Catmull-Clark base mesh consists of 5448 patches, and 3439 creased edges, with sharpnesses ranging from 1.0 to 3.0, including some fractional sharpnesses of 1.5 and 2.3. Achieving the tight radii of curvature without semi-sharp creases would significantly increase the memory footprint because the base mesh would need to be significantly more dense in most regions. Another example of semi-sharp creasing is the *Car Body* shown in Figure 4.1. Semi-sharp creases were particularly helpful near the boundary of the hood to achieve a tight radius of curvature there.

While feature adaptive subdivision handles Catmull-Clark subdivision features generically, we present an approach in Chapter 5 that particularly focuses rendering surfaces with semi-sharp creases.

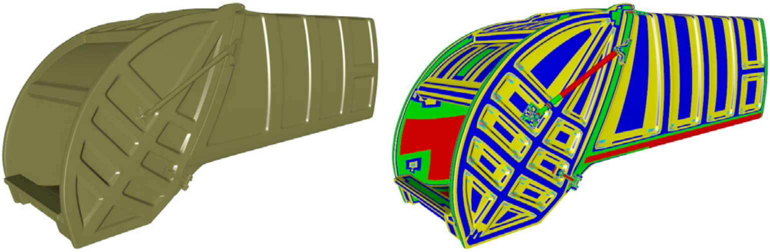


Figure 4.9: Garbage Truck with semi-sharp creases. Adaptive subdivision levels indicated by color (right). © Disney/Pixar

4.6.3 Hierarchical Detail

Multi-resolution modeling has received considerable attention in the graphics community, and can be traced back at least to the early work of Forsey and Bartels [FB88]. When applied to subdivision surfaces, the idea is to represent a shape using a relatively sparse base mesh that captures the coarse features of the shape, together with hierarchical edits that are applied during subdivision to describe variation at finer scales. The scheme used by RenderMan is typical for these methods [Pix05]. Each vertex generated through subdivision is assigned a unique index. A hierarchical edit consists of such an index together with a vector displacement. When the vertex with that index is created during subdivision, its position is offset by the vector displacement prior to subsequent subdivision. An example of this process is shown in Figure 4.10(b), where the base mesh captures the overall shape of the terrain, while the hierarchical edits are used to describe the footprints which occur at much finer scales.

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

Similar to hierarchical edits, T-splines [SCF*04] enable the representation of multiple resolutions of mesh data within a compact framework. While we have not validated our algorithm on T-spline meshes, we expect the feature adaptive principles underlying our work to be complementary to the T-spline paradigm.

4.6.4 Displacement Mapping

Displacement mapping involves offsetting a surface point in the normal direction by a scalar value stored in a texture map. In the context of hardware tessellation, it is crucial that both the position and normal of a surface on shared patch boundaries are bitwise identical; otherwise, cracks may appear in surface. Fortunately, a variant of our algorithm (see Section 4.5) guarantees this, as illustrated in Figure 4.10(a). Here we have used the traditional combination of a scalar displacement texture and a object space normal map. In Chapter 8 we present a more elaborate displacement mapping technique that gets along without a normal map.

4.7 Adaptive Level of Detail

The flexibility of our algorithmic framework can be used in a variety of ways to achieve adaptive level-of-detail (**LOD**) control. There are two independent factors that can be used for LOD control for our rendering technique: tess factor assignment and the depth at which subdivision is terminated. We examine each of these separately, then make specific recommendations for two important applications; namely, the rendering of characters and terrain. Please note that we also present a level-of-detail technique in the context of displacement mapping (see Chapter 8) that builds on the ideas shown here.

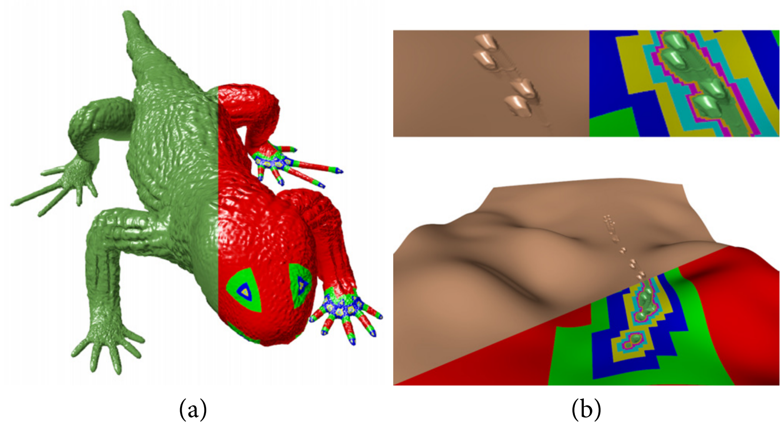


Figure 4.10: Displacements (a) and hierarchical edits (b). Subdivision levels are visualized by distinct colors. © Disney/Pixar

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

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

Since we have eliminated T-junctions from the patch structure of a model, we are free to assign these tess factors arbitrarily to shared patch edges (see Section 4.4). Tess factor assignment can either be done locally on a per edge basis requiring tess factor computations in the hull shader, or globally for an entire mesh. Local tess factors can be assigned according to some local edge based metric. Doing this based on screen space length, which approaches zero near silhouettes, may cause artifacts. More sophisticated approaches could avoid this, but at higher cost. For our applications we have achieved good results by simply using the viewing distance to edge midpoints. Alternatively, we can assign tess factors globally so that all patch edges of an object get the same value. The global tess factor could for example, be computed based on the distance from the camera to the centroid of the object. Such a global tess factor assignment would result in patches from higher levels of subdivision to appear more densely tessellated than those from lower subdivision levels. To avoid this, we assign the global tess factor to the zeroth subdivision level, halving it for each level of subdivision, resulting in a more uniform tessellation density. This strategy has proven to be effective for objects with small spatial extent, such as characters.

Adaptive subdivision level:
Additional LOD management is obtained by terminating feature adaptive subdivision after an adaptively determined maximum level. This level could be based on an object's camera distance, similar to global assignment of a tess factor. This approach may result in irregular patches having a tess factor greater than 1.0, which means that a surface approximation such as Loop et al. [LSNC09] would be required. Nevertheless, this might be reasonable for models containing very sharp creases. Such a scheme would be particularly effective for objects with a large spatial extent, such as terrains.

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

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

Adaptive local tess factors with adaptive subdivision level:
The maximum tess factor that could possibly be assigned to any edge (based

on the viewing distance) of a certain object is computed taking the respective bounding geometry into account. Hence, the most distant point on the bounding geometry determines the maximum subdivision level. As a result it is possible to locally assign tess factors on a per edge basis in the hull shader. If extraordinary vertices have been isolated, then irregular patches will never receive a tess factor greater than 1.0.

4.8 Results

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

4.8.1 Comparison to Global Mesh Refinement

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

Table 4.1: Timing using the *Big Guy* model for our scheme (feature adaptive patching) compared against our global table driven subdivision method and the previously published GPU subdivision algorithm by Shiue et al. [SJP05]. Note that all timings include final rendering, while we additionally break out draw time for our global subdivision scheme.

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

To demonstrate this point, we compare our scheme to two global refinement algorithms implemented on the GPU. The first utilized our table driven subdivision approach (see Section 4.2) to globally refine a mesh for each subdivision level. The second algorithm is a modern GPU implementation of the subdivision kernel proposed by Shiue et al. [SJP05]. Shiue's algorithm requires extraordinary vertices to be isolated (that is, no edge can be adjacent to two extraordinary vertices). We achieve this by statically pre-subdividing the mesh on the CPU. Note that neither our feature adaptive patching scheme nor our table driven subdivision has this limitation.

As shown in Table 4.1 our feature adaptive patching scheme outperforms global subdivision for all but the first subdivision levels. This can be easily explained because for the first subdivision levels little refinement is required; while our feature adaptive patching must always setup patches even if evaluations are only done at patch corners. However, beyond the first subdivision levels feature adaptive patching pays off since hardware tessellation requires less memory I/O. As the subdivision level increases this difference quickly becomes large. For subdivisions levels beyond level 4 we cannot do the comparison since global subdivision runs out of memory.

Furthermore, our feature adaptive patching scheme is faster than Shiue's algorithm for all subdivision levels and our table driven subdivision outperforms Shiue's algorithm in all cases (see Table 4.1). Our implementation of Shiue differs slightly from the original approach in that we use the compute shader in order to launch threads instead of the pixel shader.

This change was made to provide more freedom to optimize thread allocation in order to achieve best performance for Shiue's algorithm on modern GPUs. Due to its fundamental design their depth-first approach requires both a significant number of compute kernel invocations and draw calls. Redundant computations within the subdivision kernel and between distinct kernel calls further harm performance. This may have been a reasonable design when launching blocks of threads had to be done by the graphics pipeline using pixel shaders. In contrast to this depth-first approach, today's more generally programmable GPUs prefer breadth-first algorithms such as our table driven subdivision. Nevertheless, all GPU subdivision algorithms contain a significant amount of memory I/O due to their iterative nature and independent rendering of the resulting triangles. Our patching scheme, however, utilizes the tessellation unit which minimizes memory I/O; once patches are set up, patch evaluations and final rendering is done on-chip without additional memory transfer. This behavior is clearly demonstrated in Table 4.1 where the draw time alone for global subdivision is already larger than our entire feature adaptive subdivision scheme (including all kernel launches and patch renderings) beyond subdivision level 2.

4.8.2 Comparison to Direct Evaluation and Approximate Patching Algorithms

We have shown in the previous section that using hardware tessellation allows rendering with a minimum of memory I/O, making it faster than iterative mesh refinement. We now compare our feature adaptive algorithm and its watertight variant (see Section 4.5) against other patching schemes. Therefore, we consider Stam's direct evaluation algorithm [Sta98] and the approximate Gregory patching scheme proposed by Loop et al. [LSNC09]. In the following comparison we first perform one level of global subdivision for all algorithms, since Stam evaluation requires isolated extraordinary vertices (neither approximate Gregory patching nor our approach has this limitation). We use the *Big Guy* and *Monster Frog* models since they do not possess semi-sharp creases or hierarchical detail (only our approach can handle those features). The respective images including the patch structure is illustrated in Figure 4.11 while the timing results are shown in Figure 4.12.

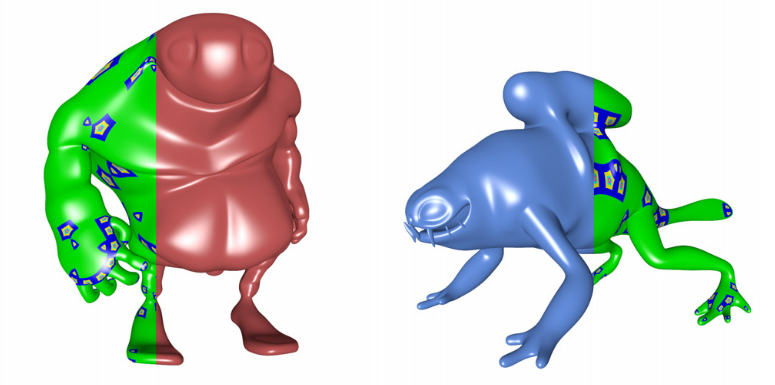


Figure 4.11: Exact evaluation of subdivision surfaces using our adaptive scheme applied on the Big Guy (left) and *Monster Frog* (right) model. Respective timings are shown in Figure 4.12.

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

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

Our watertight evaluation method produces bit-wise identical results, but

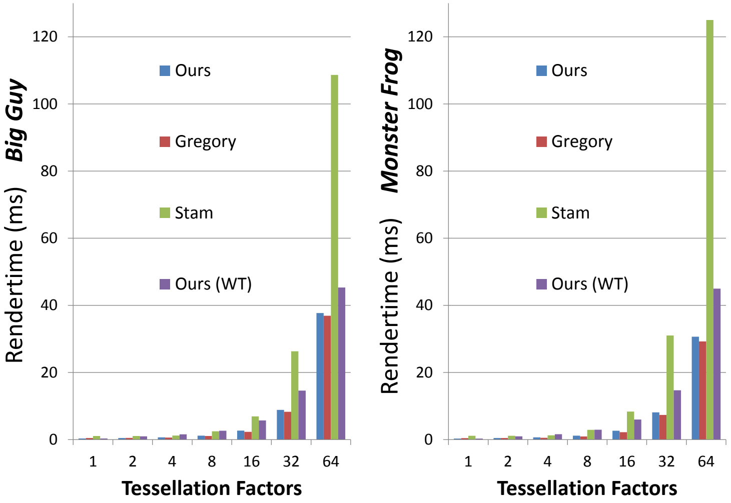


Figure 4.12: Comparison of our method (w/ and w/o watertightness) against Stam evaluation (both exact) and the approximate Gregory scheme using different tess factors applied to the *Big Guy* (top) and *Monster Frog* (bottom) models.

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

4.8.3 Semi-Sharp Creases and Hierarchical Edits

Models containing semi-sharp creases cannot be handled by previous algorithms. Performance measurements of our algorithm for such models (the *Car Body* and the *Garbage Truck*) are given in Table 4.2. Note that even for high tess factors real-time frame rates are achieved. Also note that employing the tessellator allows us to generate and render nearly one billion triangles per second on our test hardware.

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

Table 4.2: Performance of our method on the *Car Body* and *Garbage Truck* models as a function of tess factor (TF).

Our method is also the first capable of interactively rendering models containing hierarchical detail. An example is shown in Figure 4.10(b) which depicts a sandy terrain consisting of very few faces in the base mesh (a 12×19 grid), together with fine scale footprints that are modeled using hierarchical detail. Note that footprints can be easily animated since hierarchical detail can be updated at runtime.

4.8.4 Memory Requirements

As mentioned in Section 4.3, memory on the GPU needs to be allocated to store subdivision tables for each feature adaptive patch, and the vertex buffer needs to be large enough to store patch vertices at all levels of subdivision. The exact memory requirements for various levels of subdivision for particular models are shown in Figure 4.13. Modern GPUs are typically equipped with a gigabyte or more of buffer memory, so the memory requirements of our algorithm make it possible to simultaneously handle many such models.

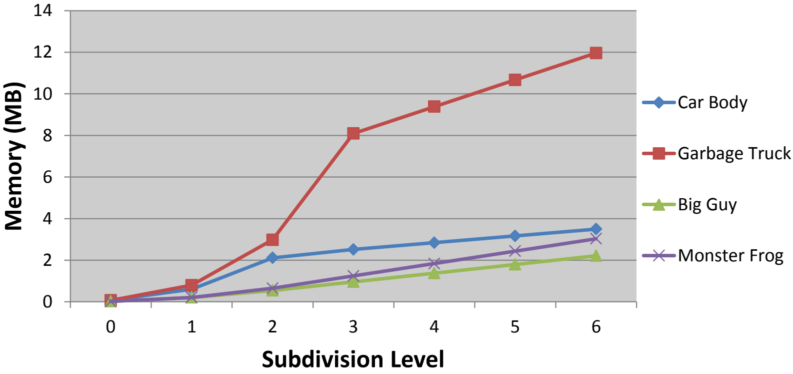


Figure 4.13: Memory requirements to store vertex buffers and subdivision tables as a function of the maximum subdivision level.

4.9 Conclusion

We have presented a novel method of GPU based rendering of arbitrary Catmull-Clark surfaces. In contrast to previous algorithms, our method is exact and implements the full RenderMan [Pix05] specification of Catmull-Clark surfaces, including arbitrary base mesh topology, semi-sharp creases, and hierarchically defined detail. We also presented a variant of the algorithm that produces watertight positions and normals, allowing for the crack-free rendering of displaced surfaces. We demonstrated the method on feature film quality models, and showed that even for such complexity we are able to generate nearly one billion triangles per second.

We would also like to point out that our approach has been released as an open source project that is maintained by Pixar [Pix12]. The project is being used as a plugin for Autodesk Maya [Auta] and Autodesk Mudbox [Autb].

Though we were inspired by feature film applications, our algorithm can be also used to increase the realism and cinemagraphic experience in the next generation of games.

CHAPTER 5

Efficient Evaluation of Semi-Sharp Creases

5.1 Introduction and Algorithm Overview

In Chapter 4 we have proposed an approach for rendering Catmull-Clark subdivision surfaces [CC78] using feature adaptive subdivision [NLMD12]. The presented approach is fully compliant to the RenderMan [Pix05] specification since it can resolve all features such as semi-sharp creases or hierarchical edits by adaptive subdivision. While this makes feature adaptive subdivision a general approach, rendering meshes with semi-sharp creases of higher sharpness is relatively slow. This is due to the exponential growth of patches being processed at edges with semi-sharp crease tags (see Section 4.2). However, semi-sharp creases as specified in RenderMan and proposed by DeRose et al. [DKT98] are an important and widely used extension of Catmull-Clark subdivision surface that allow realistic edges to be defined while keeping memory footprint small.

In order to speed up rendering of objects containing creased edges, our goal is to reduce memory I/O; i.e., to reduce the number of patches that are being created by adaptive subdivision. The key idea of our method is to analyze the polynomial structure of semi-sharp creases and directly evaluate these rather than applying iterative subdivision. We present a direct evaluation scheme for regular patches and another for irregular patches that is inspired by Stam evaluation [Sta98]. In our GPU implementation we directly evaluate regular patches with sharpness tags and we perform adaptive subdivision around extraordinary vertices. This turned out to be faster than performing a Stam-like evaluation for irregular patches. However, compared to previous feature adaptive subdivision, the number of patches being created by subdivision is only linear with respect to sharpness tags and tessellation density (instead of exponential). This results in a considerable performance gain and reduces memory consumption significantly (see Figure 5.3). In the end, rendering speed becomes independent of sharpness tags and makes the usage of high-order sharpness features in real-time applications feasible.

To sum up, the contributions of our approach are:

- Efficient direct evaluation of regular patches with a single semi-sharp crease (see Section 5.2)
- Direct evaluation scheme for arbitary patch setups; with features such as extraordinary vertices and/or bent creases (see Section 5.3).
- Rapid rendering on the GPU using hardware tessellation (see Section 5.4).

5.2 Evaluation of Semi-Sharp Creases in Regular Patches

In this section we assume that a patch contains at most a single semi-sharp crease (or two creases at opposite sides of a 4-sided patch) and does not contain any extraordinary vertices. This is achieved by adaptive subdivision following Nießner et al. [NLMD12] (see Section 5.4). We evaluate bicubic B-spline patches using their tensor product form with parameters u, v. That allows us to simplify the problem to the curve case with a single semi-sharp crease tag. Our goal is to transform the control points of a cubic B-spline curve in a way such that the transformation exactly corresponds to the semi-sharp crease rules of Catmull-Clark subdivision defined by DeRose et al. [DKT98]. In addition, the control points obtained by the transformation must again define a valid cubic B-spline curve. In the following we use respective refinement matrices to define the transform.

A uniform cubic B-spline curve can be refined applying the refinement matrix $\overline{R}$ (or $\overline{R_p}$ at a curve boundary):

$$\overline{R} = \tfrac{1}{8}\begin{pmatrix} 4 & 4 & 0 & 0 \\ 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 1 & 6 & 1 \\ 0 & 0 & 4 & 4 \end{pmatrix} \quad \text{and} \quad \overline{R_p} = \tfrac{1}{8}\begin{pmatrix} 4 & 4 & 0 & 0 \\ 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 0 & 8 & 0 \\ 0 & 0 & 4 & 4 \end{pmatrix}.$$

Thus, subdividing the initial curve control points $\vec{P} = (P_0, P_1, P_2, P_3)^T$ can be represented as $\vec{P'} = \overline{R}\vec{P}$. The matrices $\overline{R}$ and $\overline{R_p}$ correspond to the smooth and sharp Catmull-Clark subdivision rules, respectively.

We now split the cubic B-spline curve $f(t) = N(t)\vec{P}$ (with $N(t)$ a 1×4 matrix containing the cubic B-spline basis functions and $\vec{P}$ defining the B-spline control points) into two curve segments: the infinitely sharp segment $f_\infty(t)$ defined for $0 \leq t \leq 1 - 2^{-s}$ and the transition segment $f_s(t)$ to the crease defined for $1 - 2^{-s} < t \leq 1$.

$$f(t) = \begin{cases} f_\infty(t) = N(t)\vec{P}_\infty & \text{for } 0 \leq t \leq 1 - 2^{-s} \\ f_s(t) = N(t)\vec{P}_s & \text{for } 1 - 2^{-s} < t \leq 1 \end{cases}$$

In order to directly evaluate the curve, we need to obtain the control points for both curve segments $\vec{P}_\infty$ and $\vec{P}_s$. The Catmull-Clark subdivision rules for boundaries can be transferred to the curve case using the transformation matrix

$$M_\infty = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 2 & 0 \end{pmatrix}.$$

The control points of the infinitely sharp section of the curve are then given by $\vec{P}_\infty = M_\infty\vec{P}$. For $f_s(t)$ we use modified 4×4 refinement matrices R and

R_p derived from $\overline{R}$ and $\overline{R_p}$:

$$R = \tfrac{1}{8}\begin{pmatrix} 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 1 & 6 & 1 \\ 0 & 0 & 4 & 4 \end{pmatrix} \quad \text{and} \quad R_p = \tfrac{1}{8}\begin{pmatrix} 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 0 & 8 & 0 \\ 0 & 0 & 4 & 4 \end{pmatrix}.$$

These reduced matrices still subdivide the curve, however, the set of resulting control points defines only a part of the curve. This corresponds to $f_s(t)$ which is only valid for $t \in]1 - 2^{-s}, 1]$.

The control points required to define $f_s(t)$ can be obtained by $R_p^s\vec{P}$. However, this results in a wrong parameterization since the curve's velocity is changed. Thus, we back-transform these interim control points using R^{-1} in order to maintain the original parameterization:

$$\vec{P}_s = (R^{-1})^s R_p^s \vec{P}$$

The resulting control points $\vec{P}_s$ define an extrapolated curve, however, with $t \in]1 - 2^{-s}, 1]$ so that the curve exactly matches the desired shape with the parameterization corresponding to the initial curve. Examining the eigenstructures of R and R_p (both non-defective) allows us to define $M_s = (R^{-1})^s R_p^s$ and diagonalize:

$$R_p^s = V_{R_p} \Lambda_{R_p}^s V_{R_p}^{-1} \quad \text{and} \quad (R^{-1})^s = V_R^{-1} \Lambda_R^{-s} V_R.$$

Thus, we are able to simplify M_s (with $\sigma = 2^s$):

$$M_s = \frac{1-\sigma}{6\,\sigma} \begin{pmatrix} \frac{6\,\sigma}{1-\sigma} & 6\sigma^2 - 5\,\sigma + 1 & -12\,\sigma^2 + 10\,\sigma - 2 & 6\,\sigma^2 - 5\,\sigma + 1 \\ 0 & \frac{2\,\sigma^2 + 3\,\sigma + 1}{1-\sigma} & 4\,\sigma - 2 & 1 - 2\,\sigma \\ 0 & \sigma + 1 & \frac{2\,\sigma^2 + 6\,\sigma - 2}{1-\sigma} & \sigma + 1 \\ 0 & 1 - 2\sigma & 4\sigma - 2 & \frac{2\,\sigma^2 + 3\,\sigma + 1}{1-\sigma} \end{pmatrix}$$

Further, $f(t)$ is given by:

$$f(t) = \begin{cases} N(t)M_\infty\vec{P} & \text{for } 0 \leq t \leq 1 - 2^{-s} \\ N(t)M_s\vec{P} & \text{for } 1 - 2^{-s} < t \leq 1 \end{cases}$$

Two such curves with sharpness 1 and 2 are shown in Figure 5.1. In the end, we can directly evaluate regular patches with a single semi-sharp crease using $f(t)$ to construct the tensor product ($f_i(u)$ refers to the evaluation of one row of the 4×4 control points of a bicubic patch):

$$S(u, v) = (f_0(u), f_1(u), f_2(u), f_3(u))N^T(v)$$

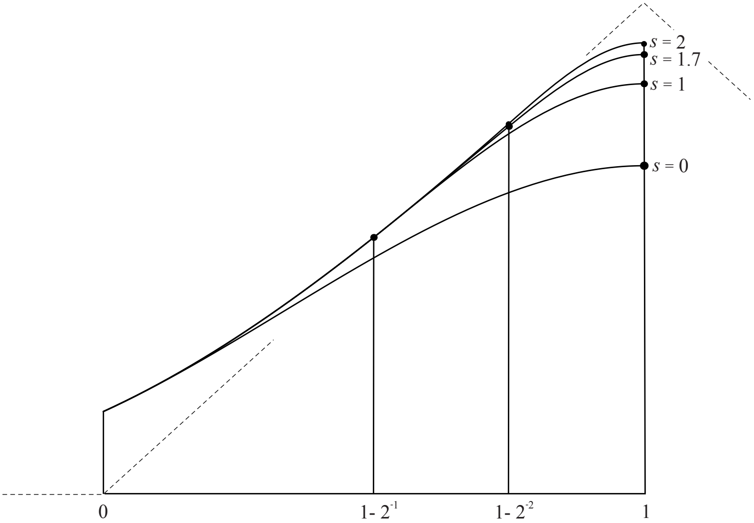


Figure 5.1: Curve segments generated for a particular control polygon; the sharpness tags shown are $s = 0, 1, 1.7, 2$. For $s = 0$ the curve is a single segment defined on $[0, 1]$; for $s = 1, 2$ the curves have two segments defined on $[0, 1 - 2^{-s}]$ and $[1 - 2^{-s}, 1]$; for $s = 1.7$ the curve has three segments defined on $[0, 1 - 2^{-\lfloor s \rfloor}]$, $[1 - 2^{-\lfloor s \rfloor}, 1 - 2^{-\lceil s \rceil}]$, and $[1 - 2^{-\lceil s \rceil}, 1]$.

