Real-Time View-Dependent Rendering of Parametric Surfaces

Christian Eisenacher*
University of Erlangen-Nuremberg

Quirin Meyer†
University of Erlangen-Nuremberg

Charles Loop‡
Microsoft Research

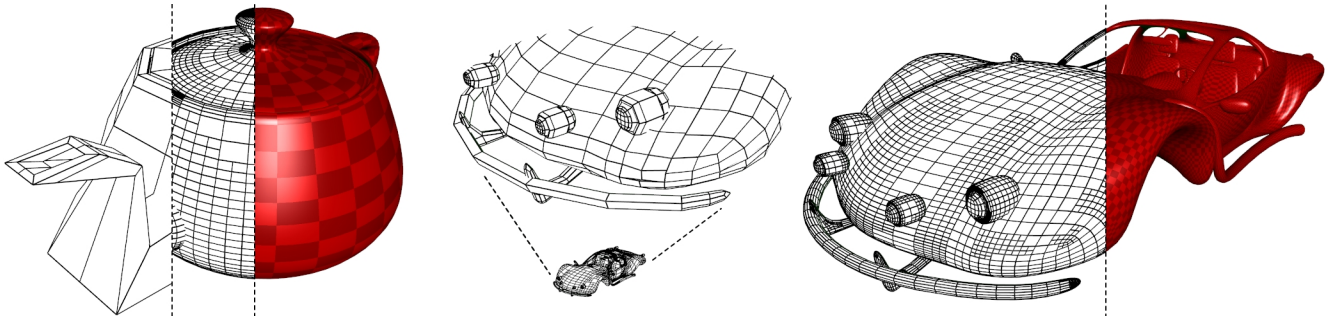

Figure 1: *We adaptively subdivide rational Bézier patches until a view-dependent error metric is satisfied. For a 1600x1200 image of the car model (right) we render 192k quads at 143 fps on a NVIDIA GTX 280 – including CUDA transfer overheads, texturing, Phong shading, and 16x multisampling.*

Abstract

We propose a view-dependent adaptive subdivision algorithm for rendering parametric surfaces on parallel hardware. Our framework allows us to bound the screen space error of a piecewise linear approximation. We naturally assign more primitives to curved areas while keeping quads large for flatter parts of the model and avoid cracks resulting from the polygonal approximation of non-uniform patch subdivision. The overall algorithm is simple, fits current GPUs extremely well, and is surprisingly fast while producing little to no artifacts.

CR Categories: I.3.7 [Computing Methodologies]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture;

Keywords: adaptive surface rendering, GPGPU, real-time rendering

1 Introduction

Polygon rendering has been the workhorse of computer graphics for decades. It is straightforward to efficiently cull, clip, and determine pixel coverage for these simple primitives. Smoothly varying shapes can be approximated to an arbitrary precision by polygons. When this approximation is done as an offline pre-process, silhouette faceting and normal field inaccuracies occur when the shape is viewed up close. Similarly, computation is wasted for sub-pixel polygons when the shape is viewed from afar. Moreover, finely approximated objects can consume considerable disk space, memory, and bus bandwidth. Finally, dense approximations are expensive to animate, since a high number of vertices must be touched for each new pose.

A much more flexible and compact approach would be to retain a higher order surface representation well into the graphics pipeline, where a polygonal approximation could be generated on-the-fly. We propose to do this using adaptive patch subdivision and design an algorithm that takes advantage of the massive parallelism available with current (ca. 2008) GPGPU technology. At a high level our approach can be described as:

> *Recursively subdivide each initial patch until its screen space projection satisfies an error metric. Approximate each of these patches with a quadrilateral and render.*

On parallel hardware this poses a number of challenges. To overcome them we present:

- a very efficient mapping onto parallel hardware,
- a simple, effective screen space error metric, and
- a method to avoid cracks induced by adaptive subdivision.

After summarizing previous work, we will describe our algorithm and its individual components. Then we will present some representative renderings and timings for our implementation in CUDA [Nvidia Corporation 2008]. Finally we will discuss the advantages and shortcomings of our approach and identify avenues for future research.

2 Previous Work

Many papers concerned with rendering parametric surfaces have appeared over the last 35 years; we mention a few highlights and recent work that is relevant to GPU acceleration.

