EUROGRAPHICS 2009 / P. Dutré and M. Stamminger
(Guest Editors)

Volume 28 (2009), Number 2

Fast GPU-based Adaptive Tessellation with CUDA

Michael Schwarz and Marc Stamminger

University of Erlangen-Nuremberg

Abstract

Compact surface descriptions like higher-order surfaces are popular representations for both modeling and animation. However, for fast graphics-hardware-assisted rendering, they usually need to be converted to triangle meshes. In this paper, we introduce a new framework for performing on-the-fly crack-free adaptive tessellation of surface primitives completely on the GPU. Utilizing CUDA and its flexible memory write capabilities, we parallelize the tessellation task at the level of single surface primitives. We are hence able to derive tessellation factors, perform surface evaluation as well as generate the tessellation topology in real-time even for large collections of primitives. We demonstrate the power of our framework by exemplarily applying it to both bicubic rational Bézier patches and PN triangles.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Display algorithms; I.3.5 [Computer Graphics]: Curve, surface, solid, and object representations

1. Introduction

Triangles have become the standard rendering primitive in real-time graphics with graphics hardware being highly optimized for their processing. However, when it comes to modeling, usually other primitives like higher-order smooth surfaces are employed. Not only are they often less tedious to work with and provide a compact representation, but also naturally maintain visual smoothness irrespective of ever increasing display resolutions. For rendering, they are then tessellated to get a triangular mesh approximation.

In many applications, it is desirable to perform this tessellation on-the-fly instead of in a preprocessing step. For instance, it is much easier to animate a coarse control mesh than to deal directly with a fine triangular mesh. Also, by dynamically varying the tessellation's sampling density, adapting to changing view points becomes straightforward. For high performance, the surfaces should only be tessellated as fine as actually required because the evaluation per sample point can be rather expensive and unnecessarily small triangles negatively impact the GPU's effective parallelism. Consequently, an adaptive tessellation is performed where each surface primitive is tessellated to its optimal degree. To avoid cracks, the tessellation factors along boundary curves shared by multiple primitives must be chosen consistently.

The actual tessellation should preferably be carried out on the GPU, both to harness its computational power and to avoid having to transfer huge amounts of geometry data onto the GPU every frame. This is traditionally performed in a vertex-parallel way, where for each primitive vertices according to sample points in the primitive's domain are issued, either by explicitly rendering a tessellation pattern mesh or from a dedicated tessellation unit. In the vertex shader, the provided (u,v) domain coordinates are then employed to evaluate the actual surface points. On the other hand, recent non-graphics APIs like CUDA [NVI08] or CAL/Brook+ [AMD08] expose scatter memory writes to the programmer, i.e. a thread can write to multiple arbitrary memory locations. It thus becomes possible to approach the tessellation task at a different granularity of parallelism.

In this paper, we introduce a new framework for on-the-fly adaptive tessellation utilizing CUDA, called *CudaTess*. All major steps like deriving consistent tessellation factors, determining sample points, evaluating actual surface points, and creating the topology are run completely on the GPU. By adopting a single surface primitive as unit of parallelism, we are able to efficiently construct vertex and index buffers and can readily employ primitive-scale techniques like forward differencing [Wal90] which are not applicable in vertex-parallel settings. We demonstrate the potential of our framework on two concrete examples: bicubic rational Bézier patches (Sec. 3) and PN triangles (Sec. 4).

1.1. Related work

There exist many papers on CPU-guided adaptive tessellation targeted towards single modeling primitives like spline surfaces [CK01] and subdivision surfaces [SMFF04], as well as on dedicated hardware solutions for tessellating primitives like Bézier tensor-product patches [EBAB07], subdivision surfaces [ABS*05], or PN triangles [CK03]. Due to their often recursive and sequential nature, they are in general not well suited for full GPU implementation.

In contrast, tessellation pattern meshes for primitives with a triangular [BS05] or rectangular [GABK05] domain offer good GPU utilization, especially for higher tessellation degrees. Here, a primitive's domain is pre-tessellated for a number of tessellation factor configurations and the resulting *refinement patterns* are stored in vertex and index buffers. At runtime, for each primitive an appropriate pattern is rendered and the provided (u, v) domain coordinates are used in the vertex shader to derive the actual surface points.

Non-uniform tessellations can either be achieved by resorting to uniform refinement patterns and performing gap filling [GABK05, SSS06], or by creating non-uniform refinement patterns for all tessellation factor configurations [BS08]. Alternatively, dyadic uniform refinement patterns can be adapted on-the-fly in the vertex shader by collapsing vertices on the boundary to yield a semi-uniform tessellation [Tat08b, DRS09].

In a variant for coarse triangle meshes where only some triangles are refined, Dyken et al. [DRS08] first render all coarse triangles, degenerating those tagged for refinement. Then the remaining triangles are rendered with uniform refinement patterns, to which a kind of geomorphing is applied to geometrically achieve continuity across patches. However, since topologically still multiple inconsistent uniform tessellations are employed, T-vertices and hence rendering artifacts can occur.

In principle, tessellation can also be realized using the geometry shader stage and its amplification capability. Targeting applications with small refinement levels, Lorenz and Döllner [LD08] employ a geometry shader to emit a precalculated refinement pattern for each coarse triangle by copying it from a vertex buffer.

It is further possible to perform adaptive tessellation via recursive subdivision. In concurrent work, Patney and Owens [PO08] present a control-point- and micropolygon-parallel implementation for Reyes-style (sub)pixel-size tessellation of bicubic Bézier patches. Unfortunately, they only address the simple parts of the problem and postpone the challenging and non-obvious ones for future work, most notably the efficient avoidance/stitching of cracks both between adjacent sub-patches in the tessellation of a single Bézier patch and among Bézier patches. Moreover, they first write to equal-sized buffer slots and then compact the buffer after each subdivision step whereas we first determine the actually required slot sizes and then write to contiguous optimal-sized slots, avoiding any unnecessary copying and thus saving memory bandwidth.

Older-generation graphics hardware featured some native support for tessellating selected primitives like Bézier patches by NVIDIA's GeForce 3 [Mor01a] or PN triangles by ATI's TruForm [ATI01]. In contrast, recent AMD GPUs like the Xbox 360's Xenos or the Radeon R600 and R700 [Tat08a] provide a dedicated tessellation unit which supports a wider range of primitives in a more generic way by basically emitting refinement patterns. A similar but more general and optimized tessellation support will be introduced by future hardware for the upcoming Direct3D 11 [Gee08], which adds three more pipeline stages (hull shader, tessellator, and domain shader).

