State of the Art Report on Real-time Rendering with Hardware Tessellation

H. Schäfer[1] and M. Nießner[2] and B. Keinert[1] and M. Stamminger[1] and C. Loop[3]

[1]University of Erlangen-Nuremberg; [2]Stanford University; [3]Microsoft Research

Abstract

For a long time, GPUs have primarily been optimized to render more and more triangles with increasingly flexible shading. However, scene data itself has typically been generated on the CPU and then uploaded to GPU memory. Therefore, widely used techniques that generate geometry at render time on demand for the rendering of smooth and displaced surfaces were not applicable to interactive applications. As a result of recent advances in graphics hardware, in particular the GPU tessellation unit's ability to overcome this limitation, complex geometry can now be generated within the GPU's rendering pipeline on the fly. GPU hardware tessellation enables the generation of smooth parametric surfaces or application of displacement mapping in real-time applications. However, many well-established approaches in offline rendering are not directly transferable, due to the limited tessellation patterns or the parallel execution model of the tessellation stage. In this state of the art report, we provide an overview of recent work and challenges in this topic by summarizing, discussing and comparing methods for the rendering of smooth and highly detailed surfaces in real-time.

1. Introduction

Today's graphics cards are massively parallel processors composed of up to several thousands of cores [Nvi12a]. While GPUs comprise a vast amount of raw computational power, they are mainly limited by memory bandwidth. In particular, this becomes a bottleneck for real-time rendering techniques where highly detailed surface geometry needs to be updated (e.g., for animation) and rasterized in every frame. In order to tackle this problem, *hardware tessellation* was introduced along with the Xbox 360 [AB06] and the DirectX 11 API [Mic09].

The key idea is to generate highly detailed geometry on-the-fly from a coarser representation. Therefore, meshes are defined as a set of patches, rather than a purely triangle-based representation. At run-time the patches are sent to the GPU's streaming processors, where they are directly refined and rasterized without further memory I/O. Tessellation densities are specified on a per-patch basis, enabling flexible level-of-detail schemes. Further, high-frequency geometric detail can be added on-the-fly by displacing generated vertices. This enables low-cost animations since only input patch control points need to be updated.

Hardware tessellation has gained widespread use in computer games for the display of highly detailed, possibly animated objects. In the animation industry, where displaced subdivision surfaces are the typical modeling and rendering primitive, hardware tessellation has also been identified as a useful technique for interactive modeling and fast previews. Much of the work presented in this report has been incorporated into `OpenSubdiv` [Pix12], an open source initiative driven by Pixar Animation Studios, for use in games and authoring tools. In the near future, hardware tessellation will also be available on mobile devices [Nvi13, Qua13], opening the door for new applications in mobile graphics.

Although tessellation is a fundamental and well-researched problem in computer graphics, the availability of fast hardware tessellation has inspired researchers to develop and significantly advance techniques specifically crafted for hardware tessellation. This includes higher-order surface rendering methods that focus on different patch-based representations able to be processed by the tessellator. In particular, much effort has been devoted to both accurately and approximately rendering subdivision surfaces, which are a modeling standard in the motion picture industry. Hardware tessellation is also ideally suited for displacement mapping, where high-frequency geometric detail is efficiently encoded as image data and applied as surface offsets at run-time. Several approaches for incorporating such high-frequency details on top of smooth surfaces have been developed to date.

DOI: 10.2312/egst.20141037

These include methods for data storage, displacement evaluation, as well as smooth level-of-detail schemes. Additional research focuses on improving performance by avoiding the rendering of hidden patches (i.e., back-patch and occlusion culling). Further techniques deal with patch-based physics interactions such as collision detection for real-time rendering.

In this state of the art report, we contribute a summary of these methods, set them in a proper context, and outline their contributions. In addition, we analyze and compare these approaches with respect to the usability and practicality for different scenarios. The techniques to be covered involve solutions for

- smooth surface rendering,
- low-cost animations and surface updates,
- adaptive level of detail,
- high-frequency detail; i.e., displacements,
- compact, consistent, and efficient texture storage for color and displacement data,
- dynamic memory management for local, on-the-fly texture allocation,
- culling techniques for improved rendering performance,
- tessellation-based collision detection for correct physics.

Previous GPU Tessellation Work Dynamic CPU-based tessellation methods are hardly applicable to real-time rendering, as the tessellated meshes must be transferred to the GPU continuously. As soon as GPUs became programmable, tessellation started being performed directly on the GPU, avoiding costly CPU-GPU data transfers. Vertex shaders made it possible to reposition vertices, so that the evaluation of vertex positions can be moved to the GPU, as long as a direct evaluation of the surface at a particular parameter position is possible. Geometry shaders can perform simple tessellation; however, they usually slow down the pipeline significantly, particularly if a single shader outputs a large number of triangles.

Boubekeur et al. [BS05,BS08] proposed the use of instantiation for tessellation. In their method, simple triangles are tessellated according to a set of tessellation factors, and are kept in GPU memory as so called *tessellation patterns*. The tessellation patterns are then rendered using instantiation and applied to the patches of the base mesh. The requirement to keep patterns for all used tessellation factors results in a limited number of tessellation levels.

Later, methods were presented to adaptively generate tessellation using GPGPU methods. The hierarchical subdivision process is typically mapped to a parallel breadth-first traversal that successively generates smaller patches until they are considered to be fine enough [EML09, PEO09].

Schwarz et al. [SS09] parallelize the process patch-wise using a single thread per patch. This allowed them to use more efficient evaluation methods based on finite differences, but parallelization could only be exploited if a large number of single patches is subdivided and if the subdivision levels do not vary largely. Other GPGPU-based approaches consider contour and silhouette information to perform adaptive mesh refinement while stitching tessellation disparities [BA08], [FFB*09].

2. Hardware Tessellation

2.1. GPU Architectures

Modern graphics processing units (GPUs) are composed of several streaming multiprocessors (SMs) each of which is a vector processing unit. SMs process data chunks in parallel in a single-instruction-multiple-data (SIMD) fashion. The specific implementation of this kind of architecture is vendor-dependent. For example, NVIDIAs Kepler GK110 architecture [Nvi12b] consists of 15 streaming multiprocessors of which each unit features 192 single-precision cores resulting in a total number of 2880 cores. In this architecture threads are dispatched by the streaming multiprocessors' schedulers in groups of 32 parallel threads called warps.

In contrast to conventional CPUs, GPUs spend more die area on computational units rather than on caches. While there is a small amount of shared memory available per SM (64 KB for the GK110) which can be used as L1 cache, most data must be obtained from global GPU memory. Access to global memory is costly given that it introduces high latency. Typically, latency is partially hidden by running a sufficient number of threads simultaneously and issuing computational instructions without requiring any memory accesses.

2.2. Graphics Pipeline and Hardware Tessellation

The typical graphics pipeline on current GPUs consists of five programmable shader stages (see Figure 1). GPUs can be programmed for rendering using the OpenGL or DirectX API. Hardware tessellation has been accessible to developers in DirectX since DirectX 11 [Mic09] and in OpenGL since Version 4.0 [SA12]. In the following, we will use the DirectX nomenclature.

Highly-tessellated meshes result in large memory footprints in GPU memory and are costly to render. Hardware tessellation allows for larger amounts of output polygons since global memory access is only required for a sparse set of input control points. We benchmark this by generating 2 million output polygons on a NVIDIA GTX 780 graphics card using planar grids comparing conventional rendering to hardware tessellation. While conventional rendering using an index and vertex buffer takes about 1.5 ms, rendering the same amount of polygons using hardware tessellation takes only 0.25 ms, i.e., more efficient by a factor of ~ 6. However, enabling hardware tessellation *without* further tessellation of input patches is ineffective, that is, rendering the 2 million triangles with the tessellation unit and treating every triangle as a separate patch primitive with a tessellation factor of 1 is 5 times slower. Therefore, hardware tessellation

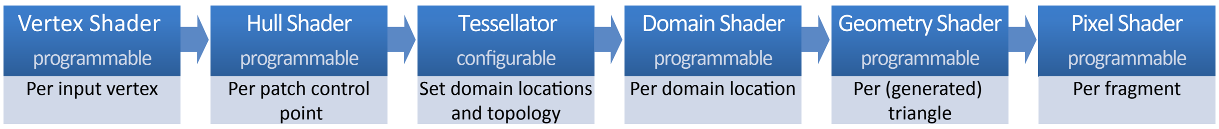


Figure 1: *Graphics pipeline according to DirectX 11 nomenclature involving programmable shader and configurable hardware stages. For the sake of simplicity, the fixed-function stages input assembly, rasterization and output merger are omitted.*

should only be used where required, i.e., when the applications applies further patch tessellation. Hardware tessellation elevates patch primitives to first class objects in the graphics pipeline. These patches are each defined by a set of control points, and processed in parallel by the GPU. Therefore, the tessellation unit generates parameter sample values within corresponding patch domains where patches are evaluated. Currently, triangular, quadrilateral, and isoline domains are supported. Based on the tessellation configuration, patches are evaluated at the sample locations in order to generate an output stream composed of triangles. The key advantage is that these polygons are directly processed by the GPU streaming processors without involving further global memory access, thus minimizing memory I/O. This allows for high-performance, dynamic (potentially dense) patch tessellation.

Hardware tessellation introduces three new pipeline stages between vertex and geometry shading (see Figure 1): the hull shader stage, the tessellator stage and the domain shader stage.

The hull shader stage is divided into two logical parts: a per-patch constant function and the actual hull shader program. The per-patch constant function is executed once per input patch and is used to determine the patch tessellation density. As such, per-patch tessellation factors are computed and sent to the fixed-function tessellator stage in order to specify the amount of generated domain sample points. For tri- and quad-domains there are edge (3 for tris, 4 for quads) and interior (1 for tris, 2 for quads) tessellation factors. The isoline domain is only controlled by two edge tessellation factors that determine the line density and the line detail. Aside from integer tessellation factors, fractional tessellation factors are also available, enabling smooth level-of-detail transitions. The hull shader program is executed for every patch control point. While one thread processes a single (output) control point, all patch input point data is shared among hull shader programs of the same patch. Currently, the number of control points is limited to 32 per patch.

The hull shader stage is followed by the fixed-function tessellator, generating sample points and topology for a given patch domain based on input tessellation factors. Examples of resulting tessellation patterns are shown in Figure 2, including tri- and quad-domains, as well as integer and fractional tessellation factors.

The last stage of the tessellation pipeline is the programmable domain shader stage where a shader program is invoked for each sample location. The input for this stage is composed of the hull shader output (i.e., tessellation factors, control points) and domain sample parameters. These are either barycentric coordinate triples *uvw* (tri-domain), or two-dimensional *uv* coordinates (quad and isoline domains). Based on the input, patches are evaluated, and an output vertex is generated for every domain sample. In addition, per-vertex attributes such as texture coordinates and surfaces normals must be computed and passed along with the positions of emitted vertices. These vertices are then triangulated using the topology generated by the fixed function tessellator and processed by the remaining stages of the graphics pipeline.

The new pipeline additions allow for the dense adaptive tessellation of given input patches. In contrast to previously available geometry shaders, the hardware tessellation stage is significantly faster, more flexible, and allows for much higher tessellation densities.

3. Higher Order Surface Rendering

One of the main purposes of hardware tessellation is to support higher order surface primitives, in particular parametric patches. A parametric patch is a mapping from a unit square or triangle (parameter) domain into 3D space. The precise nature of this mapping is specified by the programmer, both in terms of the assigned meaning of the input data, and the evaluation procedures executed by the hull and domain shader programs.

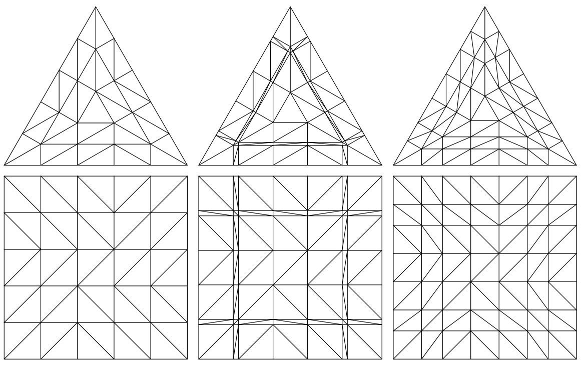


Figure 2: *Generated tessellation patterns of triangle and quad patch domains for different tessellation factors. From left to right: 5, 5.4, 6.6.*

We will present several examples of how hardware tessellation can be used to implement well-known parametric surface types. We start with the simple case of tensor product Bézier patches, and then move on to tensor product B-splines. We consider the subtle problem of *bitwise exact* evaluation in these contexts. This is important when trying to avoid *cracks* between individual patch primitives.

Next, we consider how to use hardware tessellation to render widely used *Subdivision Surfaces* [Cat74, Loo87, DS78]. This is a challenging problem since the piecewise parametric form of a subdivision surface contains an infinite number of patches. We consider why naïve subdivision on the GPU is ineffective. Then, we consider Stam's [Sta98] direct evaluation procedure and explain why it performs poorly on today's GPU architectures. We then move on to several variants of *approximate* subdivision techniques. These are schemes designed to mimic the shape of a subdivision surface without compromising hardware tessellation performance.

Finally, we move on to the state-of-art and consider *exact* subdivision surface rendering using a technique called *feature adaptive subdivision*. We show that this method is nearly as fast as approximate subdivision, but gives the exact limit surface defined by the subdivision surface. This is much more attractive in industry where authoring tools already operate on this limit surface and the expectation is for the rendering algorithm to honor this.

3.1. Bézier Patches

We begin with a tensor product bi-cubic Bézier patch, written

$$P(u,v) \;=\; \sum_{i=0}^{3}\sum_{j=0}^{3} \mathbf{b}_{i,j} B_i^3(u) B_j^3(v),$$

where u, v are the coordinates of a point in a unit square domain $[0,1] \times [0,1]$, $\mathbf{b}_{i,j}$ are 16 three-dimensional *control points* that determine the shape of the patch, and $B_k^3(t)$ are Bernstein polynomials that are defined as

$$B_k^d(t) = \binom{d}{k}(1-t)^{d-k}t^k.$$

The 16 patch control points are transmitted to the GPU in a vertex buffer (either contiguously, or referenced by an index buffer). These vertices are transformed by a *WorldViewProjection* matrix in a vertex, or hull shader program. A hull shader program is executed once for each output vertex (in parallel); if the coordinate transformation is not done here, then the input control points can be passed through to the domain shader.