Catmull [1974] proposed that bicubic polynomial patches could be recursively subdivided until their screen space projection was no larger than a pixel; the depth and color of these patches were then assigned to the nearest pixel. Due to the large number of arithmetic

*e-mail: Christian.Eisenacher@cs.fau.de
†e-mail: Quirin.Meyer@cs.fau.de
‡e-mail: Charles.Loop@microsoft.com

operations needed for patch subdivision, together with the large amount of memory needed to store sub-pixel patches, this algorithm and related variants have been used only for offline rendering. Since Catmull's algorithm subdivides patches to pixel level, cracks caused by differing subdivision levels between adjacent patches are not apparent. Trying to combine adaptive subdivision with polygon rasterization will lead to these artifacts. A clever solution to this problem was proposed by Clark [1979].

An algorithm for adaptively tessellating trimmed NURBS surfaces was presented by Rockwood et al. [1989]. Their idea was to a priori determine a sampling rate for the screen space projection of patch edges. This was estimated by finding an upper bound on the magnitude of the first derivative of patch boundary curves. A dynamic triangulation process was used to tessellate a patch domain and the surface was evaluated at the resulting *uv* vertices. Due to the triangulation step, the performance of this algorithm was not impressive. Also, the a priori sampling bound only guarantees maximum (not minimum) sample spacing, so oversampling may occur.

In the work of Guthe et al. [2005], bicubic control points are passed from the CPU to the GPU and a vertex program evaluates the patch with the help of pre-tessellated domain meshes that uniformly sample the parameter space of the patch. Several domain meshes are placed on the GPU to account for different tessellation levels. In order to hide cracks between adjacent patches of different tessellation levels, patch boundary curves are re-sampled and rendered as line strips. A similar idea has appeared in the work of Boubekeur and Schlick [2007]. While these schemes leverage parallelism to evaluate surface patches, the adaptivity is limited by the relatively small number of domain mesh tessellations.

Dynamic domain tessellation will become a new fixed function pipeline stage called the *tessellator unit* on the next generation of graphics hardware (expected release summer 2009) [Microsoft Corporation 2008]. This unit will leverage SIMD parallelism via concurrent surface evaluations. The locations of these surface samples will be determined a priori by the tessellator unit based on user provided edge tessellation factors. The adaptivity of the resulting polygonal approximation will be determined by relatively few inputs that apply to the entire patch. Inner-patch adaptivity to guarantee pixel accuracy at silhouettes, to avoid oversampling flat regions, or inefficient small triangles, is not possible with this approach to GPU tessellation.

Hardware implementations of recursive subdivision have previously not gained traction due to the large high speed stack memory needed to push and pop sub-patches visited in a depth first traversal. Recently, Patney and Owens [2008] have shown that by using a breadth first traversal along with parallel scan primitives, recursive subdivision can be efficiently mapped to data parallel hardware. Their objective was to mimic the *bound and split* approach of the Reyes renderer [Cook et al. 1987] and generate sub-pixel polygons via dicing. Our objective here is to only generate as many approximating polygons as needed to capture the parametric surface faithfully. Due to algorithmic improvements, and since we do not overwhelm the pipeline with an excessive number of small, compute intensive primitives, the performance we report is two orders of magnitude better than Patney and Owens.

3 View-Dependent Rendering of Parametric Surfaces

In order to render a parametric surface we transform its initial control structure – the *ur-patches* – into a view-dependent polygonal approximation of the real surface. We do this by recursively subdividing each ur-patch until the sub patches satisfy our error metric.

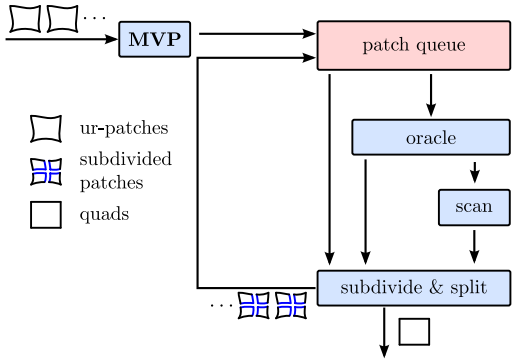

Figure 2: *Algorithmic overview: We subdivide the initial ur-patches breadth first until they satisfy our error metric.*

Figure 2 shows a high level view of how this can be done very effectively on parallel hardware using a breadth first approach. The key idea is to maintain a *patch queue* and examine all patches independently and in parallel. When they satisfy our metric we render them as quads[1]. Otherwise they are subdivided and the sub patches are placed in the queue to be tested again in the next iteration. In the following sections we will describe the individual components and our design decisions in detail.