1.2. CUDA

CUDA [NVI08] is a non-graphics API for NVIDIA's G8x, G9x and GT200 GPUs that mainly targets compute-intense data-parallel applications. The provided hardware abstraction exposes more details about and additional capabilities of the GPUs than ordinary graphics APIs. In particular, memory writes are more flexible and data parallelism can be exploited at different granularities, enabling new applications.

A G8x/G9x/GT200 GPU features several streaming multiprocessors (SMs) which themselves are composed of eight scalar processors and 16 KB of *shared memory*, each. The processors are able to run multiple threads in a time-sliced way, with one thread being spawned per vertex, primitive or fragment in graphics mode, executing the current shader program. In CUDA, computations are organized in *kernels* which are run by a user-specified number of consecutively enumerated threads. The threads are grouped into *blocks* with all threads of a block being executed on the same SM. Each block is split into *warps*, groups of 2×16 threads which are executed in SIMD fashion. Multiple blocks are further structured in a *grid*.

In CUDA each thread can perform uncached reads from and writes to arbitrary locations in *global memory*. It is also possible to perform cached reads by resorting to textures. To exchange data with an OpenGL context, buffer objects can be mapped to CUDA's global memory. On the other hand, the SM-local shared memory allows for communication between threads of the same block, and is often employed as fast data cache where common data is first brought in from global memory collectively by several threads which then operate on it.

2. CudaTess framework: General approach

Our CUDA-based CudaTess framework adaptively tessellates all surface primitive instances in a scene, referred to as *patches*, in parallel and outputs the resulting triangle meshes

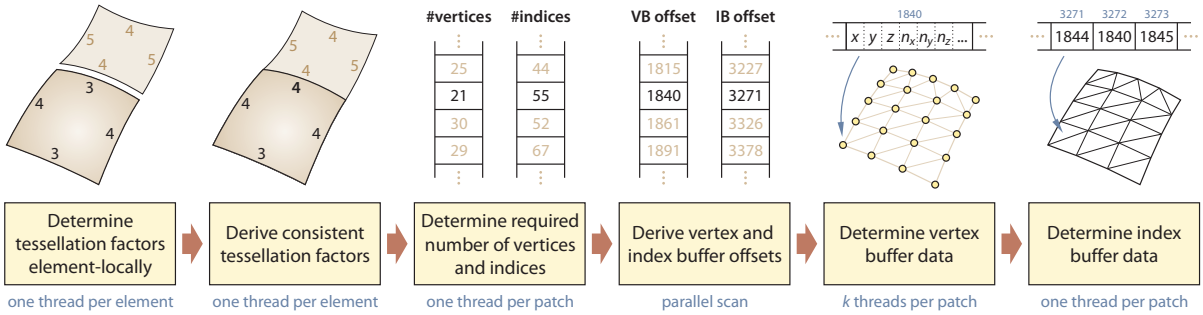


Figure 1: *Overview of the CudaTess framework for adaptive tessellation.*

into vertex and index buffers for rendering. While the general approach outlined in Fig. 1 is the same for all kinds of surface primitives, the actual implementation is usually specific to each kind of primitive.

In the first stage, the tessellation factors are determined using criteria like distance, screen-space extent, curvature or influencing silhouette. Depending on the adopted criterion, the computation is performed either on the level of patches, edges and boundary curves, or vertices. Each such *element* is usually treated independently of other elements. It is also possible to perform view-frustum or back-face culling at this stage if patches serve as elements. Affected patches can be flagged by setting their tessellation factors to zero.

To avoid any cracks in the tessellation, boundary curves shared by multiple patches must be sampled consistently. Therefore, we adapt the elements' tessellation factors appropriately in the next step using both neighborhood information provided by the application and the original element tessellation factors from the first stage. Note that such factor modifications are usually only required if the elements are patches.

Once the final tessellation factors have been computed, we derive the number of vertices and of indices required for the tessellation of each patch and store them in two arrays. Subsequently, we run an exclusive parallel scan [SHZO07] on the arrays to obtain the offsets within the vertex and the index buffer for the vertex and index data of each patch. To get the required total buffer sizes, we pad the input arrays with a zero and then read back the last entry of the scan results. If necessary, we resize the vertex and index buffer appropriately.

After that, a patch's surface is evaluated at sample points generated on-the-fly according to the tessellation factors. The resulting vertices are stored sequentially in the patch's slot within the vertex buffer (mapped into global memory). For several kinds of surface primitives, it is advantageous to employ more than one thread per patch for surface evaluation, e.g. one per xyz component ($k = 3$ threads). In particular, loading control points to fast shared memory becomes effective, the register count stays lower (enabling surfaces of rather high degrees) and memory writes are more coherent within a warp. Finally, the index buffer data is written for each patch, thus creating the topology of its tessellation.

The resulting buffers can then be used directly for rendering. Usually, the vertex data features an object id which allows for selecting object-local shading options analogously to instanced rendering. In case of multi-pass rendering, the buffer data can readily be reused without having to reevaluate the patch surfaces.

Since an explicit representation of the tessellation result is available, it is also possible to post-process it before rendering. For instance, assume only tessellations corresponding to dyadic subdivision are performed and geomorphing is desired. Then the initial surface evaluation can be done completely at the finest involved subdivision level. In a post-process on the vertex buffer, the vertices to be morphed are adapted. They easily get their coarser-level positions by interpolating between their adjacent vertices, which can readily be accessed, thus avoiding many redundant computations. As another example, it is possible to use neighborhood information to copy generated vertex position data for boundary curves across adjacent patches, ensuring absolute crack-freeness irrespective of numerical inaccuracies. Note that even in case control points and sampling parameters are consistent among patches, the numerical results may slightly differ if the involved parameterization directions differ and hence mathematically equivalent terms are evaluated in a different order. Also note that this problem affects all tessellation approaches which treat each patch individually, including refinement patterns and AMD's tessellation units. It can also be alleviated by reorganizing the evaluation in a parameterization-direction-independent way [Cas08], which however usually entails an increased arithmetic operation count.

To provide more insight into the actual realization as well as the flexibility of the framework, we describe two concrete examples in the following sections which were chosen to often differ significantly in the single steps.