5.2.1 Fractional Sharpness

DeRose et al. [DKT98] also specify fractional sharpness as a linear blend between integer sharpness levels (see Section 1.3). Hence, it is required to compute a separate $M_{\lfloor s \rfloor}$ and $M_{\lceil s \rceil}$ and apply the linear blend manually. This yields three curve segments, an infinitely sharp part, a linear blend between sharp and the smaller sharpness factor, and a linear blend between the smaller and higher sharpness factor; see Figure 5.1. These segments are defined by the respective control points $\vec{P}_\infty$, $\vec{P}_{\hat{\infty}}$ and $\vec{P}_{\hat{s}}$:

$$\vec{P}_\infty = M_\infty \vec{P}$$

$$\vec{P}_{\hat{\infty}} = (1-(s-\lfloor s \rfloor))(R^{-1})^{\lfloor s \rfloor} R_p^{\lfloor s \rfloor} \vec{P} + (s-\lfloor s \rfloor) M_\infty \vec{P}$$

$$\vec{P}_{\hat{s}} = (1-(s-\lfloor s \rfloor))(R^{-1})^{\lfloor s \rfloor} R_p^{\lfloor s \rfloor} \vec{P} + (s-\lfloor s \rfloor)(R^{-1})^{\lceil s \rceil} R_p^{\lceil s \rceil} \vec{P}$$

Multiple transform operations can be avoided by directly computing the required transformation matrices $M_{\hat{\infty}}$ and $M_{\hat{s}}$:

$$M_{\hat{\infty}} = (1-(s-\lfloor s \rfloor)) M_{\lfloor s \rfloor} + (s-\lfloor s \rfloor) M_\infty$$

$$M_{\hat{s}} = (1-(s-\lfloor s \rfloor)) M_{\lfloor s \rfloor} + (s-\lfloor s \rfloor) M_{\lceil s \rceil}$$

Note that these transformation matrices correspond to the semi-sharp subdivision rules with fractional sharpness tags. Now the initial control points are transformed and the resulting function $f(t)$ is given by the three curve segments:

$$f(t) = \begin{cases} f_\infty(t) = N(t) M_\infty \vec{P} & \text{for } 0 \leq t \leq 1 - 2^{-\lfloor s \rfloor} \\ f_{\hat{\infty}}(t) = N(t) M_{\hat{\infty}} \vec{P} & \text{for } 1 - 2^{-\lfloor s \rfloor} < t \leq 1 - 2^{-\lceil s \rceil} \\ f_{\hat{s}}(t) = N(t) M_{\hat{s}} \vec{P} & \text{for } 1 - 2^{-\lceil s \rceil} < t \leq 1 \end{cases}$$

Figure 5.1 shows an example curve with a fractional sharpness of 1.7. The computation of the tensor product surface $S(u, v)$ is the same as for integer sharpness.

5.3 Evaluation of Semi-Sharp Creases in Irregular Patches

Section 5.2 handles regular patches with at most one semi-sharp crease. Patches with extraordinary vertices or bent creases have been omitted so far. At a bent crease (i.e., a setup where two neighboring edges incident to one vertex have crease tags) an adjacent patch has at least two creased edges that cannot be resolved by subdivision. In this section we deal with both of these setups by extending Stam's direct evaluation method [Sta98]. That is to encode the semi-sharp subdivision rules into subdivision matrices, analyze their eigenstructure and directly obtain the control points of a regular subpatch that corresponds to the original patch evaluation parameters u and v.

The key idea is to subdivide irregular patches and analyze the obtained patch structure. Each set of obtained control points for a subdivision level i has three regular patches $\Omega^i_{1,2,3}$ and one irregular patch (see Figure 5.2). While evaluation points within a regular patch can be evaluated directly, irregular patches require further refinement. Thus, for an arbitrary evaluation point on an irregular patch (defined by the control points C_0) at domain parameters u, v, it is possible to determine a regular bicubic B-spline subpatch that can be directly evaluated; that is after $n = \lfloor \min(\log_2 u, \log_2 v) \rfloor$ subdivision steps.

In order to obtain the child control points after n subdivision steps, we use the Catmull-Clark subdivision matrices $\overline{A}$ and A (see [Sta98] for definition). Let C_0 be the initial 1-ring vertices of an irregular patch, then multiplication with $\overline{A}$ provides the control points $\overline{C_1}$ that define all four child patches: $\overline{C_1} = \overline{A}C_0$. In contrast, multiplying A with C_0 will result in a set of control points that defines only the child patch next to the extraordinary vertex (i.e., omitting patches $\Omega^1_{1,2,3}$): $C_1 = AC_0$. Note that A has dimension $(2N + 8) \times (2N + 8)$ (where N is the valence of the extraordinary vertex) and the extended subdivision matrix $\overline{A}$ has dimension $(2N+17) \times (2N+8)$. The desired control points $\overline{C_n}$ that define the regular patch Ω^n_k can be obtained by: $\overline{C_n} = \overline{A}A^{n-1}C_0$. Stam additionally defines a $16 \times (2N + 17)$

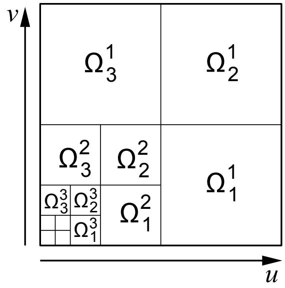


Figure 5.2: Recursive partitioning of the domain of a patch with one extraordinary vertex at the bottom left. All Ω_k^i are regular patches that can be directly evaluated.

picking matrix P_k that selects the 16 control points $B_{k,n}$ from $\overline{C_n}$ that define Ω_k^n: $B_{k,n} = P_k\overline{C_n}$. Basically, after some parameterization corrections these control points $B_{k,n}$ can be used for direct evaluation (details see [Sta98]).

Since the matrices $\overline{A}$ and A do not take care for semi-sharp creases, we introduce additional subdivision matrices S_q. Each S_q corresponds to one specific crease setup q around an extraordinary vertex. In contrast to $\overline{A}$ and A, S_q incorporates the sharp subdivision rules (see Section 1.3). In the following we assume a fixed setup q with the corresponding subdivision matrix S and the extended subdivision matrix $\overline{S}$ (analogous to $\overline{A}$).

Hence, for an irregular patch containing at least one semi-sharp crease, the child control points $\overline{C_n}$ are given by:

$$\overline{C_n} = \begin{cases} \overline{A}A^{n-1-s}S^sC_0 & \text{for } s \leq n-1, \qquad (1) \\ \overline{S}S^{n-1}C_0 & \text{else.} \qquad (2) \end{cases}$$

In case (1) the resulting control points $\overline{C_n}$ define a regular bicubic B-spline patch that can be evaluated directly. The obtained patch of case (2) contains a single edge with a semi-sharp crease where we use the evaluation approach as shown in Section 5.2.

A special case exists if the patch is evaluated at the extraordinary vertex; i.e., $u = v = 0$. In that case we compute the control points $\overline{C_s} = S^s C_0$ that define a patch without crease tags (in fact we only require the control points that belong to the 1-ring of the extraordinary point). We then apply the Catmull-Clark limit stencils corresponding to the eigenvectors of the subdivision matrix $\overline{A}$ (see [HKD93], [LS08])).

Similar to Stam, we perform an eigenvalue decomposition of A and S in order to determine $\overline{C_n}$:

$$A = V_A \Lambda_A V_A^{-1} \quad \text{and} \quad S = V_S \Lambda_S V_S^{-1}$$

Thus, $\overline{C_n}$ can be computed efficiently:

$$\overline{C_n} = \begin{cases} \overline{A} V_A \Lambda_A^{n-1-s} V_A^{-1} V_S \Lambda_S^s V_S^{-1} C_0 & \text{for } s \leq n-1, \quad (1) \\ \overline{S} V_S \Lambda_S^{n-1} V_S^{-1} C_0 & \text{else.} \quad (2) \end{cases}$$

Note that for a fixed sharpness s there are $\sum_{i=1}^{N} \binom{N}{i}$ different matrices S_q. If we allow edges with different sharpness tags on a single extraordinary vertex there are substantially more possible matrices S_q.

While we consider this direct evaluation method for irregular patches with creases an important addition to theory and interesting for CPU applications, the mentioned combinatorical issues pose a severe limitation on GPUs. Hence, for a GPU application we suggest a combination of iterative refinement around extraordinary vertices and direct evaluation for regular patches (see Section 5.4).

5.4 GPU Implementation using Hardware Tessellation

For efficient rendering on the GPU we use hardware tessellation due to its memory-I/O-friendly design (see Section 1.4.2). We perform perform adaptive subdivision iteratively around extraordinary vertices (see Chapter 4) and then directly evaluate the resulting bicubic B-spline patches using hardware tessellation. This turned out to be faster than the proposed alternative in Section 5.3; already standard Stam subdivision was significantly slower (see Figure 4.12). The reason for the efficiency of feature adaptive subdivision is that the number of adaptively created patches around extraordinary vertices is only linear with respect to the number of subdivision steps k ($3N \cdot k$ where N is the vertex valence). In Chapter 4 we use adaptive subdivision around features such as semi-sharp creases, however, this creates an exponential amount of subdivided patches (2^k) after k subdivision steps.

We now replace adaptive subdivision for creases in regular regions by direct evaluation as shown in Section 5.2. Adaptive subdivision is still applied at extraordinary vertices and semi-sharp creases with varying sharpness. It is also used to enforce the condition that regular patches must not contain more than one semi-sharp crease tag. The benefit of this is that the number of child patches created by adaptive subdivision becomes linear instead of exponential with respect to the sharpness tag.

The GPU implementation for regular patches with one edge tagged sharp is straight forward. In a preprocess we rotate these patches so that the semi-sharp crease tag is always at the same side. Additionally, we store the sharpness s for each patch in a buffer. This allows us to render patches with distinct sharpness tags within the same render pass. In the hull shader we obtain the sharpness according to the patch ID and compute the respective transformation matrix M_s. Further, the transformed control points of the different curve segments of $f(t)$ are computed according to M_∞ and M_s, or in the fractional case according to M_∞, $M_{\hat{\infty}}$ and $M_{\hat{s}}$. The two resulting sets (three with fractional sharpness tags) of control points are then passed to the domain shader, where we determine which set of control points is required in order to evaluate the sub-patch according to the domain parameters u, v.

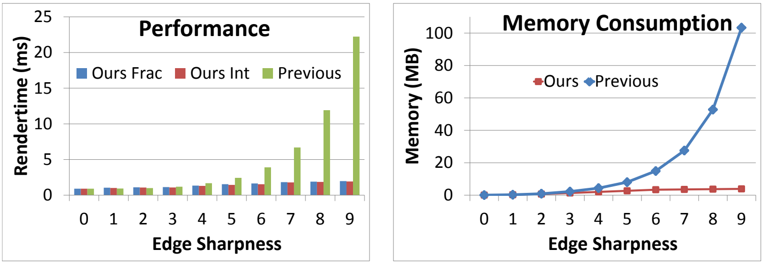


Figure 5.3: Performance (rendered with hardware tessellation at a tess factor of 8) and memory consumption for the *Sportscar* model (see Figure 5.4); our direct evaluation approach (fractional and integer sharpness) and standard feature adaptive subdivision [NLMD12].

5.5 Results

Our implementation uses DirectX 11 running on an NVIDIA GeForce GTX 480. Figure 5.3 shows performance and memory consumption of our direct evaluation method (fractional and integer sharpness variant) and standard feature adaptive subdivision [NLMD12] as shown in Chapter 4 for the *Sportscar* model (see Figure 5.4) with different sharpness tags. Rendering is performed using a tess factor of 8. Performance without any sharpness is the same for all methods since the same number of patches are being created by adaptive subdivision. Having set sharpness tags to 1 or 2 standard feature adaptive subdivision is marginally faster due to lower patch setup costs. At a sharpness above 2 our direct evaluation algorithm is faster. With higher sharpness tags, standard feature adaptive subdivision becomes significantly slower (note the exponential behavior). In contrast, render time using our direct evaluation approach remains almost constant. Render time using this method increases slightly since we still perform adaptive subdivision at extraordinary vertices. The same relation between our direct evaluation approach and standard feature adaptive subdivision can be observed in terms of memory consumption, however, the memory consumption of our direct evaluation approach is always less. A visualization of the different subdivision levels is provided by Figure 5.4 (left ours; right standard feature adaptive subdivision). Also note that our direct evaluation approach allows the modification of sharpness tags at runtime.

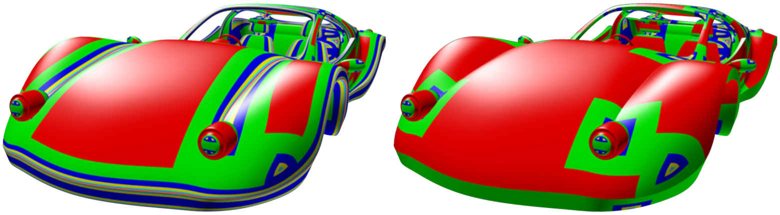


Figure 5.4: *Sportscar* model consisting of 1519 patches with 314 semi-sharp crease tags (sharpness of 6) rendered with standard feature adaptive subdivision (left) and our direct evaluation approach (right). Subdivision levels are indicated by different colors.

5.6 Conclusion

We have presented a novel and GPU-friendly method that allows efficient evaluation of semi-sharp creases as defined by the RenderMan specification [Pix05]. Our algorithm significantly improves upon standard feature adaptive subdivision as shown in Chapter 4. Memory footprint is reduced from an exponential to a linear amount with respect to the subdivision level. That keeps render time low and memory I/O small, and thus allows dealing with high-order sharpness tags in real-time applications. While we have demonstrated how to integrate direct evaluation of semi-sharp creases for hardware tessellation, our method can be also used by offline renderers. For instance, subdivision surfaces with semi-sharp crease tags can be efficiently ray-traced without costly iterative subdivision.

PART II

High-frequency Detail on Subdivision Surfaces

CHAPTER 6

Introduction

Displacement mapping has been used as a means of efficiently representing and animating 3D objects with high-frequency surface detail. Where texture mapping assigns color to surface points at u, v parameter values, displacement mapping assigns vector offsets. The advantages of this approach are two-fold. First, only the vertices of a coarse (low-frequency) base mesh need to be updated each frame to animate the model. Second, since the only connectivity data needed is for the coarse base mesh, significantly less memory is required to store the equivalent highly detailed mesh. Further memory reductions are realized by storing scalar, rather than vector offsets. The displacement is then achieved by offsetting a base surface point in its normal direction according to the value stored in a scalar displacement map. While scalar displacement mapping is not as flexible from a modeling standpoint as vector displacement mapping, it significantly reduces data throughput in the graphics pipeline, as well as the overall storage space and transmission requirements for digital models.

Hardware tessellation is ideally suited to displacement mapping. Higher order parametric patches provide a base surface that is evaluated on-chip to form a dense triangle mesh and immediately rasterized with low memory I/O. Displacing triangle vertices in their normal direction according to a value stored in texture memory has very little performance impact. However, while conceptually simple and highly efficient, hardware displacement mapping has not been widely adopted in real-time applications due to several subtle artifacts.

Displacement Mapping Artifacts:
Meshes are typically endowed with a parameterization in the form of a 2D texture atlas. Conceptually, a few seams are introduced on edges to *unfold*

the surface into the plane, creating a mapping (an atlas) from the plane to the surface. Points on seams map to more than one point in texture space resulting in inconsistent values; bilinear texture filtering exacerbates this problem. For displacement mapping, this can lead to unacceptable cracks in a rendered surface.

The normal of the base surface serves as the direction of displacement. However, this base surface normal is, in general, *not* the normal of the resulting displaced surface, thus complicating accurate shading. To overcome this problem, *normal mapping* has been used to assign more plausible surface normals over displaced vertices to reduce shading artifacts. To allow the base surface to be deformed, *tangent space* normal mapping is used, where the *xyz* coordinates of the normal relative to a tangent frame are stored. The computation of tangent frames is costly and technically challenging since these must be globally consistent across mesh edges. While the resulting shading is often plausible, the deformed normal field does not correspond to the displaced surface; hence it is not accurate. Furthermore, re-computation of normal maps on-the-fly is necessary at displacement authoring time to give instant feedback. Finally, normal map textures add significantly to the storage and data throughput costs of models.

Hardware tessellation is based on the idea of dynamic re-tessellation of patches (see Section 1.4.2). That is, the underlying sampling pattern of patch vertices should be updated every frame to keep the resulting triangle sizes just right; not too small or rasterization becomes inefficient, and not too big so that faceting and interpolation artifacts are kept to a minimum. However, changing this sampling pattern creates *swimming* artifacts in the displaced surface; the surface appears to fluctuate wildly as the sampling pattern changes. This artifact is caused by under-sampling the displacement map.

In Chapter 8 we present an approach for rendering high-frequency detail while avoiding typical artifacts [NL13]. Therefore, we add an analytic and dynamic displacement function on top of a Catmull-Clark subdivision surface that is rendered by hardware tessellation using our approach presented in Chapter 4.

CHAPTER 7

Previous Work

Texturing:
Parameterizing polygon meshes is a well-known issue in computer graphics. Texture atlases are widely used, however, providing consistent values across chart boundaries is challenging ([SWG*03], [GP09]). The resulting minor color errors are often tolerable in the context of texture mapping; the resulting cracks in the context of displacement mapping are not. Purnomo et al. [PCK04] address these color errors by finding quadrilateral regions that are aligned in texture space. They compute boundary overlap to obtain seamless texturing, and they propose several strategies to access texture entries. Our texturing approach of Section 8.2 is similar, but an important difference is that a power-of-two size constraint for tile edges is enforced. Ours is on the interior of tiles (excluding overlap); theirs is on the entire tile (including overlap). While their design choice enables perfect packing (i.e., no wasted space), ours allows for ideal mip pyramids; at a cost of some unused texture space.

Ray et al. [RNLL10] introduce *Invisible Seams*, a method for providing consistent texture accesses and mip mapping using a traditional texture atlas approach. We do something similar, but avoid the use of a global parameterization (*uv* atlas) and do not require storage for texture coordinates. Our (virtual) texture coordinates must align with the parameterization that comes from the underlying Catmull-Clark base surface in order to leverage parametric continuity among patches to achieve a smooth displaced surface; this is straightforward using a tile-based approach.

Rather than redundantly storing overlaps, Burley and Lacewell [BL08] developed Ptex for offline rendering, and use adjacency pointers to access neighboring tiles. While Ptex does not pre-filter tile data, we form full mip pyramids over tiles to accelerate level-of-detail management and avoid under-sampling. *Mesh colors* [YKH10] is another per-face texturing method. Instead of overlap or adjacency pointers, mesh colors stores data indices in texture maps; consistency is achieved by index sharing. However, this scheme requires an extra level of indirection that reduces performance.

Displacement Mapping:
Blinn [Bli78] proposed perturbing surface normals using a wrinkle function. While this mimics the shading of a high-resolution surface, the geometry itself remains unchanged. Hence, Cook [Coo84] developed displacement mapping in order to give objects more realistic silhouettes. The use of scalar displacements in the context of multi-resolution modeling has been proposed by Guskov et al. [GVSS00]. Lee et al. [LMH00] use a similar idea, but they apply displacements on top of a Loop subdivision surface [Loo87]. Additionally, they obtain the displacement function and its derivatives via costly iterative subdivision; our approach involves direct evaluation. Mapping discrete displacement values on Catmull-Clark subdivision surfaces was proposed by Bunnell [Bun05]. This is effective in terms of storage, however, again applying iterative subdivision for displacements is costly. Our approach presented in Chapter 8 also uses Catmull-Clark [CC78] as a base surface, but we apply a displacement function using biquadratic B-splines with a Doo-Sabin [DS78] subdivision structure. An overview of traditional displacement mapping approaches on the GPU is given by Szirmay-Kalos and Umenhoffer [SKU08]. Implementing displacement mapping in the context of hardware tessellation is shown by Tatarchuck et al. [TBB10]. Schäfer et al. [SPM*12] also use the tessellator to apply displacements. They assign vertex attributes (such as displacement values) in the domain shader using a shared index format similar to mesh colors.

Related work with respect to subdivision surface rendering on the GPU is depicted in Chapter 3. For rendering high-frequency surface detail we use feature adaptive subdivision (see Chapter 4). Adaptive subdivision is performed around extraordinary vertices using GPGPU compute kernels and process the resulting bicubic patches with the hardware tessellator. This method is exact and significantly faster than the direct evaluation approach by Stam [Sta98].

CHAPTER 8

Analytic GPU Displacement Mapping for Subdivision Surfaces

8.1 Introduction and Algorithm Overview

In this chapter we present an approach for rendering high-frequency detail using hardware tessellation. Therefore, we use a Catmull-Clark subdivision surface as a base mesh and an analytic displacement function on top of it.

8.1.1 Solutions and Contributions

We propose solutions, in the context of displaced Catmull-Clark subdivision surfaces, to all of the artifacts mentioned in Chapter 6. Following Lee et al. [LMH00], we write the displaced surface as

$$f(u,v) = s(u,v) + N_s(u,v)D(u,v), \tag{8.1}$$

where $s(u,v)$ is a base Catmull-Clark limit surface defined by a coarse base mesh, $N_s(u,v)$ is its corresponding normal field, and $D(u,v)$ is a scalar valued displacement function. We chose Catmull-Clark since it is an industry standard, but our ideas could be extended to Loop subdivision as well; the important property we leverage is that the base surface is everywhere C^2, except at a limited number of *extraordinary vertices* where it is still C^1. Requiring the base surface $s(u,v)$ to be C^2 ensures that its normal field $N_s(u,v)$ will be C^1. Furthermore, by constructing the displacement function $D(u,v)$ to be C^1 with vanishing first derivatives at extraordinary vertices, we can guarantee that the displaced surface $f(u,v)$ will be C^1 smooth everywhere.

Our displacement function $D(u, v)$ is a scalar valued biquadratic B-spline with a Doo-Sabin subdivision surface structure. Therefore, the discrete values found in our displacement maps are the coefficients of this surface. Our motivation in making this choice was to minimize the cost of (per-pixel) evaluation while also providing a C^1 smooth displacement function; biquadratic splines are optimal for this.

In order to deal with the problem of texture seam misalignment, we devise a tile-based texture format, similar to Ptex [BL08], that corresponds to the quad faces of the base mesh (possibly after one level of local subdivision). Unlike Ptex, our format is specifically designed for the GPU, and we eliminate the neighbor face pointers that hampers its data parallel implementation. We also deal gracefully with non-uniform tile sizes so that surface detail is appropriately distributed over a surface. This includes down-sampled displacements (i.e., mip levels) while providing matching displacements at tile boundaries.