3.1 Ur-patches and Their Transformation

We work with rational bicubic Bézier patches due to their popularity and expressive power (e.g. exact representation of spheres, cones, tori, etc.). They are also invariant under projective mappings, allowing us to transform the ur-patches into clip space by the composite Model-View-Projection matrix **(MVP)** and directly subdivide the transformed patches. This reduces the number of vertex transformations, both in the oracle and during the final rendering, by 2-4 orders of magnitude.

3.2 Oracle

The oracle kernel, shown in Figure 3, prepares patches for subdivision. We examine each patch from the patch queue independently and compute a *decision bit field* to guide the split stage described in Section 3.3. Further we prepare indices for child patches and polygons using a standard parallel prefix scan technique [Blelloch 1990; Sengupta et al. 2007].

Algorithm 1 outlines the oracle. The individual functions are designed to exploit the Single-Instruction-Multiple-Threads **(SIMT)** [Nvidia Corporation 2008] nature of current hardware and we have one lightweight thread running in parallel for each control point. The relevant functions are:

- *computeDegreeElevatedPatch(CPs)*: Each thread determines its control point of the degree elevated bilinear patch (Figure 4(b)) by averaging the corner vertices of the bicubic patch.
- *computeBilinearErrorBitfield(CPs, DEP_CPs)*: Each thread computes the distance between its control point and the degree elevated bilinear approximation in screen space. If the desired precision is reached, the corresponding bit is set (see Figure 4 and Section 6.3). Since corner bits will always be

[1] As direct access to the frame buffer is currently not supported in CUDA and context switches are expensive, we store the quads in a vertex buffer and render them in a single draw call after subdivision is finished.

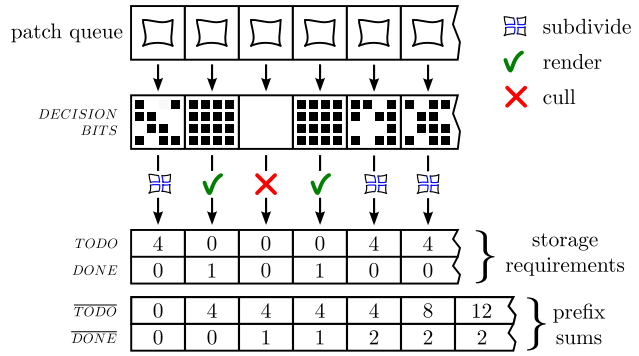

Figure 3: *Oracle: We examine all patches in parallel and compute a decision bit field and storage requirements. A parallel prefix scan transforms the latter into array positions for the subdivision kernel.*

set, we cannot confuse culled patches (no bits set) with those requiring subdivision.

- *patchReady(BF)*: If all bits in BF are set, the patch fulfills our precision requirements and the split stage will generate primitives. Otherwise the patch will be subdivided.

- *backfacing(DEP_CPs)*: We compute the z components of the bilinear patch normals at the corners. If all are negative, the quad is back-facing and can be culled. This reduces the number of normals to be computed and primitives to be rendered roughly by half.

- *computeStorage(DECISION_BITS)*: The split kernel will need space for four subdivided patches (TODO) or one finished patch (DONE).

Algorithm 1 Oracle kernel: for each patch in parallel

```
1:  CPs = loadControlPoints(patchQueue)
2:  BB = computeBoundingBox(CPs)
3:  if (outsideFrustum(BB)) then
4:      DECISION_BITS = CULL
5:  else
6:      DEP_CPs = computeDegreeElevatedPatch(CPs)
7:      BF = computeBilinearErrorBitfield(CPs, DEP_CPs)
8:      DECISION_BITS = BF
9:      if (patchReady(BF) and (backfacing(DEP_CPs)) then
10:         DECISION_BITS = CULL
11:     end if
12: end if
13: (TODO, DONE) = computeStorage(DECISION_BITS)
```

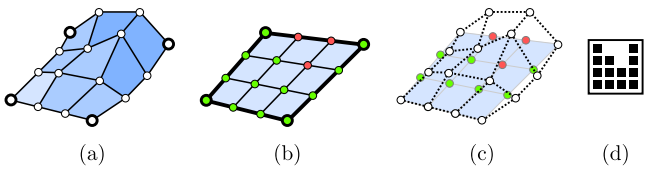

Figure 4: *ComputeBilinearErrorBitfield(): If the difference (c) between a control point (a) and its degree elevated bilinear counterpart (b) in screen space is within a given tolerance, the corresponding bit is set (d). We subdivide patches until all bits are set.*

3.3 Subdivide and Split

The split kernel, visualized in Figure 5, subdivides patches that need further refinement into four sub-patches (see Section 6.2) and generates primitives for patches that are ready to be rendered. The prefix sums $\overline{\text{TODO}}$ and $\overline{\text{DONE}}$ determine the locations in patch queue and vertex buffer. As illustrated by Algorithm 2 this is conceptually very simple. The interesting functions are:

- *notCulled(DECISION_BITS)*: At least the corner bits are set.