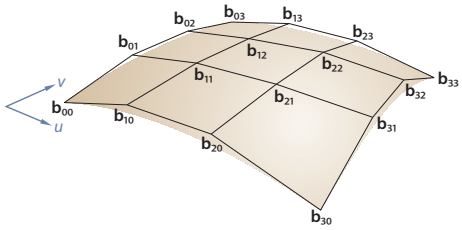


Figure 2: *Control net of a bicubic Bézier patch.*

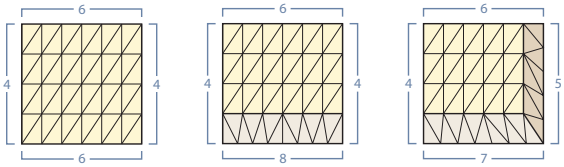


Figure 3: *Quad tessellation patterns: completely uniform (left), with one (center) or two transition regions (right).*

3. Bicubic rational Bézier patches

As our first example, we consider bicubic rational Bézier tensor-product patches [Far02]. A patch (cf. Fig. 2) is completely specified by 16 homogeneous 4D control points $\mathbf{b}_{ij} = (w_{ij}\mathbf{p}_{ij}, w_{ij})$, $i,j = 0,\ldots,3$, composed of 3D points $\mathbf{p}_{ij}$ and associated weights w_{ij}. The patch surface is given by

$$\mathbf{p}(u,v) = \frac{\big(\mathbf{b}(u,v)\big)_{xyz}}{\big(\mathbf{b}(u,v)\big)_{w}} = \frac{\sum_{i=0}^{3}\sum_{j=0}^{3} w_{ij}\mathbf{p}_{ij}B_i^3(u)B_j^3(v)}{\sum_{i=0}^{3}\sum_{j=0}^{3} w_{ij}B_i^3(u)B_j^3(v)},$$

where $B_i^n(t) = \binom{n}{i}t^i(1-t)^{n-i}$ are the Bernstein polynomials. To get the (unnormalized) normal $\tilde{\mathbf{n}}(u,v) = \mathbf{t}_u(u,v) \times \mathbf{t}_v(u,v)$, we first compute the tangent in u direction

$$\mathbf{t}_u(u,v) = \frac{\big(\mathbf{b}_u(u,v)\big)_{xyz} - \big(\mathbf{b}_u(u,v)\big)_{w}\mathbf{p}(u,v)}{\big(\mathbf{b}(u,v)\big)_{w}},$$

making use of the 4D derivative

$$\mathbf{b}_u(u,v) = 3\sum_{i=0}^{2}\sum_{j=0}^{3}\big(\mathbf{b}_{i+1,j} - \mathbf{b}_{i,j}\big)B_i^2(u)B_j^3(v);$$

$\mathbf{t}_v(u,v)$ is then obtained analogously from $\mathbf{b}_v(u,v)$.

Overview For adaptive tessellation, the control points are provided in an array. Our CudaTess implementation then derives a bounding box for each patch's control points and performs view-frustum culling. If a patch doesn't get culled, tessellation factors in u and v direction are computed. Taking neighborhood information into account, we then derive consistent tessellation factors for the four boundary curves of each patch. Based on these factors, patch-wise vertex and index counts are determined, and then buffer offsets are derived. Finally, the actual tessellation is performed by generating vertex and index buffer data.

Tessellation factors Consider a patch's uniform tessellation $\mathbf{l}(u,v)$ over the domain $U = [0,1]^2 \ni (u,v)$ with tessellation factors m and n in u and v direction, respectively; i.e. the samples $\mathbf{p}(i\delta_u, j\delta_v)$, $0 \le i \le m$, $0 \le j \le n$, $\delta_u = 1/m$, $\delta_v = 1/n$ are used. According to Zheng and Sederberg [ZS00], the approximation error $\sup_U \|\mathbf{p}(u,v) - \mathbf{l}(u,v)\| \le \varepsilon$ if the domain step sizes δ_u and δ_v have been chosen such that

$$D_{uu}\delta_u^2 + 2D_{uv}\delta_u\delta_v + D_{vv}\delta_v^2 \le 8\varepsilon \min_{ij}\{w_{ij}\}$$

holds, where

$$D_{uu} = 6 \max_{\substack{i=0,1\\ j=0,\ldots,3}} \Big\{\big\|\big(\mathbf{b}_{i+2,j} - 2\mathbf{b}_{i+1,j} + \mathbf{b}_{i,j}\big)_{xyz}\big\| + (r-\varepsilon)\big|w_{i+2,j} - 2w_{i+1,j} + w_{i,j}\big|\Big\}$$

and similarly D_{uv} and D_{vv} are bounds on the second derivatives, with $r = \max_{ij}\|\mathbf{p}_{ij}\|$.

For each patch, we therefore first derive ε from a user-specified screen-space error bound and then determine r, $\min_{ij}\{w_{ij}\}$, D_{uu}, D_{uv} and D_{vv}. Note that the last three quantities depend on ε and hence cannot be precomputed. Next, we derive step sizes δ_u and δ_v [ZS00] and the corresponding tessellation factors. For each patch, we finally store one tessellation factor per boundary curve. To avoid cracks among two adjacent patches, we always take the maximum of the two involved patches' tessellation factors.

Tessellation pattern In general three different classes of tessellation factor configurations can arise (cf. Fig. 3). If the two tessellation factors in u direction (for $\mathbf{p}(u,0)$ and $\mathbf{p}(u,1)$) are equal as well as those in v direction, the patch is tessellated uniformly. Otherwise, the tessellation is composed of a uniform core and one or two transition regions where corresponding tessellation factors vary. Each factor for the interior part is always derived from the minimum of the two tessellation factors for the same direction, which by construction provides an upper bound on the required sampling density.

Each transition region is tessellated by a Bresenham-like approach [Mor01b], yielding well-shaped triangles in domain space. A state variable Q is initialized to the difference of the two tessellation factors and subsequently updated. Its sign controls from which of the two involved transition sides the next segment is picked for creating a triangle.