3.1.1. Avoiding Cracks

Bézier patches have the property that the 4 patch control points along the shared edges between pairs of patches must be identical to maintain a continuous, C^0 surface. In order to avoid cracks in a rendered surface, adjacent patches must be C^0. Furthermore, the tessellation factor must be assigned identically to the shared edge between a pair of adjacent patches; otherwise, the tessellation unit may not sample the shared patch boundary at corresponding parameter values of adjacent patches, leading to cracks. Sharing control point data and assigning identical tessellation factors along shared edges is a necessary condition to avoid cracks, but it is not sufficient. Additionally, the domain shader program must take into account the different ordering of the control points with respect to the two patches sharing an edge. If it does not, then different numerical roundings may accumulate sufficiently to cause output vertices to project to different pixels, resulting in visible cracks between patches. The degree to which this matters is of course application dependent; however, this causes severe artifacts that must be avoided to guarantee high rendering quality.

Fortunately, it is relatively easy to evaluate the Bernstein basis in a *reversal invariant* way; that is, independent of parameter direction. This can be achieved using the following procedure

```
void EvalBezCubic(in float u, out float B[4]) {
   float T = u, S = 1.0 - u;

   B[0] = S*S*S;
   B[1] = 3.0*T*S*T;
   B[2] = 3.0*S*T*S;
   B[3] = T*T*T;
}
```

Note that replacing u by $1-u$ in `EvalBezCubic` interchanges the values of `S` and `T`, leading to the reversal of basis function values. The boundary curve is evaluated by taking the dot product of the 4 boundary control points and these basis function values. Guaranteeing that the results are bitwise identical on both sides of a shared boundary requires commutativity of both addition and multiplication. These commutativity requirements are satisfied by using IEEE floating point strictness when compiling shader programs.

While guaranteeing bitwise identical geometry along shared patch boundaries is fairly straightforward using Bézier patches, guaranteeing bitwise identical normals is not. The problem is that cross-boundary derivatives (e.g., the v direction, if u is along the boundary) may not be computed identically since they are constructed from different (non-shared) input data. Even though the C^1 (tangent plane) continuity conditions for a pair of Bézier patches along a shared boundary can be enforced on a patch-based model, it is unlikely that this condition will be preserved (bitwise identically) by coordinate transformations. This will result in slightly different normal vectors along a shared patch boundary. When used for lighting, the small color differences that might result may not be a problem. However, when used for displacement mapping, these differences will likely lead to visible cracks along patch boundaries. These problems are much easier to avoid by using the B-spline basis.

3.2. B-spline Patches

The B-spline basis has deep theoretical roots far beyond the scope of this report; we give B-splines only superficial treatment here. B-splines are a special case of the more general Non-Uniform Rational B-splines (NURBS), that are uniform and polynomial (non-rational).

A tensor product bi-cubic B-spline surface can be written as a piecewise mapping from a planar domain $[0, m+1] \times [0, n+1]$

$$P(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} \mathbf{d}_{i,j} N^3(u-i) N^3(v-j),$$

where the $\mathbf{d}_{i,j}$ are a rectangular array of B-spline control points, and $N^3(t)$ is a cubic B-spline basis function. $N^3(t)$ is a C^2 smooth, piecewise cubic polynomial curve comprised of 4 non-zero curve segments. Since these basis functions are C^2 (curvature continuous), a linear combination of them is also C^2. This makes constructing a curvature continuous surface easy with cubic B-splines.

Each individual cubic curve segment is determined by 4 contiguous control points, and each bi-cubic B-spline patch by 16 control points. A pair of adjacent curve segments will have 3 control points in common. Similarly, a pair of adjacent patches will have 12 control points in common. These 12 shared control points are exactly the ones needed to construct the positions and normals along the shared boundary. In addition, the B-spline basis function and its derivatives must be computed in a reversal invariant fashion. This can be done in a fashion similar to `EvalBezCubic` presented previously.

A B-spline basis function can be written as a linear combination of scaled copies of itself over a refined parameter domain. By replacing $N^3(t)$ with this linear combination of scaled copies, and finding the control points with respect to the scaled copies, the B-spline control mesh is *refined*. The control points of the new (subdivided) patches can be written as a weighted average of the old control points. These weighted averages are called the bi-cubic B-spline *subdivision rules*.

3.3. Catmull-Clark Subdivision Surfaces

Catmull-Clark subdivision [CC78] is a generalization of bi-cubic B-spline subdivision to irregular control meshes; that is, a mesh with faces and vertices not incident on 4 edges (so-called *extraordinary* vertices). By repeating the subdivision process, a smooth limit surface is obtained as shown in Figure 3.

The algorithm is defined by a simple set of (smooth) subdivision rules, which are used to create new face points (f_j), edge points (e_j) and vertex points (v_j) as a weighted average of points of the previous level mesh.

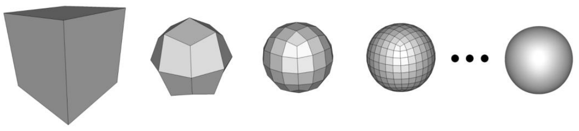


Figure 3: *Several iterations of Catmull-Clark subdivision applied to a cube-shaped mesh.*

3.3.1. Subdivision Rules

The Catmull-Clark smooth subdivision rules for face, edge, and vertex points, as labeled in Figure 4, 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: $v^{i+1} = \frac{n-2}{n} v^i + \frac{1}{n^2} \sum_j e_j^i + \frac{1}{n^2} \sum_j f_j^{i+1}$.

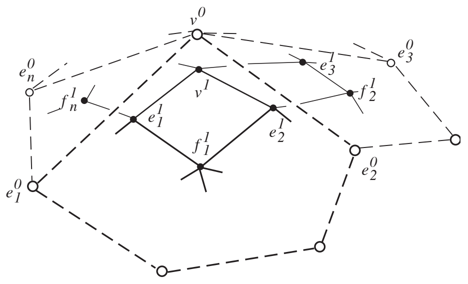


Figure 4: *Labeling of vertices of a Catmull-Clark [CC78] base mesh around the vertex v^0 of valence n.*

In the case that the input control mesh is locally regular, the resulting surface will locally be a bi-cubic B-spline. Additional refinements over these regions, while needed to refine the mesh, are not needed to determine the polynomial structure of the surface that is known once an individual B-spline patch has been determined. This patch structure is illustrated in Figure 6. We will see in Section 3.7 that this structure can be exploited using hardware tessellation to create a fast rendering algorithm for the Catmull-Clark limit surface.

While subdivision surfaces freed designers from the topology constraints of B-splines, further development was needed to make them truly useful. The original subdivision rules did not consider the case of a mesh with boundary; that is, a mesh that is not closed. Nasri [Nas87] treated the boundary of a mesh as a B-spline curve, and used the cubic B-spline curve subdivision rules to subdivide these edges. Subsequent work took the idea a step further and allowed edges within a Catmull-Clark control mesh to have a variable amount of sharpness – called creases [HDD*94, DKT98].

3.3.2. Boundary and Crease Rules

Following DeRose et al. [DKT98], a crease is defined by adding sharpness tags to edges. Subdividing a sharp edge creates two child edges, each of which are tagged with the sharpness value of the parent minus one. A vertex v_j containing exactly two crease edges e_j and e_k is considered to be a crease vertex. The following *sharp rules* are used for both boundaries and sharp edges (crease rules):

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

If a vertex is adjacent to three or more sharp edges or located on a corner, then we derive its successor by $v^{i+1} = v^i$ (corner rule). In order to deal with fractional and linearly varying smoothness, additional rules have been defined (cf. [DKT98]).

As a mesh is subdivided, sharpness tag values decrease and different subdivision rules are applied accordingly. In this way, the smoothness of edges corresponding to the original mesh can be controlled. These *semi-sharp* creases give designers considerable flexibility for modeling with Catmull-Clark Subdivision Surfaces and have significantly increased their popularity.

3.4. Subdivision on the GPU

As shown before, the rules for constructing new mesh points involve taking weighted averages of small, local collections of old mesh points. Gathering these points on the CPU usually involves neighborhood queries using a mesh connectivity data structure, e.g., winged-edge, half-edge, or quad-edge. While the details between these structures differ slightly, the basic idea is the same. Incidence relations are captured by linking edge `structs` with pointers. In order to satisfy a neighborhood query (e.g., given a vertex, return all the vertices sharing an edge with the vertex in order), the connectivity structure must be traversed, one edge at a time, by following links and dereferencing pointers. Doing this on the GPU is not practical, due to the length and varying sizes of these edge chains.

3.4.1. Subdivision Tables

By making the assumption that mesh connectivity is static, or at least does not change often, then a simple table-driven approach becomes feasible. The idea is to encode the gather patterns of vertex indices, needed to construct new mesh points from old mesh points, and store these in tables. Since there are 3 types of new points being constructed (face, vertex, and edge), 3 compute kernels are used. Each of these compute kernels will execute a single thread per new mesh point, and use the indices stored in the tables to gather the old mesh vertices needed in the weighted average.

3.4.2. GPU Subdivision versus Hardware Tessellation

Given the table-based approach to realizing Catmull-Clark subdivision surfaces on the GPU, it may seem that patch-based hardware tessellation is not needed for rendering these primitives. One can simply transfer an unrestricted control cage to the GPU and let its massive data-parallelism generate and render millions of triangles. The problem with this idea is GPU memory bandwidth. Each level of the refined mesh must be streamed to and from the GPU and off-chip GPU memory. Each newly generated level is roughly $4\times$ as large as the old, an exponential growth rate. While the table-driven approach efficiently maps the subdivision rules to the GPU, the naïve use of this approach will quickly become I/O bound. Hardware tessellation, on the other hand, is incredibly efficient since it avoids global (off-chip) memory accesses. Once a compact patch description is streamed to a local GPU cache, no further global memory accesses are needed. The hull, tessellation, domain, and even rasterization stages are all performed using fast, local GPU cache memory.

3.5. Stam's Algorithm

Stam [Sta98] proposed a method for the exact evaluation of Catmull-Clark Subdivision Surfaces. The advantage of Stam's algorithm is that it treats a subdivision surface as a parametric surface, which is seemingly ideal for hardware tessellation. In order to make the combinatorics tractable, Stam's algorithm requires that extraordinary vertices be isolated, so that no quadrilateral face be incident on more than one extraordinary. This can be achieved by first applying two steps of Catmull-Clark subdivision to a control mesh.

A face incident on a single extraordinary vertex, together with the vertices of all incident faces, is the input to Stam's algorithm, see Figure 5a. Let this collection of $2n+8$ vertices be labeled $\mathbf{v}_0$. A feature of this decomposition of the control cage into quadrilaterals is that it immediately induces a parameterization on the limit surface, as follows: Consider a collection of unit square domains in one-to-one correspondence with the faces of a subdivided control cage (all quads). As each of these quad faces is further subdivided, its unit square domain is similarly subdivided. In the limit, a point on the limit surface will correspond to a parameter value in the unit square domain.

Given this parameterization, we can enumerate all the various sub-patches generated by the subdivision process over a single quadrilateral face

$$F_{j,k}(u,v) \; = \; N^3(u)N^3(v) \cdot P_j \cdot S^k \cdot \mathbf{v}_0,$$

where $j = 1,2,3$ is the index of a quad sub-domain (see Figure 5b), k is the level of subdivision, $N^3(t)$ is the cubic B-spline basis function, P_j is a *picking* matrix that generates the corresponding B-spline control points for the chosen patch, and S is the subdivision matrix whose entries correspond to

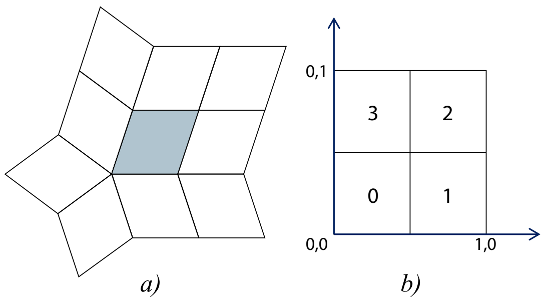


Figure 5: *a) Collection of control points input into Stam's algorithm. b) Labeling of subdomains for Stam's algorithm.*

the weights of the Catmull-Clark subdivision algorithm. Performing the eigen decomposition of S, the patches are rewritten

$$F_{j,k}(u,v) = \underbrace{N(u,v)\cdot P_j \cdot V}_{\text{eigen basis functions}} \cdot \Lambda^k \cdot \underbrace{V^{-1}\cdot \mathbf{v}_0}_{\text{eigen space projection}},$$

where V and V^{-1} are the left and right eigenvectors and Λ is the diagonal matrix of eigenvalues of S[†]. Writing $F_{j,k}(u,v)$ this way shows that *subdivision*, or taking the matrix S to the k^{th} power, can be replaced by raising the diagonal elements of Λ to the k^{th} power. This requires substantially less computation, $O(c)$ operations per parametric evaluation. However, the constant c is rather large, due to the large number of floating point operations needed to evaluate the eigen basis functions and their derivatives. Furthermore, Stam's algorithm does not handle sharp subdivision rules, and obtaining bitwise-exact boundary evaluation of positions and normals is problematic due to the eigen space projection. For these reasons, researchers have explored alternatives to exact limit surface evaluation, instead opting for reasonable approximations that are much faster to evaluate for various real-time applications.

3.6. Approximate Subdivision Methods

Since hardware tessellation is patch-based, in order to achieve maximum performance, researchers have considered ways to render smooth higher order surfaces that behave similarly to subdivision surfaces.

3.6.1. PN-triangles

The so-called PN-triangle technique [VPBM01] was developed to add geometric fidelity to the large body of existing triangle mesh models. For each triangular face, a cubic Bézier triangle patch is constructed using only the position and normal data of the three incident vertices. A cubic Bézier triangle is determined by 10 control points. The position and normal at each vertex determines 3 control points that span the corresponding tangent planes. The final control is determined by a *quadratic precision* constraint; that is, if the 9 previously constructed control points happen to be consistent with a quadratic, then this 10^{th} control point will be consistent with the same quadratic.

Note that this construction guarantees that a PN-triangle surface is continuous (C^0), but not smooth (C^1). To overcome this shortcoming, a PN-triangle contains a quadratic normal patch, whose coefficients are derived from the triangle's vertex normals. These normal patches, which are also continuous, are used for shading rather than the cubic geometry patch. This is reminiscent of Phong shading, where the (interpolated) normal surface used for shading differs from the underlying triangle geometry.

PN-triangles predate hardware tessellation, but their relatively simple and local construction is well suited to the hardware tessellation paradigm. A hull shader program can be used to convert a single mesh triangle and associated vertex data into a cubic geometry, and quadratic normal, patch pair. The domain shader stage will evaluate the input cubic and quadratic patches at the input (barycentric) *uvw* coordinate triple. While PN-triangles are effective at improving the appearance of traditional triangle mesh models, they are not the best choice for obtaining results that approach the geometric quality of Subdivision Surfaces. Other more recent schemes have been explicitly designed for this purpose.