- *makeEdgesLinear(CPs, DECISION_BITS)*: This is to avoid cracks. Details are described in Section 3.4.

- *subdivide(CPs)*: Allocate four threads per control point row, one for each component. Subdivide into shared memory along u. Subdivide the two sub-patches into global memory along v. Note: The stores to global memory are completely coalesced.

- *generatePrimitives(CPs)*: Evaluate normals from the ur-patches using one thread per normal. Store vertices, normals, and texture coordinates at the locations specified by $\overline{\text{DONE}}$.

Algorithm 2 Split kernel: for each patch in parallel

```
1: if (notCulled(DECISION_BITS)) then
2:     CPs = loadControlPoints(patchQueue)
3:     CPs = makeEdgesLinear(CPs, DECISION_BITS)
4:     if (patchReady(DECISION_BITS)) then
5:         newPatchQueue[TODO] = subdivide(CPs)
6:     else
7:         vertexBuffer[DONE] = generatePrimitives(CPs)
8:     end if
9: end if
```

3.4 Crack Prevention

To prevent cracks without losing patch independence and parallelism, we adapt the idea presented by Clark [1979] to Bézier curves: We consider the decision bits computed previously. If all bits corresponding to a patch edge are set, indicating that its screen space projection is within tolerance of being a line, we replace the inner two control points by their linear counterparts. This will turn the curved patch edge into a line. Since corresponding decision bits of adjacent patches will be set identically, the patches will share the now linear edge. Further subdivision cannot deviate from this line, so cracks cannot appear, as illustrated in Figure 6.

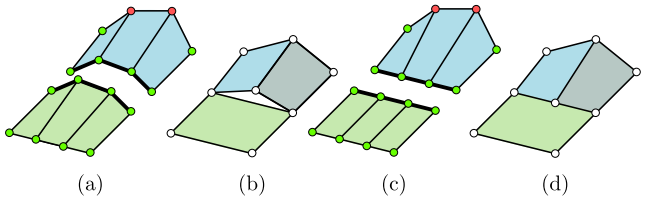

Figure 6: *Crack prevention: Patches sharing an edge (a), are subdivided to different levels, generating a crack (b). If an edge is close to being linear (control points green) we set it to linear (c) and subdivision cracks are avoided (d).*

Though not reported by Clark, forcing nearly linear edges to lines does not preserve C^1 continuity between patches, resulting in visible shading discontinuities across patch boundaries. As we keep track of domain coordinates for texture mapping purposes, we can evaluate shading normals from the unmodified ur-patches and avoid shading artifacts.

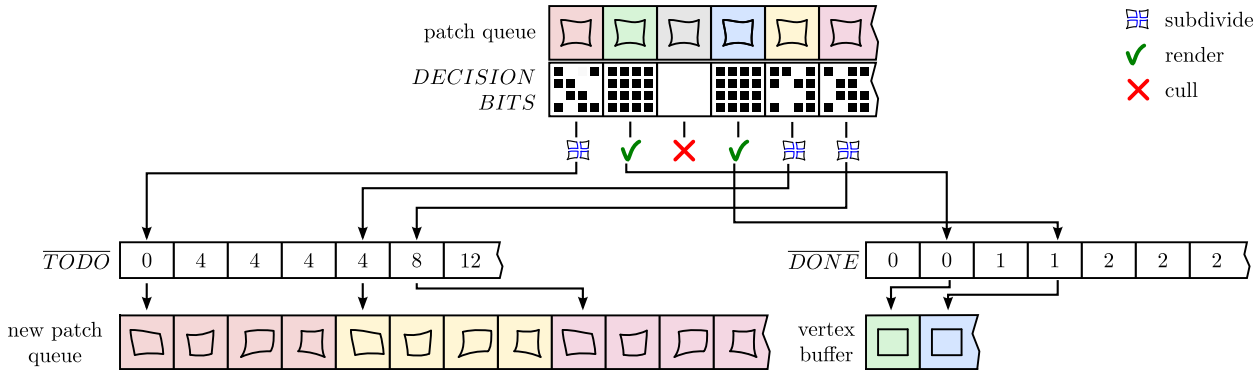

Figure 5: *Split kernel: Using the previously computed decision bit fields, we either subdivide the patches from the patch queue or create primitives to be rendered. Prefix sums obtained from a parallel scan determine the position of new elements.*

4 Results

To benchmark our implementation, we render four models with different geometric complexity at 512x512 and 1600x1200 pixels. Figure 7 shows the models and the pose used for the measurements in this section. We subdivide the patches until the screen space error is below 0.5 pixel and evaluate 16 normals for each finished bicubic patch. By writing 9 quads, created from the vertices of the degree elevated bilinear patch, into the vertex buffer we avoid managing a tiny normal map for each patch. As shown in Table 2, we still generate a rather low number of primitives.