Vertex data update We distribute the vertex data generation for each patch across four consecutive threads, one for each component (x, y, z, w). Note that a patch's threads belong to the same warp and hence run in lockstep and can easily communicate via shared memory. First, the control points are collectively loaded to shared memory. Then vertex data is successively determined according to the tessellation pattern implied by the tessellation factors, and written to the vertex buffer. Each thread first evaluates its component of $\mathbf{b}(u,v)$, $\mathbf{b}_u(u,v)$ and $\mathbf{b}_v(u,v)$. Then a patch's first three threads compute the position $\mathbf{p}(u,v)$ and normal $\tilde{\mathbf{n}}(u,v)$, with required

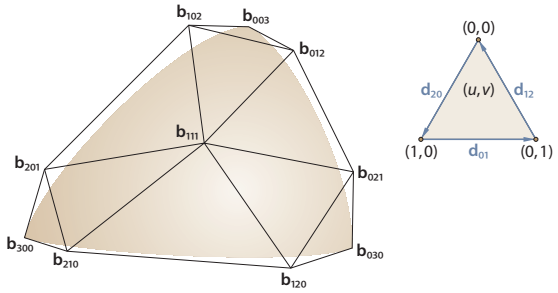


Figure 4: *Cubic triangular Bézier patch: control net (left) and parametric domain (right).*

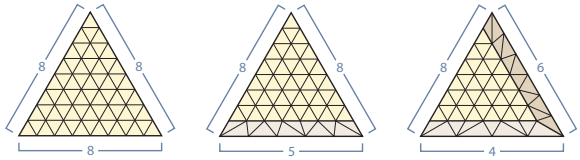


Figure 5: *Triangle tessellation patterns: completely uniform (left), with one (center) or two transition regions (right).*

quantities like $\big(\mathbf{b}(u,v)\big)_w$ being exchanged via shared memory. In addition, for each vertex, we emit (u,v) coordinates and an object id obtained from a texture.

Thanks to performing surface evaluation patch-wise and thus processing a single patch's vertices sequentially instead of in parallel, we are able to employ techniques reusing results from computations carried out for previous vertices. We exemplarily adopted forward differencing [Wal90], which reduces the evaluation of $\mathbf{b}(u,v)$, $\mathbf{b}_u(u,v)$ and $\mathbf{b}_v(u,v)$ to a small number of additions for all vertices with the same v (or u) coordinate but the first.

4. PN triangles

For our second example, we applied the CudaTess framework to PN triangles [VPBM01]. Recall that a PN triangle features a geometric component described by a cubic triangular Bézier patch (cf. Fig. 4)

$$\mathbf{b}(u,v) = \sum_{i+j+k=3} \mathbf{b}_{ijk} \frac{3!}{i!\,j!\,k!} u^i v^j (1-u-v)^k$$

as well as a normal field used for shading which is specified by a quadratic Bézier triangle $\mathbf{n}(u,v)$ with control points $\mathbf{n}_{ijk}$. All control points $\mathbf{b}_{ijk}$ and $\mathbf{n}_{ijk}$ are constructed from a given coarse triangle with vertex positions $\mathbf{P}_\ell$ and normals $\mathbf{N}_\ell$, $\ell = 1,2,3$. In particular, $\mathbf{b}_{300} = \mathbf{P}_1$, $\mathbf{n}_{200} = \mathbf{N}_1$, etc.

Overview A collection of coarse triangle meshes is provided as input for adaptive tessellation. For each corresponding PN triangle, we first derive the control points $\mathbf{b}_{ijk}$ as well as their bounding box and test it against the viewing frustum. If the patch is potentially visible, we further determine its normal field control points $\mathbf{n}_{ijk}$ and store them along with the $\mathbf{b}_{ijk}$ for the vertex data generation stage, before finally computing the tessellation factors along the patch's three boundary curves. Next, we utilize adjacency information for the input triangles to make the factors consistent across patches. After determining the number of vertices and indices required for tessellating each PN triangle, buffer offsets are computed. In a last step, the actual vertex and index buffer data is generated, yielding the final tessellation.

Tessellation factors According to Filip et al. [FMM86], the error of approximating a C^2 surface $\mathbf{f}$ by a flat triangle $\mathbf{l}$ over the domain triangle $T = \triangle\big((u_0,v_0);(u_0+\delta_u,v_0);(u_0,v_0+\delta_v)\big)$ is bounded by

$$\sup_T \|\mathbf{f}(u,v) - \mathbf{l}(u,v)\| \le \tfrac{1}{8}\left(D_{uu}\delta_u^2 + 2D_{uv}\delta_u\delta_v + D_{vv}\delta_v^2\right)$$

where

$$D_{uv} = \sup_T \left\| \frac{\partial^2 \mathbf{f}(u,v)}{\partial u \partial v} \right\|,$$

$$D_{uu} = \sup_T \left\| \frac{\partial^2 \mathbf{f}(u,v)}{\partial u^2} \right\|, \quad D_{vv} = \sup_T \left\| \frac{\partial^2 \mathbf{f}(u,v)}{\partial v^2} \right\|.$$

Applied to our triangular setting with the domain directions

$$\mathbf{d}_{01} = (-1,1), \qquad \mathbf{d}_{12} = (0,-1), \qquad \mathbf{d}_{20} = (1,0),$$

and using the identity $2\delta_u\delta_v \le \delta_u^2 + \delta_v^2$, we get

$$\tfrac{1}{8}\Big(\max\big\{\sup_T \big\|D^{1,1}_{\mathbf{d}_{01},\mathbf{d}_{20}}\mathbf{b}(u,v)\big\|, \sup_T \big\|D^{1,1}_{\mathbf{d}_{01},\mathbf{d}_{12}}\mathbf{b}(u,v)\big\|\big\} + \sup_T \big\|D^2_{\mathbf{d}_{01}}\mathbf{b}(u,v)\big\|\Big)\,\delta^2_{\mathbf{d}_{01}} \le \tfrac{1}{2}\varepsilon$$

for the step size $\delta_{\mathbf{d}_{01}}$ in direction $\mathbf{d}_{01}$ and a given error bound ε. $D^2_{\mathbf{d}_{01}}\mathbf{b}(u,v)$ denotes a second and $D^{1,1}_{\mathbf{d}_{01},\mathbf{d}_{20}}\mathbf{b}(u,v)$ a mixed directional derivative of $\mathbf{b}(u,v)$ [Far02]. Since these derivatives are themselves (linear) Bézier triangles, their magnitudes can easily be bounded by utilizing the magnitudes of the corresponding control points. Analogous relationships hold for the other two step sizes $\delta_{\mathbf{d}_{12}}$ and $\delta_{\mathbf{d}_{20}}$.

For each PN triangle we first derive the bounds on the directional derivatives required for determining the step sizes. In principle, they can be precomputed if the coarse base triangles are only subjected to rigid transformations. Next, we compute the ε value corresponding to a user-specified screen-space error bound, and derive the three step sizes $\delta_{\mathbf{d}_{01}}$, $\delta_{\mathbf{d}_{12}}$, $\delta_{\mathbf{d}_{20}}$ such that the error bound is satisfied. From these the corresponding tessellation factors for the patch's boundary curves are determined and stored. Finally, to obtain consistent tessellation factors among neighboring PN triangles, we again take the maximum of the involved patches' factors.