3.6.2. ACC-1

The first of these schemes was one approximating based on bi-cubic patches [LS08]. It is assumed that the input control cage contains only quadrilaterals; if not, then a single step of Catmull-Clark subdivision will ensure this. Given a quad mesh as input, the idea is similar to PN-triangles in that a geometry patch is constructed to interpolate the position and tangent plane of the Catmull-Clark subdivision limit surface at the corners of a mesh quadrilateral. This bi-cubic patch, while designed to approximate the Catmull-Clark limit surface, will only be continuous, not smooth. Therefore, a *normal patch* is generated to create a smooth normal field for shading. As with PN-triangles, this algorithm is easily implemented for hardware tessellation by using the hull shader for patch construction, and the domain shader for patch (geometry and normal) evaluation. Unlike PN-triangles, ACC-1 geometry and normal patches are both bi-cubic. This simplifies domain shader evaluation in that only a single set of basis functions needs to be computed.

One of the advantages of approximating Catmull-Clark subdivision surfaces for hardware tessellation is that many authoring tools already exist for creating Catmull-Clark control cages for off-line applications like CG movies. The ACC-1 scheme made these tools available for real-time ap-

[†] this form differs slightly from Stam's original formulation in order to simplify this exposition

plications like games. However, professionally created subdivision models often employ crease and boundary rules. These are special subdivision rules that are applied to mesh boundaries, or chains of edges that an artist has decided should appear to be more creased. A full treatment of this issue in the context of ACC-1 patches appeared in [KMDZ09].

While the ACC-1 scheme was relatively fast, the requirement that patch control points be 6 dimensional (3 for position, 3 for surface normal), was less than ideal. Furthermore, the underlying surface was not geometrically smooth. This led researchers to further develop approximate Catmull-Clark schemes. Several papers that split quad mesh faces into four triangles along both diagonals appeared [MNP08, MYP08, NYM*08]. These works generated smooth surfaces, but did so by increasing the patch count so that the effective throughput, the ratio of input control point data to output amplified geometry, went down. Ideally, we should have large patches that are evaluated many times, thereby increasing this throughput. This was the design goal behind the next approximating scheme.

3.6.3. ACC-2

Constructing a piecewise smooth surface of arbitrary topological type from polynomial patches is a non-trivial problem. The difficulties lies in honoring the requirement that mixed partial derivatives ($\frac{\partial^2 F}{\partial uv} = \frac{\partial^2 F}{\partial vu}$) are consistent among patches at a shared extraordinary vertex. Adding degrees of freedom by domain splitting (as mentioned in the previous paragraph) can be used to solve this problem. Another approach is use non-polynomial patches; this is the idea behind a scheme we call ACC-2 [LSNC09], based on *Gregory* patches [Gre74].

A bi-cubic Gregory patch is defined by 20 control points. Like a (16 control point) bi-cubic Bézier patch, the 4 corner and 8 edge control points completely define the boundaries of the patch. The remaining interior control points are similar. However, each interior control point of a Bézier patch corresponds to two interior points of a Gregory patch. The reason for this is closely related to mixed partial derivatives. For a polynomial Bézier patch these must be equal (corresponding to a single interior control point per vertex). For non-polynomial Gregory patches, the mixed partial derivatives can disagree (hence the two interior control points per vertex). These additional degrees of freedom are used to solve the patch-to-patch smoothness constraints independently for Gregory patches, rather than as a (often times singular) system involving all patches incident on an extraordinary patch. The drawback to this approach is that the Gregory patch basis functions contain singularities ($\frac{0}{0}$) at corners. However, since we already know the limit position we want for these points, this issue causes no real concern in practice. The construction of ACC-2 control points is very similar to those for ACC-1.

To the best of our knowledge, the Gregory patch-based ACC-2 algorithm is the faster method for interpolating limit position corners and smoothly approximating the surface elsewhere. However, an artist who has painstakingly constructed a model using a Catmull-Clark subdivision surface is unlikely to be pleased if the renderer produced a slightly different result than intended. These differences are even more obvious when considering surface parameterization, though He et al. [HLS12] presented a way to mitigate this to a certain extent. Artist and games developers would be much happier if the hardware tessellator could be used to render a Catmull-Clark Subdivision Surfaces accurately and efficiently. This brings us to the state-of-the-art technique, and the topic of our next section.

3.7. Feature-Adaptive Subdivision

We now present an algorithm, called *feature-adaptive subdivision*, for rendering Catmull-Clark limit surfaces with crease edge tags, that combines table-driven subdivision and hardware tessellation [NLMD12, Nie13]. As the term *feature-adaptive* implies, subdivision is only used where needed, to create as few bi-cubic patches as possible. These patches are then efficiently rendered using hardware tessellation. This algorithm has become the basis for Pixar's *"OpenSubdiv"* [Pix12], which is now used in major modeling packages [Auta, Autb].

It is well known that the limit surface defined by Catmull-Clark subdivision can be described by a collection of bi-cubic B-spline patches, where the set has infinitely many patches around extraordinary vertices, as illustrated in Figure 6(left). Similarly, near creases, the number of limit patches grows as the crease sharpness increases, as shown in Figure 6(right).

Feature adaptive subdivision proceeds by identifying regular faces at each stage of subdivision, rendering each of these directly as a bi-cubic B-spline patch using hardware tessellation. Irregular faces are refined, and the process repeats at the next finer level. This strategy uses a table-driven approach; however, the subdivision tables are restricted to irregular faces. A face is regular only if it is a quad with all regular vertices and if none of its edges or vertices are tagged

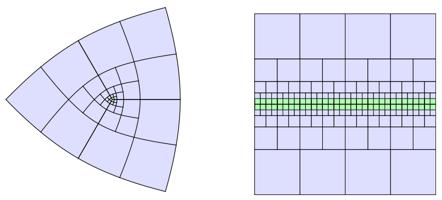


Figure 6: *The arrangement of bi-cubic patches (blue) around an extraordinary vertex (left), and near an infinitely sharp crease (right). Patches next to the respective feature (green) are irregular.*

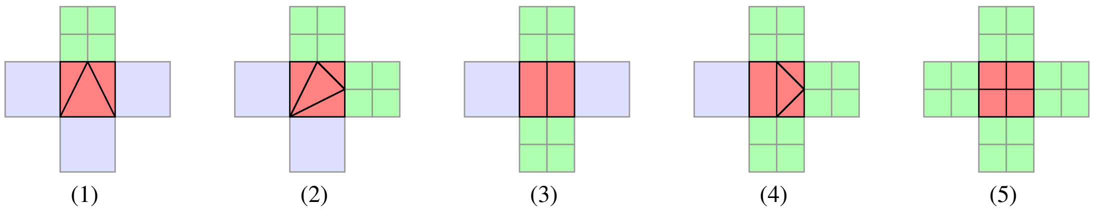


Figure 7: *There are five possible constellations for TP. While TP are colored red, the current level of subdivision is colored blue and the next level is green. The domain split of a TP into several subpatches allows full tessellation control on all edges, since shared edges always have the same length.*

as sharp. In all other cases the face is recognized as irregular, and subdivision tables are generated for a minimal number of subfaces. 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 8).

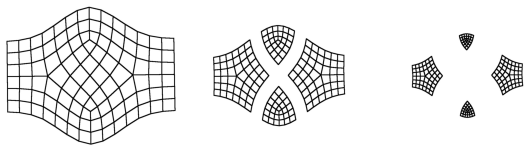


Figure 8: *The feature-adaptive subdivision scheme applied on a grid with four extraordinary vertices. Subdivision is only performed in areas next to extraordinary vertices.*

3.7.1. Patch Construction

Once the subdivision stage is complete, the resulting patches are sent to the hardware tessellator for rendering. For each subdivision level there are two kinds of patches: full patches and transition patches.

3.7.2. Full Patches

Full patches (FPs) are patches that only share edges with patches of the same subdivision level. Regular FPs are passed through the hardware tessellation pipeline and rendered as bi-cubic B-splines. It is ensured by feature-adaptive subdivision that irregular FPs are only evaluated at patch corners. This means that for a given tessellation factor, $\lceil \log_2 \mathit{tessfactor} \rceil$ adaptive subdivision steps must be performed. Since current hardware supports a maximum tessellation factor 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, a special vertex shader is used.

3.7.3. Transition Patches

Note that the arrangement of bi-cubic 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). It is additionally required that TPs are always regular. This constraint is enforced 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 tessellation factors of shared edges to the same value will ensure this. However, TPs share edges with neighboring patches at a different subdivision level by definition. One solution to this problem would be using compatible power of two tessellation factors so that the tessellations will line up. However, allowing only power of two tessellation factors is a severe limitation that reduces the available flexibility provided by the tessellation unit.

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

Each subdomain corresponds to a logically separate subpatch, though each shares the same bi-cubic 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 these are batched according to subpatch type. However, since the control points within a TP are shared for all subpatches, the vertex and index buffers are the same. The overhead of multiple draw calls with different shaders but the same buffers becomes negligible for a larger number of patches.

By rendering TPs as several logically separate patches,

	speed	exact limit surface	order of continuity	bitwise-exact edge		meshes with boundary	semi-sharp creases
				position	normal		
Stam	+	yes	C^2	no	no	no	no
PN-triangles	++	no	C^0	yes	no	yes	no
ACC-1	++	no	C^0	yes	no	yes	no
ACC-2	+++	no	C^1	yes	no	yes	no
Feature-Adaptive	+++	yes	C^2	yes	yes	yes	yes

Table 1: *A comparison of properties of the various approaches to rendering Catmull-Clark subdivision surfaces using hardware tessellation presented in this report.*

all T-junctions in the patches are eliminated from the structure of a surface. This means that as long as consistent tessellation factors are assigned to shared edges, in principle a crack-free rendered surface is obtained. In practice however, due to the behavior of floating point numerics, additional care is required as discussed previously.

3.7.4. Optimizing Semi-Sharp Creases

Excluding crease edges, the time and space requirements of feature-adaptive subdivision is linear in the number of extraordinary vertices in a mesh. Including crease edge, the time and space requirements are exponential, although this growth is less than näive subdivision. In order to improve this behavior, Nießner et al. [NLG12] create optimized shaders for handling semi-sharp creases. The idea is similar to Stam's algorithm. They consider only patches with isolated constant sharpness tags, which will appear after each subdivision level. Within such a patch, the unit square domain can be partitioned into 2 (integer sharpness) or 3 (fractional sharpness) zones. Using eigen analysis, the polynomial structure of each of these zone is determined. In a domain shader specialized for such patches, the basis functions appropriate for the zone containing the u, v parameter values are used to evaluate the patch. Using this technique, large patches containing a single crease tag can be immediately tessellated or rather subdivided, greatly improving the time and space requires of feature-adaptive subdivision.

3.8. Level of Detail: Tessellation Strategies

Assigning tessellation factors properly is important to achieving maximum efficiency from hardware tessellation. Too large, and a surface will be *under tessellated*, leading to interpolation artifacts and faceting on silhouettes. Too small, and a surface will be *over tessellated*, reducing rasterization efficiency. Modern GPUs (circa 2014) are still not optimized for pixel-sized triangles; ideally, triangles should still cover at least tens of pixels. We now consider some strategies for assigning tessellation factors that lead to reasonable quality and efficiency tradeoffs.

3.8.1. Heuristic methods

A simple approach to computing tessellation factors is as a function of the distance from the eyepoint to the midpoint of a patch edge [NLMD12]. Another option is to fit a sphere around an edge and determine the tessellation factors from the projected spheres diameter in screen space [Can11]. Both methods are easy to compute and (critically) give the same result for a shared edge regardless of which patch is being considered. Further heuristics are shown in the context of displacement mapping [NL13].

We caution against using the projected length of patch edges [TBB10]. This length can easily shrink to zero if the edge happens to line up the view ray. Setting the tessellation factor of any edge in a patch to zero instructs the hardware to cull the patch.

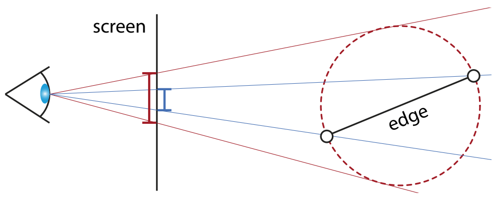


Figure 9: *Tessellation heuristics: In contrast to edge length projection (blue) the heuristic using an edges projected bounding sphere diameter is rotation invariant (red).*

3.8.2. Pixel Accurate Rendering

The concept of pixel accurate rendering [YBP12] is used to determine a tessellation factor that guarantees a tessellation of a surface differs from the true surface, when measured in screen-space pixel coordinates, by at most half a pixel. This is accomplished using a bounding structure called a slefe; an acronym meaning Subdividable Linear Efficient Function Enclosures [LP01]. These are piecewise linear upper and lower bounding polylines that can be computed using a precomputed table of numbers and a curve's Bézier control points. This notion is extended to surfaces using *slefe boxes*, whose projections can be used to estimate a tessellation factor for an entire patch. Since the tessellation factors for neighboring patches will likely differ on a shared edge between patches, the edge's tessellation factor is set to the larger tessellation factor estimated for the two incident patches.

4. Displacement Mapping

In 1978, Blinn proposed perturbing surface normals using a wrinkle function [Bli78]. While this mimics the shading of a high resolution surface, the geometry itself remains unchanged. This led Cook [Coo84] to develop *displacement mapping* to give objects more realistic silhouettes. Displacement mapping has since been used as a means to efficiently represent and animate 3D objects with high-frequency surface detail. Where texture mapping assigns color to surface points, displacement mapping assigns offsets, which are either scalar- or vector-valued. The advantages of displacement mapping are two-fold. First, only the vertices of a coarse base mesh need to be updated to animate the model. Second, since only the connectivity for the coarse mesh is needed, less space is required to store the equivalent, highly detailed mesh. In the following, we primarily consider scalar-valued displacements since they are faster to render and take up less storage. The displacement is then achieved by tessellating the base mesh and moving the generated vertices along their normal according to the value stored in the displacement map.

To that end, Hardware tessellation is ideally suited for displacement mapping. A coarse mesh provides a base surface that is tessellated on-chip to form a dense triangle mesh and immediately rasterized without further memory I/O. Displacing triangle vertices in their normal direction has little performance impact. However, while conceptually simple and highly efficient, there are two major sources for artifacts that have to be addressed.

First, before a displacement map can be applied, the base mesh is typically endowed with an explicit parameterization, often in the form of a 2D texture atlas (see [FH05] for a survey). Conceptually, seams must be 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, these discontinuities in surface offset or normal can lead to unacceptable cracks in a rendered surface as shown in Figure 10.

Second, hardware tessellation is based on the dynamic retessellation of patches, so the underlying sampling pattern is continuously updated to achieve triangles of uniform size (see Section 3.8). However, continuously changing the sampling pattern creates swimming artifacts – the surface appears to fluctuate and the sampling pattern becomes visible. This effect is caused by under-sampling the displacement map while changing the sampling positions over time.