For real-time applications with very complex scenes, we can reduce memory and bandwidth consumption considerably by evaluating only four normals and placing a single quad in the vertex buffer. While this sacrifices shading quality for speed, we maintain artifact-free silhouettes (see Section 6.3); this approach would work well in combination with normal mapping.

In fact, the killeroo was the only model to show reduced shading quality with four normals (see inset in Figure 7). We believe the reason for this is its very fine initial control mesh, resulting in a low number of subdivisions needed to guarantee smooth silhouettes. Rendering only one quad per patch, we basically reduce detail in a very crude way.

Table 3 shows the total time needed per frame. We observe that the view-dependent subdivision accounts only for a fraction of the time spent per frame. Evaluating normals and generating primitives, mapping memory, and the final draw call require considerable time. For our most complex model, the killeroo, we present a detailed analysis of where time is spent in Table 1.

	no MSAA		16x MSAA	
shading normals	4	16	4	16
system overhead	0.3	0.3	1.4	1.4
ur-patch transform	0.1	0.1	0.1	0.1
oracle	1.4	1.4	1.4	1.4
scan	0.1	0.1	0.1	0.1
sub/split	2.0	5.0	2.0	5.0
transfer overhead	1.9	1.9	1.9	1.9
draw call	0.5	3.7	1.6	4.4
total	6.4	12.6	8.6	14.3

Table 1: *Detailed timing analysis for rendering the 1600x1200 image of the killeroo model. Time in ms on a GTX 280 card.*

5 Implementation Details

We believe we found some very good optimization and programming strategies for SIMT architectures. This section gives a few very technical pointers on what we do to make our kernels fast.

Some calculations are naturally parallel, some are not. Forcing thread cooperation with shared memory and arcane address calculations turned out to be slower than just having one active thread and "wasting" the remaining threads in many cases. In order to maximize SIMT efficiency and memory bandwidth usage, we allocate a "thread pool"; i.e. we process patches in bundles and switch our level of parallelism inside kernels. We demonstrate this idea on our oracle kernel:

For each CUDA thread block we bundle several bicubic patches and allocate 16 threads per patch. First we load control points, perform perspective division, compute bounding boxes and decision bits. Those computations are trivially parallel using *one thread per control point* and all allocated threads are running in parallel.

Then we synchronize the threads of the block, and switch to *one thread per patch* for oracle logic and predicate computation, both having no "natural" parallelism at all. As current GPUs provide a *virtual* vector processor, only the few active threads are scheduled and this works surprisingly well.

Current hardware hints at bundle sizes of 32 to maximize the compute density. However, this uses large amounts of the limited shared memory and reduces the number of concurrently running blocks, preventing global memory latency hiding. We found bundle sizes of four performing best, being almost four times as fast as processing single patches: The decision logic (back-face culling) is comparably complex and the aggregated global memory writes waste less memory bandwidth. Using larger bundles reduced overall performance moderately.

6 Discussion

Our algorithm is simple and should scale well with future hardware and an increasing number of cores[2]. We expect it to be useful in a wide range of applications. Since no publicly available API currently exists, we cannot compare our performance directly to hardware tessellation. Due to the more direct (and arguably brute force) approach to adaptivity, combined with dedicated silicon, we do not expect to outperform a tessellator unit. That being said, we believe

[2]We have fixed overheads of about 2.5 ms per frame. This results in a sublinear speedup in total frame time from G80 to GT200 for small models.