Since the position and derivatives of the displaced surface $f(u, v)$ can be evaluated analytically, no normal maps are required. Furthermore, our approach uses an evaluation procedure where the low-frequency base surface $s(u, v)$ is evaluated at triangle vertices in the domain shader, and the derivatives of the high-frequency displacement function $D(u, v)$ are evaluated in the pixel shader; resulting in highly accurate surface shading, even during animation. Our scheme also supports *dynamic* displacement mapping where the displacement function can change at runtime; we prototype a simple authoring tool to demonstrate this.

Finally, we provide a novel and efficient level-of-detail scheme based on a multi-resolution analysis of the displacement function $D(u, v)$. This includes computing the tessellation density on-the-fly and selecting the appropriate mip level of the displacements. This allows us to avoid the under-sampling problems that cause the swimming artifacts typically encountered when varying hardware tessellation factors. Furthermore, we are able to eliminate popping artifacts by employing fractional tessellation factors and filtering between corresponding mip levels.

To summarize, we provide explicit solutions to the following problems with displacement mapping:

- Texture (atlas) seams cause cracks
- Computing displaced surface normals
- Swimming artifacts (under-sampling)

This is achieved by the following contributions:

- Tile-based texture format for displacements on the GPU
- Analytic displaced surface: on-the-fly normal computation
- Efficient GPU rendering using hardware tessellation
- Smooth level-of-detail scheme in order to prevent under-sampling

8.1.2 Algorithm Overview

Our base surface $s(u, v)$ is the limit surface of Catmull-Clark subdivision defined by a two-manifold control mesh, possibly with mesh boundaries. While this surface is traditionally defined as the result of the repeated application of a set of subdivision rules, we (following [HKD93], [Sta98], and [NLMD12]) treat this surface in a parametric form. The topology, geometry, and parameterization of this surface are characterized by its defining control mesh. If the faces of the control mesh are not exclusively quadrilateral, then one refinement step will ensure this. A one-to-one correspondence between these quadrilateral faces and unit square domains is established, giving rise to a global parameterization of the surface (via a face ID; $u, v \in [1, 0] \times [0, 1]$ triple). This is defined by the corresponding subdivision of quadrilateral control mesh faces and unit square domains. This process has well-defined limits and yields a closed form (via eigenbasis functions [Sta98], or bicubic subpatches [NLMD12]). In the interest of simplicity, we assume consistency of quad face ID and unit square domain ID; we therefore safely exclude this book keeping detail from our notation.

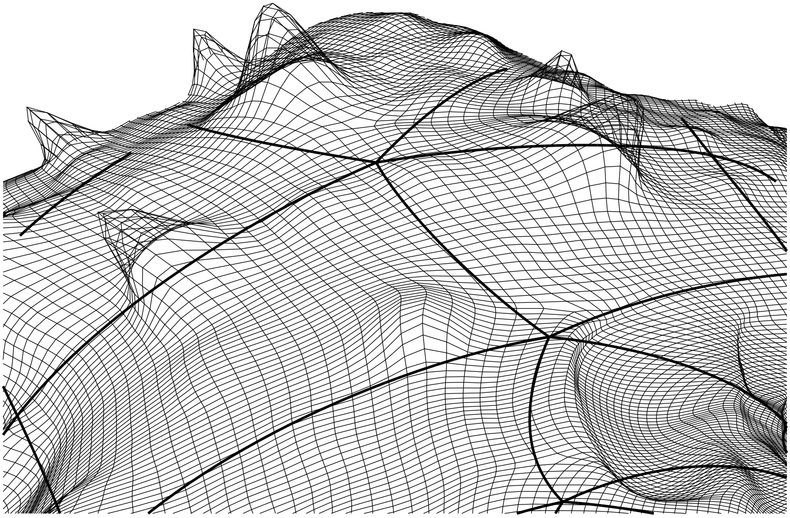


Figure 8.1: Base surface: Catmull-Clark limit patches - patch boundaries shown as thick lines. Displacement surface: biquadratic Doo-Sabin B-splines - scalar coefficients on top of base surface normal field shown as thin lines.

For our analytic displacement function $D(u, v)$, we use biquadratic B-splines. These patches have an arrangement that is consistent with Doo-Sabin subdivision. This means that the control mesh for our displacement coefficients is *dual*, with refinements, to the control mesh of the base mesh. Note however, that $D(u, v)$ is scalar valued and can be thought of as a *height field*. In other words, both the base surface $s(u, v)$ and the displacement function $D(u, v)$ correspond to the same topological two-manifold; though embedded in $\mathbb{R}^3$ and $\mathbb{R}^1$, respectively. Note again, that choosing biquadratic B-splines is closely related to Doo-Sabin subdivision and gives us a globally C^1 displacement function that is less costly to compute than higher order alternatives.

Figure 8.1 shows a detail view of a model with base patch edges (thick curves) and the displacement function coefficients over the base surface (thin grid). As a practical matter, we deal with extraordinary vertices by imposing a constraint that causes first derivatives of the displacement function $D(u,v)$ to vanish at these points. This degeneracy implies that $D(u,v)$ is a globally C^1 function that can be evaluated over the entire manifold without special case handling, see Section 8.2.2 for a detailed explanation.

For each quad face of the base surface control mesh, we define a *texture tile* that contains the coefficients of displacement function. For non-quad faces, we locally subdivide once to obtain quads. To evaluate the displacement function near tile boundaries, we need coefficient data from adjacent tiles. Trying to explicitly access adjacent tile data on the GPU would degrade performance since only boundary evaluations would require this, causing a branch and breaking data parallelism. Instead, we pad our tiles with a one texel overlap region; this ensures good data parallel performance since boundary evaluations will not be a special case. We devise a straightforward tile-based texture format in Section 8.2 that contains these overlaps, as well as a simple solution to tile access issues caused by extraordinary vertices.

Treating the displaced surface in a smooth analytic form means that it will have a well-defined, smooth normal field; this will eliminate many shading artifacts. Furthermore, the separation of the displaced surface into a low-frequency base surface and a high-frequency displacement function is ideally suited to a modern graphics pipeline implementation. We evaluated the base surface and its partial derivatives at a relatively low frequency in the domain shader. The derivatives of the displacement function are then evaluated at a higher frequency in the pixel shader. The interpolated low-frequency data, combined with the evaluated high-frequency data results in a highly accurate normal field, ideal for lighting calculations (see Section 8.3).

To deal with swimming artifacts caused by displacement under-sampling, we form mip pyramids over texture tiles via a multi-resolution analysis of the displacement function $D(u,v)$. In order to remove swimming artifacts, we employ a smooth level-of-detail scheme that matches sampling density to the appropriate displacement function mip level (see Section 8.4).

An overview of our rendering algorithm is provided by Figure 8.2.

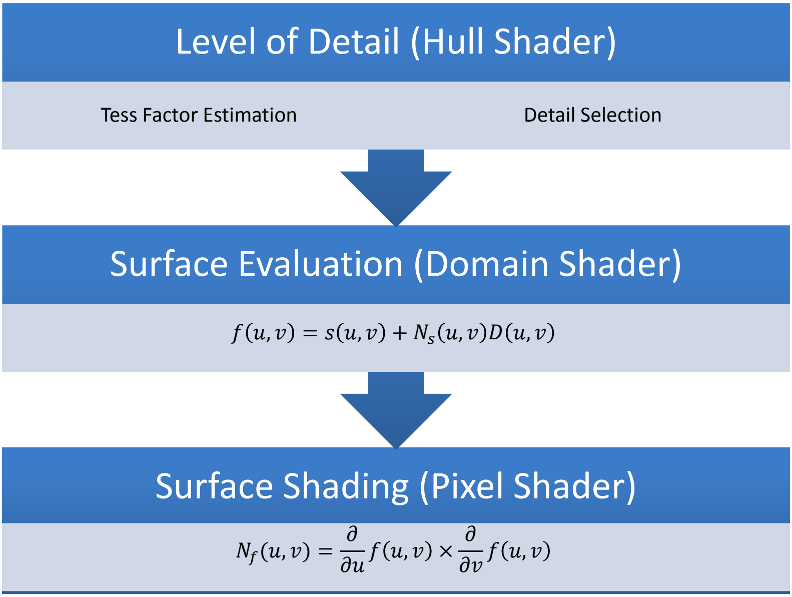

Level of Detail (Hull Shader)
Tess Factor Estimation
Detail Selection
Surface Evaluation (Domain Shader)
$f(u,v) = s(u,v) + N_s(u,v)D(u,v)$
Surface Shading (Pixel Shader)
$N_f(u,v) = \frac{\partial}{\partial u}f(u,v) \times \frac{\partial}{\partial v}f(u,v)$

Figure 8.2: Analytic displacement mapping: algorithm overview.

8.2 Tile-Based Texture Format

We store our biquadratic displacement function coefficient data in an axis-aligned tile-based texture format. This avoids seam misalignment problems that plague classic u, v atlas parameterization texture methods. Our format can be seen as a GPU version of Ptex [BL08], however, we do not rely on adjacent tile pointers since these are impractical on the GPU. Instead, we store a one texel overlap per tile to enable filtering while matching displacements at tile boundaries. Two of these tiles are shown in Figure 8.3. Overlap computation, particularly at extraordinary vertices, is described in Section 8.2.2. Each tile corresponds to a quad face of the Catmull-Clark control mesh. We require tile edges to be power-of-two (plus overlap) in size; that is for a $tileSize = 2^k$ (for integer $k \geq 1$) , tile edge lengths are of the form $tileSize + 2$. However, adjacent tiles do not need to be the same size. We currently only support square tiles; but rectangular tiles could be supported at a cost of a few additional storage bits per tile.

8.2.1 Displacement Data Generation

The base surface and displacement function needed by our algorithm could be generated by a conversion process from a scanned dense triangle mesh, or directly authored using a sculpting tool. Our work is agnostic to this choice, but we discuss the trade-offs here for the sake of completeness.

Lee et al. [LMH00] assume that a high-resolution triangle mesh is given, and then simplified to obtain a coarse (Loop subdivision) base mesh. The displacement data are then found by extraction using ray casting. Rays are fired for each tile entry from the base mesh in the normal direction and intersected with the (high-resolution) source mesh. Unfortunately, extraction using ray casting has problems. In particular, rays can miss the source surface, and typically requires manual adjustments. These geometry processing issues remain as open research problems that we do not address here. However, assuming *clean* displaced sample data at all tile locations, we are able to convert surfaces with traditional displacements into our tile-based format. Therefore, we resample a given displacement map and solve for the biquadratic B-spline coefficients to interpolate the displacement data. We have performed this conversion process to generate the displacement data for the sample models *Dragon Head* and *Monster Frog* (see Figure 8.4).

An alternative approach is to integrate scalar displacement modeling into the authoring tool. This implies that sculpting data are restricted to translations along normals of the base surface. Artists can then directly create the displaced surface exactly as it will appear in the final application. For demonstration we have prototyped such a tool that allows direct authoring of the displacement function. The tile-based data format we provide allows a user to directly *paint* on the surface and modify displacements in place. While we apply edits on the CPU, we only update modified tiles in GPU memory in order to keep CPU-to-GPU memory transfer small. Modeling is analogous to multi-resolution editing as used by typical sculpting tools such as MudBox [Autb] or ZBrush [Spe08].

We do not claim that the work-flow shown in our authoring tool is necessarily original (details of how professional authoring tools work are proprietary). We do claim however, that by using our techniques, the delay and limitations imposed by the intermediate step of dense triangle mesh creation can be avoided. Far less GPU memory is needed, and the model is readily animated without any internal conversions. Furthermore, the analytic nature of our method provides for correct shading at all scales.

8.2.2 Overlap at Extraordinary Vertices

Near extraordinary vertices, where more (or less) than four tiles meet, we need a way to efficiently evaluate our displacement function. Our scalar displacement spline has a Doo-Sabin subdivision structure; however, a direct evaluation approach based on eigenbasis functions does not exist since the subdivision matrix for this case is defective [Sta98].

Instead, we impose a constraint that will allow us to evaluate the displacement function $D(u, v)$ as a standard biquadratic B-spline over its entire domain. The idea is to set all tile corners corresponding to the same extraordinary vertex to the same value. We find this value by averaging. The result is that $\frac{\partial}{\partial u}D = \frac{\partial}{\partial v}D = 0$ at these tile corners. While this limitation is unfortunate from a modeling perspective, it is beneficial from a rendering perspective. Evaluation of the displacement function $D(u, v)$ is fast and consistent since extraordinary vertices do not require branching to specialize code. Furthermore, we can guarantee that our displacement spline $D(u, v)$, will be C^1 across tile boundaries, for proof see [Rei97]. This means that extraordinary vertices will not cause any shading discontinuities.

8.2.3 Mip Levels and Global Texture Design

The swimming artifacts associated with the dynamic tessellation patterns generated by the hardware tessellation unit are under-sampling artifacts.

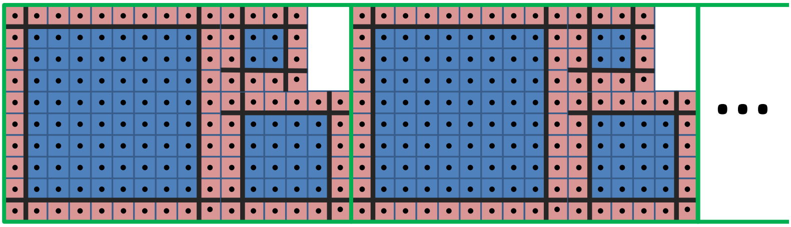


Figure 8.3: Snippet of our texture format used for displacement values (8×8 per tile; blue) showing two tiles (bordered green) including overlap (red) and mip levels.

That is, the underlying displaced surface is a high-frequency signal that is sampled below its Nyquist rate by the tessellator. This is a classic problem in other context within computer graphics and signal processing that can effectively be resolved using mip mapping [Wil83]. Therefore, we precompute a full mip map pyramid for each tile. To generate these mip levels, we tried both Haar wavelets and a wavelet based on quadratic B-splines, so-called *B-wavelets*. Haar wavelets correspond to classic 4-way averaging to down-sample mips levels. The quadratic B-wavelets we used are based on the work of Bertram et al. [Ber04] and involve a kernel with larger support.

Once all mip levels have been generated, we pack all tiles including its mip levels in a single texture. In order to leverage cache coherence, mip levels of individual tiles are stored next to each other (see Figure 8.3). While this leaves some unused space, our experiments show that it provides superior performance. In the end, we require $(1.5 \cdot tileSize + 4) \cdot (tileSize + 2)$ texture entries for a single tile including overlap and mip levels. Additionally, for each quad face we must store its tile size and an offset to the tile location within the global texture in a separate buffer. Tile data is then indexed by the face ID. Note that local mip level offsets (within a tile) are computed at runtime and do not require additional storage.

8.2.4 Non-uniform Tile Sizes

We support distinct tile sizes in order to allow localized detail within a mesh. As a result there may be adjacent tiles with distinct resolutions. To avoid cracks, we must ensure the consistency of data accessed along boundaries between mixed resolution tiles. To this end, we require that coarser mip levels of a higher resolution tile correspond to its lower resolution neighbor. This is achieved by computing tile overlap for each mip level separately at matching resolutions. Since not all tiles have the same number of mip levels (i.e., they have different resolutions), there are boundaries (at particular mip levels) where overlap computation cannot be performed.

Tile resolution is characterized by k, where $\mathit{tileSize} = 2^k$ (tile edge length, not including overlap). For each base mesh control vertex, we determine the incident tile with the highest resolution k and use this number k as a base value for that vertex. We then find the differences between the base value of a vertex, and each incident tile's highest resolution (one of these is guaranteed to be zero); we store these differences for each of the four tile corners. These values are packed into a single 32 bit integer stored per tile (see Section 8.3.4).

At runtime we bilinearly interpolate these difference values for a given u, v parameter value. The tile's own resolution minus this interpolated value tells us the maximum (possibly fractional) mip level that can be accessed for that parameter value. Along edges between mixed resolution tiles, we will always obtain a consistent maximum mip level, and hence consistent data accesses. In Section 8.4 we discuss a vertex based level-of-detail scheme and how mip levels are selected at runtime.

8.3 Surface Rendering

Hardware tessellation generates triangle meshes on-the-fly by sampling a user defined evaluation procedure for a parametric surface patch. While this allows efficient and parallel geometry processing, this paradigm is not compatible with the traditional recursive refinement construction of subdivision surfaces. Several G^1 patch-based approximate Catmull-Clark schemes have appeared in recent years (e.g., [MNP08], [LSNC09]) to overcome this difficulty. However, these are not adequate for our purposes, since we require a C^2 base surface (in order to guarantee that the final displaced surface is C^1). The direct evaluation procedure from Stam [Sta98] could be used to evaluate the base surface. However, our experiments on the GPU indicate that feature adaptive subdivision described in Chapter 4 performs significantly better (see Section 8.3.4). In the following, we first describe how we evaluate the displacement function and then discuss how to efficiently render the resulting surface using hardware tessellation.

8.3.1 Surface Evaluation

For given u, v coordinates and face ID, we evaluate the displaced surface

$$f(u,v) = s(u,v) + N_s(u,v)D(u,v),$$

corresponding to a texture tile, by evaluating the base patch $s(u,v)$, its normal $N_s(u,v)$, and the corresponding displacement function $D(u,v)$. The scalar displacement function is evaluated by selecting the 3×3 array of coefficients for the biquadratic subpatch of $D(u,v)$, corresponding the u, v value within its tile domain. We transform the patch parameters u, v into the subpatch domain $(\hat{u}, \hat{v})$ using the linear transformation T:

$$\hat{u} = T(u) = u - \lfloor u \rfloor + \tfrac{1}{2}, \quad \text{and} \quad \hat{v} = T(v) = v - \lfloor v \rfloor + \tfrac{1}{2}.$$

We then evaluate the scalar displacement function

$$D(u,v) = \sum_{i=0}^{2} \sum_{j=0}^{2} B_i^2(T(u)) B_j^2(T(v)) d_{i,j},$$

where $d_{i,j}$ are the selected displacement coefficients, and $B_i^2(u)$ are the quadratic B-spline basis functions.

The base surface normal $N_s(u, v)$ is obtained from the partial derivatives of $s(u, v)$:

$$N_s(u, v) = \frac{\frac{\partial}{\partial u}s(u, v) \times \frac{\partial}{\partial v}s(u, v)}{\left\| \frac{\partial}{\partial u}s(u, v) \times \frac{\partial}{\partial v}s(u, v)\right\|_2}.$$

In order to obtain the normal of the displaced surface $f(u, v)$, we compute its partial derivatives:

$$\frac{\partial}{\partial u}f(u, v) = \frac{\partial}{\partial u}s(u, v) + \frac{\partial}{\partial u}N_s(u, v)D(u, v) + N_s(u, v)\frac{\partial}{\partial u}D(u, v),$$

$\frac{\partial}{\partial v}f(u, v)$ is similar. Note that the derivatives of the displacement function are a scaled version of subpatch derivatives:

$$\frac{\partial}{\partial u}D(u, v) = \mathit{tileSize} \cdot \frac{\partial}{\partial \hat{u}}\hat{D}(\hat{u}, \hat{v}).$$

Further, $\frac{\partial}{\partial u}s(u, v)$ can be directly obtained from the base surface. To find the derivative of $N_s(u, v)$, we note that the derivatives of the (unnormalized) normal $N_s^*(u, v)$ are found using the Weingarten equation [DC76] (E, F, G and e, f, g are the coefficients of the first and second fundamental form):

$$\frac{\partial}{\partial u}N_s^*(u, v) = \frac{\partial}{\partial u}s(u, v)\frac{fF - eG}{EG - F^2} + \frac{\partial}{\partial v}s(u, v)\frac{eF - fE}{EG - F^2},$$

$\frac{\partial}{\partial v}N_s^*(u, v)$ is similar. From this, we find the derivative of the normalized normal:

$$\frac{\partial}{\partial u}N_s(u, v) = \frac{\frac{\partial}{\partial u}N_s^*(u, v) - N_s(u, v)(\frac{\partial}{\partial u}N_s^*(u, v) \cdot N_s(u, v))}{\|N_s^*(u, v)\|_2},$$

$\frac{\partial}{\partial v}N_s(u, v)$ is similar. Finally, we compute $\frac{\partial}{\partial u}f(u, v)$ (analogously $\frac{\partial}{\partial v}f(u, v)$) and thus $N_f(u, v)$.

8.3.2 Approximate Shading

Since the computation of the derivatives of the base surface normal using the Weingarten equation is relatively costly, it is possible to approximate the normal computation of the displaced surface $N_f(u, v)$. Blinn [Bli78] suggests ignoring the Weingarten term, resulting in the approximate partial derivative:

$$\frac{\partial}{\partial u} f(u, v) \approx \frac{\partial}{\partial u} s(u, v) + N_s(u, v) \frac{\partial}{\partial u} D(u, v).$$

This is a reasonable assumption when the displacements are small since the term $\frac{\partial}{\partial u} N_s(u, v) D(u, v)$ becomes negligible. $\frac{\partial}{\partial v} f(u, v)$ can be approximated the same way. We discuss this further in Section 8.5, and quantify the performance of approximate versus accurate shading.

8.3.3 Rendering using Hardware Tessellation

We evaluate the base surface $s(u, v)$, its derivatives $\frac{\partial}{\partial u} s(u, v)$, $\frac{\partial}{\partial v} s(u, v)$ and the displacement function $D(u, v)$ in the domain shader. Additionally, the derivatives of the normal $\frac{\partial}{\partial u} N_s(u, v)$, $\frac{\partial}{\partial v} N_s(u, v)$ can be evaluated. These computations are used in order to determine the vertices of the triangle mesh that are generated by the tessellator.

The vertex attributes computed in the domain shader are then interpolated by hardware and available in the pixel shader. In the pixel shader, we evaluate the derivatives of the displacement function $\frac{\partial}{\partial u} D(u, v)$ and $\frac{\partial}{\partial v} D(u, v)$. This allows us to compute the derivatives of the displaced surface normal $\frac{\partial}{\partial u} f(u, v)$, $\frac{\partial}{\partial v} f(u, v)$ at each pixel. Therefore, we obtain $N_f(u, v)$ at each pixel that corresponds to the displaced surface. Evaluating the surface normal $N_f(u, v)$ on a per vertex basis would degrade rendering quality, due to interpolation artifacts.

As a base surface we tested Stam evaluation [Sta98] and feature adaptive subdivision [NLMD12]. Both schemes work well, however, [NLMD12] is faster, especially for high levels of tessellation.

8.3.4 Base Surface Evaluation

Feature adaptive subdivision [NLMD12] as shown in Chapter 4 leverages the nested polynomial patch structure of Catmull-Clark subdivision. Regular regions of a control mesh define bicubic B-spline patches that can be rendered by the hardware tessellator. Further subdivision is only needed near extraordinary vertices (a type of feature), to generate more regular regions and more patches. The limit surface will contain an infinite number of smaller and smaller patches around extraordinary vertices. However, after only a few subdivision levels, these patches will only cover a few pixels. At that point, adaptive subdivision can stop, and final patches are rendered as quads. Subdivision is carried out by GPGPU compute kernels driven by precomputed index buffers. At each level of subdivision, regular patches corresponding to that level are generated. Additionally, the final extraordinary vertex hole filling quads are generated for rendering in a separate final pass. The advantage of feature adaptive subdivision is that the number of subdivision operations grows linearly with respect to subdivision level, rather than exponentially as it does when refining the entire mesh at each level. The high compute-to-memory bandwidth ratio of modern GPUs is exploited since evaluating bicubic B-splines using hardware tessellation performs better than streaming refined mesh vertices to and from GPU memory.

Applying the displacement function for level 0 patches is trivial since tiles correspond to patches. At higher subdivision levels, however, a patch will correspond to a subdomain of a base patch. Additionally, feature adaptive subdivision may rotate patches (by $j\frac{\pi}{2}$) to reduce combinatorics. So for each patch we store a local offset within a tile and its rotation j. The size of a patch domain is provided by its subdivision level. From these we can selectively access displacements belonging to a subpatch. After taking patch subdomains and rotations into account, all patches are rendered uniformly as described in Section 8.3.3.