Tessellation pattern Each PN triangle's tessellation consists of a uniformly tessellated part and, if its tessellation factors differ, one or two additional transition regions (cf. Fig. 5). The tessellation factor for the uniform part is derived from the maximum of all three boundary curve factors. Like

(a) Tea table (b) Killeroo (c) Killeroo herd (d) Spheres

Figure 6: *Example scenes composed of adaptively tessellated bicubic rational Bézier patches.*

Scene	Patches	Triangles	Vertices	Indices	Adaptive tessellation	Adaptive tessellation with forward differencing	Only shading (reusing buffer data)	Tessellation factors	Final factors & buffer offsets	Vertex buffer data update	Vertex buffer data update with forward differencing	Index buffer data update
Tea table	332	41372	25798	55316	206.5 Hz	260.6 Hz	758 Hz	0.39 ms	0.09 ms	2.40 ms	1.40 ms	0.56 ms
Killeroo	11532	100930	110515	213709	156.2 Hz	194.2 Hz	706 Hz	0.82 ms	0.16 ms	3.32 ms	2.04 ms	0.61 ms
Killeroo herd	92256	345751	514753	827389	48.1 Hz	62.6 Hz	366 Hz	4.18 ms	0.42 ms	12.22 ms	7.38 ms	1.19 ms
Spheres	32000	402000	393600	675600	98.4 Hz	108.8 Hz	362 Hz	1.62 ms	0.20 ms	4.54 ms	3.54 ms	0.96 ms
Spheres (close-up)	32000	272301	216925	391401	92.8 Hz	123.1 Hz	320 Hz	1.04 ms	0.19 ms	5.54 ms	2.92 ms	0.77 ms

Table 1: *Bézier patches: scene statistics, rendering performance and tessellation time break-down for some example scenes.*

in the quad case, for each transition region, a Bresenham-like algorithm is executed with the tessellation factors of the interior uniform part and of the boundary curve as input. However, in case two transition regions occur, we apply a small modification to get triangles of better shape in domain space: The algorithm is run starting from where the transition regions meet with the tessellation factor of the interior part increased by one but skipping the first generated triangle (which always gets constructed from the first (only virtual) segment of the virtually extended interior part's boundary).

Vertex data update For each patch, the vertex data generation is distributed across three consecutive threads, one for each component. At first, the control points corresponding to the vertex positions and normals of the underlying coarse triangle are collectively loaded to shared memory. If all tessellation factors equal one, the coarse triangle is not refined and we just output the corresponding vertex data. Otherwise, the remaining control points are brought into shared memory and vertex data is successively computed according to the tessellation pattern and written to the mapped vertex buffer. In addition to evaluating the position $\mathbf{b}(u,v)$ and the normal $\mathbf{n}(u,v)$, we also output (u,v) coordinates and an object id.

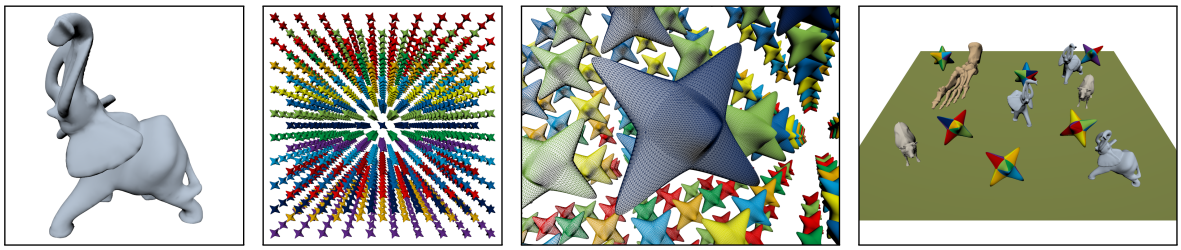
5. Results and discussion

All results were obtained on a Pentium IV 3 GHz with an NVIDIA GeForce 8800 GTS (G80) video card at a viewport of 1024×768 and a screen-space error bound of 0.5 pixels.

Bicubic rational Bézier patches Some example scenes to which we applied our implementation are shown in Figs. 6 and 8. As listed in Table 1, we achieve real-time performance even for large numbers of patches to tessellate. Note that in case of the close-up view of the spheres scene (Fig. 8), only those patches get actually tessellated which pass the view-frustum test. Also recall that the adaptive tessellation is generated from scratch each frame, requiring only the patches' control points and their neighborhood relationships as input. Consequently, both the view-point and the patch control points can be freely animated without negatively affecting the tessellation performance. Since CudaTess outputs a vertex buffer and an index buffer, the data can be reused for multi-pass rendering without necessitating any recomputations, enabling even higher frame rates.

As the time break-down shows, generating the vertex data and hence evaluating the surface is the most dominating part. Even if all tessellation factors are low, like in the spheres scene, this costly computation can be sped up significantly via forward differencing. In scenes with a large number of visible patches, like the Killeroo herd scene, determining tessellation factors also consumes a significant share of time, mainly because of having to compute the bounds D_{uu}, D_{uv}, and D_{vv}. Note that these two most time-demanding stages are required by all on-the-fly tessellation approaches.

PN triangles Example scenes are depicted in Figs. 7 and 9, covering a wide range of both coarse triangle counts and generated tessellation triangle counts. As shown in Table 2, even large tessellation loads can be coped with in real-time.



(a) Elephant (b) Star field (c) Star field (close-up) (d) Model zoo

Figure 7: *Example scenes composed of triangle meshes adaptively refined according to the PN triangle scheme.*

Scene	Base triangles	Triangles	Vertices	Indices	Adaptive tessellation	Adaptive tessellation using pre-computed data	Only shading (reusing buffer data)	Tessellation factors	Tessellation factors using pre-computed data	Final factors & buffer offsets	Vertex buffer data update	Index buffer data update
Double torus	1536	15320	16048	43328	408 Hz	409 Hz	1413 Hz	0.39 ms	0.35 ms	0.21 ms	0.60 ms	0.48 ms
Elephant	21540	79208	116399	257894	230 Hz	259 Hz	1076 Hz	0.91 ms	0.43 ms	0.26 ms	1.49 ms	0.68 ms
Star field	41496	1050688	864928	2547040	60.1 Hz	63.6 Hz	186 Hz	1.41 ms	0.47 ms	0.33 ms	6.81 ms	2.57 ms
Star field (close-up)	41496	379982	249385	692306	118 Hz	122 Hz	301 Hz	0.75 ms	0.45 ms	0.30 ms	2.84 ms	1.11 ms
Model zoo	80578	160200	311469	548730	109 Hz	135 Hz	524 Hz	2.42 ms	0.65 ms	0.50 ms	3.33 ms	0.93 ms