Avoiding these artifacts is an active research area. In the following we discuss displacement mapping methods and provide details on selected recent publications. Hiding seams in displacement mapping mainly depends on the underlying parameterization. Therefore, the approaches can be categorized into the following classes:

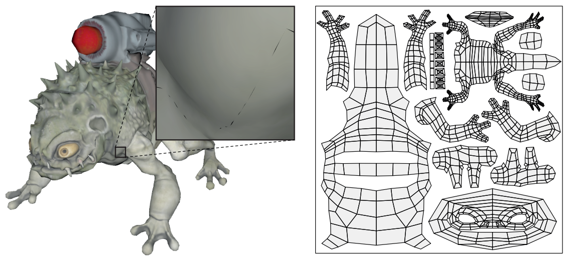


Figure 10: *Cracks on the displacement-mapped Monsterfrog (left) appear at uv chart boundaries (atlas right) when sampled displacement and normal values differ on both sides of a seam.*

- *Texture Atlases 4.1:* The general case of explicitly parameterized meshes, as described above. The atlas can consist of multiple unconnected regions (charts).
- *Heightmaps 4.2:* Typically used for planar surface displacement using a texture atlas consisting of only a single, rectangular chart without seams.
- *Procedural Displacement 4.3:* Avoid texture related problems by amplifying geometry computationally.
- *Tile-Based Approaches 4.4:* Methods, where each patch maps to a unique texture tile with displacement values.

4.1. Texture Atlases

Providing consistent transitions between chart boundaries in a texture atlas is challenging [SWG*03] and leads to a wide range of approaches.

Artifacts can be reduced by optimizing seam positions [SH02, LPRM02] or by creating textures with matching colors across seams [LH06]. However, minor discontinuities will always remain as long as chart boundaries differ in length or orientation in the texture domain. Small discontinuities are well acceptable for color textures, however for displacement textures they result in cracks and thus become greatly visible. Thus, parameterizations were proposed with consistent chart boundaries and orientation to obtain a perfect alignment of texel grids. Carr et al. [CHCH06, CH04] and [PCK04] employ quad segmentation on the input mesh and map the resulting charts to axis-aligned squares in the texture domain. This mapping induces distortions unless all quads in the segmentation are of equal size. Ray et al. [RNLL10] solve this issue by aligning matching boundary edges with an indirection map containing scaling and lookup information.

A different class of approaches aims at creating watertight surfaces without modifying the parameterization. For instance, Sander et al. [SWG*03] close gaps after displacement by moving boundary vertices to a common, precomputed *zippering* path. González and Gustavo [GP09] insert a ribbon of triangles between charts in the texture domain to interpolate between boundaries in a specialized shader program.

Texture coordinates are typically stored and processed as floating point values. Unfortunately, precision is unevenly distributed (see Goldberg [Gol91]), i.e., precision decreases with distance from the origin. This results in different sample positions, and thus displacement discontinuities, when interpolating texture coordinates on seam edges in different locations in the atlas, even if the edges' lengths are equal and consistently oriented. Castaño [Cas08] avoids these precision problems by pre-selecting one of the matching edges or vertices for sampling the seam. The texture coordinates of the preselection are stored for every patch vertex and edge, resulting in a consistent evaluation during rendering. Thereby, this approach is independent from boundary orientation at the cost of storing additional texture coordinates (16 for quads, 12 for triangles).

The majority of parameterization methods used for color texturing minimize only geometric metrics and assumes no prior knowledge of the signal to be stored. However, when the texture to be stored is known (as is mostly the case for displacement mapping), this prior knowledge can well be used to optimize the parameterization, and to allocate more texel in regions with high detail. This inspired Sloan et al. [SWB98] and Sander et al. [SGSH02] to optimize the parameterization based on an importance measurement, e.g., the size of a triangle in the detailed mesh or the reconstruction error introduced. Following these approaches, Jang and Han [JH13, JH12] displace vertices generated with hardware tessellation into regions with high detail.

4.2. Heightmaps

In applications such as digital elevation modeling and terrain rendering the underlying base model is typically spherical or locally planar. For example, a planet can be subdivided into mostly rectangular regions, which are then mapped to a square in the texture domain.

When using such *heightmaps*, various problems arise. First, the amount of available elevation information typically does not fit into the GPU memory for rendering (e.g., the Earth's surface at a resolution of one texel per $1m^2$ would require 1.855 TB of memory). This problem can be solved by creating a hierarchy of different resolutions and only keeping a few levels in GPU memory at a time, as proposed by Tanner et al. [TMJ98]. Second, for watertight rendering, seams between heightmaps, possibly at different resolutions, need to be handled. Third, the surface geometry should ideally be sufficiently dense to reconstruct details close to the observer and coarser at distance to allow for interactive rendering. For watertight rendering, this requires handling of patch boundaries with different mesh resolutions in addition to seams between heightmaps. This led to a wide range of tessellation and level-of-detail schemes, see [PG07] for a survey.

Interactively rendering high density heightmaps requires displacing a densely tessellated mesh. Hardware tessellation is ideally suited for this task. Massive amounts of geometry can be adaptively generated from a coarse mesh, saving precious GPU memory, which in turn, allows for using higher resolution heightmaps, since a large portion of the mesh to be displaced is generated on the fly.

Tatarchuk et al. [TBB10] subdivide the planar input geometry into equally sized sub-patches to overcome hardware-imposed tessellation density limitations. Instead of a uniform subdivision, Cantlay [Can11] subdivides the input geometry using a chunked-LOD approach [Ulr02] to allow higher tessellation close to the observer. During construction Cantlay restricts the patches to a power-of-two in relative size, which is then stored with each patch. Watertight rendering is achieved by making adjacent edges concur on a matching tessellation factor depending on the relative patch sizes (see Section 3.8.1).

In contrast to the previous methods, Bonaventura [Bon11] considers different heightmap resolutions (mipmaps) during rendering. The input mesh is subdivided into equally sized quads, and tessellation factors at edges are restricted to powers-of-two. The mipmaps for trilinear sampling are selected such that the average vertex density matches the texel density. Yusov and Shevtsov [YS11] compress heightmaps in a GPU-friendly format that can be used to reduce memory I/O for out-of-core rendering of large data sets.

Tatarchuk [Tat09] and Yusov [Yus12] apply deformations to heightmap terrains by blending between two maps or directly updating a single heightmap. For multi-resolution heightmaps this requires updating all mip-levels.

4.3. Procedural Displacement

Modeling realistic environments such as cities or natural-looking plants is a time consuming process. Man-made structures and plants are often composed of simple shapes with small variations (e.g., plants with different numbers of leaves and variations in shape). Procedural approaches aim at creating massive content computationally. Applications include modeling cities, large realistic terrains, complex plants, or extremely small surfaces details (e.g., skin). Procedural methods offer a large variety of uses: Content can be automatically generated, parameterized, stored and combined with artist-created content. On-the-fly generation enables rendering massive amounts of content with small I/O requirements with only a small set of parameters having to be uploaded. Therefore, it is reasonable to combine procedural techniques with hardware tessellation.

Bonaventura [Bon11] shows how to apply hardware tessellation to ocean rendering. Their ocean is represented by a procedural heightmap function defined over a ground plane. In the context of planar surface displacements Cantaly [Can11] shows how to amplifiy a terrain, which fol-

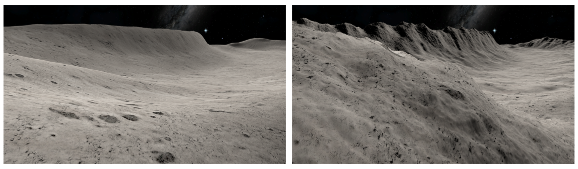


Figure 11: *Low resolution heightmap (left) and procedurally amplifying geometry with high-frequency detail (right).*

lows a base heightmap, with procedurally generated detail displacements as shown in Figure 11.

Instead of directly evaluating procedural heightmaps, geometry and scene properties can also be generated by a set of rules, affording potentially more complex kinds of content. This approach was first proposed by Lindenmayer [Lin68] for describing and researching plant growth. Each species is described by a *shape grammar* [Sti80] and individuals by a small set of parameters. Large amounts of plants are then obtained by interpreting the grammar for each individual. Shape grammars found widespread use in architectural modeling, city planning, simulation and content creation for movie production and games (see [WMWF07] for a survey).

Marvie et al. [MBG*12] propose a method for evaluating shape grammars directly within the tessellation shaders. After grammar evaluation, a set of terminal symbols remain that are replaced by detailed geometry instances or simple textured quads (e.g., facade walls) in a geometry shader. For LOD rendering they switch between geometry and texture terminal symbols depending on the viewer distance. Experimental results show a massive city scene with about 100k buildings and 550k trees rendered at interactive rates on an Nvidia GTX 480, whose buildings and trees were generated from 2 and 7 different grammars, respectively. Grammars and parameters are adjustable, allowing for interactive feedback and verification. An explicit polygonal representation of the scene with detailed geometry would require 2.3 TB of GPU memory in contrast to 900 MB with the method of Marvie et al., mostly occupied by geometry terminal symbols.

4.4. Tile-Based Approaches

Most automatic global parameterization approaches either aim at creating as few seams, and thus charts, as possible or striving towards minimizing distortions resulting in more distinct charts. When packed into a texture atlas, charts must not overlap and are typically enclosed by a texel margin to prevent sampling from neighboring charts. This results in many texels not covered by charts and, thus, in an ineffective utilization of the available texture space. The use of mipmapping [Wil83] to prevent undersampling artifacts exacerbates the problem, requiring larger margins such that charts in lower resolution textures do not overlap.

Maruya [Mar95] proposes to increase coverage by tightly packing triangles in arbitrary order into equally sized blocks. Therefore, each mesh triangle is mapped to a isosceles right triangle with edges being power-of-two fractions of the block size. Carr et al. [CH02] improve this approach by mapping adjacent triangles to blocks. This reduces discontinuities and enables mipmapping individual blocks.

Instead of a global parameterization, Burley and Lacewell [BL08] propose per-face texturing with each quad face implicitly parameterized by the order of its corner vertices $[(0,0)(0,1),(1,1)(1,0)]$. Each face is assigned a power-of-two sized texture block enabling mipmapping each face individually. The full resolution and down-sampled face blocks are then packed into a global texture atlas. Further, indices to adjacent face blocks are stored with each face to enable seamless evaluation by sampling neighboring face textures at boundaries in their offline renderer. While this approach works well for offline rendering, evaluating and sampling neighboring faces negatively impacts the performance when applied on the GPU.

4.4.1. Multiresolution Attributes

Schäfer et al. [SPM*12] present a method to avoid texture-related artifacts by extending the concept of vertex attributes to the volatile vertices generated by hardware tessellation. Therefore, they use a data structure that follows the hardware tessellation pattern, where for each generated vertex a set of attributes such as surface offset, normal or color is stored. This direct vertex to attribute mapping enables them to overcome under-sampling artifacts appearing when sampling attributes from textures.

Conceptually, their data structure is similar to the per-face texture tiles [BL08], however they store a linear array of attributes corresponding to hardware generated vertices for each face. Providing consistent transitions between faces in tile-based texturing approaches is of paramount importance for preventing discontinuities and cracks. Hence, Schäfer et al. separately store attributes of inner face vertices from those at corners and edges shared between multiple faces. Attributes shared by multiple faces are only stored once resulting in consistent evaluation and, thus, transitions between faces. This completely avoids discontinuities except for discontinuities present in the input mesh.

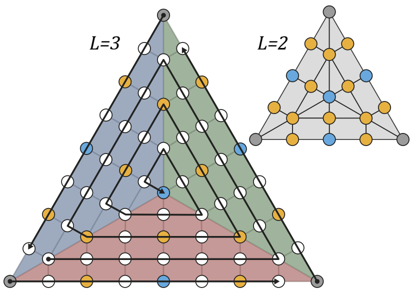
Unfortunately, creating a mapping between generated vertices and their attributes is intricate, as vertices are only represented by a patch-local coordinate (barycentric for triangular, bilinear for quadrangular patches) in the domain shader. Therefore, Schäfer et al. propose a method to compute a unique linear index for each vertex from its implicit patch coordinate. The idea is to interpret and parameterize the vertex locations in terms of the tessellation algorithm: Let $\mathbf{T}^{\mathbf{I}}$ and $\mathbf{T}_i^{\mathbf{E}}$ be the *inside* and *edge* tessellation factors defined in the hull shader, where i is the corresponding *edge index*. Vertex positions are generated by first subdividing a patch into

$\lfloor \mathbf{T^I}/2 \rceil + 1$ concentric topological *rings* (triangles or quads). Edges on each ring are then subdivided into $\mathbf{T^I} - \mathbf{R} \cdot 2$ segments for interior edges whereas boundary edges ($\mathbf{R} = 0$) are subdivided using the outer tessellation factors $\mathbf{T}_i^{\mathbf{E}}$. The end points of each segment represent vertices $\mathbf{V}$ that are then further processed in a domain shader. Additionally, the coordinate of each vertex is given as barycentric or bilinear coordinates $\mathbf{P}$ for triangular or quadrangular patches, respectively.

Following this scheme, the number of vertices on each ring's edge $C_{r,e}$ is fixed for a given tessellation pattern ($[\mathbf{T^I}, \mathbf{T}_i^{\mathbf{E}}]$). Thus, the number of vertices on each ring is given by the sum of the corresponding edge vertices. Further, each vertex is uniquely defined by a ring, edge and integral position on that edge ($[\mathbf{R}, \mathbf{E}, \mathbf{V}]$). In the domain shader Schäfer et al. convert patch coordinates to the $[\mathbf{R}, \mathbf{E}, \mathbf{V}]$ representation. This representation is then used to compute a unique index for each vertex by linearizing edge and face vertices separately as depicted in Fig. 12 (left).

Schäfer et al. employ a preprocessing step to convert existing data, for example uv displacement or color textures, into their representation along with tile locations to the edge and inner data per primitive. During rendering, each generated vertex then fetches an attribute from the corresponding face or shared edge tile in the domain shader.

For rendering, adaptive tessellation factors are desirable (see Section 3.8). By introducing filtering, rendering tessellation factors are decoupled from the storage pattern. Schäfer et al. show that nearest neighbor, bilinear and trilinear filtering can be efficiently applied using the $[\mathbf{R}, \mathbf{E}, \mathbf{V}]$ representation. A unique property of tessellation patterns is that for power-of-two tessellation factors (in the following referred to as levels), all vertices of coarser levels are present at exactly the same patch coordinate of higher levels (see Fig. 12 (right)). Schäfer et al. employ this property for trilinear filtering by creating a mipmap-like pyramid of patterns for storage. In fact they employ an ARAP-like fitting scheme in a preprocessing step to fit attributes at each power-of-two pattern to the input data.Thereby, they adapt the number of required mipmap-levels to reconstruct the input data in a signal-optimized manner.