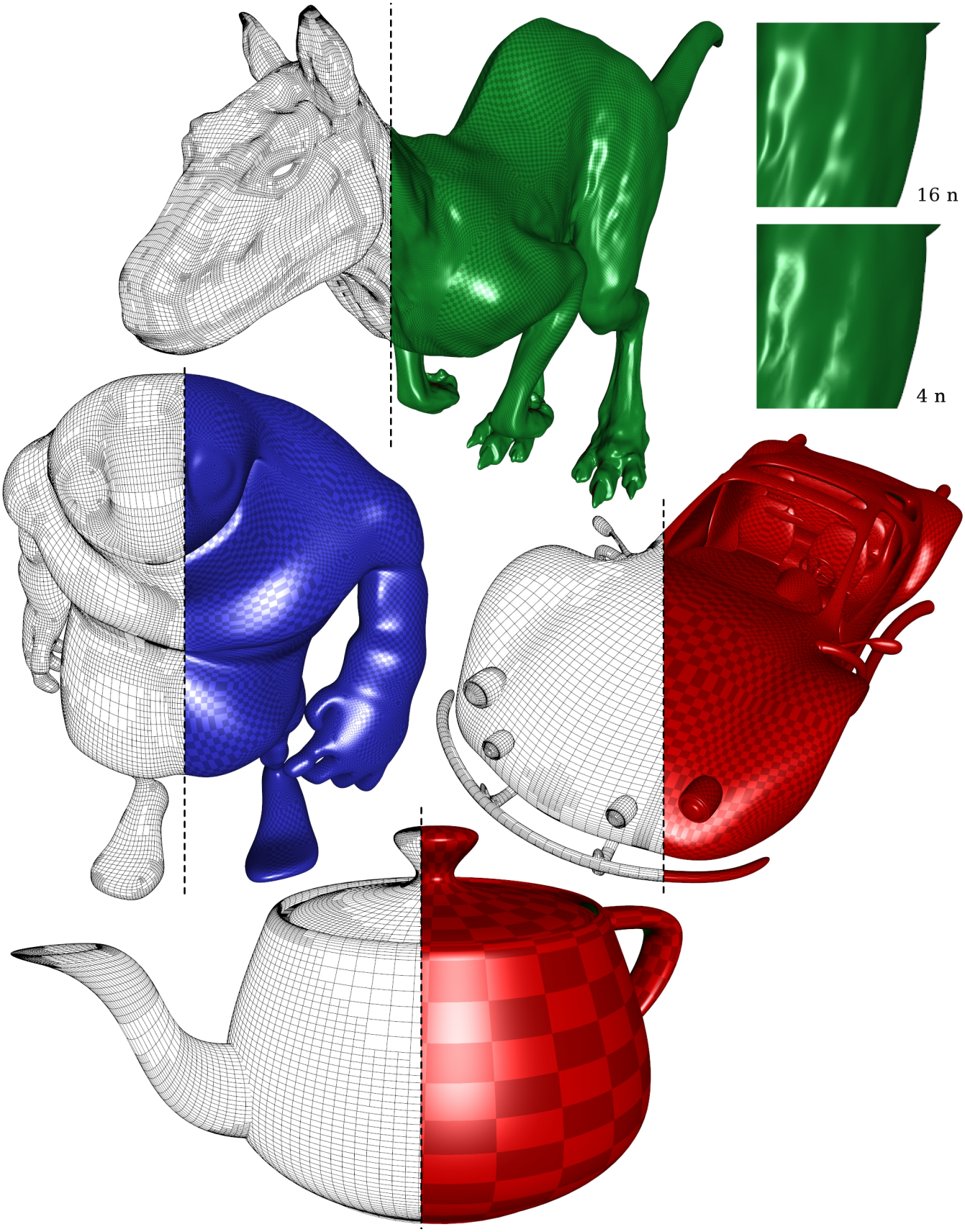

Figure 7: *Models used for measurements: We subdivide until we can guarantee a silhouette error of less then 0.5 pixels. For a 1600x1200 image we obtain the sub-patches shown as line drawing and render them as quadrilaterals with texturing, Phong shading, and 16x MSAA. The sub-patches are rendered evaluating 16 normals each. The magnified crops illustrate the reduced shading quality using only four normals.*

	ur-patches	512x512 [Patney and Owens 2008]	512x512 16 normals	512x512 4 normals	1600x1200 16 normals	1600x1200 4 normals
teapot	32	1 234 688 (4 823 x 256)	39 870	4 430	103 770	11 530
big guy	3 570	-	87 840	9 760	200 106	22 234
car	5 067	-	142 677	15 853	335 511	37 279
killeroo	11 532	3 693 056 (14 426 x 256)	173 646	19 294	408 465	45 385

Table 2: *Primitives sent to the rasterizer: Using a threshold of 0.5 pixels, our algorithm produces a comparably low number of primitives. For applications with a tight computational budget the number of primitives can be reduced further (4 normals). While this sacrifices shading quality, it maintains the 0.5 pixel guarantee for silhouettes. Renderings with 16 normals at 1600x1200 are shown in Figure 7.*

