Real-Time Patch-Based Sort-Middle Rendering on Massively Parallel Hardware

Charles Loop[1] and Christian Eisenacher[2]
[1]Microsoft Research
[2]University of Erlangen-Nuremberg

May 2009

Technical Report
MSR-TR-2009-83

Recently, sort-middle triangle rasterization, implemented as software on a manycore GPU with vector units (Larabee), has been proposed as an alternative to hardware rasterization. The main reasoning is, that only a fraction of the time per frame is spent sorting and rasterizing triangles. However is this still a valid argument in the context of geometry amplification when the number of primitives increases quickly? A REYES like approach, sorting parametric patches instead, could avoid many of the problems with tiny triangles. To demonstrate that software rasterization with geometry amplification can work in real-time, we implement a tile based sort-middle rasterizer in CUDA and analyze its behavior: First we adaptively subdivide rational bicubic Bézier patches. Then we sort those into buckets, and for each bucket we dice, grid-shade, and rasterize the micropolygons into the corresponding tile using on-chip caches. Despite being limited by the amount of available shared memory, the number of registers and the lack of an L3 cache, we manage to rasterize 1600×1200 images, containing 200k sub-patches, at 10-12 fps on an nVidia GTX 280. This is about 3x to 5x slower than a hybrid approach subdividing with CUDA and using the HW rasterizer. We hide cracks caused by adaptive subdivision using a flatness metric in combination with rasterizing Bézier convex hulls. Using a k-buffer with a fuzzy z-test we can render transparent objects despite the overlap our approach creates. Further, we introduce the notion of true back patch culling, allowing us to simplify crack hiding and sampling.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

[1]e-mail: Charles.Loop@microsoft.com
[2]e-mail: Christian.Eisenacher@cs.fau.de

1 Introduction

Modern GPUs have evolved into general purpose floating point processing devices containing many parallel SIMD cores. Intel's upcoming Larabee chip [13] even replaces specialized rasterization hardware with a tile based sort-middle triangle rasterizer, implemented in software on such a manycore processor.

Given the abundant computational power, higher order surfaces have entered the field of real-time rendering. They are tessellated directly on the GPU and the resulting triangles are processed by the traditional triangle pipeline. However, this creates many tiny triangles that are not trivial for customized hardware and even more difficult to manage for a software based triangle rasterizer: Sorting triangles into tiles becomes as expensive as sorting fragments, and shading individual triangles with an area of around one pixel will leave most components of the SIMD vector units idle.

A straightforward idea is to use something similar to the REYES algorithm, i.e. move the sorting further to the front of the pipeline and sort parametric (sub) patches instead of triangles. Each tile would then tessellate and rasterize the patches associated with it; since no additional synchronization is needed and the temporary triangles only reside in on-chip caches, this should scale well with the increasing number of available cores.

To demonstrate the viability of this approach, we present a working CUDA [8] implementation of such a REYES-like pipeline: We divide the screen into a set of tiles with associated buckets. Then we adaptively subdivide rational bicubic Bézier patches breadth-first and sort the sub-patches into the buckets. When rendering the individual tiles, we tessellate each of its sub-patches into a grid of micropolygons, shade the grid and perform stochastic multisampling of the micropolygons. In order to deal with the limited memory resources available on GPUs and the memory demands of a data-parallel breadth-first algorithm, we

- subdivide in the parameter domain, and
- cull back facing *patches*. Further we present a
- local way to hide cracks caused by adaptive subdivision that does not require connectivity information or linearized edges.

We are mainly limited by the size of on-chip memory, but despite storing the visible point samples in slower off-chip (global) memory, we can rasterize 1600×1200 images with 200k sub-patches, at 10-12 fps on an nVidia GTX 280.

2 Previous Work

Many parallel renderers with different advantages and drawbacks exist. Molnar et al. [6] try to classify them into three groups, depending on where in the pipeline they sort primitives to the screen. To illustrate: Parametric patches get tessellated into triangles, which in turn get sampled. A *sort-first* renderer distributes the patches among screen space tiles, a *sort-middle* renderer does

the same with the triangles after tessellation, and a *sort-last* renderer sorts the samples into pixels. As we perform most of the tessellation before sorting, we classify our approach as sort-middle.