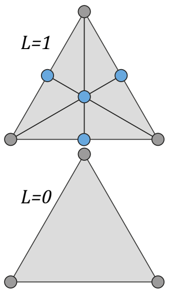


Figure 12: *Linearization of triangle tessellation pattern in inward-spiraling order (left). Vertex locations on different power-of-two tessellation patterns are present in higher levels at exact same uvw positions (vertices colored by level of first appearance).*

During rendering, attributes from the bracketing pattern mipmap-levels are fetched and smoothly blended, resulting in an LOD rendering, while greatly reducing *swimming* artifacts. By constraining the rendering tessellation to the same power-of-two factors used for storage, these artifacts can be completely mitigated enabling continuous level-of-detail rendering, at the cost of higher tessellation density. To this end, they compute the desired tessellation factors using a distance-based metric (see Section 3.8), selecting the upper bracketing tessellation factor from the storage scheme for rendering. Then, trilinear filtering is applied resulting in a smooth blending in of detail present only at the higher mipmap-level (e.g., white vertices in Figure 12 when blending between level two and three).

Experimental results show comparisons to uv-texturing. In terms of quality both are almost equal when using bi- and tri-linear filtering making their approach a valid alternative to uv textures with the benefit of avoiding undersampling artifacts and cracks with their representation. Major differences are visible when comparing nearest neighbor sampling making the underlying storage pattern (rectangular shape of texel for uv textures, triangular pattern using their representation) visible. Performance measurements show that their method is on-par with uv textures. Surprisingly, nearest and bilinear sampling slightly faster compared to uv texture sampling, although hardware filtering cannot be used with their data structure. This behavior is attributed to texture cache inefficiency when undersampling uv textures. The presented storage scheme relies on patch-local coordinates only. Providing these in the pixel shader ([SPM*13]) enables this scheme to act as a complete replacement of uv-textures avoiding all afore mentioned texture-related artifacts.

In digital content creation, instant visual feedback during sculpting or painting operations on an asset is desired to improve the design process. Adding detail, e.g., wrinkles around the eye, requires sufficient texture samples in this area. Where artists decide to add detail is unpredictable. This requires either pre-allocating texture samples globally for all faces in advance or a requesting texture resizing and resampling at run-time. Both approaches have severe disadvantages: First, the available memory and texture size on the GPU is limited. This restricts the number texture samples per face and thus the ability to add local detail. Second, resizing the texture on demand or allocating new face textures involves expensive memory I/O between host and device.

Schäfer et al. [SKS13] propose a dynamic memory management scheme for per-face texturing methods running entirely on the GPU to overcome this issue. The idea is to independently pre-allocate face blocks from the number of mesh faces at different resolutions. At run-time these blocks are dynamically assigned to mesh faces for storing detail when editing the mesh. Providing a custom amount of face blocks for each resolution allows for using higher detail textures on a subset of faces in contrast to global sample distribution. This makes their approach suited for applications that demand varying sample densities on different regions of a mesh. Obviously, there are cases where more face blocks are requested than available in the block pool. In these cases Schäfer et al. [SKS13] either allocate and attach new sets of face blocks to the block pool or use the following deallocation strategies in case the available GPU memory has been exhausted: First, for digital content creation face blocks can be downloaded and stored to disk. The face block is then replaced with a lower resolution version and finally deallocated by writing its block index back to the pool. Second, for gaming scenarios they propose decaying displacement over time, e.g., the imprint of a footstep, before the block is deallocated.

Although Schäfer et al. [SKS13] apply GPU memory management to their storage method, it can easily be extended to other per-face texturing schemes. In their experimental results they report handling local edits, including memory management and normal re-computation, in less than a millisecond, making this approach perfectly suited for real-time content creation and games.

4.4.2. Analytic Displacement Mapping

A major drawback to traditional displacement maps is the requirement for an additional map to retrieve normal information. This causes a significant amount of memory overhead, and makes updating displacement data inconvenient since normals need to be re-computed with every surface update.

Nießner and Loop [NL13] present an approach that eliminates this problem by obtaining normal information directly from an *analytic* offset function. Their method appeared in the context of displaced subdivision surfaces, and formulates the displaced surface as

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

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. The important property is that the base surface is consistently C^2, except at a limited number of extraordinary vertices where it is still C^1. $D(u,v)$ is then defined by constructing a scalar-valued bi-quadratic B-spline with a Doo-Sabin subdivision surface structure [DS78], which is C^1 with vanishing first derivatives at extraordinary vertices. Thus, the displaced surface $f(u,v)$ is also C^1, facilitating smooth surface normals that can be derived analytically without requiring an explicit normal map. While this reduces memory consumption and rendering time, it also allows for efficient dynamic surface edits.

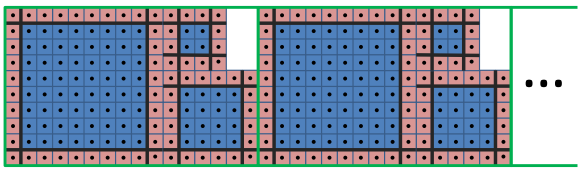


Figure 13: *Snippet of the tile-based texture [NL13] format used for displacement data (8×8 per tile; blue) showing two tiles (green outline) including overlap (red) and mip levels.*

Tile-Based Texture Format In order to avoid texture seam misalignments, which plague classic u,v atlas parameterization texture methods, Nießner and Loop [NL13] propose a tile-based texture format to store their displacement data. They store a bi-quadratic displacement function whose coefficients are stored in an axis-aligned fashion in parameter and texture space. This can be seen as an improved GPU version of Ptex [BL08]; however, adjacent tile pointers, which are impractical on the GPU, are absent. Instead, a one-texel overlap per tile is stored to enable filtering while matching displacements at tile boundaries. An example with two texture tiles is shown in Figure 13 where each tile corresponds to a quad face of the Catmull-Clark control mesh. Tile edges are required to be a 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* $+ 2$. Adjacent tiles do not need to be the same size.

Additional care must be taken with overlap computations at extraordinary vertices, where four tiles do not exactly meet. The idea is to make all tile corners that correspond to the same extraordinary vertex equal in value by averaging corner values. 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, it is guaranteed that the displacement spline $D(u,v)$, will be C^1 across *all* tile boundaries (see [Rei97] for the proof).

The format also stores a full mipmap pyramid [Wil83] for each tile in order to avoid undersampling artifacts. Since displacement values are coefficients of a bi-quadratic surface, mip pyramids are computed based on quadratic B-wavelets [Ber04]. All tiles are then efficiently packed in a global texture with mip levels of individual tiles stored next to each other (cf. Figure 13). While this leaves some unused space, it provides efficient data access due to cache coherency. Additionally, the tile size and an offset to the tile location within the global texture is stored in a separate buffer for each quad face. Tile data is then indexed by the face/patch ID.

Surface Evaluation The key idea behind this approach is to consider a higher-order displacement function which facilitate analytically determining normals on the displaced

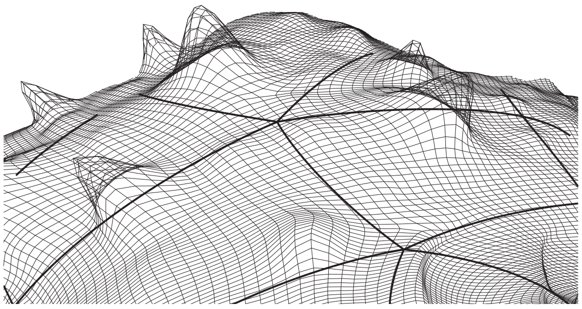


Figure 14: *Base surface: Catmull-Clark limit patches - patch boundaries shown as thick lines. Displacement surface: bi-quadratic Doo-Sabin B-splines - scalar coefficients on top of base surface normal field shown as thin lines.*

surface. The base surface $s(u,v)$ is the limit surface of the Catmull-Clark subdivision defined by a two-manifold control mesh, possibly with mesh boundaries and can be efficiently evaluated and rendered using feature-adaptive subdivision [NLMD12]. A one-to-one correspondence between these quadrilateral faces and unit square domains of tiles is established, giving rise to a global parameterization of the surface (via a face ID; $u,v \in [1,0] \times [0,1]$ triple). The analytic displacement function $D(u,v)$ is defined by a scalar-valued bi-quadratic B-spline. These patches have an arrangement that is consistent with the Doo-Sabin [DS78] subdivision, meaning that the control mesh for the scalar displacement coefficients is *dual*, with refinements, to the control mesh of the base mesh. Note, 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. Figure 14 shows a detailed view of a model with base patch edges (thick curves) and the displacement function coefficients over the base surface (thin grid). For practicality's sake, a constraint is imposed at extraordinary vertices that causes first derivatives of the displacement function $D(u,v)$ to vanish at these points [Rei97]. This degeneracy implies that $D(u,v)$ is a globally C^1 function that can be evaluated over the entire manifold without particular case handling.

For given u,v coordinates and face ID, the displaced surface $f(u,v)$ of a patch can then be computed 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 bi-quadratic sub-patch of $D(u,v)$, corresponding to the u,v value within its tile domain. The patch parameters u,v are transformed into the sub-patch domain $(\hat{u},\hat{v})$ using the linear transformation T:

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

The scalar displacement function

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

is then evaluated, 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 $f(u,v)$, its partial derivatives are computed:

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

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

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

Further, $\frac{\partial}{\partial u}s(u,v)$ can be directly obtained from the base surface. To find the derivative of $N_s(u,v)$, 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, the derivative of the normalized normal is found:

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

$\frac{\partial}{\partial v}N_s(u,v)$ is similar. Finally, $\frac{\partial}{\partial u}f(u,v)$ is computed (analogous to $\frac{\partial}{\partial v}f(u,v)$) and thus $N_f(u,v)$.

Further, Nießner and Loop propose an approximate variant where the Weingarten term is omitted following Blinn [Bli78]. This is much faster from a rendering perspective and provides similar shading quality for small displacements.

Rendering surfaces with analytic displacements can be trivially performed by employing the hardware tessellator. 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 $D(u,v)$ are evaluated in the domain shader along with the derivatives of the normal $\frac{\partial}{\partial u}N_s(u,v)$, $\frac{\partial}{\partial v}N_s(u,v)$. These computations are used in order to determine the vertices of the triangle mesh that are generated by the tessellator. The vertex attributes computed

	seamless	mipmapping	mapping overhead	no normal storage	render performance	arbitrary meshes
UV-Atlas	no	limited	++	no	++	yes
UV-Atlas w. zippering [Cas08]	yes	limited	+	no	+	yes
Heightmaps	yes	yes	+++	yes	+++	only planar
Multires. Attributes [SPM*13]	yes	yes	+++	no	++	yes
Analytic Displacement [NL13]	yes	yes	+++	yes	+++	quad dominant

Table 2: *Comparison and properties of displacement mapping storage schemes (+++ is best): UV-Atlas parameterizations cannot fully hide displacement seams unless zippering is applied. Depending on the margin size between uv-boundaries only a few mip-levels are supported without overlapping charts. Mapping overhead indicates the storage and lookup requirements to access the texture data (e.g. one uv-coordinate per vertex for classical parameterizations vs. four uv-coordinates with zippering). Approaches providing low-cost seam handling or direct normal evaluation from displacement are superior in performance due to fewer texture fetches.*

in the domain shader are then interpolated by hardware and available in the pixel shader where the derivatives of the displacement function $\frac{\partial}{\partial u}D(u,v)$ $\frac{\partial}{\partial v}D(u,v)$ are computed. This allows computing the derivatives of the displaced surface normal $\frac{\partial}{\partial u}f(u,v)$, $\frac{\partial}{\partial v}f(u,v)$ at each pixel independently providing per pixel surface normals for shading. Evaluating the surface normal $N_f(u,v)$ on a per-vertex basis would degrade rendering quality, due to interpolation artifacts.

4.5. Displacement Sampling

In order to recreate the shape of a detailed mesh from a tessellated coarse mesh, the differences between both meshes are sampled and stored to a displacement map. The following methods are used in practice: Surface offsets can be computed by casting a ray from each texel's position on the coarse mesh in positive and negative normal direction and measuring the closest-hit distance with the detailed geometry. Unfortunately, the closest hit may not be the desired intersection point. In the case of the detail mesh was being created from several subdivisions of the coarse mesh, each detail patch maps to a subspace of coarse mesh patches. This induces a direct mapping between both representations, enabling an easy transfer of detail to the displacement map.

Xia et al. [XGH*11] overcome these problems and restrictions by presenting a method for transferring detail from an arbitrary model to a semi-automatically created Catmull-Clark mesh based on polycubes [THCM04, HWFQ09]. Given a detailed input mesh M and a user created coarse approximation P of the model consisting of equally sized cubes, the user defines correspondences by painting strokes on both meshes. A bijective map between both representations is found by computing a global distance field emerging from stroke endpoints. The distance field induces a triangulation on M and P. Therefore, positions inside each triangle are uniquely defined by distances to the triangle corners in M and P, resulting in a bijective map between both representations. With the polycube only consisting of square faces a texture atlas is created containing all boundary faces, with faces between cubes omitted. Then, each chart in the texture atlas is populated by sampling positions in the detailed mesh using a distance field induced mapping between P and M. Finally, the polycube mesh is converted to a Catmull-Clark surface, that can then be rendered using methods outlined in Section 3. Although a few user inputs are required, the repositioning of strokes on the polycubes enables control over distortions and assigned texture space in the resulting parameterization. The particular strength of this approach is the ability to compute a bijective map between two arbitrary meshes for transferring detail to a displacement map.

5. Culling

On current hardware back-facing triangles are typically removed from the pipeline (culled) to avoid unnecessary rasterization and pixel shading. Also, the application of backface culling to geometry that is generated by GPU hardware tessellation is straightforward. However, if the plane normals of all generated triangles for a given surface patch point away from the viewer, or patches are entirely occluded, considerable amounts of computation are still wasted for surface evaluation and triangle setup.

In this section, we explore the feasibility and performance of culling approaches for parametric patches. We will show an overview of state-of-the-art methods, including backpatch and occlusion culling techniques. Further, we will evaluate the practicality of these approaches considering the trade-off between *efficiency* and *effectiveness* [KM96] – how much computational effort is needed to reach a culling decision and how many patches are actually culled. Note that a strict requirement of patch culling techniques is its conservatism, meaning that back-facing patches may occasionally be rendered (needlessly), but front-facing patches are always rendered; i.e., false positives are not tolerated.

5.1. Back-patch Culling Techniques

5.1.1. Cone of Normals

Culling parametric patches can be achieved by computing a *cone of normals* for surface patches. A conservative bound of normals for a given parametric patch is defined by a cone axis $\mathbf{a}$, a cone apex $\mathbf{L}$ and an aperture angle α.

Shirmun and Abi-Ezzi [SAE93] introduce such a technique by using Bézier patches and computing corresponding normal patches with the help of the analytic surface derivatives $N(u,v) = \partial B(u,v)/\partial u \times \partial B(u,v)/\partial v$. Hence, resulting normal patches are of degree $(3d_u - 1, 3d_v - 1)$ in the rational case, and of degree $(2d_u - 1, 2d_v - 1)$ in the polynomial case, where (d_u, d_v) are the degrees of the original patch. In the context of hardware tessellation, polynomial bi-cubic patches with $d_u = d_v = 3$ are commonly used.