In the end, we use the following control structure to access displacement data (we only need 16 byte per patch compared to the 32 byte required by traditional texturing):

```
struct {
  ushort[2] globalTextureOffsetXY;
  ushort[2] localDomainOffsetXY;
  ushort tileSize;
  ushort rotation;
  uchar[4] tileSizeDifferences;
};
```

Rendering the extraordinary vertex quads (i.e., faces at the finest subdivision level that are not being further tessellated) requires a separate rendering pass. In this case, an exception to the rendering rule for regular patches occurs. Due the singularity in the subdivision surface parameterization at extraordinary vertices, the directions $\frac{\partial}{\partial u}s$ and $\frac{\partial}{\partial v}s$ are not consistently defined (as they are at all other points of the surface). However, the base surface limit normal N_s is well defined at extraordinary vertices. So to obtain a consistent normal over these final quads (in particular along quad edges), we bilinearly blend between the limit normal N_s, and the displaced surface normal N_f involving the interpolated partials $\frac{\partial}{\partial u}s \times \frac{\partial}{\partial v}s$. The blending function equals 1 at extraordinary vertices (for N_s) and 0 at other quad corners (for N_f), and is performed on a per fragment basis without requiring the evaluation of the Weingarten term at the extraordinary point. Fortunately, since the quads incident on extraordinary vertices are rendered separately, there are no measurable extra costs for this special treatment. Also note that the partial derivatives of the displacement function $D(u, v)$ are restricted to be 0 at extraordinary vertices (see Section 8.2.2); thus, the resulting surface will be C^1 and correspond to the displacement data.

8.4 Level of Detail

Hardware tessellation is controlled by user specified tessellation (tess) factors assigned per patch in a hull shader program. The fixed function tessellation unit then generates an appropriate sampling pattern to match these inputs. This is particularly effective in the context of displacement mapping since detail can be added and removed at runtime. However, under-sampling occurs when the resolution of the sampling pattern is insufficient to reconstruct the high-frequency displacement detail. This can lead to swimming artifacts since minor tess factor changes can cause significant changes in the resulting surface. Our solution is to select mip levels based on the tessellation density in order to avoid under-sampling artifacts.

8.4.1 Tessellation Factor Estimation

We first estimate tess factors; this includes two interior tess factors per patch, as well as a tess factor for edge patch. Adjacent patches must have the same tess factors assigned along shared edges in order to guarantee crack-free rendering. Our approach is to determine a tessellation density value for each vertex of the base mesh in a compute shader kernel, and then to propagate these values to the (sub) patches corners using bilinear subdivision.

We determine tess factors *TF* for base mesh vertices *v* (with edges *e*) using one of these simple methods (*c* is a user defined constant):

- Distance based: $TF = c \cdot \|eye - v\|_2$
- Screen space area based: $TF = c \cdot \sqrt{\sum e_i \times e_{i+1}}$
- Screen space edge length based: $TF = c \cdot \max_i \|e_i\|_2$

More elaborated methods, for instance [FMM86], that take surface curvature into account are also possible, but would require greater computational effort.

Once tess factors have been computed for patch corners, we assign patch edge tess factors as the maximum of the two incident corner tess factors. We treat inner tess factors analogously (maximum of opposite edge tess factors in *u* and *v* directions).

Please note that our tess factor estimation is orthogonal to the level-of-detail schemes shown in the context of feature adaptive subdivision (see Section 4.7).

8.4.2 Mip Level Selection

The tess factors computed for each of the four patch corners are passed to the domain shader and are bilinearly interpolated producing the function $TF(u, v)$. Based on this interpolant we select the two adjacent mip levels $\lfloor \log2(TF(u, v)) \rfloor$ and $\lceil \log2(TF(u, v)) \rceil$ and linearly interpolate the resulting displacements (including the derivatives). We must also clamp this value to the maximum mip level determined in Section 8.2.4 in order to provide consistent results at shared boundaries where tiles with distinct resolutions meet. This allows us to guarantee a specific sampling rate of the displacement map in order to avoid under-sampling.

8.5 Results

Our implementation uses DirectX 11 with shader code written in HLSL. Timing measurements were made on an NVIDIA GeForce GTX 480 and are provided in milliseconds.

In order to test our method, we extracted displacement values from two representative models, the *Dragon Head* and the *Monster Frog*. Figure 8.4 shows the rendering with and without displacements including the control mesh of the base mesh. For these images the approximate variant (see Section 8.3.2) without the Weingarten term is used and the level-of-detail computation (see Section 8.4) is omitted. Displacements are stored using 16 bit floating points and a tile size of 16×16 (overall 2.7 MB) and 8×8 (overall 413 KB), respectively. Tessellation factors are determined adaptively based on the camera distance (637K and 328K triangles are generated) so that the models are rendered with our algorithm in 1.7 ms and 1.3 ms. Note that we also tried 32 bit floating point displacement values; both performance and visual quality changes were insignificant.

Figure 8.5 depicts the level-of-detail scheme proposed in Section 8.4 (again the Weingarten term is ignored). By blending between adjacent mip levels we achieve smooth level-of-detail transitions. We found that using Haar

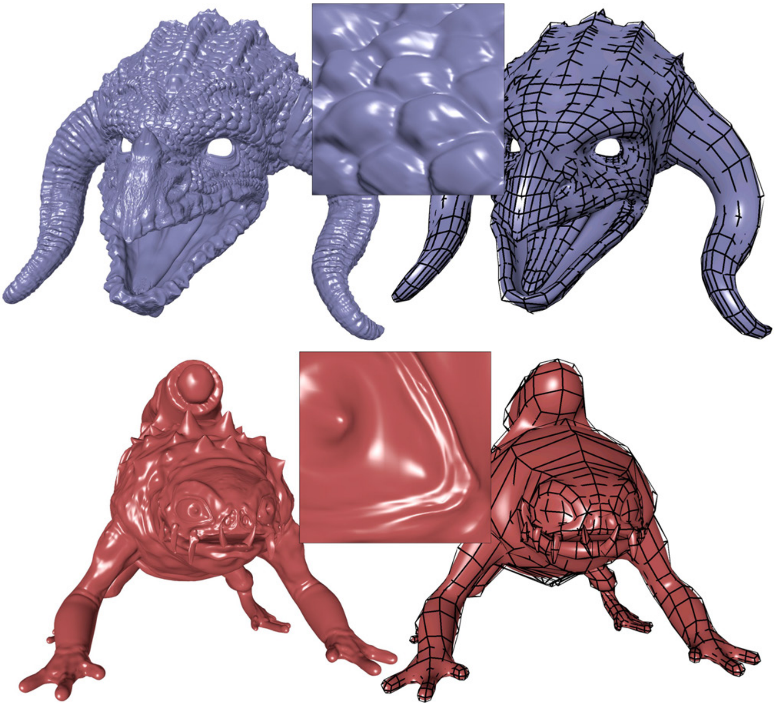


Figure 8.4: The *Dragon Head* (2706 patches; 1.7 ms and 1.2 ms; 2.7 MB for displacement values) and *Monster Frog* (1292 patches; 1.3 ms and 0.85 ms; 413 KB for displacement values) model rendered w/ and w/o our displacement method. We displace vertices according to an analytic displacement function on top of a Catmull-Clark base surface using hardware tessellation. Since normal computation is solely based on the displacement function, no normal map is required. This allows modifying displacements at runtime and increases performance due to a reduced memory I/O.

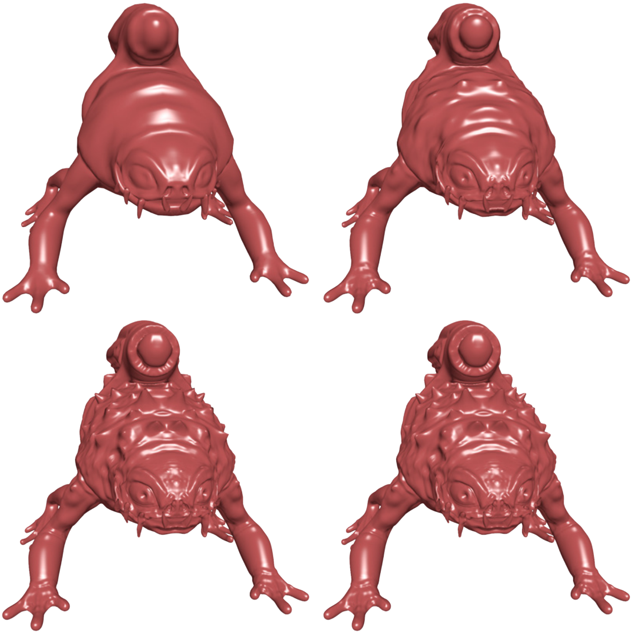


Figure 8.5: Our level-of-detail scheme; precomputed mip levels are selected based on the tess factors. These images (from left to right) are rendered using tess factors 2, 4, 8, 16 resulting in render times 0.6 ms, 1.0 ms, 1.7 ms, 3.3 ms. Also note that we are linearly blending between two adjacent mip levels in order to obtain a smooth transition between selected levels.

wavelets provided smoother results, whereas biquadratic B-wavelets preserved more detail in down sampled mip levels. Results shown here use the Haar wavelet mip level variant. Detail is dynamically added with an increased tessellation rate and vice versa. This enforces a certain sampling rate of the displacement values and thus prevents under-sampling artifacts. For these images (from left to right) tess factors 2, 4, 8, 16 are chosen resulting in render times 0.6 ms, 1.0 ms, 1.7 ms, 3.3 ms, respectively.

The difference between accurate and approximate normal computation (with and without the Weingarten term) is shown in Figure 8.6 for the *Dragon*

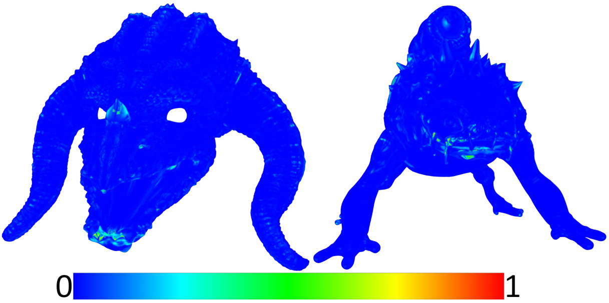


Figure 8.6: Difference (Euclidean color distance in HSV) between the approximate normal computation and the accurate variant that takes the Weingarten term into account. Patches with large displacement offsets and high curvature are most different. For this setting (same as in Figure 8.4) the accurate variant is 46% and 53%, respectively, slower than the approximation.

Head and *Monster Frog*. Patches with large displacement offsets and high curvatures show the most difference; elsewhere the results are visually indistinguishable. For these render settings (same as in Figure 8.4), taking the Weingarten term into account increases the render time from 1.7 ms to 2.6 ms and from 1.3 ms to 1.9 ms, respectively. Considering the minor shading improvements but the major performance decrease the accurate variant seems to be less appropriate for real-time entertainment applications.

Figure 8.7 shows performance results of our method with various setups using different tess factors. Rendering the basic version of our method (i.e., without level of detail and without Weingarten term) is only slightly slower than rendering the base surface without displacements but comprises high-frequency surface detail. Both level-of-detail and accurate normal computation come at some costs, whereas taking the Weingarten term into account affects performance more drastically. We also compare our method against traditional displacement mapping combined with tangent space normal mapping. While the resulting surfaces of our method (basic version) and the traditional approach are about the same, our method achieves higher frame rates. This is attributed to the fact that we only require a single displacement texture rather than a separate displacement and normal map. Also note that tangent space normal mapping has issues with texture seams, under-sampling artifacts, and does not support dynamic displacements.

8.6 Conclusion

We have presented a method for rendering displaced Catmull-Clark subdivision surfaces that avoids typical displacement mapping artifacts that have limited their use on GPUs with hardware tessellation. These include texture seams, the need for normal maps to provide appropriate high-frequency shading, and under-sampling that causes swimming. We introduced a tile-based texture format for the GPU and defined an analytically smooth displacement surface using a biquadratic displacement function. Data for this function can be obtained by traditional displacement extraction (e.g., MudBox [Autb] or ZBrush [Spe08]) or direct authoring. We believe these advances will be useful in both the context of authoring, as well as in the runtime engine, where our method provides highly accurate shading of detailed models at high frame rates.

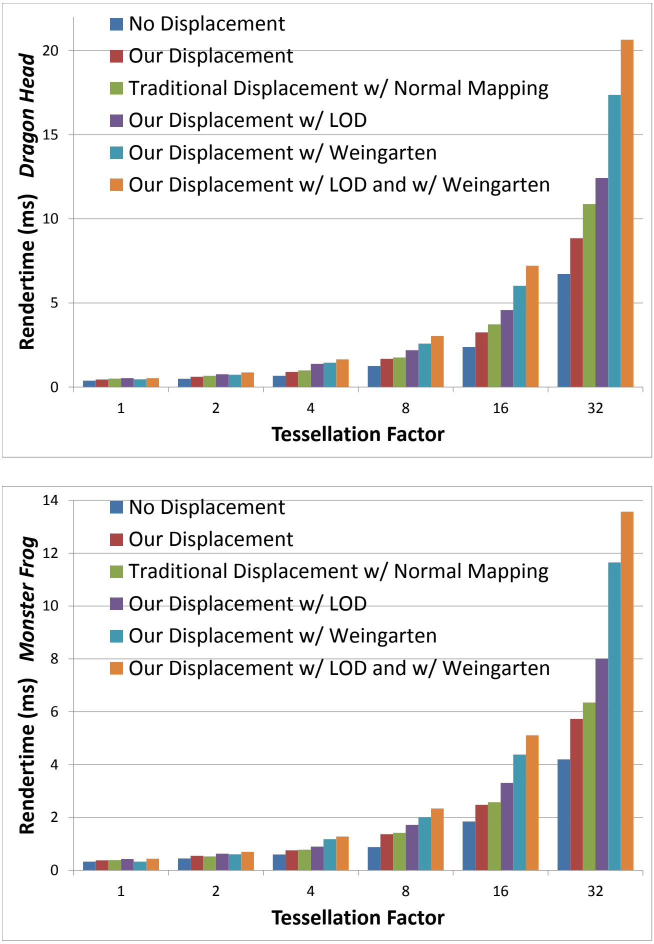


Figure 8.7: Performance results for rendering the base surface, rendering the displaced surface using our method with various setups, and a comparison against traditional displacements combined with tangent space normal mapping. Note that traditional displacement mapping is always slower than our basic method even so dynamic tangent and bitangent computation (required for animation) is omitted.

PART III

Performance Enhancement by Patch Culling

CHAPTER 9
Introduction

In Chapters 4, 5, 8 we have presented techniques for rendering surfaces using hardware tessellation. The corresponding elevation of patch primitives to first class objects in the graphics pipeline offers a unique opportunity to revisit classic culling methods. Therefore, we improve upon those techniques with new insights tailored to the computational demands of patch processing. On the one hand we present an approach for back-patch culling (see Chapter 11) and on the other hand we show how to apply patch-based occlusion culling (see Chapter 12). While the first algorithm identifies back-patches based on patch normals, the second approach relies solely on visibility information, thus allowing surfaces to have displacements.

Our culling techniques are agnostic to a particular patching scheme; our only assumption is that patches obey the convex-hull property, and the first partial derivatives can be bounded. For simplicity, we consider widely used bicubic Bézier patches, though our algorithm is easily extensible to other patch types. Those could be the bicubic B-spline patches generated by our feature adaptive subdivision approach presented in Chapter 4.

In the end, both of our culling approaches significantly speed up rendering by avoiding unnecessary surface evaluation and shading operations of hidden patches, invariant of the used patching scheme.

CHAPTER 10

Previous Work

10.1 Back-Patch Culling Techniques

Back-face culling is a standard method in today's graphics hardware to reduce triangle rasterization and pixel/fragment shading operations. To avoid calculating the dot-product between plane normal and viewing direction for each polygon, hierarchical approaches cluster polygons by normal and cull entire polygon groups [KM96]. This concept can be transferred to parametric surfaces, where *back-patch* culling removes an entire patch before it is tessellated into polygons. Such schemes take patch normals or tangent planes into account in order to determine whether a patch is back or front-facing.

The *cone of normals* presented by Shirmun and Abi-Ezzi [SAE93] is one such technique. In a preprocess, they determine the *normal patch*, for a given Bézier patch and compute its bounding cone of normals defined by $apex : \mathbf{l}$, $axis : \mathbf{a}$, $angle : \alpha$. $N(u,v) = \partial B(u,v)/\partial u \times \partial B(u,v)/\partial v$, During runtime, for eyepoint $\mathbf{e}$ the vector $\mathbf{v} = (\mathbf{e} - \mathbf{l})/||\mathbf{e} - \mathbf{l}||$ is used in the simple test $\mathbf{v} \cdot \mathbf{a} \leq \sin(\alpha)$ to determine if the patch can be culled safely. The bound is comparably tight and the runtime test fast, but the calculation of the normal patch is expensive. This is not a problem for static models, but a draw back for patches that are animated or generated on-the-fly.

Though the cone of normals provides tight normal bounds, its computation is relatively costly. An approximate cone of normals can be computed at low cost by combining a tangent and bitangent cone [SM88]. Munkberg et al. [MHTAM10] used this in the context of bounding displaced patches and applied this approach on a modern GPU with hardware tessellation.

They perform the calculations in the constant hull shader and set the tessellation factors of back-facing patches to zero in order to cull them. While they describe a general strategy to bound displaced Bézier patches, it is important to note that their GPU implementation only considers constant displacements (i.e., a constant displacement value per patch). We use a similar idea for our occlusion culling approach in Chapter 12 to deal with displaced patches. However, our bounds are optimized with respect to screen space area, thus provide better culling results. The work of Kumar et al. [KML96] focuses on NURBS models. For each surface patch, they compute a bounding box for the normalized control vectors of the normal patch. At runtime, the vertices of this bounding box are tested against the viewing direction to see if all surface normals point away from the viewer. If so, then the patch is back-facing and culled. This is similar to the cone of normals approach, but does not take into account that rays from the eye to points on the patch may differ from the view direction. A general problem with back-patch culling is the inability to deal with displaced patches. Hasselgren et al. [HMAM09] address this by using a Taylor series to represent the displaced surface. Nevertheless, this has severe limitations (e.g., cannot deal with fractional tessellation) and its culling rate is poor. In Chapter 11 we present a near optimal back-patch culling technique [LNE11] that utilizes the parametric tangent plane.

10.2 Occlusion Culling

In addition to a back-patch culling approach, we also present a technique that performs occlusion culling of patches (see Chapter 12, [NL12]). There exist many occlusion algorithms in the context of polygon rendering. A survey of early methods is provided by Cohen-or et al. [COCSD03].

On modern GPUs, hardware occlusion queries provide information about whether an object contributes to the current frame. In DirectX 11 *predicate* rendering (`GL_NV_conditional_render` in OpenGL) allows conditional rendering without GPU-CPU synchronization; i.e., if an occlusion query is not yet complete until the next conditional draw call, rendering will be performed ignoring the actual query result. There are several methods that efficiently use hardware occlusion queries by reducing the number of issued queries in the context of per object occlusion culling [Sek04], [BWPP04], [GBK06], [MBW08]. However, all these algorithms share several problems that makes their application on a per patch level inefficient: they require separate draw calls for each cull primitive (this severely affects performance since thousands of separate draw calls could be necessary to render a single object); spatial hierarchies are required on the CPU side to limit the number of issued queries (updates become costly under animation); rasterizing bounding geometry creates additional overhead (latency and compute).

Engelhardt and Dachsbacher [ED09] propose two methods for granular visibility queries that make query results available on the GPU: pixel counting with summed area tables and hierarchical item buffers. The first method assigns query objects to color channels and uses summed area tables to count covered pixels. Thus, only four query objects per region are supported, which is insufficient considering thousands of patches whose bounds overlap particularly when considering displacements. Hierarchical item buffers write IDs of query objects to resulting pixels. The resulting buffer is interpreted as a point list and in a second (count) renderpass a vertex shader distributes points (i.e., query IDs) to pixels accordingly. With alpha blending enabled the number of covered pixels can be obtained for each query object. While this might be possible for a larger number of query objects, rendering a single point for each pixel (i.e., in practice over a million points) seems to be significant overhead. In addition, on-the-fly bounding geometry computation and rasterization remains a problem.

We base our occlusion culling approach on the hierarchical Z-buffer proposed by Greene et al. [GKM93]. They use an object-space octree of the scene geometry and a screen space Z-pyramid (Hi-Z map). The pyramid's lowest level is the Z-buffer; each higher level is constructed by combining four Z values into a Z value at the next lower level by choosing the farthest Z value from the observer. Then the cubes of the octree are tested against the best fitting entry in the Hi-Z map. Shopf et al. [SBOT08] use the Hi-Z map to perform culling based on geometry instances. As an extension to the original hierarchical Z approach, they use bounding spheres and four

Hi-Z map samples to obtain better coverage. However, in their approach occluders (i.e., the terrain) are fixed and occlusion culling is only applied to selected objects (i.e., characters). Since they use geometry instances all tested objects (must) have exactly the same topology and obtaining the cull decisions involves a CPU query.

In contrast to previous occlusion culling methods, our approach shown in Chapter 12 supports fully animated objects and does not require any precomputed scene data structures. Furthermore, we are able to cull subsets of objects since our algorithm works at the patch level; there are no static occluder lists, so all objects can act as occluders or be occluded.

CHAPTER 11

Effective Back-Patch Culling for Hardware Tessellation

11.1 Introduction and Algorithm Overview

On current hardware back-facing triangles can be culled to avoid unnecessary rasterization and pixel shading. However, if the plane normals of all generated triangles for a given patch point away from the viewer, considerable amounts of computation are wasted for surface evaluation and triangle setup. In this chapter we explore the feasibility and performance of back-patch culling. That is, we perform a culling test on entire patch primitives earlier in the pipeline, to avoid computations that happen before rasterization.

This is not a new idea, but previous to our approach [LNE11] only low order approximations have appeared [SAE93, KML96]. This is likely due to the limited compute resources of previous generation graphics processors. As pointed out by Kumar and Manocha [KM96], patch culling is a trade-off between *efficiency*: how much computational effort is needed to reach a culling decision, and *effectiveness*: how many patches are actually culled.

We present a novel approach, based on the *parametric tangent plane* of a patch to accurately partition the space of eyepoint positions into *front-facing*, *back-facing*, or *silhouette* regions. As shown in Figure 11.1, this is more effective than previous methods.

Operating in clip space simplifies our test considerably, and all steps vectorize very well. Our computation times are comparable to existing approaches, but the increased effectiveness gives us a performance advantage as patch tessellation density increases.

In addition to back-patch culling, we believe that accurate visibility classification of patches has other potentially useful applications.

To sum up, our contributions of our approach are:

- Classification between front-facing, back-facing and silhouette patches
- Effective culling: best cull ratio compared to other algorithms
- SIMD efficient GPU implementation

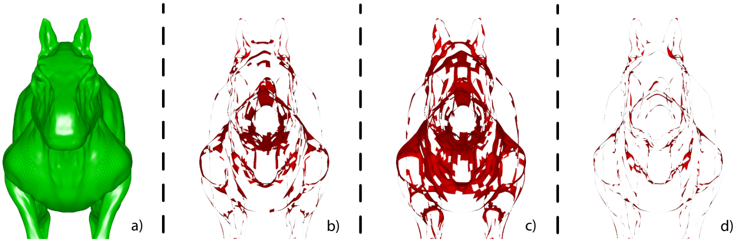


Figure 11.1: The *Killeroo* (a, 11532 Bézier patches), is rendered with different strategies for back-patch culling. For each algorithm we visualize wasted computations: areas processed by the tessellator but back-facing and hence not visible; less area is better. The cone of normals (b, 3697 patches culled) is effective, but costly for dynamic scenes. Its approximation from tangent and bitangent cones is faster to compute, but less precise (c, only 2621 patches culled). Our approach is faster than the cone of normals, and more effective (d, 4604 patches culled).