	[Patney and Owens 2008], G80 512x512	G80 512x512		GT200 512x512		GT200 1600x1200		GT200, 16x MSAA 1600x1200	
normals per sub-patch	256	16	4	16	4	16	4	16	4
teapot	82 (**12.41 fps**)	4.3	3.5	3.8	3.3	5.4	4.1	7.5 (**133 fps**)	5.7 (**175 fps**)
big guy	-	5.9	3.8	4.4	3.2	7.4	4.5	9.3 (**108 fps**)	6.5 (**154 fps**)
car	-	8.4	5.0	5.7	3.6	10.4	5.5	12.3 (**81 fps**)	7.5 (**133 fps**)
killeroo	241 (**4.15 fps**)	10.0	5.7	6.6	3.9	12.6	6.4	14.3 (**70 fps**)	8.6 (**116 fps**)

Table 3: *Total time per frame in ms (including subdivision, transfer overheads, rasterization, and shading). Note that Patney and Owens use vertex shading, while we use Phong shading with drastically fewer primitives. Except for the killeroo model, everything shades surprisingly well with only 4 normals per patch. We use a GTX 8800 Ultra (G80) and a GTX 280 (GT200) on Windows XP.*

that in the long run a relatively simple software solution on parallel hardware may prove superior to one implemented as a fixed function pipeline stage. In the following sections we will discuss our algorithm in more detail.

6.1 Memory and Bandwidth Requirements

Due to the low number of primitives generated, our algorithm uses little memory in addition to the vertex buffers we need for interaction between CUDA and the rasterizer. A 16 MB patch queue is sufficient for our largest example, the killeroo at 1600x1200. However, as the rasterizer is currently not accessible from CUDA, we need to allocate additional 80 MB for vertices, normals, and texture coordinates. We hope that this limitation will disappear in future versions of CUDA.

A related issue is the memory bandwidth used during subdivision. Compared to the "subdivide and compact" approach of Patney and Owens [2008], our "compute indices and subdivide" algorithm uses less than half the bandwidth, as it does not touch every patch again during compactification. Due to the bandwidth hungry nature of both algorithms, this directly translates to an almost twofold increase in performance.

Our algorithm naturally clusters subdivided patches by ur-patch. This allows us to use the available caching mechanisms very efficiently when we evaluate normals from the ur-patches, further reducing the pressure on global memory bandwidth.

6.2 Performance

Our algorithm is simple and the bicubic patches used fit current hardware nicely. We observe excellent performance, and even for large images and complex models we can spend additional resources on multi sampling.

Guaranteeing an error of less than 0.5 pixels in screenspace, we subdivide to a number of patches comparable to Patney and Owens [2008] (see Table 2, 4 normals for 512x512). However, as we do not aim to mimic the Reyes pipeline, we do not dice them into 256 pixel sized micropolgons. This means significantly fewer primitives need to be computed and transfered to the rasterizer, resulting in less vertex processing[3] and triangle setup. About half of the reported performance advantage can be attributed to the reduced primitive count.

During development we found that subdividing each patch into four sub-patches, instead of two, leads to a significant increase in performance for two reasons: First, there is no need to determine the subdivision direction in the oracle and no need to branch for directions in the split kernel. Second, we need fewer iterations until all sub-patches meet our precision requirements. This saves memory bandwidth and computations considerably.

The obvious drawback is that we generate more primitives for long thin patches than the asymmetric subdivision into two patches, e.g. at the teapot rim and the killeroo legs. In our experiments the total number of patches increased about 5%. However the cost for additional patches is dwarfed by the reduced complexity and the reduced number of iterations. Overall we observe performance up to two orders of magnitude faster than reported by Patney and Owens [2008].

While we archive comparable frame rates, it is difficult to compare our approach to the GPU based mesh refinement techniques of Boubekeur [2007]. As we aim to keep the final number of primitives small, we do not intend to reach the high polygon counts reported. Further, our approach is very simple and does not have any CPU involvement besides launching three kernels and one draw call. Our algorithm is completely screen space driven and operates at the final patch granularity: We do not need to store pre-tessellated meshes or compute a subdivision level per ur-patch from distance and curvature.

6.3 View-Dependent Metric

Our view-dependent error metric measures the distance in screen space between control points of the bicubic patch and the (degree elevated) bilinear patch sharing the same corner points, see Figure 4. We chose this metric for its simplicity, accuracy, and ideal fit to SIMT hardware. When the maximum difference is below a pixel unit threshold, we render the bilinear approximation. Due to the

[3] Note that our algorithm delivers the vertices already transformed to clip space, so the vertex shader has little work to do.

convex hull property of Bézier curves, this gives us a guaranteed error bound on silhouettes, densely approximates highly curved regions, and uses fewer polygons in more flat regions.