Once the normal patch has been computed, the floating cone is constructed. It contains all normal directions of the original patch and is floating since it has no position, but rather only an orientation. The goal is to find the smallest enclosing sphere of a given set of (normalized) points in space. While there are exact, albeit computationally expensive, algorithms for its computation [Law65]), a practical solution is to exploit the convex hull property of Bézier patches and construct the corresponding bounding box (rather than a sphere) of the normal tip points $\mathbf{N}_i$. Therefore, a feasible guess for the cone axis $\mathbf{a}$ is

$$\mathbf{a} = \frac{1}{2}\begin{pmatrix} \max_i \mathbf{N}_{ix} - \min_i \mathbf{N}_{ix} \\ \max_i \mathbf{N}_{iy} - \min_i \mathbf{N}_{iy} \\ \max_i \mathbf{N}_{iz} - \min_i \mathbf{N}_{iz} \end{pmatrix}$$

with a corresponding aperture angle α given by $\cos(\alpha) = \min_i(\mathbf{a} \cdot \mathbf{n}_i)$ where $\mathbf{n}_i$ are the normals at patch control points. Note that the cone may be undefined if $\alpha > \pi/2$.

Once the floating cone is constructed, it has to be translated in order to contain the patch itself, i.e., including the patch control points. Such a cone is called an anchored truncated cone. First, a bottom control point $\mathbf{B}$ with respect to $\mathbf{a}$ is computed $\mathbf{B} = \mathbf{P}_i$, with i given by $\min_i(\mathbf{a} \cdot \mathbf{P}_i)$. An analogous point $\mathbf{T}$ on the top plane is also computed. Thus, the cone bottom plane is defined by $(\mathbf{B} - \mathbf{x}) \cdot \mathbf{a} = 0$. Second, every control point $\mathbf{P}_i$ is checked for enclosure in the translated cone. Points outside the cone are projected to the bottom plane. Projected points $\mathbf{P}'_i$ are then given by $\mathbf{BP}'_i = r(\mathbf{BP}_i - h\mathbf{a})$, with $h = \mathbf{BP}_i \cdot \mathbf{a}$, and $r = \left(\sqrt{\|\mathbf{BP}_i\|^2 - h^2} - h\tan(\alpha)\right) / \sqrt{\|\mathbf{BP}_i\|^2 - h^2}$. Upon applying this operation for all control points, the cone can be bounded on the bottom plane by the projected bounding box. Finally, the top plane of the cone is determined analogously, which results in a bounding radius for both planes r_b (bottom plane) and r_t (top plane), and a center of gravity $\mathbf{c}$ of projected points on the bottom plane. The cone apex $\mathbf{q}$ of the cone containing all control points and normals is then given by $\mathbf{q} = \mathbf{c} - (r_b \cdot (\cos(\alpha)/\sin(\alpha))) \cdot \mathbf{a} + \mathbf{B}$. This provides the distance from the control points to the bottom and top plane $d_t = \mathbf{a} \cdot \mathbf{T} - \mathbf{a} \cdot \mathbf{q}$ and $d_b = \mathbf{a} \cdot \mathbf{B} - \mathbf{a} \cdot \mathbf{q}$. The distance z from the bottom plane to the front plane cone apex $\mathbf{L}$ is then given by $z = d_b \cdot \tan^2(\alpha)$, and leads directly to $\mathbf{L} = \mathbf{q} + \mathbf{a} \cdot (d_b - z)$. The distance to the top plane $y = d_t \cdot \tan^2(\alpha)$ and the back plane cone apex $\mathbf{F} = \mathbf{q} + \mathbf{a} \cdot (d_t - y)$ is provided accordingly.

Given the cone of normals, culling for a given camera point $\mathbf{E}$ can be easily performed. First, normalized viewing directions are computed in respect to the cone apexes $\mathbf{V}_b = (\mathbf{E} - \mathbf{F})/\|\mathbf{E} - \mathbf{F}\|^2$ and $\mathbf{V}_t = (\mathbf{E} - \mathbf{L})/\|\mathbf{E} - \mathbf{L}\|^2$.

In the case where $\mathbf{a} \cdot \mathbf{V}_b \geq \sin(\alpha)$, the cone is back-facing and can thus be culled. Fully front-facing patches are identified if $\mathbf{a} \cdot \mathbf{V}_t \leq -\sin(\alpha)$. Otherwise, the patch is considered to be a silhouette containing front- as well as back-facing regions.

5.1.2. Approximate Cone of Normals

While the cone of normals technique provides relatively tight bounds, having to compute the normal patches is costly. Munkberg et al. [MHTAM10] propose an approximation of the cone of normals, which relies on a tangent and bi-tangent cone following Sederberg and Meyers [SM88].

The first step of their approach is to efficiently determine a cone axis $\mathbf{a}$, which is simply approximated by the four patch corner points $\mathbf{a} = ((\mathbf{P}_{0n} - \mathbf{P}_{00}) + ((\mathbf{P}_{nn} - \mathbf{P}_{n0})) \times ((\mathbf{P}_{n0} - \mathbf{P}_{00}) + ((\mathbf{P}_{nn} - \mathbf{P}_{0n}))$. Note that the cone axis is also considered to be normalized. Next, aperture angles α_u and α_v of the tangent $\partial B(u,v)/\partial u$ and bi-tangent $\partial B(u,v)/\partial v$ patches, respectively, are derived by employing the convex hull property of the derivative Bézier patches. The angles of the tangent and bi-tangent cones are then combined in order to compute the cone angle α [SM88]:

$$\sin(\alpha) = \frac{\sqrt{\sin^2(\alpha_u) + 2\sin(\alpha_u)\sin(\alpha_v)\cos(\beta) + \sin^2(\alpha_v)}}{\sin(\beta)},$$

where β is the smallest of the two angles between the u and v directions used to construct $\mathbf{a}$.

Given the cone axis $\mathbf{a}$ and the aperture α, patch culling is conducted the same way as described in Section 5.1.1.

5.1.3. Parametric Tangent Plane

Another way to perform patch culling is to consider the *parametric tangent plane* as proposed by Loop et al. [LNE11].

The parametric tangent plane $T(u,v)$ of a Bézier patch (also applicable to other polynomial patch types) $B(u,v)$ satisfies

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

Thus, $T(u,v)$ can be directly computed as

$$T(u,v) = \text{cross4}\left(B(u,v), \frac{\partial}{\partial u}B(u,v), \frac{\partial}{\partial v}B(u,v)\right), \quad (1)$$

where *cross4*() is the generalized cross product of 3 vectors in $\mathbb{R}^4$. For bi-cubic $B(u,v)$, the parametric tangent plane is a polynomial of bi-degree 7 and can be written in Bézier form as

$$T(u,v) = \mathbf{B}^7(u) \cdot \begin{bmatrix} \mathbf{t}_{00} & \mathbf{t}_{01} & \cdots & \mathbf{t}_{06} & \mathbf{t}_{07} \\ \mathbf{t}_{08} & \mathbf{t}_{09} & \cdots & \mathbf{t}_{14} & \mathbf{t}_{15} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{t}_{48} & \mathbf{t}_{49} & \cdots & \mathbf{t}_{54} & \mathbf{t}_{55} \\ \mathbf{t}_{56} & \mathbf{t}_{57} & \cdots & \mathbf{t}_{62} & \mathbf{t}_{63} \end{bmatrix} \cdot \mathbf{B}^7(v),$$

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 patch control points of $B(u,v)$. Note that $T(u,v)$, being of bi-degree 7, is one less in both parametric directions than expected from adding the polynomial degrees of inputs to the equation.

The next step is to employ the parametric tangent plane for visibility classification, i.e., determining whether a patch is front-facing, back-facing, or silhouette with respect to the eye point. The visibility for a patch $B(u,v)$ is classified by using its parametric tangent plane $T(u,v)$, $u,v \in [0,1]^2$, with respect to the homogeneous eye point $\mathbf{e}$ using the continuous visibility function:

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

Computing CVis $(B,\mathbf{e})$ precisely will require costly iterative techniques to determine the roots of a bivariate polynomial. Instead, Loop et al. [LNE11] suggest 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:

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

The classification produced by DVis $(B,\mathbf{e})$ is then a conservative approximation of CVis $(B,\mathbf{e})$. Therefore, it is possible for DVis $(B,\mathbf{e})$ to classify a front- or back-facing patch as a silhouette in error. Again, culling can be performed by omitting back-facing patches. Loop et al. [LNE11] also provide an efficiently parallelized version to compute and evaluate the parametric tangent plane on the GPU.

5.1.4. Summary

To evaluate the different back-patch culling approaches, the SimpleBezier example from the DirectX 11 SDK is extended, running on an NVIDIA GTX 480. Note that it is very efficient to implement culling tests in a separate compute shader and to feed the decision into the constant hull shader using a small buffer. This provides more flexibility and is considerably faster, as the constant hull shader seems to execute only a single thread per patch and multiprocessor. In order to obtain a meaningful metric for the effectiveness of back-patch culling, the number of culled patches is determined for 10K random views. Corresponding average cull rates for three popular models are listed in Table 3. Results are shown for the accurate cone of normals (NCONE), the approximate cone of normals (TCONE), and the parametric tangent plane (PPLANE) approach.

	Big Guy (3570)	Monster Frog (5168)	Killeroo (11532)
TCONE	1260 (35%)	1911 (37%)	3790 (33%)
NCONE	1601 (45%)	2286 (44%)	4685 (40%)
PPLANE	1729 (48%)	2478 (48%)	5206 (45%)

Table 3: *Average cull rates of different culling approaches: the accurate cone of normals [SAE93] (*NCONE*), the approximate cone of normals [SM88] (*TCONE*), and the parametric tangent plane [LNE11] (*PPLANE*).*

Results from the varying approaches for a more representative view are exhibited in Figure 15. Corresponding frame times for varying tessellation factors are shown in Figure 16. PPLANE requires 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 PPLANE's time per frame is lower than TCONE, which needs 0.36 ms, but only culls 2621 patches. Overall, PPLANE provides the best culling results, thus making it ideally suited for medium and higher tessellation rates. While TCONE provides the lowest culling rates, it is computationally the fasted. The computation costs and culling rates of NCONE are between the others. In almost every scenario (i.e., tessellation factos greater than 4), back-patch culling comes out ahead, making it beneficial to many real-time applications. A general drawback to back-patch culling methods, however, is the inability to address displaced patches. Hasselgren et al. [HMAM09] attempt to tackle this problem by introducing a Taylor expansion and applying interval arithmetic. Unfortunately, this involves significant computational overhead and restricts dynamic tessellation densities. A more practical solution is to rely on occlusion information (cf. Section 5.2), rather than to classify back-facing patches.

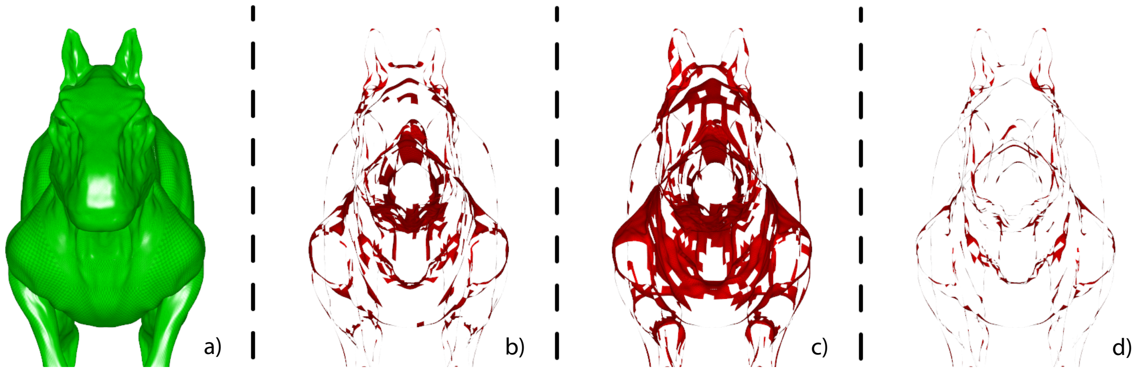


Figure 15: *Comparison of different back-patch culling strategies applied to the* Killeroo *model shown in a, which is composed of 11532 Bézier patches. For each comparison, wasted computations are visualized; i.e., areas processed by the tessellator with back-facing surface normals (less area is better). The cone of normals [SAE93] (b, 3697 patches culled) is effective, but costly for dynamic scenes. Its approximation from tangent and bi-tangent cones is faster to compute, but less precise [SM88] (c, only 2621 patches culled). Parametric tangent planes are faster than the cone of normals, and more effective [LNE11] (d, 4604 patches culled).*

5.2. Patch-based Occlusion Culling

Patch-based rendering can efficiently be sped up by employing back-face culling as described in Section 5.1. However, a major drawback is the inability to handle displaced surfaces, which are relevant the context of hardware tessellation. In addition, back-patch culling does identify occluded patches that are not visible in the rendered image. Nießner and Loop [NL12] introduce an approach to overcome this limitation by solely relying on occlusion information.

Occlusion Culling Pipeline Patch occlusion culling 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 beginning 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*) [GKM93] 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 and visible are rendered to complete the frame.

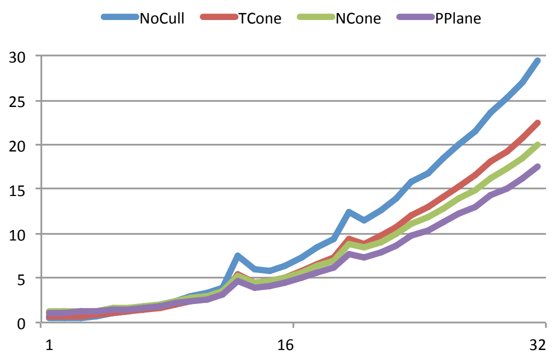


Figure 16: *Rendering times of the view shown in Figure 15 using different tessellation factors. Performance for the other test models and views are similar.*

Computing Occlusion Data In order to obtain occlusion information, the depth buffer resulting from rendering the visible patches is used to generate a Hi-Z map [GKM93], [SBOT08]. The Hi-Z map construction is similar to standard mipmapping where four texels are combined to determine a single texel in the next mip level. Instead of averaging texels, the value of the new texel is set to the maximum depth of the corresponding four child texels. Thus, within a region covered by a particular texel (no matter which mip level) a conservative bound is given, such that no objects with larger distance to the observer are visible at the texel's location.

Occlusion Cull Decision Nießner and Loop [NL12] use bicubic Bézier patches as a representative patch primitive, but the approach could be applied to other patching schemes as well. In order to determine the visibility of a patch, its axis-aligned bounding box (AABB) is computed in clip space. The AABB's front plane is then tested against the Hi-Z map. 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 for achieving better coverage.