11.2 Parametric Tangent Plane

In this section, we develop the key geometric concepts behind our patch culling algorithm and introduce the *parametric tangent plane*. We work with homogeneous vectors in $\mathbb{R}^4$, and maintain a distinction between *points*, represented by row vectors, and *planes* represented by column vectors. We note the distinct transformation rules

$$\mathbf{q} = \mathbf{p} \cdot \mathbf{P} \qquad \text{and} \qquad \mathbf{s} = \mathbf{P}^{-1} \cdot \mathbf{t}$$

for points **p** and **q**, and planes **s** and **t**, given the 4×4 transformation matrix **P**.

We focus on widely used bicubic Bézier patches, but extending our ideas to other polynomial patch types could be done in a similar fashion. For instance our approach could be easily integrate into the feature adaptive subdivision method presented in Chapter 4.

A (rational) bicubic Bézier patch is defined by

$$B(u,v) = \mathsf{B}^3(u) \cdot \begin{bmatrix} \mathbf{b}_0 & \mathbf{b}_1 & \mathbf{b}_2 & \mathbf{b}_3 \\ \mathbf{b}_4 & \mathbf{b}_5 & \mathbf{b}_6 & \mathbf{b}_7 \\ \mathbf{b}_8 & \mathbf{b}_9 & \mathbf{b}_{10} & \mathbf{b}_{11} \\ \mathbf{b}_{12} & \mathbf{b}_{13} & \mathbf{b}_{14} & \mathbf{b}_{15} \end{bmatrix} \cdot \mathsf{B}^3(v),$$

where $u, v \in [0,1]^2$, $\mathsf{B}_i^d(t) = \binom{d}{i}(1-t)^{d-i}t^i$ are the degree d Bernstein basis functions, and $\mathbf{b}_j \in \mathbb{R}^4$ are homogeneous 3D control points. The parametric tangent plane $T(u,v)$ of $B(u,v)$ satisfies

$$\begin{bmatrix} B(u,v) \\ \frac{\partial}{\partial u}B(u,v) \\ \frac{\partial}{\partial v}B(u,v) \end{bmatrix} \cdot T(u,v) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

We can compute $T(u,v)$ directly as

$$T(u,v) = \text{cross4}\left(B(u,v), \tfrac{\partial}{\partial u}B(u,v), \tfrac{\partial}{\partial v}B(u,v)\right), \qquad (11.1)$$

where *cross*4() is the generalized cross product of 3 vectors in $\mathbb{R}^4$, see Section 11.4.1. For bicubic $B(u,v)$, the parametric tangent plane is a polynomial of bidegree 7 and can be written in Bézier form as

$$T(u,v) = \mathbf{B}^7(u) \cdot \begin{bmatrix} \mathbf{t}_0 & \mathbf{t}_1 & \cdots & \mathbf{t}_6 & \mathbf{t}_7 \\ \mathbf{t}_8 & \mathbf{t}_9 & \cdots & \mathbf{t}_{14} & \mathbf{t}_{15} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{t}_{48} & \mathbf{t}_{49} & \cdots & \mathbf{t}_{54} & \mathbf{t}_{55} \\ \mathbf{t}_{56} & \mathbf{t}_{57} & \cdots & \mathbf{t}_{62} & \mathbf{t}_{63} \end{bmatrix} \cdot \mathbf{B}^7(v),$$

where the $\mathbf{t}_i$ form an 8×8 array of *control planes*. Each $\mathbf{t}_i$ results from a weighted sum of $cross4()$ products among the control points of $B(u,v)$.

Note that $T(u,v)$ being of bidegree 7 is less by one in both parametric directions than expected from adding the polynomial degrees of inputs to equation (11.1). This is easily verified with symbolic algebra software, but can be traditionally proven using properties of DeCastlejau's algorithm [FH00]. We have not seen the object that we are calling the parametric tangent plane formally defined or used in previous work. We acknowledge the strong probability that the parametric tangent plane has a classical definition, but we have not found one.

11.3 Visibility Classification

We use the generic term *visibility* here to mean that a point on an oriented surface can be seen from a given eyepoint. We do not consider the effects of occlusion, self or otherwise. Our goal is to classify entire surface patches, with respect to a given eyepoint, as front-facing, back-facing, or silhouette. We first note an optimization that will significantly reduce the computational cost of our algorithm; we do this in the context of familiar triangle culling.

11.3.1 Triangle Culling

Given a triangle defined by points $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{v}_2$, its oriented spanning plane is $\mathbf{t} = \text{cross4}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2)$. We say that triangle $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ cannot be seen from eyepoint $\mathbf{e}$, if $\mathbf{e}$ lies in the negative half-space defined by $\mathbf{t}$. We express this as a dot product, if $\mathbf{e}\cdot\mathbf{t} < 0$ then triangle $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ is *back-facing*. Conversely, if $\mathbf{e}\cdot\mathbf{t} > 0$ we say that triangle $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ is *front-facing*. Otherwise if $\mathbf{e}\cdot\mathbf{t} = 0$ then triangle $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ appears edge-on; we classify such triangles as *silhouette*.

Note that this visibility classification does not depend on a coordinate system. Given the composite world, viewing, and perspective transform $\mathbf{P}$ that maps world space to clip space, we can write

$$\mathbf{e}\cdot\mathbf{t} \;=\; \mathbf{e}\cdot\mathbf{I}\cdot\mathbf{t} \;=\; (\mathbf{e}\cdot\mathbf{P})\cdot\left(\mathbf{P}^{-1}\cdot\mathbf{t}\right) \;=\; \mathbf{f}\cdot\mathbf{s},$$

where $\mathbf{f}$ and $\mathbf{s}$ represent the transformations of eyepoint $\mathbf{e}$ and plane $\mathbf{t}$ to clip-space, respectively. By convention $\mathbf{f} = \left[\begin{array}{cccc} 0 & 0 & \alpha & 0 \end{array}\right]$ in clip-space, so that $\mathbf{f}\cdot\mathbf{s} = \alpha\, s_z$, where s_z is the z component of $\mathbf{s}$, and α is of known sign. This means that in clip-space, visibility classification can be done by simply checking the sign of s_z. So instead of computing the plane containing triangle $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ in world space and dotting the result with $\mathbf{e}$, we only need to compute the z component of the plane containing the transformed vertices in clip space. Similarly for patch culling, we only need to compute the clip-space z component of the parametric tangent plane.

11.3.2 Patch Culling

We classify the visibility for a patch $B(u,v)$ using its parametric tangent plane $T(u,v),\;\; u,v \;\in\; [0,1]^2$, with respect to homogeneous eyepoint $\mathbf{e}$ using the *Continuous Visibility* function:

$$\text{CVis}\,(B,\mathbf{e}) = \begin{cases} \text{back-facing,} & \text{if}\,(\mathbf{e}\cdot T(u,v)\;<\;0)\,,\;\forall\; u,v\in[0,1]^2, \\ \text{front-facing,} & \text{if}\,(\mathbf{e}\cdot T(u,v)\;>\;0)\,,\;\forall\; u,v\in[0,1]^2, \\ \text{silhouette,} & \text{otherwise.} \end{cases}$$

A similar viewing space *back-patch condition* appears in [KM96]. Though equivalent, our classification is more general in that it is invariant to projective transformation. Computing CVis $(B,\mathbf{e})$ precisely will require costly iterative techniques to determine the roots of a bivariate polynomial. Instead, we compute a more practical discrete variant, based on the Bézier convex hull of the scalar valued patch

$$\mathbf{e} \cdot T(u,v) \;=\; \mathbf{B}^7(u) \cdot \begin{bmatrix} \mathbf{e}\cdot\mathbf{t}_0 & \mathbf{e}\cdot\mathbf{t}_1 & \cdots & \mathbf{e}\cdot\mathbf{t}_6 & \mathbf{e}\cdot\mathbf{t}_7 \\ \mathbf{e}\cdot\mathbf{t}_8 & \mathbf{e}\cdot\mathbf{t}_9 & \cdots & \mathbf{e}\cdot\mathbf{t}_{14} & \mathbf{e}\cdot\mathbf{t}_{15} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{e}\cdot\mathbf{t}_{48} & \mathbf{e}\cdot\mathbf{t}_{49} & \cdots & \mathbf{e}\cdot\mathbf{t}_{54} & \mathbf{e}\cdot\mathbf{t}_{55} \\ \mathbf{e}\cdot\mathbf{t}_{56} & \mathbf{e}\cdot\mathbf{t}_{57} & \cdots & \mathbf{e}\cdot\mathbf{t}_{62} & \mathbf{e}\cdot\mathbf{t}_{63} \end{bmatrix} \cdot \mathbf{B}^7(v).$$

Patch visibility classification reduces to counting the number of negative values, *Ncnt*, produced by taking the 64 dot products $\mathbf{e}\cdot\mathbf{t}_i$ using the *Discrete Visibility* function:

$$\text{DVis}\,(B,\mathbf{e}) = \begin{cases} \text{back-facing}, & \text{if } (Ncnt = 64)\,, \\ \text{front-facing}, & \text{if } (Ncnt = 0)\,, \\ \text{silhouette}, & \text{otherwise.} \end{cases}$$

It is important to note that the classification produced by $\text{DVis}\,(B,\mathbf{e})$ is a conservative approximation of $\text{CVis}\,(B,\mathbf{e})$: sign differences among the Bézier coefficients are a necessary, but not sufficient condition for determining the presence of a root. Therefore, it is possible for $\text{DVis}\,(B,\mathbf{e})$ to classify a front or back facing patch as silhouette in error. While we can construct such cases, they seem to be rare in practice, and as demonstrated in Section 11.6, we are able to cull significantly more patches than previous techniques.

11.4 Serial Algorithm

Using symbolic algebra software, we expand equation (11.1) for the parametric tangent plane and find its Bézier representation. Each Bézier coefficient $\mathbf{t}_i$ is the result of a weighted sum of *cross*4() products among the control points of the bicubic patch $\mathbf{B}(u, v)$

$$\mathbf{t}_i = \cdots + wgt \cdot \mathrm{cross4}(\mathbf{b}_j, \mathbf{b}_k, \mathbf{b}_l) + \cdots$$

For each of these *cross*4() products, we extract a destination index i, and source indices j, k, and l, as well as the corresponding scalar weight *wgt*. These values will be the same for all bicubic Bézier patches, and we place them in a header file with format

```
uint idx[4][] = {{i1, j1, k1, l1},
                 ...
                  {im, jm, km, lm}};
float wgt[] =     {w1, ..., wm};
```

For the bicubic Bézier case, we require 516 *cross*4() products that can be precomputed according to Section 11.4.1 and stored in a buffer.

As noted earlier, we operate in clip space, and thus only need the z component of the parametric tangent plane. Using the predetermined indices and weights, the serial algorithm to compute this is

```
for (uint k = 0; k < 516; k++)
   t[idx[k][0]] += wgt[k]
                * cross4Z(
                     b[idx[k][1]],
                     b[idx[k][2]],
                     b[idx[k][3]]);
```

where `float4 b[16]` contains the values of the patch control points after transformation to clip space, *cross*4*Z*() computes just the z component of the four-dimensional cross product *cross*4() (see Section 11.4.1), and `float t[64]` will contain the z components of the control planes which are all initialized to zero.

11.4.1 The 4D Cross Product

Our back-patch culling relies on the 4D cross product (see Blinn [Bli03]). It is defined as the function cross4($\mathbf{a}, \mathbf{b}, \mathbf{c}$) $= \begin{bmatrix} x & y & z & w \end{bmatrix}^T$:

$$x = \det \begin{bmatrix} a_y & b_y & c_y \\ a_z & b_z & c_z \\ a_w & b_w & c_w \end{bmatrix}, \qquad y = -\det \begin{bmatrix} a_x & b_x & c_x \\ a_z & b_z & c_z \\ a_w & b_w & c_w \end{bmatrix},$$

$$z = \det \begin{bmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_w & b_w & c_w \end{bmatrix}, \qquad w = -\det \begin{bmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{bmatrix}.$$

Geometrically, $\begin{bmatrix} x & y & z & w \end{bmatrix}^T$ is the oriented plane that spans homogeneous points $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^4$. In addition, $\begin{bmatrix} x & y & z & w \end{bmatrix}^T$ is orthogonal to all three vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$ and has the length of the volume of the parallelotope (an N-dimensional parallelepiped) spanned by $\mathbf{a}, \mathbf{b}, \mathbf{c}$.

11.5 Parallel Algorithm

Creating a parallel version of the parametric tangent plane algorithm is relatively straightforward, though a little care is needed to avoid write hazards. The first observation is, that each weighted *cross*4() product is independent of every other. So our strategy will be to have each thread in a group compute such a weighted *cross*4() product, and add its result to a target location in shared memory.

There are 516 weighted *cross*4() products, but only 64 target locations. Allocating more than 64 threads per patch will guarantee a write hazard, since a single target location will simultaneously be written by more than one thread. Allocating exactly 64 threads would not be efficient, since the distribution of weighted *cross*4() products is non-uniform over the 64 target locations, so many threads will idle after only a few computations. This distribution is illustrated in the matrix below:

$$\left[\begin{array}{cccc|cccc}
1 & 2 & 4 & 5 & 5 & 4 & 2 & 1 \\
2 & 4 & 8 & 10 & 10 & 8 & 4 & 2 \\
4 & 8 & 11 & 17 & 17 & 11 & 8 & 4 \\
5 & 10 & 17 & 21 & 21 & 17 & 10 & 5 \\
\hline
5 & 10 & 17 & 21 & 21 & 17 & 10 & 5 \\
4 & 8 & 11 & 17 & 17 & 11 & 8 & 4 \\
2 & 4 & 8 & 10 & 10 & 8 & 4 & 2 \\
1 & 2 & 4 & 5 & 5 & 4 & 2 & 1
\end{array}\right].$$

Each entry of this matrix shows the number of times the corresponding target location is accessed by the 516 weighted *cross*4() products. By partitioning the target locations into 4 × 4 blocks as shown above, and summing the blocks we get the following much more uniform distribution

$$\begin{bmatrix}
32 & 33 & 33 & 32 \\
33 & 31 & 31 & 33 \\
33 & 31 & 31 & 33 \\
32 & 33 & 33 & 32
\end{bmatrix}$$

This suggests a strategy where we allocate 16 threads per patch, and each thread is responsible for the 4 corresponding target locations of the 8 × 8 target array. Each thread will need to loop 33 times, compute a weighted *cross*4() product, and add the result to a target shared memory location. After reordering the array elements within `idx` and `wgt` according to this load distribution, the parallel code to compute the parametric tangent plane looks like

```
// ceil(516 / 16) = 33 iterations max.
for (uint k = threadIdx; k < 516; k += 16)
   t[idx[k][0]] += wgt[k]
                 * cross4Z(
                     b[idx[k][1]],
                     b[idx[k][2]],
                     b[idx[k][3]]);
```

As in the serial variant of our approach (see Section 11.4), we assume that the array `b[16]` contains the patch control points after transformation to clip space. This means that we only need to compute the z component of $cross4()$. The final step is counting the signs of `t[64]` using a simple parallel reduction strategy on a per patch level.

11.6 Results and Discussion

To evaluate our approach we extend the SimpleBezier example from the DirectX 11 SDK. As real world applications will spend additional resources to determine tessellation factors, or construct tangent patches and evaluate those, this serves as a lower bound for the performance gains expected due to the better cull precision. We are mainly interested in dynamic surfaces, and hence only use the Bézier control points as input for the cull tests for each frame.

In contrast to Munkberg et al. [MHTAM10], we implement our cull tests using DirectX 11 compute shaders, and feed the decision into the constant hull shader using a small texture. This gives us more flexibility and is considerably faster, as the constant hull shader seems to execute only a single thread per patch and multiprocessor. Also, the performance difference between tangent cone [SM88] (TCone) and normal cone [SAE93] (NCone) is much less dramatic than reported by Munkerberg et al. [MHTAM10]; we attribute this difference to a combination of implementation details and more recent hardware. For TCone and NCone we use 1 thread per patch and 128 patches per block. For ours we use 16 threads per patch and 8 patches per block. Those settings were determined empirically to give the best performance.

The effectiveness of back-patch culling strongly depends on the used model and viewpoint. Our test culls more patches than the previous methods for any view. To quantify the improvement, we determine the number of culled patches for 10K random views, and list the average cull rates for three popular models in Table 11.1.

	Big Guy	Monster Frog	Killeroo
	(3570)	(5168)	(11532)
TCone	1260 (35%)	1911 (37%)	3790 (33%)
NCone	1601 (45%)	2286 (44%)	4685 (40%)
Ours	1729 (**48%**)	2478 (**48%**)	5206 (**45%**)

Table 11.1: Average cull rates for 10K random views. Our method consistently performs best, and culls close to 50% of the patches.

For a particularly challenging view of the *Killeroo*, shown in Figure 11.1, we measure the total time per frame for different tessellation factors, and graph it in Figure 11.2. We need 0.76 ms per frame to cull 4604 patches. This is faster than NCone, which needs 0.86 ms to cull 3697 patches. For tessellation factors larger than 8 the additional cull precision pays off, and our time per frame is lower than with TCone, which needs 0.36 ms, but only culls 2621 patches.

These timings seem counterintuitive, as the arithmetic cost of our algorithm is roughly 10 to 20 times higher than that of NCone and TCone. However, both algorithms require many registers, limiting the number of active blocks per multiprocessor. In our method each individual thread needs few registers, and shared memory is naturally used very efficiently. As result we have much more active threads, but a comparable number of patches per multiprocessor, and hence similar computation times.

Aside from performance and precision advantages, our classification technique is projectively invariant, and we support *rational* bicubic patches directly. Further, by counting *positive* values ($Pcnt$) among the 64 dot products in DVis $(B, \mathbf{e})$ and changing the back-facing condition to: if$(Pcnt = 0)$, we can correctly classify degenerate patches with collapsed edges, e.g., the top of the Utah teapot.

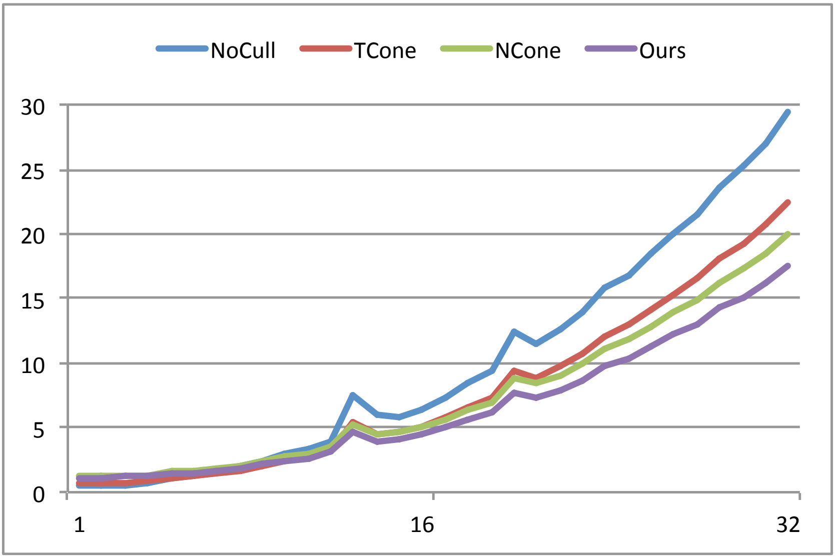


Figure 11.2: Time per frame for Figure 11.1 using different tessellation factors. The rendering times for the other models and views behave similar. Time in ms on an Nvidia GTX 480.

11.7 Conclusion

We presented a novel strategy to cull back-facing patches, to avoid unnecessary work in the hardware tessellator. We demonstrated its feasibility for bicubic Bézier patches, but we are not limited to this patch type. The calculations vectorize very well, and compared to the popular cone of normals approach it is both more effective and more efficient on current hardware. Compared to the fast approximation using tangent and bitangent cones it is about 2x slower, but the better cull rate pays off quickly as tessellation density increases.

In addition to back-patch culling, we feel that our precise visibility classification technique could be useful for other applications as well. One area we plan to explore is better handling of adaptive tessellation for silhouette patches, as these are generally the areas where most over-tessellation occurs.

CHAPTER 12

Patch-Based Occlusion Culling for Hardware Tessellation

12.1 Introduction and Algorithm Overview

In this chapter we present an patch-based occlusion culling approach for hardware tessellation. Previous occlusion culling algorithms, typically being applied to triangle meshes, required a scene graph structure whose leaves bound a sufficiently dense, and static collection of geometry in order amortize their cost. This limited occlusion culling to fixed scene graphs that must be traversed at runtime. Our new patch-based approach is *unstructured* in the sense that we can process standard lists (buffers) of patch primitives and still achieve significant performance gains.

The key observations we leverage are 1) that patches are compact and are (relatively) easy to find bounds for in screen space; and 2) that committing to the evaluation, tessellation, and generation and rejection of triangles, corresponding to a patch is (relatively) expensive. Since the cost of processing patches that are not seen is high, and the cost of finding screen space bounds for patches is low, the extra computation needed to perform occlusion culling is easily amortized. Our algorithm is easy to implement and requires no pre-processing of patch-based models. Since the screen space bounds of individual patches are re-computed every frame, we automatically support animated models. Furthermore, to maximize utility, we allow patches to have displacement maps applied and offer a novel technique for bounding such patches.

The strategy behind our algorithm is to use temporal coherence to maintain the visible/occluded status of individual patches, use the visible patches to build a hierarchical Z-buffer [GKM93] on-the-fly, and then to update the visibility status of patches. Our method is conservative in that occluded patches may occasionally be rendered (needlessly), but visible patches are always rendered. The culling overhead for our algorithm is small compared to the computational savings, resulting in significant performance gains (see Figure 12.1). Note that even for simple scenes (e.g., single objects) and small tess factors our algorithm is effective (see Figure 12.9).

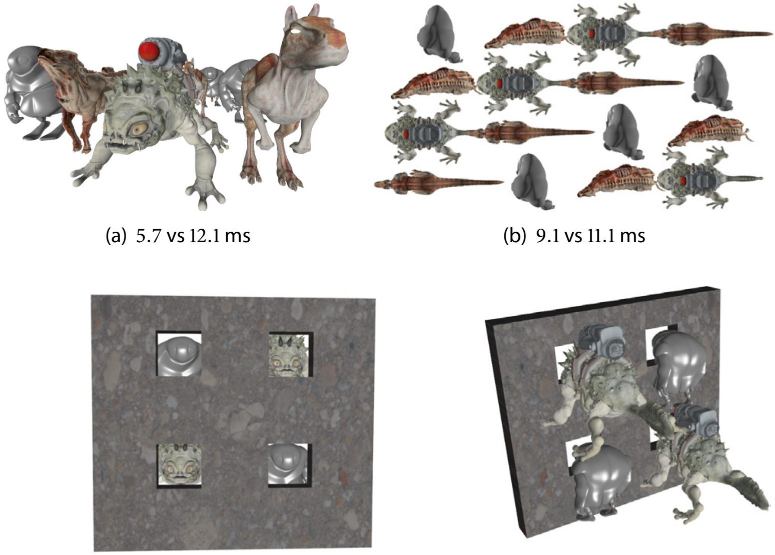
We summarize our main contributions as follows:

- Fast and efficient occlusion culling for patches
- Novel bounds for patches with displacements
- Dynamic scenes with patch occluders and occludees

12.2 Culling Pipeline

Our algorithm works by maintaining visibility status bits (visible, occluded, or newly-visible) of individual patches as each frame is rendered. Assume that a frame has already been rendered and these status bits have been assigned to patches. At the start of a new frame, patches marked as visible are rendered. From the Z-buffer of this frame, a hierarchical Z-buffer (or *Hi-Z map*) is built using a compute shader (or CUDA kernel). Next, all patches are occlusion tested against the newly-constructed Hi-Z map. Patches passing this test (i.e., not occluded) are either marked as visible if they were previously visible, or newly-visible if they were previously occluded; otherwise they are marked as occluded. Finally, all patches marked as newly-visible are rendered to complete the frame. See Figure 12.2 for a flow diagram of this process.