Table 2: *PN triangles: scene statistics, rendering performance and tessellation time break-down for some example scenes.*

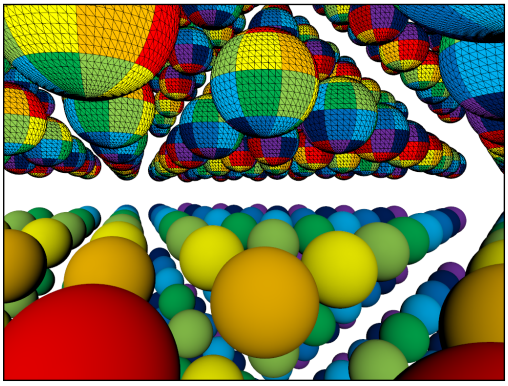


Figure 8: *Close-up view of the spheres scene from Fig. 6 d: adaptive tessellation (top) and rendering result (bottom).*

Both the PN triangles' control points as well as the adaptive tessellation are derived anew each frame, thus freely supporting animations. Again, computing the vertex data is the most time-consuming part. Note that the kernel for deriving tessellation factors can be stream-lined in case of static scenes by precomputing the control points, their bounding box and the bounds on the directional derivatives, yielding a measurable speed-up for complex scenes.

Limitations While adopting a patch as unit of parallelism is crucial for efficient GPU-guided generation of varying amounts of geometry, and enables acceleration techniques like forward differencing, it may prevent utmost utilization of the GPU's processors. First, since each patch is processed by one single thread (or a small number of threads), the number of patches must be reasonably high to not leave processors completely idle. Nevertheless, as the tea table scene demonstrates, even smaller patch counts with high tessellation rates are handled very well. Second, threads within a warp may diverge and finish at different times if the tessellation patterns of the warp's patches differ, which impacts the effective parallelism. However, even when manually imposing a kind of worst-case workload, we only observed a reasonably low reduction in throughput compared to a best case for SIMD parallelism.

Since CudaTess requires the tessellation to be stored in a vertex and an index buffer, it may consume a noticeable amount of memory. Especially in case of a large number of patches being excessively tessellated, one hence may consider applying CudaTess sequentially to subsets of the scene. On the other hand, only the explicit availability of the tessellation result enables post-processing of the vertex data as well as fast buffer reuse for multi-pass rendering.

Driver dependency All reported performance data was obtained with ForceWare 175.19 under Windows XP. However, during our tests we observed that the employed driver version often measurably impacts performance. For instance, compared to the listed frame rates, a different driver resulted in a 25% performance gain for the Killeroo herd scene. Another driver showed only small performance impacts of using forward differencing. Moreover, the alternative approach

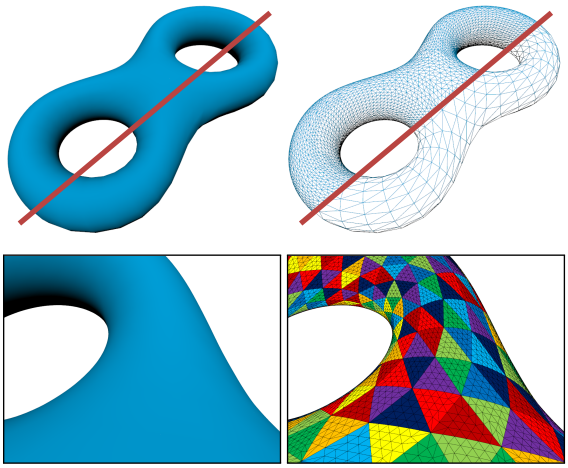


Figure 9: *Double torus: PN triangle surface with its base triangle mesh (top), and close-up (bottom).*

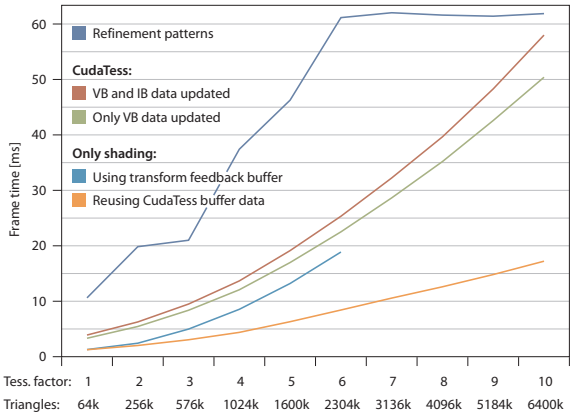


Figure 10: *Rendering performance comparison between (instanced) refinement patterns [BS05] and CudaTess for the spheres scene (Fig. 6 d) with uniformly tessellated patches.*

of instanced rendering of refinement patterns [BS08] was up to four times slower on a recent CUDA-specific driver than the figures reported below.

Geometry shader We note that, in principle, a patch-wise approach could also be pursued utilizing the geometry shader stage. However, we don't consider this worthwhile due to several severe restrictions faced. For instance, the shader's output is basically a vertex list, requiring interior vertices (with all their attributes) to be emitted at least twice. Also, a geometry shader can output at most 1024 float values per input primitive, limiting the maximum tessellation factors, e.g. to 12 for a triangular domain if just positions and normals are emitted. Even worse, in practice, the shader output must be restricted to far less than the possible 1024 values to avoid severe performance drops. Although this limitation may be alleviated by a multi-pass approach, overall performance will be negatively affected by the overhead entailed. Furthermore, available performance data for approaches employing a geometry shader for outputting a tessellation pattern [DRS09, LD08] suggest that even in case of small tessellation factors, alternative approaches like rendering refinement patterns [BS08] perform significantly better.

Comparison to refinement patterns Rendering a refinement pattern for each input patch [BS08] excels in case of extremely high tessellation degrees, even for small patch counts. On the other hand, such setups are not encountered in a large class of scenes. For a first comparison at more common tessellation degrees, we picked the spheres scene, selecting identical tessellation factors for all patches. Note that consequently only a single refinement pattern is utilized and instancing can be employed to limit API overhead to just one draw call. As the performance data in Fig. 10 shows, our CudaTess implementation, which simplifies to generating vertex and index buffer data, turns out to be faster. In particular, when only regenerating the vertex data and using a precomputed index buffer, our superior surface evaluation performance becomes obvious. We attribute this mainly to caching the control points in shared memory. A further speed-up would be possible if forward differencing were employed.