Occlusion Culling of Displaced Patches A key advantage of occlusion culling is the ability to address displaced patches. While obtaining occlusion information (i.e., Hi-Z map computation) is the same as without displacement, determining patch bounds is different since vertices are being offset along the surface normal direction. One way to bound displaced patches is to extend patch OBBs with respect to patch displacements and corresponding patch normals [MHTAM10]. However, optimal patch bounds are obtained when applying a camera-aligned frustum (CAF) as introduced by Nießner and Loop [NL12], that is, a frustum whose side planes contain the origin in camera space, which corresponds to the eye point in viewing space. In order to determine a patch's CAF, control points are transformed to camera space. Then the minimum Z value (*minZ*) among the control points is determined, as well as the front plane of the CAF (a plane perpendicular to the viewing direction). Next, patch control points are projected onto the CAF's front plane. Side planes of the CAF are then obtained by computing the minimum and maximum x, y values of the projected control points $\mathbf{P}'_i$. The resulting frustum is the generalization of the screen space AABB and its projection will give the same result as the approach for non-displaced patches, if all displacements are zero.

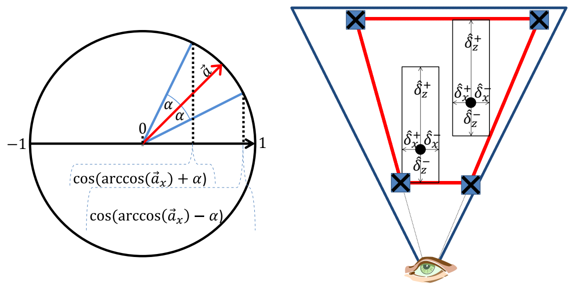


Figure 17: *The construction of a camera-aligned frustum (CAF). Left: the cone axis* **a** *and the aperture* α *determine the extension in the positive x direction (*δ_x^+*). Right: the CAF (red) is determined in camera space for two control points.*

In order to bound the range of patch normals to which displacements are applied, a cone of normals for each patch is computed. This can be achieved by employing either the accurate [SAE93] or approximate [SM88] cone of normal variant. In practice, typically the approximate variant is used since its computation is significantly cheaper. Further, scalar patch displacement values are bound with the maximum $D_{\max}$ as positive (or zero) and the minimum $D_{\min}$ as negative (or zero). The displacement bounds, the cone axis **a**, and aperture α are used to extend the CAF so that it will conservatively bound the displaced patch (see Figure 17 left). Therefore, bounds for each control point are computed according to the displacement extends. These bounds are given by an AABB, defined by $\mathbf{P}_{\max}$ and $\mathbf{P}_{\min}$. In order to construct the CAF, the corner points of all AABBs are projected onto the CAF front plane (see Figure 17, 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 above using a lookup in the Hi-Z map [GKM93]. Note that for a given cone of normals the CAF provides an optimal screen space bound. Figure 18 shows the difference between OBB bounds [MHTAM10] and CAF bounds [NL12] using the same cone.

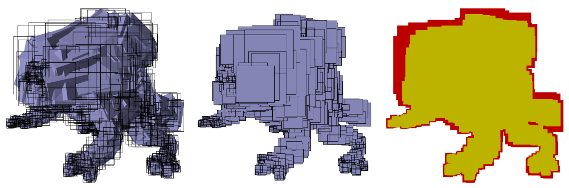
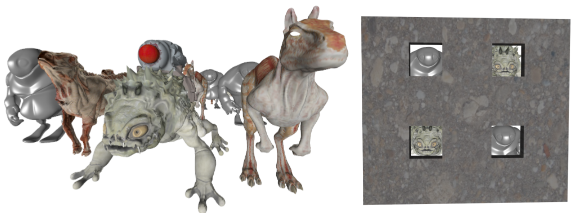


Figure 18: *Different bounding methods for displaced surfaces visualizing object (purple) and screen space (black quads) bounds: OBB (object-oriented bounding box [MHTAM10]), CAF (camera aligned frustum [NL12]) and a comparison between OBB (red) and CAF (yellow); both using the same, approximate cone of normals.*

5.2.1. Summary

Similar to back-patch culling (cf. Section 5.1), computations for occlusion culling can be most efficiently obtained in a separate compute shader. Resulting culling decisions are then read by the hull shader, and patch tessellation factors are set to zero for culled patches.

Occlusion Culling within Individual Objects Since culling is performed on a per-patch basis, it can be applied to individual models and ignore occluded patches. In order to obtain meaningful culling rate measurements, average culling rates are determined using 1K different cameras views and models with and without displacement. Each view contains the entire object to prevent view frustum culling influencing the measurements.



(a) 5.7 vs 12.1 ms (b) 3.4 vs 9.0 ms

Figure 19: *Patch-based occlusion culling [NL12] performed on per-patch basis: of patches with and without displacements. The images above are rendered with culling disabled and enabled; performance gains are below the respective image. (a)* 64.2% *patches culled, (b)* 70.6% *patches culled.*

	OBB		CAF	
	ACoN	CoN	ACoN	CoN
Frog	12.1%	14.0%	17.0%	18.4%
Frog^2	25.1%	26.4%	29.4%	30.9%
Cow	14.1%	15.6%	17.6%	18.7%
Cow^2	27.9%	29.1%	31.2%	32.7%

Table 4: *Average occlusion culling rates for displaced models; ^{2} denotes the respective model after one level of subdivision (i.e., having four times more patches). While* OBB *performs occlusion culling using the bounds by Munkerberg et al. [MHTAM10],* CAF *uses the camera-aligned frustum by Nießner and Loop [NL12].* ACoN *and* CoN *refer to the approximate and accurate cone of normal variant, respectively.*

Occlusion culling for non-displaced models is tested on the *Killeroo* and *Big Guy* model, achieving an average culling rate of 27.9% and 26.76%, respectively. In contrast, the best back-patch culling algorithm [LNE11] culls 38.7% and 37.8% of the patches. However, occlusion culling detects and culls significantly more patches with increased depth complexity since back-patch culling algorithms cannot take advantage of inter object/patch occlusions.

Occlusion culling for models with displacements is tested on the *Monster Frog* and the *Cow Carcass* model. Corresponding culling rates for different cull kernels are shown in Table 4. The camera-aligned frustum (CAF) is always more effective than the object-oriented bounding box (OBB) due to its better screen space bounds. Since the computational cost is equal, it is always better to use the CAF.

General Occlusion Culling for Scenes Realistic applications involve scenes with multiple objects consisting of both triangle and patch meshes. Two simple example scenes are shown in Figure 19 (the ACoN kernel is used for displaced models). In the first scene containing 27K patches a culling rate of 64.2% for the view shown in Figures 19(a) is achieved. Rendering is sped up by a factor of 2.1 (using a tessellation factor of 16). As expected, higher depth complexity results in more patches to be culled. Occlusion culling also benefits from triangle mesh occluders as shown in the second test scene (5.5K patches and 156 triangles). A culling rate of 70.6% and 30.5% is achieved for the view shown in Figures 19(b). As a result, render time is reduced by a factor of 2.6 (using a tessellation factor of 32).

Overall, patch-based occlusion culling significantly reduces tessellation and shading work load allowing for faster rendering. In contrast to back-patch culling approaches, the culling rate on single objects is slightly lower. However, occlusion culling is capable of handling displaced objects which are widely used in the context of hardware tessellation with increased effectiveness at higher depth complexities.

6. Collision Detection for Hardware Tessellation

Collision detection for hardware tessellation is particularly challenging since surface geometry is generated on-the-fly, based on dynamic tessellation factors and displacement values. Ideally, the same geometry used for rendering should be also taken into account to simulate physics. Nießner et al. [NSSL13] propose such an approach which provides accurate collision results for dynamic objects and runs entirely on the GPU. The key idea of this method is to detect collisions using voxelized approximations of hardware generated, possibly displaced and animated, detail geometry.

Collision Candidates The first step of the algorithm is to identify potential collision candidates. Therefore, object oriented bounding boxes (**OBBs**) are computed, and each object pair is tested for the corresponding OBB intersection. If there is no intersection, there can be no collision (early exit). Otherwise, a shared intersection volume between the two respective objects is defined. Next, a compute kernel is used to determine patches that are included in the shared volumes. In order to account for patch displacements, the cone of normals techniques, [SAE93] (accurate cone) or [SM88] (approximate cone), is used. Computing the corresponding patch bounds is similar to patch-based occlusion culling [NL12].

Voxelization The core idea of the collision test is to voxelize the rendering geometry of the current frame. In this stage, only patches that were identified as collision candidates are considered in order to improve performance. These are all transformed into the space of the intersecting volume, which is used as a basis for a binary voxelization. Inside patches are then voxelized using a modified version of the practical algorithm by Schwarz [Sch12]. In order to account for non-closed meshes, a backward and forward voxelization is required. Pseudo code for the latter is shown below.

```
//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);
}
```

Collision Detection Collision detection is based on the binary voxelizations, which are obtained every frame for each non-zero shared volume of all object pairs. Given two solid voxelizations, embedded in a shared space, collisions are detected by pairwise voxel comparisons. Since voxel representations are binary, 32 voxel comparisons can be performed using a single bitwise AND-operation. In addition to collision positions, corresponding surface normals are obtained based on voxel neighborhoods, i.e., normals are derived by using the weighted average of the vectors between the current voxel and its 26 neighbors. This test may be extended by an additional pass to determine patch IDs and u, v coordinates of collision points.

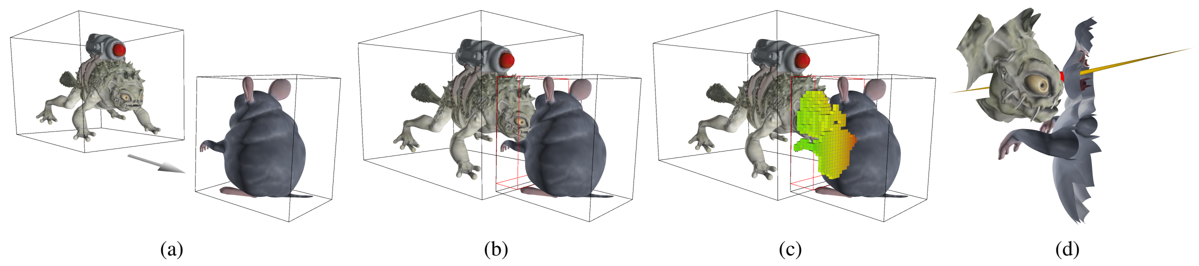


Figure 20: *Visualizing collision detection for dynamically tessellated objects. The* Monster Frog *is moving towards the* Chinchilla *(a). At one point the OBBs of both objects intersect (b, red box) and the containing geometry is voxelized (c). This provides a collision point and corresponding surface 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.*

Figure 20 shows how collision detection is applied to a simple test scene with two objects. Computational overhead is well below a millisecond.

7. Conclusion

In this survey we provided an overview of state-of-the-art rendering techniques for hardware tessellation. Many of these methods have already found application in industry; e.g., Pixar's *Open Subdiv* [Pix12], which is based on [NLMD12], is now part of various authoring tools such as Autodesk Maya [Auta], and is about to become an industry standard. Also, many games (e.g., the recently released game Call-of-Duty Ghosts) rely on terrain tessellation and subdivision surface rendering. However, we believe this is only the tip of the iceberg and future generations of video games and authoring tools will benefit from the massively parallel hardware tessellation architecture.

We hope this report will inspire future research in this very exciting area. For instance the fully automatic and artifact-free conversion process from polygonal meshes to a displaced surface representation remains to be an open-ended problem within this field of research. Such a conversion approach would greatly simplify the content creation pipeline, providing scalable content on different architectures and applications. In addition, locally adaptive detail at an intra-patch level for displaced surface representations would reduce memory requirements and facilitate richer mesh geometry.

Hardware vendors are currently working on tessellation support in next-generation mobile graphics processors [Nvi13], [Qua13]. This will greatly widen the range of applications of the algorithms presented in this survey and inspire new research. For instance, power and memory consumption are of paramount importance in mobile graphics. Therefore, future research could include novel displacement storage and (de)compression schemes to reduce memory consumption, or tessellation algorithms with less computational overhead, e.g. refraining from fractional tessellation while still providing continuous level-of-detail rendering.

References

[AB06] ANDREWS J., BAKER N.: Xbox 360 System Architecture. *IEEE Micro 26*, 2 (2006), 25–37. 1

[Auta] AUTODESK: Maya. http://usa.autodesk.com/maya. 8, 23

[Autb] AUTODESK: Mudbox. http://autodesk.com/mudbox. 8

[BA08] BOUBEKEUR T., ALEXA M.: Phong Tessellation. *ACM Trans. Graph. 27*, 5 (2008), 141:1–141:5. 2

[Ber04] BERTRAM M.: Lifting Biorthogonal B-spline Wavelets. In *Geometric Modeling for Scientific Visualization*. Springer, 2004, pp. 153–169. 15

[BL08] BURLEY B., LACEWELL D.: Ptex: Per-Face Texture Mapping for Production Rendering. In *Proceedings of EGSR* (2008), pp. 1155–1164. 13, 15

[Bli78] BLINN J. F.: Simulation of Wrinkled Surfaces. *Computer Graphics (Proceedings of SIGGRAPH) 12*, 3 (1978), 286–292. 11, 16

[Bon11] BONAVENTURA X.: Terrain and Ocean Rendering with Hardware Tessellation. In *GPU Pro 2*. A K Peters, 2011, pp. 3–14. 12

[BS05] BOUBEKEUR T., SCHLICK C.: Generic Mesh Refinement on GPU. In *Proceedings of HWWS* (2005), ACM, pp. 99–104. 2

[BS08] BOUBEKEUR T., SCHLICK C.: A Flexible Kernel for Adaptive Mesh Refinement on GPU. *Computer Graphics Forum 27*, 1 (2008), 102–114. 2

[Can11] CANTLAY I.: DirectX 11 Terrain Tessellation. Nvidia whitepaper, 2011. 10, 12

[Cas08] CASTAÑO I.: Next-Generation Rendering of Subdivision Surfaces. Talk at SIGGRAPH 08, 2008. 12, 17

[Cat74] CATMULL E.: *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, The University of Utah, 1974. 4

[CC78] CATMULL E., CLARK J.: Recursively Generated B-Spline Surfaces on Arbitrary Topology Meshes. *Computer-Aided Design 10*, 6 (1978), 350–355. 5

[CH02] CARR N. A., HART J. C.: Meshed Atlases for Real-Time Procedural Solid Texturing. *ACM Trans. Graph. 21*, 2 (2002), 106–131. 13

[CH04] CARR N. A., HART J. C.: Painting Detail. *ACM Trans. Graph. 23*, 3 (2004), 845–852. 11