The simplest way to initialize the visibility status bits of patches would be to mark all patches visible. However, this would cause all patches to be rendered in the first frame. To avoid this worst-case behavior, we mark half of the patches as visible and the other half occluded. Even randomly assigning visibility status bits will allow at least some patches to be occluded; which is



(a) 5.7 vs 12.1 ms

(b) 9.1 vs 11.1 ms

(c) 3.4 vs 9.0 ms

(d) 6.7 vs 8.7 ms

Figure 12.1: Our algorithm performs culling on a per patch basis and supports patches w/ and w/o displacements. The images above are rendered both using and not using our culling method; see the performance gain below the respective image. Our method significantly speeds up rendering of scenes with even moderate depth complexity (a, 64.2% culled) including triangle mesh occluders (c, 70.6% culled). Even when having intra-object occlusion only, render time can be reduced (b, 30.4% culled; d, 30.5% culled).

better than none. After the first frame has been rendered, however, we rely on temporal coherence to approximate these visible and occluded sets. Our observation is that between frames, most visible patches stay visible while most occluded patches stay occluded. Obviously, for dynamic scenes some visible patches will become occluded between frames, and vice versa (see Figure 12.9 right). One of the key features of our algorithm is to efficiently track these changes so that each new frame begins with a good approximation to the set of visible patches.

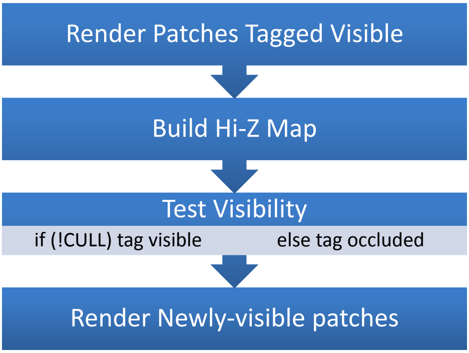


Figure 12.2: Overview of our culling pipeline within a frame: first, patches tagged visible are rendered; the resulting depth buffer is used to construct the Hi-Z map. Next, all patches are tested for visibility against the Hi-Z map. Last, all patches that were previously tagged occluded but are now visible (i.e., newly-visible) are rendered.

While the main focus of our work is on hardware tessellation of animated, patch-based objects such as characters, scene environments are often represented by triangle meshes. We render triangle meshes before generating the Hi-Z map so their depth information is included in the Hi-Z map construction. This will allow triangle meshes to be treated as occluders for patch-based objects. It is also straightforward to combine this per patch algorithm with previous approaches that focus on entire objects. The Hi-Z map information can be used to determine whether an object's bounding volume is fully occluded. However, computing bounding volumes of dynamic objects on-the-fly can be be costly.

12.2.1 Aggressive Culling

The pipeline shown in Figure 12.2 can be even simplified by omitting the last render pass. Updating patch tags will still fix rendering in the subsequent frame. This will cause patches becoming visible to appear with a one frame delay. This may be tolerable in some real-time applications since that delay may not be noticeable at high frame rates. However, we do not evaluate this option in our results.

12.3 Applying Cull Decision

We now describe how to determine patch visibility within our culling pipeline (see Section 12.2). We separate this into the problem of obtaining occlusion information (i.e., Hi-Z map creation) and the culling test itself.

12.3.1 Computing Occlusion Data

As mentioned in Section 12.2, visible patches (i.e., occluders) are rendered first. In order to obtain occlusion information, we employ a hierarchical Z-buffer approach. Therefore, we use the depth buffer resulting from rendering the visible patches to generate a Hi-Z map [GKM93], [SBOT08].

The Hi-Z map construction is similar to standard mip mapping where four texels are combined to determine a single texel of the next mip level. Instead of averaging texels, the value of the new texel is obtained by the maximum depth value of the corresponding four child texels (i.e., it is set to the largest distance value). Thus, within a region covered by a particular texel (no matter which mip level) a conservative bound is given, so that at the texel's location no objects with larger depth values are visible. Five levels of an example Hi-Z map are shown in Figure 12.3. Note that watertight patch joins are crucial for a good Hi-Z map since cracks at patch boundaries will propagate a false depth value through the hierarchy.

We found binding the hardware depth buffer to be relatively costly, see Figure 12.7 right. Though the highest-resolution level of the Hi-Z map could correspond to depth buffer, we avoid copying this data and use a half-resolution image as the highest level. This is reasonable as the full resolution map would only be useful to cull tiny patches. We assume that such tiny patches will not be tessellated with a high enough tess factor to make patch occlusion culling effective; thus the highest level is not needed.

The different Hi-Z levels are stored in a single texture with its respective mip levels to obtain fast access. We deal with non-power-of-two size images by enlarging the Hi-Z map's width and height to the next greater (or equal) power of two. This is necessary since the default size of mip map levels will always be even; e.g., a 5 × 5 texture will be down sampled to 2 × 2. The resized Hi-Z map, however, allows us to take all texels of the parent level into account; e.g., a 5 × 5 texture will be down sampled to 3 × 3. While unused texels are initialized with 0, all kernels can be used without any modification.

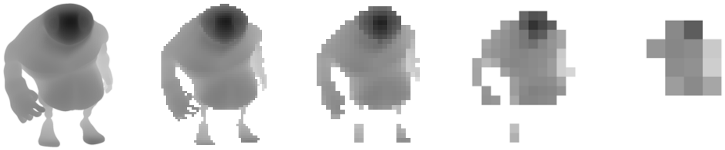


Figure 12.3: Five levels (0,4,5,6,7) of a Hi-Z map corresponding to a view of the *Big Guy* model.

12.3.2 Cull Decision

Cull decisions are applied per patch. As a representative patch primitive we use bicubic Bézier patches consisting of a 4 × 4 array of control points (the basic approach could be applied to various other patching schemes). For now we assume the patches do not have displacement maps applied; we will extend the algorithm to include displacements in Section 12.3.3. In order to determine visibility of a patch, we compute its axis-aligned bounding box (AABB) in clip space. We use the AABB's front plane to test against the Hi-Z map (see Section 12.3.1). Depending on the bounding box's width and height in screen space a particular level of the Hi-Z map is chosen: $level = \lceil \log_2(\max(width, height)) \rceil$. The bounding box's area will be conservatively covered by at most 4 texels of the selected Hi-Z map level. Considering multiple Hi-Z map entries allows us to achieve better coverage; see Figure 12.4 for the distinct Hi-Z access patterns. If the respective depth values of the Hi-Z map are all smaller than the minimum Z value of the patch's bounding box, we set the visibility status bit of the patch to occluded.

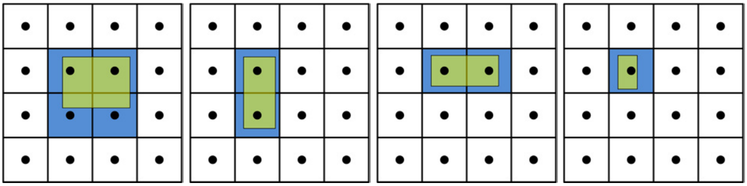


Figure 12.4: Different Hi-Z access patterns; at most four texels (blue) are chosen to cover the screen space bounding rectangle (yellow) of a patch. The case that only one texel is used (right) is special since it only applies at the finest Hi-Z level where no further refinement is possible.

Since we must compute the screen space bounding box of a patch to perform occlusion culling, we can also apply view frustum culling at virtually no cost. That is, patches whose bounding box lies entirely outside of the clipping cube are culled. We clamp bounding box corners to screen space if they are partially outside to ensure that all corners map to Hi-Z values. While we perform both culling methods in the same kernel, view frustum culling is applied first since its computation is less costly.

12.3.3 Displaced Patches

Displacement mapping is an important use case for hardware tessellation as it allows adding high-resolution geometric detail at low cost. We now adapt our algorithm to handle patches with displacements. Although determining the patch bounds is different, creating the Hi-Z map (see Section 12.3.1) is the same for patches with, and without, displacements since we obtain visibility information directly from the depth buffer.

For displaced patches, we use a camera-aligned frustum (CAF) as a bounding volume; that is, a frustum whose side planes contain the origin in camera space. In camera space the eyepoint is at the origin and the viewing direction is the positive Z-axis (in OpenGL it's the negative Z-axis). Unlike the non-displaced case, we do not yet apply the perspective transform since this projective mapping would destroy Euclidean distance metrics that we rely on to construct our bounds.

First, control points are transformed to camera space. We then find the minimum Z value (*minZ*) among the control points to determine the front plane of the CAF (a plane perpendicular to the viewing direction). Next, we project the patch control points onto the CAF's front plane. Since control points P_i are in camera space, we can achieve this by coordinate rescaling: $P_i' = P_i \cdot \frac{minZ}{P_{i,z}}$. Side planes of the CAF are then obtained by computing the minimum and maximum x, y values of the P_i'. Note that the resulting frustum is the generalization of the screen space AABB. Therefore, its screen space projection will give the same result non-displaced patches (Section 12.3.2), if all displacements were zero.

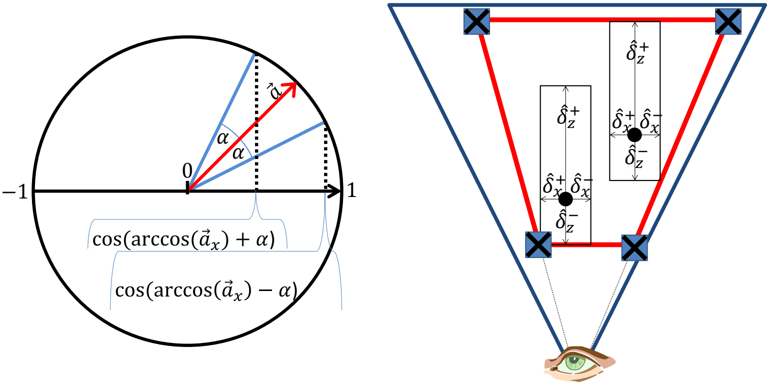


Figure 12.5: The construction of our camera-aligned frustum (CAF). Left; the cone axis $\vec{a}$ and the aperture α determine the extension in the positive x direction (δ_x^+). Right; the CAF (red) is determined in camera space for two control points.

In order to bound the range of patch normals to which displacements are applied, we compute a cone of normals for each patch. This cone is represented by an aperture angle α and cone axis $\vec{a}$. We consider the construction of Shirmun and Abi-Ezzi [SAE93] (accurate cone) or the method of Sederberg and Meyers [SM88] (approximate cone); the trade-off being accuracy versus computational cost. Since obtaining the accurate cone is relatively costly (see Figure 12.7 left), our choice for animated patches will be the approximate version. The accurate cone may be still be used for static objects where the cone can be precomputed.

We also find bounds on the scalar displacement values and require the maximum $D_{\max}$ to be positive (or zero) and the minimum $D_{\min}$ be negative (or zero). The displacement bounds, the cone axis $\vec{a}$, and aperture α, are used to extend the CAF so that it will conservatively bound the displaced patch. For each camera space coordinate, we compute positive and negative extensions as follows (see Figure 12.5 left):

```
if($\vec{a}_x \geq \cos(\alpha)$)
    $\delta_x^+ = 1$
else
    $\delta_x^+ = \max(\cos(\arccos(\vec{a}_x) + \alpha), \cos(\arccos(\vec{a}_x) - \alpha))$

if($-\vec{a}_x \geq \cos(\alpha)$)
    $\delta_x^- = 1$
else
    $\delta_x^- = \max(\cos(\arccos(-\vec{a}_x) + \alpha), \cos(\arccos(-\vec{a}_x) - \alpha))$
```

This allows us to compute a minimum and maximum bound for each control point:

$$\begin{aligned} P_{\max} &= P + \hat{\delta}_x^+ = P + \max(D_{\max} \cdot \delta^+, -D_{\min} \cdot \delta^-) \\ P_{\min} &= P - \hat{\delta}_x^- = P - \max(D_{\max} \cdot \delta^-, -D_{\min} \cdot \delta^+) \end{aligned}$$

$P_{\max}$ and $P_{\min}$ define AABBs for each control point and determine the *minZ* value, i.e., the CAF front plane. In order to construct the CAF, the corner points of all AABBs are projected on the CAF front plane (see Figure 12.5, right). Finally, the CAF is transformed into screen space by projecting the four corner points of its front plane. Visibility for the screen space bounding box is determined the same way as described in Section 12.3.2. Both occlusion and view frustum culling benefit from the tight CAF bounds.

Note that for a given cone of normals the CAF provides an optimal screen space bound. Figure 12.6 shows the difference between our bounds and state-of-the-art previous work using both the same cone.

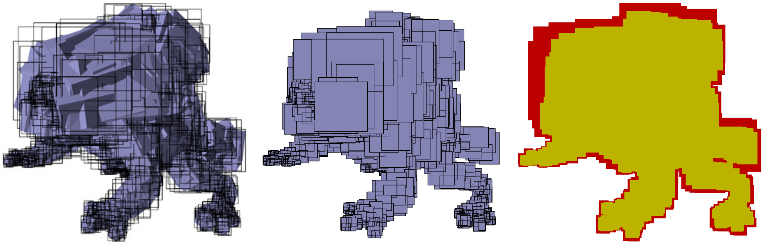


Figure 12.6: Different bounding methods for displaced surfaces visualizing object (purple) and screen space (black quads) bounds: OOBB (object-oriented bounding box [MHTAM10]), CAF (ours) and a comparison between OOBB (red) and CAF (yellow); both using the approximate cone of normals. Our bounds generalize axis-aligned screen space bounds and thus provide optimal results for a given cone of normals.

12.4 Implementation Details

Our algorithm was implemented using DirectX 11 with HLSL shaders. Since patches are tagged either *visible* or *occluded*, a binary flag is sufficient to define visibility status. A second binary flag is required to mark a patch as *newly-visible*, to be rendered in the next draw call. Both flags are combined in a per object texture that contains one value for each patch. Cull decisions are determined in a compute shader that runs one thread per patch, with the results being stored in the previously mentioned texture. This texture is then accessed in the constant hull shader, where patch culling is applied by setting respective tess factors to zero. This turned out to be faster than computing cull decisions directly in the hull shader. We attribute the poor constant hull shader performance to the fact that there is only a single constant hull shader thread running per warp. Since there is no measurable context switch overhead between rendering and compute in DirectX 11, we also use compute shaders to construct the Hi-Z map.

12.5 Results

All experimental results were made using an NVIDIA GeForce GTX 480. Timings are provided in milliseconds and account for all runtime overhead except for display of the GUI widgets, text rendering, etc.. In order to reflect a real application use case, back-face culling is always turned on to explicitly measure computations done by hardware tessellation and costs associated with front-facing fragments.

12.5.1 Cull Computations

Our algorithm requires passes for drawing, determining visibility, and Hi-Z map construction. Figure 12.7 left shows the runtime of the culling kernels for different patch counts. Culling time scales linearly with respect to the number of patches. The non-displaced surface (NoDisp), the simple uniform frustum extension (Full) and the approximate cone of normals (ACoN) kernel are approximately the same cost, while the accurate cone of normals bounding frustum extension kernel (CoN) is about an order of magnitude slower. These results suggest that the ACoN kernel is the best choice for dynamic displaced surfaces, since its slightly lower effectiveness compared to the CoN kernel (see Section 12.5.2), is offset by its significantly lower cost. In fact the CoN kernel is not suitable for animated patches since rendering the patches is cheaper in most cases than the cull kernel execution. The corresponding kernels using the bounds of Munkerberg et al. [MHTAM10] have about the same cost (±3%), however, their cull rate is always lower (see Table 12.1).

The performance of the Hi-Z map construction for various screen resolutions is shown in Figure 12.7 right (the depth buffer is down sampled to a size of 4×4 pixels). A large portion the Hi-Z map computation time is

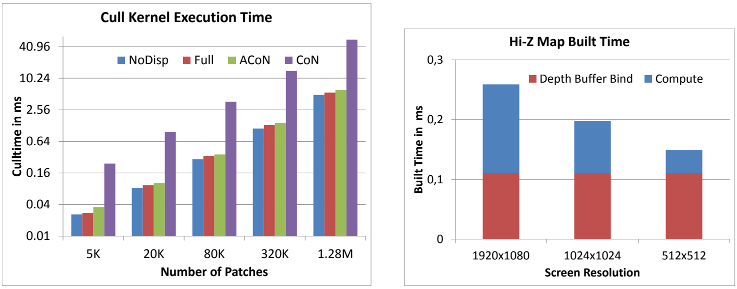


Figure 12.7: Left: execution times for different culling kernels with varying patch count. Having no displacements (NoDisp), uniform bounding volume extension (Full), and considering approximate cone of normals (ACoN) takes about the same amount of time, while using the true cone (CoN) is significantly more expensive. Right: Hi-Z map construction times for different screen resolutions. Timings are split by binding the depth buffer (depth stencil view) and running the Hi-Z build kernel.

consumed binding the current depth buffer. Our speculation is that binding the depth buffer causes a copy operation due to the driver/vendor specific internal depth buffer representation. Further, the Hi-Z map creation is independent of the number of patches and is therefore a constant factor in our culling pipeline.

While our method executes both cull and Hi-Z map construction kernels, the time required by our algorithm for a scene with 80K patches was less than a millisecond. Depending on the cull rates we expect a payoff even at low surface evaluation costs (see Section 12.5.2 and 12.5.3).

12.5.2 Culling within Individual Objects

Since we perform culling on a per patch basis, we can apply our algorithm within individual models. In order to obtain meaningful cull rate measurements, we determine average cull rates by using 1K different cameras views respectively. Each view contains the entire object so that no patches are

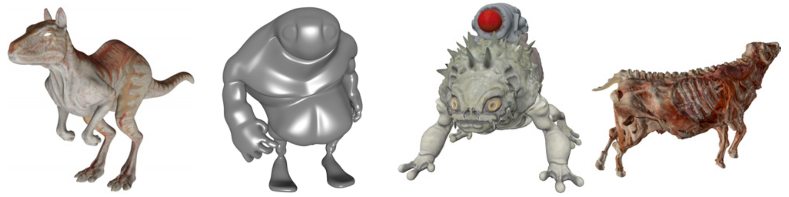


Figure 12.8: Test models: *Killeroo* and *Big Guy* (non-displaced; 2.9K and 1.3K patches); *Monster Frog* and *Cow Carcass* (displaced; 1.3K and 1.2K patches).

view frustum culled. As test objects we use both non-displaced and displaced models.

Non-displaced models:
The non-displaced version of our algorithm is applied on two representative models; *Killeroo* and *Big Guy* (see Figure 12.8 left). Our algorithm achieves an average cull rate of 27.9% and 26.76% respectively compared to our state-of-the-art back-patch culling [LNE11] that culls 38.7% and 37.8% of the patches. However, our method will cull significantly more patches with increased depth complexity (see Section 12.5.3), since back-patch culling algorithms cannot take advantage of inter object/patch occlusions. Note that the per patch computational cost of back-patch culling (for dynamic models) is an order of magnitude higher than ours. The effectiveness of our scheme also increases as patch size is decreased. For instance, if we subdivide (a 1-4 split) *Killeroo* and *Big Guy* patches (5.8K and 11.6K patches) the cull rate increases to 42.5% and 42.9% respectively.

Displaced models:
To test our algorithm on models with displacement maps, we used the *Monster Frog* and the *Cow Carcass* model (see Figure 12.8 right). The culling rates using different cull kernels are depicted in Table 12.1. As expected, the uniform bounding shape extension (Full) has the lowest cull rate. Taking the approximate cone of normals into account significantly improves the cull rate. Using the accurate cone of normals (CoN) provides an additional improvement. However, the difference between ACoN and CoN is smaller than between Full and ACoN.

	OOBB			CAF		
	Full	ACoN	CoN	Full	ACoN	CoN
Frog	10.5%	12.1%	14.0%	14.4%	17.0%	18.4%
Frog^2	17.1%	25.1%	26.4%	22.8%	29.4%	30.9%
Cow	10.1%	14.1%	15.6%	14.5%	17.6%	18.7%
Cow^2	17.8%	27.9%	29.1%	24.0%	31.2%	32.7%

Table 12.1: Average cull rates for displaced models; ^{2} denotes the respective model after one level of subdivision (i.e., having four times more patches). While OOBB is our method using the bounds by Munkerberg et al. [MHTAM10], CAF stands for our camera-aligned frustum.

Furthermore, the camera-aligned frustum (CAF) is more effective than the object-oriented bounding box (OOBB) due to its better screen space bounds. Since the cost of the respective kernels is equal, we conclude that it is always better to use the CAF.

Note that these tests used simple models with a relatively small number of patches. As shown in Table 12.1, an increased patch count yields a significantly higher cull percentage. This results from better bounds on the displacement scalars and smaller patch sizes (small patches are more likely to be occluded).

12.5.3 General Culling

Culling for scenes:
In Section 12.5.2 we considered culling within individual objects. However, more realistic applications involve scenes with multiple objects consisting of both triangle and patch meshes. Two simple example scenes are shown in Figure 12.1 (the ACoN kernel is used for displaced models). The first scene contains 27K patches and we achieve cull rates of 64.2% and 30.4% for the views shown in Figures 12.1(a) and 12.1(b) respectively. Rendering is sped up by a factor of 2.1 and 1.2, respectively (using a tess factor of 16). As expected, the higher the depth complexity, the more patches can be culled. Our method also benefits from triangle mesh occluders as shown in our second test scene (5.5K patches and 156 triangles). We achieve cull rates

of 70.6% and 30.5% for the views shown in Figures 12.1(c) and 12.1(d). Render time is reduced by a factor of 2.6 and 1.3 (using a tess factor of 32). This demonstrates our algorithm's viability for game levels containing animated characters rendered by hardware tessellation within triangle mesh environments. In such a scenario per object culling methods would not be suitable since computing the bounding box geometry of an object on-the-fly is costly and ineffective (looping over all control points is required). Hierarchical approaches (e.g., clustering patches) would be also inappropriate since hierarchy updates create significant overhead.

Payoff:
In real applications the efficiency of a culling algorithm will determine when culling becomes beneficial. This depends on patch evaluation cost, scene composition, and viewing position. By quantifying patch evaluation cost, we can provide a general statement (i.e., independent of a specific scene composition) about the cull rate needed to amortize the cost of our method. For this analysis we use relatively inexpensive bicubic Bézier patches; more expensive to evaluate patches will benefit even more from culling.

Let RT represent the time it takes to render all patches assuming no culling. Let RT_0 represent the time it takes to render all patches with tess factor set to zero; this would be the draw time if all patches were culled. Finally, let CT represent the time it takes to perform the culling tests. We do not include the cost of Hi-Z map generation since it is constant and becomes negligible for a moderately large number of patches. For a cull rate of x, our culling pipeline (see Section 12.2) will have the following patch related costs: a draw call that consists of rendering the non-culled patches ($(1 - x) \cdot RT$) including the culled patches with tess factor set to zero ($x \cdot RT_0$); plus the cull test time (CT); plus the cost of rendering all patches with a tess factor of zero (RT_0) since patch rendering is applied in two passes. Note that the cost of newly-visible patches is accounted for in the RT term. In order to determine the breakeven point (i.e., when culling becomes a win), we compare the render time for our culling method (LHS) to that of rendering without culling (RHS) in the following equation:

$$(1 - x) \cdot RT + x \cdot RT_0 + CT + RT_0 = RT$$

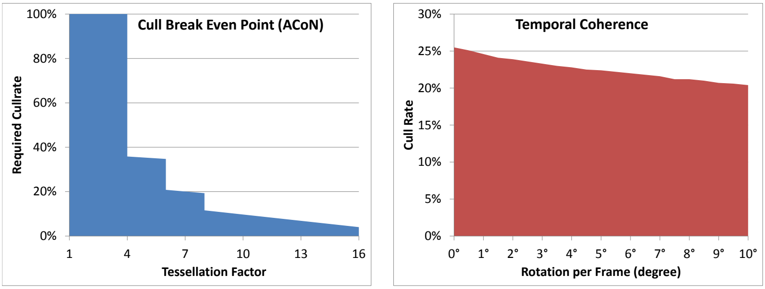


Figure 12.9: Left: required cull rates to make culling beneficial for different tess factors using the ACoN kernel. Culling becomes a win above a tess factor of 4 at a cull rate of 35.8%. Right: average cull rates using the *Big Guy* model for different speeds of rotation around the z-axis. Even for a large temporal change, our algorithm's cull rate is only slightly affected.