For a second comparison, we rendered all our example scenes with the refinement pattern approach. In a preparation step, we determined the tessellation factors and generated just the actually required refinement patterns. The frame rates listed in Table 3 show that in case of traditional "immediate" rendering [BS08], where for each (visible) patch the corresponding pattern is rendered with an own draw call, CudaTess is significantly faster. The only exception is the tea table scene with its small number of patches and hence limited potential for high GPU utilization with a patch-parallel approach like CudaTess. On the other hand, the encountered high tessellation rates make this scene an ideal case for the refinement pattern technique.

The "immediate" rendering of refinement patterns incurs a high API invocation overhead, which clearly dominates for smaller tessellation factors. This can be alleviated by batching together all patches using the same refinement pattern and employing instancing for rendering. As indicated in Table 3, for more complex scenes, the performance improves significantly. Nevertheless, our CudaTess implementation is still clearly faster for scenes composed of many bicubic rational Bézier patches. In case of PN triangles, however, batched rendering of refinement patterns appears to be often faster than CudaTess. One main reason for this difference is that the evaluation of PN triangles is cheaper than that of Bézier patches. Moreover, note that for the timings we built the batches, and rearranged and uploaded the control points

	Refinement patterns												CudaTess	
	Integer tessellation				Dyadic tessellation				Semi-uniform tessellation					Fwd.
Scene	#P	#T	Immed.	Batched	#P	#T	Immed.	Batched	#P	#T	Immed.	Batched		diff.
Tea table	81	41372	595 Hz	365 Hz	26	80808	515 Hz	262 Hz	5	237664	303 Hz	51.0 Hz	213 Hz	271 Hz
Killeroo	179	100930	43.7 Hz	146 Hz	70	125396	43.8 Hz	133 Hz	4	226518	41.2 Hz	40.7 Hz	173 Hz	225 Hz
Killeroo herd	109	345751	5.6 Hz	31.8 Hz	74	357110	5.6 Hz	31.5 Hz	4	568326	5.1 Hz	10.4 Hz	59.7 Hz	84.3 Hz
Spheres	7	402000	15.6 Hz	49.1 Hz	7	586400	15.7 Hz	37.8 Hz	2	774400	14.8 Hz	13.6 Hz	111 Hz	124 Hz
Spheres (close-up)	54	272301	49.4 Hz	86.1 Hz	13	406200	49.4 Hz	62.7 Hz	3	457472	46.3 Hz	26.0 Hz	102 Hz	139 Hz
Double torus	23	15320	307 Hz	979 Hz	11	23660	307 Hz	977 Hz	3	25104	285 Hz	979 Hz	431 Hz	
Elephant	65	79206	24.8 Hz	303 Hz	31	102542	24.6 Hz	305 Hz	4	119226	22.9 Hz	310 Hz	286 Hz	
Star field	4	1050688	13.0 Hz	110 Hz	3	2410336	12.9 Hz	60.3 Hz	2	2540544	12.0 Hz	54.9 Hz	69.5 Hz	
Star field (close-up)	57	379982	102 Hz	180 Hz	7	777472	99.8 Hz	124 Hz	3	792320	92.1 Hz	118 Hz	131 Hz	
Model zoo	74	160200	6.6 Hz	86.3 Hz	35	198540	6.6 Hz	85.9 Hz	5	224263	6.1 Hz	86.2 Hz	156 Hz	

Table 3: *Performance comparison between various approaches for rendering refinement patterns [BS08, Tat08b, DRS09] and CudaTess. The determination of tessellation factors as well as preparations for instanced rendering are not included in the timings. (#P: number of different patterns used; #T: number of rendered triangles; Immed.: one draw call per patch; Batched: one draw call per pattern)*

for instanced rendering during the preparation step. In practice, this must be done during runtime and can reasonably be expected to consume some amount of time, thus reducing the achievable frame rate. Also note that while CudaTess allows reusing buffer data in case of multi-pass rendering, the refinement pattern approach necessitates a complete rerun, which is always slower. Using transform feedback is not really a viable option because only a triangle soup is recorded, causing valence-n vertices to be stored n times and thus also necessitating a huge buffer.

Recall that refinement patterns need to be precomputed and stored in vertex and index buffers. However, the number of patterns can easily explode, especially for quad domains. For example, creating quad refinement patterns for all configurations of tessellation factors up to 32 results in almost 1 GB of 16-bit index buffer data. Remember that for our timings, we circumvented this issue by only creating the refinement patterns actually used for one specific viewpoint, which is not feasible in the general setting. Consequently, it is questionable whether providing patterns for all combinations of integer tessellation factors is reasonable or possible in practice.

One option is to restrict tessellation factors to power-of-two values, leading to a dyadic tessellation. However, as indicated by the data in Table 3, compared to ordinary integer tessellation this can incur significant evaluation overhead. Tessellating a triangular domain with a factor of 32 instead of 17, for instance, results in 228% more vertices. Adopting a semi-uniform tessellation [Tat08b, DRS09] where only dyadic uniform refinement patterns are used further decreases the pattern count but also increases the evaluation overhead. Yet another option is to decompose each refinement pattern into patterns for the uniformly tessellated core and the transition regions. But this further increases the number of draw calls and also leads to redundant evaluations. On the other hand, CudaTess can efficiently deal with any tessellation factor configuration since it constructs tessellation patterns on-the-fly.

6. Conclusion and future work

We have presented CudaTess, a novel and flexible framework for the crack-free adaptive tessellation of surfaces. Utilizing CUDA, the tessellation task is parallelized on the patch level. All major steps like deriving consistent tessellation factors, evaluating the surface at vertices, and generating index data according to the tessellation topology are executed completely on the GPU. In particular, we have shown how to successfully employ CUDA for the efficient and purely GPU-based dynamic generation of geometry without requiring any CPU assistance except invoking a few kernels.

We have demonstrated CudaTess with two concrete examples, rational Bézier patches and PN triangles. In both cases, even large collections of patches are adaptively tessellated on-the-fly in real-time. Compared to other approaches like rendering refinement patterns, CudaTess excels especially when the surface evaluation is rather expensive. Then, techniques made possible by our patch-parallel approach, like caching control points in shared memory or forward differencing, can make a significant impact on performance. Therefore, we reckon that in such cases our technique is competitive even against using a dedicated tessellation unit that outputs (u, v) domain coordinates.