[CHCH06] CARR N. A., HOBEROCK J., CRANE K., HART J. C.: Rectangular Multi-Chart Geometry Images. In *Proc. SGP'06* (2006), pp. 181–190. 11

[Coo84] COOK R.: Shade Trees. *Computer Graphics (Proceedings of SIGGRAPH) 18*, 3 (1984), 223–231. 11

[DC76] DO CARMO M.: *Differential Geometry of Curves and Surfaces*, vol. 1. Prentice-Hall, 1976. 16

[DKT98] DEROSE T., KASS M., TRUONG T.: Subdivision Surfaces in Character Animation. In *Proceedings of SIGGRAPH 98* (1998), Annual Conference Series, ACM, pp. 85–94. 5, 6

[DS78] DOO D., SABIN M.: Behaviour of Recursive Division Surfaces near Extraordinary Points. *Computer-Aided Design 10*, 6 (1978), 356–360. 4, 15, 16

[EML09] EISENACHER C., MEYER Q., LOOP C.: Real-Time View-Dependent Rendering of Parametric Surfaces. In *Proceedings of I3D'09* (2009), pp. 137–143. 2

[FFB*09] FISHER M., FATAHALIAN K., BOULOS S., AKELEY K., MARK W., HANRAHAN P.: DiagSplit: Parallel, Crack-Free, Adaptive Tessellation for Micropolygon Rendering. *ACM Trans. Graph 28*, 5 (2009), 150. 2

[FH05] FLOATER M. S., HORMANN K.: Surface Parameterization: A Tutorial and Survey. In *Advances in Multiresolution for Geometric Modelling*. Springer, 2005, pp. 157–186. 11

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Z-Buffer Visibility. In *Proceedings of SIGGRAPH* (1993), Annual Conference Series, ACM, pp. 231–238. 20, 21

[Gol91] GOLDBERG D.: What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv. 23*, 1 (Mar. 1991), 5–48. 12

[GP09] GONZÁLEZ F., PATOW G.: Continuity Mapping for Multi-Chart Textures. *ACM Trans. Graph. 28* (2009), 109:1–109:8. 11

[Gre74] GREGORY J.: Smooth Interpolation without Twist Constraints. *Computer Aided Geometric Design* (1974), 71–87. 8

[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 SIGGRAPH* (1994), Annual Conference Series, ACM, pp. 295–302. 5

[HLS12] HE L., LOOP C., SCHAEFER S.: Improving the Parameterization of Approximate Subdivision Surfaces. *Computer Graphics Forum 31*, 7 (2012), 2127–2134. 8

[HMAM09] HASSELGREN J., MUNKBERG J., AKENINE-MÖLLER T.: Automatic Pre-Tessellation Culling. *ACM Trans. Graph. 28*, 2 (2009), 19. 19

[HWFQ09] HE Y., WANG H., FU C.-W., QIN H.: A Divide-and-Conquer Approach for Automatic Polycube Map Construction. *Computers & Graphics 33*, 3 (2009), 369–380. 17

[JH12] JANG H., HAN J.: Feature-Preserving Displacement Mapping With Graphics Processing Unit (GPU) Tessellation. *Computer Graphics Forum 31*, 6 (2012), 1880–1894. 12

[JH13] JANG H., HAN J.: GPU-optimized indirect scalar displacement mapping. *Computer-Aided Design 45*, 2 (2013), 517–522. 12

[KM96] KUMAR S., MANOCHA D.: Hierarchical Visibility Culling for Spline Models. In *Proceedings of Graphics Interface* (1996), vol. 96, pp. 142–150. 17

[KMDZ09] KOVACS D., MITCHELL J., DRONE S., ZORIN D.: Real-Time Creased Approximate Subdivision Surfaces. In *Proceedings of I3D'09* (2009), ACM, pp. 155–160. 8

[Law65] LAWSON C.: The Smallest Covering Cone or Sphere. *SIAM Review 7*, 3 (1965), 415–416. 18

[LH06] LEFEBVRE S., HOPPE H.: Appearance-Space Texture Synthesis. *ACM Trans. Graph. 25*, 3 (2006), 541–548. 11

[Lin68] LINDENMAYER A.: Mathematical Models for Cellular Interactions in Development I. Filaments with one-sided Inputs. *Journal of Theoretical Biology 18*, 3 (1968), 280–299. 13

[LNE11] LOOP C., NIESSNER M., EISENACHER C.: Effective Back-Patch Culling for Hardware Tessellation. *Proceedings of Vision, Modeling, and Visualization (VMV)* (2011), 263–268. 18, 19, 20, 22

[Loo87] LOOP C.: *Smooth Subdivision Surfaces Based On Triangles*. Master's thesis, University of Utah, 1987. 4

[LP01] LUTTERKORT D., PETERS J.: Optimized Refinable Enclosures of Multivariate Polynomial Pieces. *Computer Aided Geometric Design 18*, 9 (2001), 851–863. 10

[LPRM02] LÉVY B., PETITJEAN S., RAY N., MAILLOT J.: Least Squares Conformal Maps for Automatic Texture Atlas Generation. *ACM Trans. Graph. 21*, 3 (2002), 362–371. 11

[LS08] LOOP C., SCHAEFER S.: Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Trans. Graph. 27*, 1 (2008), 8. 7

[LSNC09] LOOP C., SCHAEFER S., NI T., CASTAÑO I.: Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation. *ACM Trans. Graph. 28* (2009), 151:1–151:9. 8

[Mar95] MARUYA M.: Generating a Texture Map from Object-Surface Texture Data. *Computer Graphics Forum 14*, 3 (1995), 397–405. 13

[MBG*12] MARVIE J.-E., BURON C., GAUTRON P., HIRTZLIN P., SOURIMANT G.: GPU Shape Grammars. *Computer Graphics Forum 31*, 7 (2012), 2087–2095. 13

[MHTAM10] MUNKBERG J., HASSELGREN J., TOTH R., AKENINE-MÖLLER T.: Efficient Bounding of Displaced Bézier Patches. In *Proceedings of HPG'10* (2010), Eurographics Association, pp. 153–162. 18, 21, 22

[Mic09] MICROSOFT CORPORATION: Direct3D 11 Features, 2009. http://msdn.microsoft.com/en-us/library/ff476342(VS.85).aspx. 1, 2

[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. 8

[MYP08] MYLES A., YEO Y. I., PETERS J.: GPU Conversion of Quad Meshes to Smooth Surfaces. In *SPM '08: ACM Symposium on Solid and Physical Modeling* (2008), pp. 321–326. 8

[Nas87] NASRI A.: Polyhedral Subdivision Methods for Free-Form Surfaces. *ACM Transactions on Graphics (TOG) 6*, 1 (1987), 29–73. 5

[Nie13] NIESSNER M.: *Rendering Subdivision Surfaces using Hardware Tessellation*. Dr. Hut, 2013. 8

[NL12] NIESSNER M., LOOP C.: Patch-Based Occlusion Culling for Hardware Tessellation. *CGI* (2012). 20, 21, 22

[NL13] NIESSNER M., LOOP C.: Analytic Displacement Mapping using Hardware Tessellation. *ACM Trans. Graph. 32*, 3 (2013), 26. 10, 15, 17

[NLG12] NIESSNER M., LOOP C., GREINER G.: Efficient Evaluation of Semi-Smooth Creases in Catmull-Clark Subdivision Surfaces. *Computer Graphics Forum* (2012). 10

[NLMD12] NIESSNER M., LOOP C., MEYER M., DEROSE T.: Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces. *ACM Trans. Graph. 31*, 1 (2012), 6. 8, 10, 16, 23

[NSSL13] NIESSNER M., SIEGL C., SCHÄFER H., LOOP C.: Real-time Collision Detection for Dynamic Hardware Tessellated Objects. In *Proc. EG'13* (2013). 22

[Nvi12a] NVIDIA: CUDA C Programming guide, 2012. 1

[Nvi12b] NVIDIA: NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2012. http://www.nvidia.com/content/pdf/kepler/nvidia-kepler-gk110-architecture-whitepaper.pdf. 2

[Nvi13] NVIDIA: Kepler to Mobile, 2013. http://blogs.nvidia.com/blog/2013/07/24/kepler-to-mobile. 1, 23

[NYM*08] NI T., YEO Y. I., MYLES A., GOEL V., PETERS J.: GPU Smoothing of Quad Meshes. In *SMI '08: IEEE International Conference on Shape Modeling and Applications* (2008), pp. 3–9. 8

[PCK04] PURNOMO B., COHEN J. D., KUMAR S.: Seamless Texture Atlases. In *Proceedings of SGP'04* (2004), ACM, pp. 65–74. 11

[PEO09] PATNEY A., EBEIDA M., OWENS J.: Parallel View-Dependent Tessellation of Catmull-Clark Subdivision Surfaces. In *Proceedings of HPG* (2009), ACM, pp. 99–108. 2

[PG07] PAJAROLA R., GOBBETTI E.: Survey of Semi-Regular Multiresolution Models for Interactive Terrain Rendering. *The Visual Computer 23*, 8 (2007), 583–605. 12

[Pix12] PIXAR: OpenSubdiv, 2012. http://graphics.pixar.com/opensubdiv. 1, 8, 23

[Qua13] QUALCOMM: Qualcomm Technologies Announces Next Generation Qualcomm Snapdragon 805, 2013. http://www.qualcomm.com/media/releases/2013/11/20/qualcomm-technologies-announces-next-generation-qualcomm-snapdragon-805. 1, 23

[Rei97] REIF U.: A Refineable Space of Smooth Spline Surfaces of Arbitrary Topological Genus. *Journal of Approximation Theory 90*, 2 (1997), 174–199. 15, 16

[RNLL10] RAY N., NIVOLIERS V., LEFEBVRE S., LEVY B.: Invisible Seams. *Computer Graphics Forum (Proc. EGSR'10) 29*, 4 (2010), 1489–1496. 11

[SA12] SEGAL M., AKELEY K.: The OpenGL Graphics System: A Specification (Version 4.0 (Core Profile), Mar. 2012. http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf. 2

[SAE93] SHIRMUN L., ABI-EZZI S. S.: The Cone of Normals Technique for Fast Processing of Curved Patches. *Computer Graphics Forum 12*, 3 (1993), 261–272. 18, 19, 20, 21, 22

[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. 20

[Sch12] SCHWARZ M.: Practical Binary Surface and Solid Voxelization with Direct3D11. In *GPU Pro3: Advanced Rendering Techniques*. AK Peters Limited, 2012, p. 337. 22

[SGSH02] SANDER P. V., GORTLER S. J., SNYDER J., HOPPE H.: Signal-Specialized Parametrization. In *Proc. EGWR'02* (2002), pp. 87–98. 12

[SH02] SHEFFER A., HART J. C.: Seamster: Inconspicuous Low-Distortion Texture Seam Layout. In *VIS'02* (2002), IEEE Computer Society, pp. 291–298. 11

[SKS13] SCHÄFER H., KEINERT B., STAMMINGER M.: Real-Time Local Displacement using Dynamic GPU Memory Management. In *Proceedings of HPG'13* (2013), pp. 63–72. 14, 15

[SM88] SEDERBERG T., MEYERS R.: Loop Detection in Surface Patch Intersections. *Computer Aided Geometric Design 5*, 2 (1988), 161–171. 18, 19, 20, 21, 22

[SPM*12] SCHÄFER H., PRUS M., MEYER Q., SÜSSMUTH J., STAMMINGER M.: Multi-Resolution Attributes for Hardware Tessellated Meshes. In *Proceedings of I3D'12* (2012), ACM, pp. 175–182. 13

[SPM*13] SCHÄFER H., PRUS M., MEYER Q., SÜSSMUTH J., STAMMINGER M.: Multi-Resolution Attributes for Hardware Tessellated Objects. *IEEE Transactions on Visualization and Computer Graphics* (2013), 1488–1498. 14, 17

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

[Sta98] STAM J.: Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values. In *Proceedings of SIGGRAPH* (1998), Annual Conference Series, ACM, pp. 395–404. 4, 6

[Sti80] STINY G.: Introduction to Shape and Shape Grammars. *Environment and Planning B 7*, 3 (1980), 343–351. 13

[SWB98] SLOAN P.-P. J., WEINSTEIN D. M., BREDERSON J.: Importance Driven Texture Coordinate Optimization. *Computer Graphics Forum 17*, 3 (1998), 97–104. 12

[SWG*03] SANDER P. V., WOOD Z., GORTLER S. J., SNYDER J., HOPPE H.: Multi-Chart Geometry Images. In *Proceedings of SGP'03* (2003), pp. 146–155. 11

[Tat09] TATARCHUK N.: Fast High-Quality Rendering with Real-Time Tessellation on GPUs. In *ShaderX 7*, Engel W., (Ed.). Charles River Media, 2009. 12

[TBB10] TATARCHUK N., BARCZAK J., BILODEAU B.: Programming for Real-Time Tessellation on GPU. AMD whitepaper 5, 2010. 10, 12

[THCM04] TARINI M., HORMANN K., COGNONI P., MONTANI C.: PolyCube-Maps. *ACM Trans. Graph. 23*, 3 (2004), 853–860. 17

[TMJ98] TANNER C. C., MIGDAL C. J., JONES M. T.: The Clipmap: A Virtual Mipmap. In *Proceedings of SIGGRAPH 98* (1998), Annual Conference Series, ACM, pp. 151–158. 12

[Ulr02] ULRICH T.: Rendering Massive Terrains using Chunked Level of Detail Control. In *ACM SIGGRAPH 02 Talks* (2002), ACM, p. 34. 12

[VPBM01] VLACHOS A., PETERS J., BOYD C., MITCHELL J. L.: Curved PN Triangles. In *Proceedings of I3D'01* (2001), pp. 159–166. 7

[Wil83] WILLIAMS L.: Pyramidal Parametrics. *Computer Graphics 17*, 3 (1983), 1–11. 13, 15

[WMWF07] WATSON B., MÃIJLLER P., WONKA P., FULLER A.: Urban Design and Procedural Modelling. In *ACM SIGGRAPH 2007 Courses* (2007). 13

[XGH*11] XIA J., GARCIA I., HE Y., XIN S.-Q., PATOW G.: Editable Polycube Map for GPU-based Subdivision Surfaces. In *Proceedings of I3D'11* (2011), ACM, pp. 151–158. 17

[YBP12] YEO Y. I., BIN L., PETERS J.: Efficient Pixel-Accurate Rendering of Curved Surfaces. In *Proceedings of I3D'12* (2012), pp. 165–174. 10

[YS11] YUSOV E., SHEVTSOV M.: High-Performance Terrain Rendering Using Hardware Tessellation. *WSCG* (2011). 12

[Yus12] YUSOV E.: Real-Time Deformable Terrain Rendering with DirectX 11. In *GPU Pro 3*, Engel W., (Ed.). A K Peters, 2012. 12