Solving for x provides the breakeven point:

$$x = \frac{RT_0 + CT}{RT - RT_0}$$

Since we can measure RT at a given tess factor as well as RT_0, and CT for bicubic patches, we can create a graph for the culling breakeven points (see Figure 12.9 left). The graph plots the resulting cull rate x as a function of tess factor using the ACoN cull kernel. Graphs for the NoDisp and Full kernel are almost equal to that of ACoN. The CoN kernel, however, does not pay off until a tess factor of 6 with a cull rate of over 60%. Since the cull rate of CoN is only slightly better, but its costs are an order of magnitude higher than ACoN, the ACoN kernels generally provides better performance. An application for CoN would be static (non-animated) objects where the cone of normals can be precomputed.

Temporal Coherence:
In order to measure the effects of reduced temporal coherence on our algorithm, we rotate the *Big Guy* model around its z-axis using different speeds of rotation (see Figure 12.9 right). We choose rotations for this test as it

is worst-case in terms of violating temporal visibility coherence. Even for large view point changes, the cull rate is only slightly affected. In our example the cull rate drops from 25.5% (no movement) to 20.4% (rotation at a speed of 10° per frame). While we expect extreme scene motion to reduce cull rates, we note that due to the conservative nature of our algorithm, no visible patches will be missed.

Adaptive tessellation:
An important advantage of the hardware tessellation pipeline is the ability to assign tess factors to individual patches dynamically. While this can reduce tessellation costs significantly, an additional constant amount of per patch computation is required to determine these tess factors. Within the context of our culling method, this means that we can avoid tess factor computations for culled patches. However, since adaptive tessellation tends to assign smaller tess factors to potentially culled patches, there is less room for saving tessellation costs. In order to evaluate the culling efficiency of our algorithm when using adaptive tessellation, we use a simple camera distance based tess factor estimate (similar to the DetailTessellation11 sample of the DirectX SDK). For the scene and camera setup of Figure 12.1(a) we achieve roughly the same visual quality at render times of 3.8 ms (w/o culling) and 2.6 ms (w/ culling; ACoN), respectively. Thus, our method reduces render time by about 31.6%. Please note that more sophisticated adaptive tess factor estimates (e.g., curvature based) would favor our algorithm even more due to the extra tess factor computation costs. For the same setup using the view of Figure 12.1(b), where we have only intra-object occlusion (cull rate of 30.4%), our culling method does not pay off; i.e., about 9% slower than w/o culling. In order to avoid such a situation where our method's overhead is not amortized by savings, we suggest disabling culling for selected models. This decision can be made for each model independently. For instance, one could compute a single adaptive tess factor per model (e.g., model centroid to eye distance); if that tess factor is less than a particular threshold (see Figure 12.9 left for payoff), do not apply culling. Per patch tess factors can be still assigned afterwards to non-culled patches.

Another option for adaptive tessellation is to compute a uniform tess factor for each model at runtime. That would allow balancing costs and gains for each model individually, and provide a decision whether or not to apply culling.

12.6 Conclusion

We have presented a simple, yet effective algorithm for culling occluded bicubic patches for hardware tessellation. We used bicubic Bézier patches due to their simplicity and popularity. However, our method can be applied with different types of patches such as PN-Triangles [VPBM01], Gregory patches [LSNC09] or bicubic B-spline patches obtained from a Catmull-Clark subdivision mesh [NLMD12]. All that we require are methods to bound patch geometry, and first partial derivatives (to determine an approximate cone of normals).

Our results show that our culling method performs well on current hardware involving only minimal overhead. Thus, culling is effective even for simple scenes (e.g., single objects) and small tess factors (see Figure 12.9). In addition, our patch-based culling algorithm can easily be combined with previous per object occlusion culling methods that are applied on triangle meshes. We believe that our method is ideally suited for real-time applications that leverage the advantages of hardware tessellation.

PART IV

Collision Detection

CHAPTER 13

Introduction

Hardware tessellation allows efficient rendering of smooth surfaces, including subdivision surfaces, by processing patch primitives in parallel as shown in Chapters 4, 5, 8. On top of a smooth base surface displacement maps can be added to provide high-frequency geometric detail. Surfaces can be animated by updating only the patch control points while displacement values remain constant. Another benefit is that the patch tessellation is set at computed at runtime. That allows the realization of various level-of-detail schemes such as shown in Chapter 4.7 and 8.4. However, all these advantages make collision detection a challenging task since geometry is generated on-the-fly by the GPU. Transferring geometry to the CPU would involve significant memory I/O and is not feasible in real-time applications. Thus, traditional collision detection approaches that maintain a hierarchy of proxy primitives are too costly and provide only loose results. To the best of our knowledge, providing a satisfactory collision detection scheme for hardware tessellation has not previously been done. Therefore, we present an approach in Chapter 15 that performs collision detection for hardware tessellation entirely on the GPU [NSSL13].

CHAPTER 14

Previous Work

Collision detection:
An essential part of physics simulations (for instance in games [Mil07]) is the ability to detect collisions. Typically, a hierarchy of proxy primitives is maintained in order to facilitate efficient pairwise collision tests. Surveys of traditional approaches are shown in [JTT01] and [Eri04].

Bounding and culling displaced patches:
A key feature of our approach presented in Chapter 15 is to reduce the number of possibly colliding patches using patch-based culling. Therefore, patch normals need to be bound. This can be achieved by computing an accurate [SAE93] or approximate [SM88] cone of normals. Munkerberg et al. [MHTAM10] and our patch-based culling algorithm depicted in Chapter 12 make use of the approximate variant since resulting quality is similar but computational costs are an order of magnitude smaller. While we use the same normal bounds, the application of collision detection is different since we need to cull against a shared volume given as a bounding box.

Real-time voxelization:
A binary voxelization is a memory efficient representation to distinguish between empty and occupied space. It can be obtained in different ways on modern GPUs in real-time [DCB*04], [ED06], [ED08]. More recent approaches also provide for conservative voxelizations [SS10]. However, a conservative method does not fit our needs since it would be computationally too expensive. Instead, our collision detection algorithm relies on the more practical solid binary voxelization proposed by Schwarz [Sch12]. In contrast to his approach, we do not voxelize closed meshes; therefore we modify the approach accordingly.

CHAPTER 15

Collision Detection for Hardware Tessellation

15.1 Introduction and Algorithm Overview

In this chapter we present a novel approach that performs collision detection for hardware tessellation. This is particularly challenging since surface geometry is based on dynamic tessellation factors and displacement values. We tackle this problem by considering the geometry that is actually used for rendering. The first step of our approach is to test two objects for collision by testing their oriented bounding boxes (**OBBs**) for intersection. If there is no intersection, there can be no collision (early exit); otherwise we compute the intersection of the OBBs. Next, we determine all patches that lie within this shared volume by performing an inclusion test. Included patches are then voxelized into identical grids, one for each object. Finally, the resulting voxelizations of both objects are compared to determine collisions. We can also determine collision positions and their normals based on these voxelizations. We provide an extension of our method that will report colliding patch IDs and u, v coordinates. These can be used to evaluate object patches at collision points for accurate physics handling.

For simplicity, we demonstrate our approach using bicubic Bézier patches. However, our algorithm can be applied to any patching scheme that provides patches whose derivatives can be bound; e.g., [LSNC09], [NLMD12]. An overview of our algorithm pipeline of is shown in Figure 15.1. Our experimental results show that a collision test between two objects consisting of thousands of patches can be performed in less than a millisecond (see Section 15.1). This is well within the processing budget of real-time

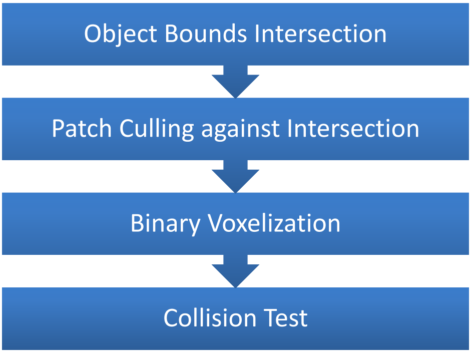

Object Bounds Intersection
Patch Culling against Intersection
Binary Voxelization
Collision Test

Figure 15.1: Overview of our collision detection approach.

applications such as video games. While we currently use our own rudimentary physics simulation for demonstration, our approach could be also integrated into any physics engine [Mil07], [BB07].

To sum up, our approach allows for

- real-time collision detection for hardware tessellation,
- supports animated objects with displacements.

15.2 Collision Candidates

A crucial part of our approach is to first reduce the number of potentially colliding patches. Therefore, we first consider the collision of the respective object bounds. Since we use bicubic Bézier patches we are able to obtain bounds using the convex hull property. Note, we could also use other surface types such as B-splines, subdivision surfaces, or triangle meshes. We compute the OBBs by applying principal component analysis (**PCA**) on the patch control points [Jol05].

For the collision test between two objects we then determine the intersec-

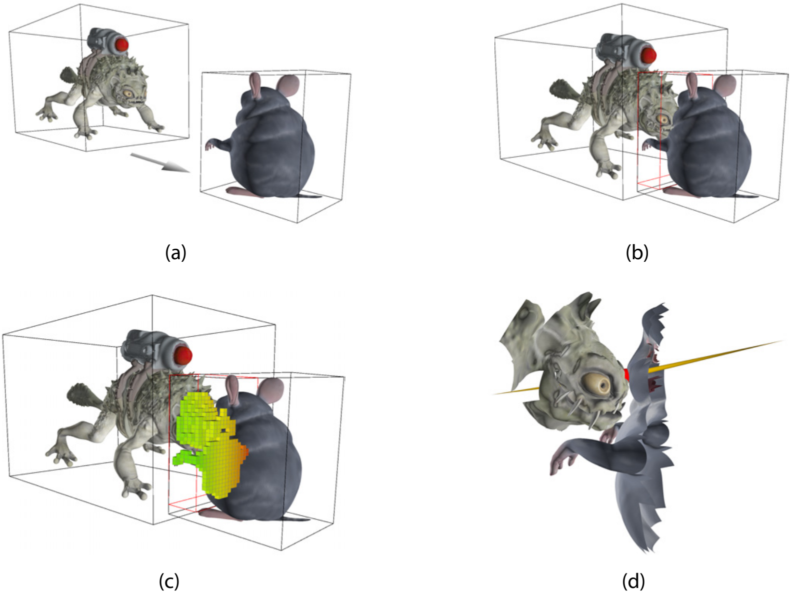


Figure 15.2: Simple test setup visualizing our approach with the *Monster Frog* moving towards the *Chinchilla* (a). At one point the OBBs of both objects intersect (b, red box) and we voxelize the containing geometry (c). This allows us to determine the collision point and corresponding surface normals (d). Patches shown in the last image could not be culled against the intersecting OBB and thus are potential collision candidates contributing to the voxelization.

tion of the two corresponding OBBs. If there is no intersection, there is no collision (early exit). Otherwise, we obtain a set of contact points by intersecting the faces of one box with the edges of the other and vice versa. OBB corner points that are inside the other box are also considered to be contact points. We then compute a new OBB based on the obtained contact points; again by using PCA. In some cases we may need to enlarge the intersecting OBB in order for our binary voxelization to work properly (see Section 15.3). For at least one pair of opposite faces, one face must be outside of one input object OBB and the opposite face must be outside of the other OBB (see Figure 15.3). We select the OBB axis that minimizes this enlargement.

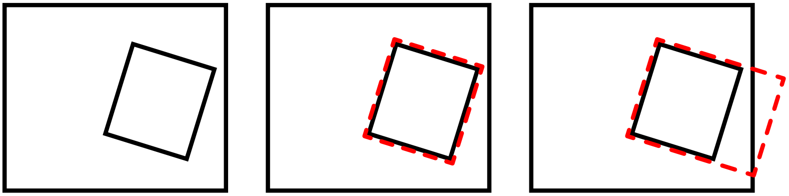


Figure 15.3: OBBs of two objects (black boxes, left) and joint volume (dotted red box). Extension of joint volume to ensure that one face of is outside of the OBB (right).

A compute kernel is used to determine patches that are included in the intersecting OBB in parallel. Each thread processes one patch and computes its OBB. This is done by transforming patch control points into the space of the intersecting OBB; in that space the OBB is the unit cube. As a result we obtain axis-aligned bounding boxes (**AABB**) in the respective OBB space per patch. If an AABB is outside of the unit cube a patch can be culled. However, we also need to incorporate displacements into the patch AABB computation. For each patch we compute a cone of normals that bounds the patch normals. Therefore, an exact [SAE93] or approximate [SM88] but also conservative method can be taken into account. We rely on the approximate variant since its computation is an order of magnitude less expensive and provides similar results. Given the cone axis $\vec{a}$ and cone aperture α we enlarge the bounding box of a patch according to the minimum D_{min} and maximum D_{max} displacement value; similar to our occlusion culling approach (see Section 12.3.3):

$$
\begin{array}{ll}
\mathit{if}(\vec{a}_x \geq \cos(\alpha)) & \delta_x^+ = 1 \\
\mathit{else} & \delta_x^+ = \max(\cos(\arccos(\vec{a}_x)+\alpha), \cos(\arccos(\vec{a}_x)-\alpha)) \\
\\
\mathit{if}(-\vec{a}_x \geq \cos(\alpha)) & \delta_x^- = 1 \\
\mathit{else} & \delta_x^- = \max(\cos(\arccos(-\vec{a}_x)+\alpha), \cos(\arccos(-\vec{a}_x)-\alpha))
\end{array}
$$

This allows the modification of patch AABBs:

$$\begin{aligned} AABB_{\max} &= AABB'_{\max} + \max(D_{\max} \cdot \delta^{+}, -D_{\min} \cdot \delta^{-}) \\ AABB_{\min} &= AABB'_{\min} - \max(D_{\max} \cdot \delta^{-}, -D_{\min} \cdot \delta^{+}) \end{aligned}$$

We precompute displacement extrema $D_{\min}$ and $D_{\max}$ per patch since displacements are considered to be static.

Note that it would be feasible to orient patch bounding boxes based on patch normals [MHTAM10]. However, we obtain better results if both patch and object intersecting OBB share the same axes. In the end, we are able to identify all patches that are within the common bounding volume intersection. Only those patches need to be considered for collision detection; we mark those by maintaining a flag list that contains a single binary value per patch.

15.3 Voxelization

The key idea of our collision test is to voxelize the rendering geometry of the current frame. For two objects those patches lie within the intersection of the objects' bounding boxes (see Section 15.2). We then setup an orthogonal camera matrix that spans that space and perform a solid binary voxelization. Therefore, we use a modified version of the algorithm proposed by Schwarz [Sch12]. The voxelization is performed within the rasterization pipeline and can be applied to any object with or without displacements. Since DirectX 11 (or OpenGL 4.0 or above) allows scattered memory writes in the pixel shader there is no need for a render target. In addition, no depth buffer is required as all fragments contribute to the voxelization. The voxel grid is represented as a linear buffer of 32 bit integer values allowing us to store 32 voxels per entry. In the pixel shader we compute the voxel index depending on the x, y and depth value. For each fragment atomic XOR-operations are used to flip all voxels behind that fragment. Since this is an integer operation, we process 32 voxels per instruction. In the end, only voxels within the volume will remain set resulting in a solid voxelization.

In contrast to the binary voxelization by Schwarz, our potentially colliding patches may not form a closed surface. Therefore, we construct the intersecting OBB to have at least one face that lies completely outside of the original object OBB (see Section 15.2). Thus, we can fill voxels towards the opposite face. We use two different kernels to perform the solid voxelization and fill voxels backwards or forward, respectively. Pseudo code of our forward and backward voxelization kernels is shown below:

```
//Forward solid voxelization
addr = p.x * stride.x + p.y * stride.y + (p.z >> 5) * 4;
atomicXor(voxels[addr], 0xffffffff << (p.z & 31));
for (p.z = (p.z | 31) + 1; p.z < gridSize.z; p.z += 32)
{
   addr += 4;
   atomicXor(voxels[addr], 0xffffffff);
}
```

```
//Backward solid voxelization
addr = p.x * stride.x + p.y * stride.y + (p.z >> 5) * 4;
atomicXor(voxels[addr], ~(0xffffffff << (p.z & 31)));
for (p.z = (p.z & (~31)); p.z > 0; p.z -= 32)
{
   addr -= 4;
   atomicXor(voxels[addr], 0xffffffff);
}
```

Note that back-face culling must be turned off for the voxelization. Performance scales linearly with the number of patches that need to be voxelized making patch culling as described in Section 15.2 essential. The resolution of the voxel grid is adaptive and computed based on the size of the intersecting OBB. However, we require the resolution in Z-direction to be a multiple of 32 to align with four byte integer values.

The use of a solid instead of a surface voxelization is essential. It prevents from missing collisions where objects entirely penetrate a surface within one time step. In addition, a resulting voxelization that is close to the original mesh can be obtained with a single render pass. In some cases we may loose one voxel width around the visual hull due to non-conservative rasterization.

15.4 Collision Detection

We perform collision detection based on binary voxelizations as shown in Section 15.3. Our basic approach is to determine collision positions and corresponding normals based on the voxelization. In addition, we propose an extended variant that provides patch IDs and u, v coordinates of collision points.

15.4.1 Basic Collision Test

Determining collisions given two solid voxelizations that occupy the same space can be obtained by performing pairwise voxel comparisons. There is a collision if equivalent voxels are set. Since our voxel representation is binary, we can perform 32 voxel comparisons using a single bitwise AND-operation of the two corresponding integer values. Therefore, we use a compute kernel with one thread for each integer value of the linear voxel buffer (each value contains 32 binary voxel entries). In addition to collision positions, we obtain corresponding normals based on voxel neighborhoods. The normal is determined by using a weighted average of the vectors between the current voxel and its 26 neighbors; if a voxel neighbor is not set, the corresponding vector is excluded from the average computation. Collisions are written into a GPU buffer using atomics that can be accessed from the CPU.

15.4.2 Extended Collision Test

The extended collision test first executes the basic variant. Then another pass is used to obtain patch IDs and u, v coordinates of collision points.

Therefore, we use an orthographic camera setup that conservatively contains the intersecting OBB and points in the direction of the average approximate collision normal obtained from the basic collision test. Thus, a maximum number of fragments is generated near collision points. Passing patch IDs and u, v coordinates to the pixel shader allows us to store these in a global linked list with an atomic counter (i.e., append buffer in DirectX 11). This allows us to accurately evaluate surface geometry on the CPU at all collision points and to determine corresponding attributes such as surface normals. In order to obtain collision attributes for both colliding objects, we must perform this test twice; once for each object while testing against the voxelization of the other object, respectively.

15.5 Results

Our implementation uses the DirectX 11 graphics pipeline and compute shaders. Performance measurements were made using an NVIDIA GeForce GTX 480 and an Intel Core i7 at 2.80 GHz. In our examples we use the same tessellation density for collision detection that is used for rendering.

Figure 15.2 shows a simple test scene with two objects. The *Monster Frog* (w/ displacements) is moving (a) towards the *Chinchilla* until we detect a collision (b). We also visualize the voxelization that we use for collision computation (c). In addition, patches that could not be culled against the intersecting OBB and the obtained collision point with its corresponding surface normals is shown (d). We used a relatively low voxel grid resolution for visualization purposes in this figure. For all our tests we use a volume of 128^3 adapted anisotropically to the intersecting OBB requiring no more than 256KB of GPU memory.

In order to validate the practicality of our approach, we implemented a rudimentary physics engine. We perform physics computations based on the collisions and corresponding attributes of our extended collision test. Figure 15.4 shows two simple examples. In the first sequence a torus is falling until it hits its counterpart, then rotating slightly, hitting the other

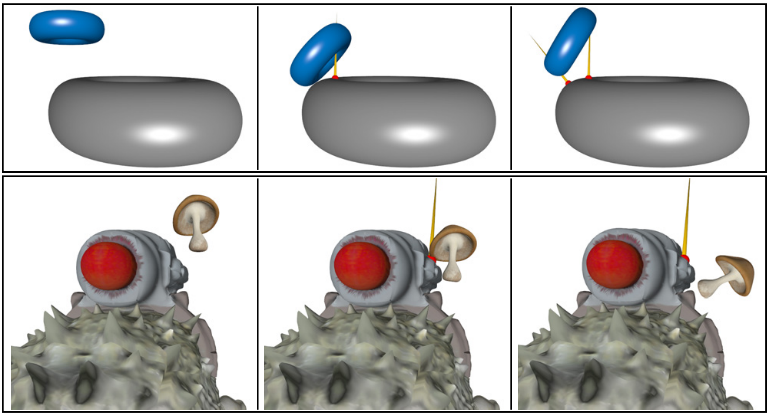


Figure 15.4: Basic physics using our collision detection approach. In each sequence one object is falling onto another.

torus again and bouncing off. In the second sequence a *Mushroom* is falling on the *Monster Frog* model with only a single hit point. Note that there are two normals at collision points (one for each object), however, the visualization of the normals in Figure 15.4 is occluded by the larger objects, respectively.

Performance results of our approach for two setups are shown in Table 15.1. With the presented approach, we are able to perform both the basic and the extended collision test including all overhead in less than a millisecond.

15.6 Conclusion

We have presented a method for real-time collision detection of dynamic hardware-tessellated objects with displacements. Our approach considers the rendering geometry of the current frame and runs entirely on the GPU. We have shown that collision tests can be performed with minimal overhead and that our method can be used for real-time physics. We can

	Frog / Chinchilla		Frog / Mushroom	
Patches	1292 / 4270		1292 / 744	
Tess Factor	4	8	4	8
Draw	0.065	0.169	0.024	0.067
OBB Intersect	0.008	0.008	0.011	0.011
Patch Culling ×2	0.016	0.016	0.012	0.012
Voxelization ×2	0.059	0.092	0.037	0.053
Collision	0.117	0.117	0.114	0.114
Collision Attributes ×2	0.210	0.262	0.222	0.232
Sum Test Basic	**0.275**	**0.341**	**0.223**	**0.255**
Sum Test Extended	0.695	0.865	0.667	0.719

Table 15.1: Performance measurements in milliseconds of our approach for the two test scenes (see Figures 15.2 and 15.4). We provide numbers for rendering (Draw) and for each step of our collision pipeline (OBB Intersect, Patch Culling, Voxelization, Collision, Collision Attributes). In addition, we show the overall performance for a pairwise collision test for both the basic and extended variant of our approach.

perform both basic collision tests and extended collision tests that find attributes at collision positions. We have also shown that the overhead of our approach for objects containing thousands of patches is less than a millisecond. This makes our scheme ideally suited for real-time games allowing physics engines to achieve a realistic behavior.

CHAPTER 16

Conlusion

In this thesis we have presented different techniques for real-time rendering subdivision surfaces (see Chapters 4, 5). We not only provide approaches for rendering the subdivision surfaces but also show how high-frequency detail can be efficiently represented by using a scalar displacement function (see Chapter 8). In addition, our culling techniques increase performance by avoiding unnecessary surface evaluation and shading computations (see Chapters 11, 12). Hardware-tessellated objects can also interact with virtual scene environments using our collision detection approach (see Chapter 15). In the end, our methods provide a comprehensive solution for using hardware tessellation in real-time applications. Those applications range from authoring tools for movies to video games. In particular, we consider our approaches relevant for dynamic objects such as characters. Our techniques also close the gap between movies and games since high-quality subdivision content from movies can now be utilized in games. On the other side interactive authoring tools help artists to create movie content and thus reduce film production costs.

Moreover, we assume that future hardware generations will be similar to current hardware in a sense that there will be even more parallelism. We argue that physical limits of processor clock speed endorse parallel architectures in order to obtain more floating point operations per seconds. That makes keeping caches synchronous challenging and favors compute over memory I/O. Since modern GPUs are such platforms and hardware tessellation optimizes performance on this, we expect our approaches to be even more relevant on future hardware generations.

To sum up, we believe that future generations of video games and authoring tools will benefit from the presented techniques by utilizing massively parallel hardware architectures. For instance, parts of our work have been realized by Pixar as the open source project *Open Subdiv* [Pix12]. As a result of our work, our subdivision surface rendering approach is now being used as a plugin for Autodesk Maya [Auta] and Autodesk Mudbox [Autb].

Outlook

While we have presented techniques for the rendering of displaced subdivision surface, we did not address the creation of suitable content. However, the creation of such specific content requires appropriate algorithms and tools, particularly designed for non-expert users. For instance as outlined in Section 8.2.1, the automated and artifact-free conversion process of polygonal meshes to displaced subdivision surfaces remains an open research problem. In addition, a compelling solution to handle vector displacements on top of subdivision surfaces efficiently would be beneficial. That would facilitate non-uniform sampling of displacements within patch domains and would increase the modeling flexibility over scalar displacements.