For future work, it would be interesting to port CudaTess to CAL/Brook+ [AMD08] or to the upcoming Direct3D 11 with its compute shaders [Boy08] and compare the performance against utilizing a dedicated tessellation unit. Furthermore, we believe that our approach is well suited for future hardware architectures like Larrabee [SCS*08] with their increased flexibility and programmability, and we would like to eventually adapt CudaTess to them.

Another avenue of future work is applying CudaTess

to rendering subdivision surfaces using recently developed Bézier approximations [LS08, MNP08]. The necessary on-the-fly conversion into Bézier patches could be incorporated analogously to how we currently derive a PN triangle's control points.

Acknowledgements

This work was funded by the European Union within the CROSSMOD project (EU IST-014891-2). The Killeroo model is courtesy of Headus; the double torus, elephant, bones, cow and star models are provided courtesy of INRIA and MPII, respectively, by the AIM@SHAPE Shape Repository.

References

[ABS*05] AMOR M., BÓO M., STRASSER W., HIRCHE J., DOGGETT M.: A meshing scheme for efficient hardware implementation of Butterfly subdivision using displacement mapping. *IEEE Computer Graphics and Applications 25*, 2 (2005), 46–59.

[AMD08] AMD INC.: AMD stream SDK, 2008. http://ati.amd.com/technology/streamcomputing/sdkdwnld.html.

[ATI01] ATI TECHNOLOGIES INC.: *TruForm*. White paper, 2001. http://ati.amd.com/products/pdf/truform.pdf.

[Boy08] BOYD C.: Direct3D 11 compute shader: More generality for advanced techniques, 2008. Presentation, *Gamefest 2008*. http://www.microsoft.com/downloads/details.aspx?FamilyID=9f943b2b-53ea-4f80-84b2-f05a360bfc6a.

[BS05] BOUBEKEUR T., SCHLICK C.: Generic mesh refinement on GPU. In *Graphics Hardware 2005* (2005), pp. 99–104.

[BS08] BOUBEKEUR T., SCHLICK C.: A flexible kernel for adaptive mesh refinement on GPU. *Computer Graphics Forum 27*, 1 (2008), 102–113.

[Cas08] CASTAÑO I.: Tessellation of displaced subdivision surfaces in DX11, 2008. Presentation, *Gamefest 2008*. http://www.microsoft.com/downloads/details.aspx?FamilyId=a484d275-9360-41dd-abd4-86d4f1218d0c.

[CK01] CHHUGANI J., KUMAR S.: View-dependent adaptive tessellation of spline surfaces. In *Proceedings of I3D 2001* (2001), pp. 59–62.

[CK03] CHUNG K., KIM L.-S.: Adaptive tessellation of PN triangle with modified Bresenham algorithm. In *Proceedings of SOC Design Conference 2003* (Nov. 2003), pp. 448–452.

[DRS08] DYKEN C., REIMERS M., SELAND J.: Real-time GPU silhouette refinement using blended Bézier patches. *Computer Graphics Forum 27*, 1 (2008), 1–12.

[DRS09] DYKEN C., REIMERS M., SELAND J.: Semi-uniform adaptive patch tessellation. *Computer Graphics Forum* (2009). To appear.

[EBAB07] ESPINO F. J., BÓO M., AMOR M., BRUGUERA J. D.: Hardware support for adaptive tessellation of Bézier surfaces based on local tests. *Journal of Systems Architecture 53*, 4 (2007), 233–250.

[Far02] FARIN G.: *Curves and Surfaces for CAGD: A Practical Guide*, 5th ed. Morgan Kaufmann, 2002.

[FMM86] FILIP D., MAGEDSON R., MARKOT R.: Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design 3*, 4 (1986), 295–311.

[GABK05] GUTHE M., ÁKOS BALÁZS, KLEIN R.: GPU-based trimming and tessellation of NURBS and T-spline surfaces. *ACM Transactions on Graphics 24*, 3 (2005), 1016–1023.

[Gee08] GEE K.: Direct3D 11 tessellation, 2008. Presentation, *Gamefest 2008*. http://www.microsoft.com/downloads/details.aspx?FamilyID=2d5bc492-0e5c-4317-8170-e952dca10d46.

[LD08] LORENZ H., DÖLLNER J.: Dynamic mesh refinement on GPU using geometry shaders. In *WSCG 2008 Full Papers Proceedings* (2008), pp. 97–104.

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

[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.

[Mor01a] MORETON H.: Higher order surfaces, 2001. Presentation. http://developer.nvidia.com/object/higher_order_surfaces.html.

[Mor01b] MORETON H.: Watertight tessellation using forward differencing. In *Proceedings of Workshop on Graphics Hardware 2001* (2001), pp. 25–32.

[NVI08] NVIDIA CORPORATION: *NVIDIA CUDA Programming Guide 2.0*, 2008.

[PO08] PATNEY A., OWENS J. D.: Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics 27*, 5 (2008).

[SCS*08] SEILER L., CARMEAN D., SPRANGLE E., FORSYTH T., ABRASH M., DUBEY P., JUNKINS S., LAKE A., SUGERMAN J., CAVIN R., ESPASA R., GROCHOWSKI E., JUAN T., HANRAHAN P.: Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3 (2008), Article 18.

[SHZO07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J. D.: Scan primitives for GPU computing. In *Graphics Hardware 2007* (2007), pp. 97–106.

[SMFF04] SETTGAST V., MÜLLER K., FÜNFZIG C., FELLNER D.: Adaptive tesselation of subdivision surfaces. *Computers & Graphics 28*, 1 (2004), 73–78.

[SSS06] SCHWARZ M., STAGINSKI M., STAMMINGER M.: GPU-based rendering of PN triangle meshes with adaptive tessellation. In *Vision, Modeling, and Visualization 2006* (2006), pp. 161–168.

[Tat08a] TATARCHUK N.: Advanced topics in GPU tessellation: Algorithms and lessons learned, 2008. Presentation, *Gamefest 2008*. http://developer.amd.com/gpu_assets/Tatarchuk-Tessellation(Gamefest2008).pdf.

[Tat08b] TATARINOV A.: Instanced tessellation in DirectX10, 2008. Presentation, *Game Developers Conference 2008*. http://developer.download.nvidia.com/presentations/2008/GDC/Inst_Tess_Compatible.pdf.

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

[Wal90] WALLIS B.: Tutorial on forward differencing. In *Graphics Gems*, Glassner A. S., (Ed.). Academic Press, 1990, pp. 594–603.

[ZS00] ZHENG J., SEDERBERG T. W.: Estimating tessellation parameter intervals for rational curves and surfaces. *ACM Transactions on Graphics 19*, 1 (2000), 56–77.