Our algorithm provides accurate silhouettes and high frame rates, but does not give guarantees for accurate shading. However, even for pathological failure cases, we get surprisingly reasonable shading by evaluating 16 normals and using Phong shading. All example figures are rendered with 16 normals per patch.

6.4 Crack Prevention

We avoid the creation of cracks between patches caused by non-uniform levels of subdivision, but we cannot remove cracks that are inherent to the model, e.g. when the given piecewise bicubic model consists of several patches that are not perfectly C^0 continuous (adjacent patches do not share Bézier control points). Small thresholds for our metric hide these artifacts to some extent, but do not solve the underlying issue. Examples for this can be found in the killeroo model as demonstrated in Figure 8.

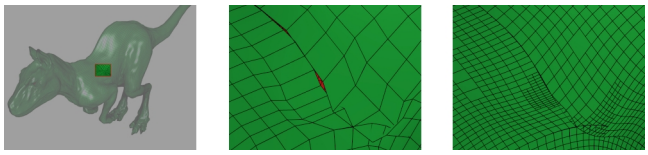

Figure 8: *We avoid cracks created by adaptive subdivision, but do not fix cracks that are inherent to the model. Small thresholds ease but do not fix the problem (right, threshold 0.5 pixels).*

A different but related artifact caused by T-junctions leads to occasional pixel dropouts between patch approximations. This artifact is the result of rasterization rules and finite precision arithmetic. Our experiments confirm the presence of this problem. However, when multi sample anti-aliasing is turned on, pixel dropouts are not detectable and image quality in general is greatly improved.

7 Future Work

While rational bicubic patches fit extremely well on current hardware, the adaptive subdivision approach presented here could be applied to arbitrary bidegree tensor product patches, as well as polynomial patches with triangular domains.

Animating a piecewise bicubic surface directly is complicated by the smoothness constraints among patch control points. By animating the vertices of a subdivision surface control mesh, this problem is avoided. However, applying our result to Catmull-Clark subdivision surfaces will require additional processing at the front-end. The challenge will be to convert a two-manifold control mesh, with extraordinary vertices, into a collection of bicubic patches at high frame-rate on the GPU.

Currently we create all sub-patches from scratch every frame. The high inter-frame coherency could be exploited to cache intermediate subdivision levels e.g. replace the subdivision tree by a forest, further increasing performance.

While our metric produces patches that shade very well with real world models, it only gives a guarantee for the silhouette error. In the future, we would like to integrate a guarantee for the shading error without sacrificing simplicity and efficiency.

The use of displacement mapping is very popular. It will be a very interesting challenge to find a simple and efficient metric that only generates fine grained geometry where variations in the displacement map require it.

Acknowledgments

We would like to express our thanks to David Luebke of NVIDIA for providing a GTX280 graphics card, and Avneesh Sud, Yuri Dotsenko and Shubhabrata Sengupta for sharing their understanding of parallel hardware. We also want to thank the anonymous reviewers for their work and detailed feedback, as well as Bay Raitt of Valve Software for the big guy and sports car models. The Killeroo model is courtesy of Headus (metamorphosis) Pty Ltd (available at `http://www.headus.com`).

References

BLELLOCH, G. E. 1990. *Vector Models for Data-Darallel Computing*. MIT Press, Cambridge, MA, USA.

BOUBEKEUR, T., AND SCHLICK, C. 2007. Generic adaptive mesh refinement. In *GPU Gems 3*. Addison-Wesley, ch. 5, 93–104.

CATMULL, E. 1974. *Subdivision Algorithms for the Display of Curved Surfaces*. PhD thesis, The University of Utah.

CLARK, J. H. 1979. A fast scan-line algorithm for rendering parametric surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 79)*, 7–11.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, 95–102.

GUTHE, M., BALÁZS, A., AND KLEIN, R. 2005. GPU-based trimming and tessellation of NURBS and T-spline surfaces. *ACM Transactions on Graphics 24*, 3 (Aug.), 1016–1023.

MICROSOFT CORPORATION. 2008. *Introduction to the Direct3D 11 graphics pipeline*.

NVIDIA CORPORATION. 2008. *NVIDIA CUDA: Compute unified device architecture*, June.

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

ROCKWOOD, A. P., HEATON, K., AND DAVIS, T. 1989. Real-time rendering of trimmed surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, 107–116.

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