Another interesting research area is the rendering of subdivision surfaces on heterogeneous hardware architectures. In particular, mobile devices such as smartphones or tablet computers have become more and more relevant for content consumption and video gaming in the recent years. Providing suitable algorithms that facilitate high-quality rendering on those tile-based rendering architectures with limited computational capabilities is challenging. We also expect that cloud and crowd computing will provide further research opportunities in the context of real-time rendering.

Bibliography

[Auta] AUTODESK: *Maya*. http://usa.autodesk.com/maya.

[Autb] AUTODESK: *Mudbox*. http://autodesk.com/mudbox.

[BB07] BOEING A., BRÄUNL T.: *Evaluation of real-time physics simulation systems*. In Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia (2007), ACM, pp. 281--288.

[Ber04] BERTRAM M.: *Lifting Biorthogonal B-spline Wavelets*. *Geometric Modeling for Scientific Visualization* (2004), 153.

[BFH04] BUCK I., FATAHALIAN K., HANRAHAN P.: *GPUBench: Evaluating GPU performance for numerical and scientific applications*. In Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors (2004), p. 63.

[BL08] BURLEY B., LACEWELL D.: *Ptex: Per-Face Texture Mapping for Production Rendering*. Computer Graphics Forum 27, 4 (2008), 1155--1164.

[Bli78] BLINN J.: *Simulation of wrinkled surfaces*. ACM SIGGRAPH Computer Graphics 12, 3 (1978), 286--292.

[Bli03] BLINN J.: *Lines in space. 1. The 4D Cross Product*. Computer Graphics and Applications, IEEE 23, 2 (2003), 84--91.

[BS02] BOLZ J., SCHRÖDER P.: *Rapid evaluation of Catmull-Clark subdivision surfaces*. In Proceedings of the seventh international conference on 3D Web technology (2002), ACM, pp. 11--17.

[Bun05] Bunnell M.: *Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping.* In GPU Gems 2. 2005, pp. 109--122.

[BWPP04] Bittner J., Wimmer M., Piringer H., Purgathofer W.: *Coherent hierarchical culling: Hardware occlusion queries made useful.* Computer Graphics Forum 23, 3 (2004), 615--624.

[Cat74] Catmull E.: *Subdivision algorithms for the display of curved surfaces.* PhD thesis, The University of Utah, 1974.

[CC78] Catmull E., Clark J.: *Recursively generated B-spline surfaces on arbitrary topological meshes.* Computer-Aided design 10, 6 (1978), 350--355.

[CCC87] Cook R., Carpenter L., Catmull E.: *The Reyes image rendering architecture.* ACM SIGGRAPH Computer Graphics 21, 4 (1987), 95--102.

[Cn08] Castaño I.: *Tessellation of Subdivision Surfaces in DirectX 11.* Gamefest 2008 presentation, 2008. https://developer.nvidia.com/content/tessellation-subdivision-surfaces-directx-11.

[COCSD03] Cohen-Or D., Chrysanthou Y., Silva C., Durand F.: *A survey of visibility for walkthrough applications.* Visualization and Computer Graphics, IEEE Transactions on 9, 3 (2003), 412--431.

[Coo84] Cook R.: *Shade trees.* ACM SIGGRAPH Computer Graphics 18, 3 (1984), 223--231.

[CT97] Coorg S., Teller S.: *Real-time occlusion culling for models with large occluders.* In Proceedings of the 1997 Symposium on Interactive 3D graphics (1997), ACM, pp. 83--ff.

[DC76] Do Carmo M.: *Differential geometry of curves and surfaces*, vol. 1. Prentice-Hall, 1976.

[DCB*04] Dong Z., Chen W., Bao H., Zhang H., Peng Q.: *Real-time voxelization for complex polygonal models.* In Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on (2004), IEEE, pp. 43--50.

[DKT98] DeRose T., Kass M., Truong T.: *Subdivision surfaces in character animation.* In Proceedings of the 25th annual conference on Computer graphics and interactive techniques (1998), SIGGRAPH '98, ACM, pp. 85--94.

[DLG90] Dyn N., Levine D., Gregory J.: *A butterfly subdivision scheme for surface interpolation with tension control.* ACM transactions on Graphics (TOG) 9, 2 (1990), 160--169.

[DS78] Doo D., Sabin M.: *Behaviour of recursive division surfaces near extraordinary points.* Computer-Aided Design 10, 6 (1978), 356--360.

[ED06] Eisemann E., Décoret X.: *Fast scene voxelization and applications.* In Proceedings of the 2006 symposium on Interactive 3D graphics and games (2006), ACM, pp. 71--78.

[ED08] Eisemann E., Décoret X.: *Single-pass GPU solid voxelization for real-time applications.* In Proceedings of graphics interface 2008 (2008), Canadian Information Processing Society, pp. 73--80.

[ED09] Engelhardt T., Dachsbacher C.: *Granular visibility queries on the GPU.* In Proceedings of the 2009 symposium on Interactive 3D graphics and games (2009), ACM, pp. 161--167.

[EL10] Eisenacher C., Loop C.: *Data-parallel micropolygon rasterization.* In Eurographics 2010 Short Papers (2010), The Eurographics Association, pp. 53--56.

[EML09] Eisenacher C., Meyer Q., Loop C.: *Real-time view-dependent rendering of parametric surfaces.* In Proceedings of the 2009 symposium on Interactive 3D graphics and games (2009), pp. 137--143.

[Eri04] Ericson C.: *Real-time collision detection.* Morgan Kaufmann, 2004.

[Far96] Farin G.: *Curves and Surfaces for Computer-Aided Geometric Design: A Practical Code.* Academic Press, Inc., 1996.

[FB88] Forsey D., Bartels R.: *Hierarchical B-spline refinement.* ACM SIGGRAPH Computer Graphics 22, 4 (1988), 205--212.

[FFB*09] Fisher M., Fatahalian K., Boulos S., Akeley K., Mark W., Hanrahan P.: *DiagSplit: parallel, crack-free, adaptive tessellation for micropolygon rendering.* ACM Transactions on Graphics (TOG) 28, 5 (2009), 150.

[FH00] Farin G., Hansford D.: *The essentials of CAGD.* AK Peters, 2000.

[FMM86] Filip D., Magedson R., Markot R.: *Surface algorithms using bounds on derivatives.* Computer Aided Geometric Design 3, 4 (1986), 295--311.

[GBK06] Guthe M., Balázs A., Klein R.: *Near optimal hierarchical culling: Performance driven use of hardware occlusion queries.* In Eurographics Symposium on Rendering 2006 (2006).

[GKM93] Greene N., Kass M., Miller G.: *Hierarchical Z-buffer visibility.* In Proceedings of the 20th annual conference on Computer graphics and interactive techniques (1993), SIGGRAPH '93, ACM, pp. 231--238.

[GP09] González F., Patow G.: *Continuity mapping for multi-chart textures.* ACM Transactions on Graphics (TOG) 28, 5 (2009), 109.

[Gre74] Gregory J.: *Smooth interpolation without twist constraints.*

Computer Aided Geometric Design (1974), 71--87.

[GVSS00] Guskov I., Vidimče K., Sweldens W., Schröder P.: *Normal meshes.* In Proceedings of the 27th annual conference on Computer graphics and interactive techniques (2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., pp. 95--102.

[HAM07] Hasselgren J., Akenine-Möller T.: *PCU: the programmable culling unit.* ACM Transactions on Graphics (TOG) 26, 3 (2007), 92.

[HAMO05] Hasselgren J., Akenine-Möller T., Ohlsson L.: *Conservative rasterization.* In GPU Gems, vol. 2. 2005, pp. 677--690.

[HDD*94] Hoppe H., DeRose T., Duchamp T., Halstead M., Jin H., McDonald J., Schweitzer J., Stuetzle W.: *Piecewise smooth surface reconstruction.* In Proceedings of the 21st annual conference on Computer graphics and interactive techniques (1994), SIGGRAPH '94, ACM, pp. 295--302.

[HKD93] Halstead M., Kass M., DeRose T.: *Efficient, fair interpolation using Catmull-Clark surfaces.* In Proceedings of the 20th annual conference on Computer graphics and interactive techniques (1993), SIGGRAPH '93, ACM, pp. 35--44.

[HLS93] Hoschek J., Lasser D., Schumaker L.: *Fundamentals of computer aided geometric design*, vol. 1. AK peters Wellesley, MA, 1993.

[HMAM09] Hasselgren J., Munkberg J., Akenine-Möller T.: *Automatic pre-tessellation culling.* ACM Transactions on Graphics (TOG) 28, 2 (2009), 19.

[HMC*97] Hudson T., Manocha D., Cohen J., Lin M., Hoff K., Zhang H.: *Accelerated occlusion culling using shadow frusta.* In Proceedings of the thirteenth annual symposium on Computational geometry (1997), ACM, pp. 1--10.

[Jol05] Jolliffe I.: *Principal component analysis.* Wiley Online Library, 2005.

[JTT01] Jiménez P., Thomas F., Torras C.: *3D collision detection: a survey.* Computers & Graphics 25, 2 (2001), 269--285.

[KM94] Krishnan S., Manocha D.: *Global Visibility and Hidden Surface Removal Algorithms for Free Form Surfaces.* Technical Report TR94-063, Department of Computer Science, University of North Carolina, 1994.

[KM96] Kumar S., Manocha D.: *Hierarchical visibility culling for spline models.* In Graphics Interface (1996), no. 1996, Citeseer.

[KMDZ09] Kovacs D., Mitchell J., Drone S., Zorin D.: *Real-time creased approximate subdivision surfaces.* In Proceedings of the 2009 symposium on Interactive 3D graphics and games (2009), ACM, pp. 155--160.

[KMGL99] Kumar S., Manocha D., Garrett W., Lin M.: *Hierarchical back-face computation.* Computers & Graphics 23, 5 (1999), 681--692.

[KML96] Kumar S., Manocha D., Lastra A.: *Interactive display of large NURBS models.* Visualization and Computer Graphics, IEEE Transactions on 2, 4 (1996), 323--336.

[Kob00] Kobbelt L.: *sqrt(3)-subdivision.* In Proceedings of the 27th annual conference on Computer graphics and interactive techniques (2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., pp. 103--112.

[LMH00] Lee A., Moreton H., Hoppe H.: *Displaced subdivision surfaces.* In Proceedings of the 27th annual conference on Computer graphics and interactive techniques (2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., pp. 85--94.

[LNE11] Loop C., Niessner M., Eisenacher C.: *Effective back-patch culling for hardware tessellation.* In Proceedings of Vision, Modeling and Visualization (2011), pp. 263--268.

[Loo87] Loop C.: *Smooth subdivision surfaces based on triangles.* Master's thesis, University of Utah, 1987.

[LS08] Loop C., Schaefer S.: *Approximating Catmull-Clark subdivision surfaces with bicubic patches.* ACM Transactions on Graphics (TOG) 27, 1 (2008), 8.

[LSNC09] Loop C., Schaefer S., Ni T., Castano I.: *Approximating subdivision surfaces with Gregory patches for hardware tessellation.* 151.

[M*09] Munshi A., et al.: *The opencl specification. Khronos OpenCL Working Group 1* (2009), l1--15.

[MBW08] Mattausch O., Bittner J., Wimmer M.: *CHC++: Coherent hierarchical culling revisited.* Computer Graphics Forum 27, 2 (2008), 221--230.

[MHTAM10] Munkberg J., Hasselgren J., Toth R., Akenine-Möller T.: *Efficient bounding of displaced Bézier patches.* In Proceedings of the Conference on High Performance Graphics 2010 (2010), Eurographics Association, pp. 153--162.

[Mic09] Microsoft Corporation: *Direct3D 11 Features*, 2009. http://msdn.microsoft.com/en-us/library/ff476342(VS.85).aspx.

[Mil07] Millington I.: *Game physics engine development.* Morgan Kaufmann Pub, 2007.

[MM02] Moule K., McCool M.: *Efficient bounded adaptive tessellation of displacement maps.* In Graphics Interface (2002), Citeseer, pp. 171--180.

[MNP08] Myles A., Ni T., Peters J.: *Fast parallel construction of* *smooth surfaces from meshes with tri/quad/pent facets.* Computer Graphics Forum 27, 5 (2008), 1365--1372.

[Mor01] Moreton H.: *Watertight tessellation using forward differencing.* In HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware (New York, NY, USA, 2001), ACM, pp. 25--32.

[MYP08] Myles A., Yeo Y., Peters J.: *GPU conversion of quad meshes to smooth surfaces.* In Proceedings of the 2008 ACM symposium on Solid and physical modeling (2008), ACM, pp. 321--326.

[Nas87] Nasri A.: *Polyhedral subdivision methods for free-form surfaces.* ACM Transactions on Graphics (TOG) 6, 1 (1987), 29--73.

[NL12] Niessner M., Loop C.: *Patch-Based Occlusion Culling for Hardware Tessellation.* In Computer Graphics International (2012).

[NL13] Niessner M., Loop C.: *Analytic Displacement Mapping using Hardware Tessellation.* ACM Transactions on Graphics (TOG) 32, 3 (2013), 26.

[NLG12] Niessner M., Loop C., Greiner G.: *Efficient Evaluation of Semi-Smooth Creases in Catmull-Clark Subdivision Surfaces.* In Eurographics 2012 Short Papers (2012), The Eurographics Association, pp. 41--44.

[NLMD12] Niessner M., Loop C., Meyer M., Derose T.: *Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces.* ACM Transactions on Graphics (TOG) 31, 1 (2012), 6.

[NSG12] Niessner M., Sturm R., Greiner G.: *Real-time simulation and visualization of human vision through eyeglasses on the GPU.* In Proceedings of the 11th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and its

Applications in Industry (2012), ACM, pp. 195--202.

[NSS10] Niessner M., Schäfer H., Stamminger M.: *Fast indirect illumination using Layered Depth Images.* The Visual Computer (Proceedings of CGI 2010) 26, 6-8 (2010), 679--686.

[NSSL13] Niessner M., Siegl C., Schäfer H., Loop C.: *Real-time Collision Detection for Dynamic Hardware Tessellated Objects.* In Eurographics 2013 Short Papers (2013), The Eurographics Association, p. to appear.

[Nvi08] Nvidia C.: *Programming guide*, 2008.

[NYM*08] Ni T., Yeo Y., Myles A., Goel V., Peters J.: *GPU smoothing of quad meshes.* In Shape Modeling and Applications, 2008. SMI 2008. IEEE International Conference on (2008), IEEE, pp. 3--9.

[PCK04] Purnomo B., Cohen J., Kumar S.: *Seamless texture atlases.* In Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing (2004), ACM, pp. 65--74.

[PEO09] Patney A., Ebeida M., Owens J.: *Parallel view-dependent tessellation of Catmull-Clark subdivisionsurfaces.* In Proceedings of the Conference on High Performance Graphics 2009 (2009), ACM, pp. 99--108.

[Pix05] Pixar Animation Studios: *The RenderMan Interface version 3.2.1*, 2005. (https://renderman.pixar.com/products/-rispec/index.htm).

[Pix12] Pixar: *Open Subdiv*, 2012. http://graphics.pixar.com/opensubdiv.

[Rei97] Reif U.: *A Refineable Space of Smooth Spline Surfaces of Arbitrary Topological Genus.* Journal of Approximation Theory 90, 2 (1997), 174--199.

[RNLL10] Ray N., Nivoliers V., Lefebvre S., Lévy B.: *Invisible Seams*. Computer Graphics Forum 29, 4 (2010), 1489--1496.

[SAE93] Shirmun L., Abi-Ezzi S.: *The cone of normals technique for fast processing of curved patches*. Computer Graphics Forum 12, 3 (1993), 261--272.

[SBOT08] Shopf J., Barczak J., Oat C., Tatarchuk N.: *March of the Froblins: simulation and rendering massive crowds of intelligent and detailed creatures on GPU*. In ACM SIGGRAPH 2008 classes (2008), ACM, pp. 52--101.

[SCF*04] Sederberg T., Cardon D., Finnigan G., North N., Zheng J., Lyche T.: *T-spline simplification and local refinement*. ACM Transactions on Graphics (TOG) 23, 3 (2004), 276--283.

[Sch12] Schwarz M.: *Practical Binary Surface and Solid Voxelization with Direct3D11*. In GPU Pro3: Advanced Rendering Techniques. AK Peters Limited, 2012, p. 337.

[Sek04] Sekulic D.: *Efficient occlusion culling*. In GPU Gems. 2004, pp. 487--503.

[SG99] Sudarsky O., Gotsman C.: *Dynamic scene occlusion culling*. Visualization and Computer Graphics, IEEE Transactions on 5, 1 (1999), 13--29.

[Shr10] Shreiner D.: *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*, vol. 1. Addison-Wesley Professional, 2010.

[SJP05] Shiue L., Jones I., Peters J.: *A realtime GPU subdivision kernel*. ACM Transactions on Graphics (TOG) 24, 3 (2005), 1010--1015.

[SKS13] Schäfer H., Keinert B., Stamminger M.: *Real-time Local Displacement using Dynamic GPU Memory Management*. In Proceedings of the Conference on High Performance

Graphics 2013 (2013), ACM, pp. 63--72.

[SKU08] Szirmay-Kalos L., Umenhoffer T.: *Displacement Mapping on the GPU-State of the Art.* Computer Graphics Forum 27, 6 (2008), 1567--1592.

[SM88] Sederberg T., Meyers R.: *Loop detection in surface patch intersections.* Computer Aided Geometric Design 5, 2 (1988), 161--171.

[Spe08] Spencer S.: *ZBrush Character Creation: Advanced Digital Sculpting.* Sybex, 2008.

[SPM*12] Schäfer H., Prus M., Meyer Q., Süssmuth J., Stamminger M.: *Multiresolution attributes for tessellated meshes.* In Proceedings of the 2012 symposium on Interactive 3D graphics and games (2012), ACM, pp. 175--182.

[SS09] Schwarz M., Stamminger M.: *Fast GPU-based Adaptive Tessellation with CUDA.* Computer Graphics Forum 28, 2 (2009), 365--374.

[SS10] Schwarz M., Seidel H.: *Fast parallel surface and solid voxelization on GPUs.* ACM Transactions on Graphics (TOG) 29, 6 (2010), 179.

[Sta98] Stam J.: *Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values.* In Proceedings of the 25th annual conference on Computer graphics and interactive techniques (1998), SIGGRAPH '98, ACM, pp. 395--404.

[SWG*03] Sander P., Wood Z., Gortler S., Snyder J., Hoppe H.: *Multi-chart geometry images.* In Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing (2003), ACM, pp. 146--155.

[SZD*98] Schröder P., Zorin D., DeRose T., Forsey D., Kobbelt L., Lounsbery M., Peters J.: *Subdivision for modeling and animation. ACM SIGGRAPH Course Notes 12* (1998).

[TBB10] Tatarchuk N., Barczak J., Bilodeau B.: *Programming for Real-Time Tessellation on GPU. AMD whitepaper 5* (2010).

[TS91] Teller S., Séquin C.: *Visibility preprocessing for interactive walkthroughs.* ACM SIGGRAPH Computer Graphics 25, 4 (1991), 61--70.

[VHB87] Von Herzen B., Barr A.: *Accurate triangulations of deformed, intersecting surfaces.* ACM SIGGRAPH Computer Graphics 21, 4 (1987), 103--110.

[VPBM01] Vlachos A., Peters J., Boyd C., Mitchell J.: *Curved PN triangles.* In Proceedings of the 2001 Symposium on Interactive 3D graphics (2001), ACM, pp. 159--166.

[VZ01] Velho L., Zorin D.: *4--8 Subdivision.* Computer Aided Geometric Design 18, 5 (2001), 397--427.

[Wil83] Williams L.: *Pyramidal parametrics.* ACM SIGGRAPH Computer Graphics 17, 3 (1983), 1--11.

[WKP11] Wittenbrink C., Kilgariff E., Prabhu A.: *Fermi GF100 GPU architecture.* Micro, IEEE 31, 2 (2011), 50--59.

[YKH10] Yuksel C., Keyser J., House D.: *Mesh colors.* ACM Transactions on Graphics (TOG) 29, 2 (2010), 15.

Rendern von Unterteilungsflächen mittels Hardware Tessellierung

Zusammenfassung

Computergenerierte Bilder werden immer wichtiger im alltäglichen Leben. Dabei benötigen zunehmend realistischer werdende Bilder mehr und mehr wahrnehmbares Detail. Die visuelle Qualität der generierten Bilder hängt dabei stark von der Beschreibung der verwendeten Szenengeometrie ab. Speziell in der Filmbranche haben sich dabei Unterteilungsflächen zu einem Industriestandard entwickelt. Während Unterteilungsflächen Grafikdesignern einen hohen Grad an Flexibilität bieten, ist die Bildgenerierung von Szenen mit solchen Oberflächen sehr aufwändig. Daher werden in Filmproduktionen typischerweise teure Hochleistungsrechner mit entsprechend hoher Rechenleistung eingesetzt.

In dieser Arbeit zeigen wir Methoden um hochqualitative Filminhalte in Echtzeitanwendungen auf handelsüblichen Desktop-Computern einzusetzen. Dabei werden moderne Grafikkarten und die zugehörige Hardware Tessellierung verwendet, welche die Oberflächengeometrie dynamisch und basierend auf einzelnen parametrischen Flächenstücken generiert. Der entscheidende Vorteil der Hardware Tessellierung liegt darin, dass die Oberflächengeometrie auf den jeweiligen Berechnungseinheiten berechnet und direkt weiterverarbeitet wird. Dadurch können generierte Polygone direkt und ohne zusätzliche Speicherzugriffe rasterisiert werden. Außerdem lassen sich Objekte sehr einfach animieren, da lediglich die Kontrollpunkte der Flächenstücke bewegt werden müssen. In einem ersten Schritt unseres Verfahrens werden dazu Unterteilungsflächen in einzelne Flächenstücke zerlegt, welche anschließend vom Hardware Tessellierer verarbeitet werden. Diese werden dann entgegen der rekursiven Definition von Unterteilungsflächen direkt ausgewertet. Zusätzlich wird hochfrequentes Oberflächendetail durch eine analytische Verschiebungsfunktion realisiert. Die endgültigen Oberflächenpunkte sowie die zugehörigen Normalen können dann anhand der Verschiebungsfunktion und der darunterliegenden Unterteilungsfläche bestimmt werden. Um die Bildgenerierung zu beschleuni-

gen, stellen wir außerdem Verfahren zur Auswahl sichtbarer Flächenstücke vor, was letztlich dazu beiträgt überflüssige Berechnungen zu vermeiden. Zudem ermöglicht ein Kollisionserkennungsverfahren die Interaktion von hardwaretessellierten dynamischen Objekten in Echtzeit.

Letztendlich stellen wir eine umfassende Lösung bereit, um Unterteilungsflächen in Echtzeitanwendungen einzusetzen. Aktuelle Entwicklungen zeigen, dass dadurch die nächste Generation von Computerspielen und Modellierungssoftware hochdetaillierte Oberflächen darstellen und animieren kann.

Lebenslauf

Name, Vorname:	Nießner, Matthias
Geburtsdatum:	14.05.1986
Geburtsort:	Gunzenhausen
Staatsangehörigkeit:	deutsch
Familienstand:	ledig

09/96 -- 07/05	Gymnasium
10/05 -- 11/09	Studium der Informatik an der Universität Erlangen-Nürnberg
09/06 -- 10/10	System & Netzwerkadministrator (Rommelwood e.V.)
09/07 -- 11/09	Studentische Hilfskraft (versch. Veranstaltungen)
11/09	**Diplom in Informatik**
12/09 --	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Graphische Datenverarbeitung
08/10 -- 10/10	Praktikum bei Microsoft Research, Redmond, USA
08/11 -- 10/11	Praktikum bei Microsoft Research, Redmond, USA
08/12 -- 11/12	Praktikum bei Microsoft Research, Redmond, USA
03/13 -- 05/13	Praktikum bei Microsoft Research, Cambridge, UK
07/13	**Promotion**