The concept of a sort-middle rasterizer gained significant attention recently with announcement of Larrabee [13]. Larrabee aims to be a GPU with a fully programmable rendering pipeline, i.e. it does not contain custom hardware for triangle setup, rasterization or fragment merging. The main argument is, while it probably cannot outperform custom silicon, a sort-middle triangle rasterizer in software is "fast enough" for existing workloads, undoubtedly more flexible and should scale well with an increasing number of general purpose cores.

The REYES image rendering architecture was developed as an off-line renderer at Pixar in the mid 1980's [2] and is an industry standard for rendering high quality complex animated scenes. REYES adaptively subdivides higher order surface patches in a process known as *bound and split* until their screen space size is small. The resulting sub-patches are *diced* (tessellated) into regular *grids* of *micropolygons*; i.e. polygons with an area on the order of one pixel [1]. The grid is shaded and pixel coverage is determined by stochastic super-sampling of the micropolygons.

Adaptive subdivision might result in visible holes or *cracks*, when adjacent sub-patches are subdivided to different levels. For off-line rendering, or on a single core machine, neighboring patches can communicate to stitch cracks [1]. However, on a machine with multiple cores and for real-time applications, a local method to address cracks is required to avoid costly synchronization.

Several attempts to implement a REYES pipeline on parallel hardware have been made: Owens et al. [9] implement REYES with Catmull-Clark subdivision surfaces on the Stanford Imagine stream processor. They subdivide depth first until all edge lengths are below a threshold. By freezing edge equations individually they avoid subdivision cracks. After subdivision they generate fragments that are composited into the final image. A comparable OpenGL implementation is one order of magnitude faster and they note that their implementation of REYES spends over 80% of the total time subdividing patches.

In 2008 Patney and Owens demonstrate a hybrid approach [11]: They subdivide Bézier patches breadth first and dice them using CUDA. Then they transfer the resulting micropolygons to the hardware rasterizer for sampling, thereby requiring considerable amounts of memory for the patch queue and transfer buffers. They are able to render a 512x512 image of the killeroo model with 14 k sub-patches at 30 fps [10].

Eisenacher et al. [4] refine the implementation of Patney and Owens, but try to generate as few sub-patches and micropolygons as possible to minimize transfer overheads. They do this by using a flatness metric as a stopping criterion and handle subdivision cracks by linearizing critical edges. They render a 1600x1200 image of the killeroo with 45k sub-patches at 150 fps, but with 2 orders of magnitude fewer polygons that are Phong shaded.

3 System Overview

In this work, we borrow many algorithmic features from REYES, but differ in a number of details in order to utilize the available parallelism and to handle the memory constraints found on current GPUs. Most prominently we use considerably smaller screen tiles (e.g. 8×8 pixels) and dice into 4×4 grids only. Conceptually we separate our pipeline into three successive steps:

Bound and Split: Following the approach of Eisenacher et al. [4] we first adaptively subdivide the input primitives into sub-patches until they are single sided, flat, and small enough. The first two criteria deviate from the original REYES algorithm and are required for our local crack hiding. In contrast to existing adaptive subdivision algorithms on the GPU, we subdivide in the parameter domain using quad tree coordinates. This saves memory and memory bandwidth considerably, but prevents previous linearization based methods to avoid subdivision cracks.

Middle Sort: Then we sort the quad tree coordinates of the sub-patches into buckets. A single bucket contains all patches that cover its associated screen space tile. This can be handled surprisingly well using CUDA's `atomicAdd()` and a parallel prefix scan as described in Section 5.

Tile Based Rasterizer: Finally we render the individual tiles. We dice the patches of each bucket into a small grid of micropolygons, shade the grid, hide cracks caused by the adaptive subdivision and sample the micropolygons. Details are described in Section 6.

4 Bound and Split

4.1 Overview

The initial patches describing the model, the *urPatches*, are subdivided in a breadth first stream computing approach [4, 11]. As visualized in Figure 1, we start by transforming the urPatches into clip space using the composite MVP matrix and place them into a double buffered patch queue. An oracle kernel examines all patches in parallel and decides what to do: We either *cull*, *keep* or

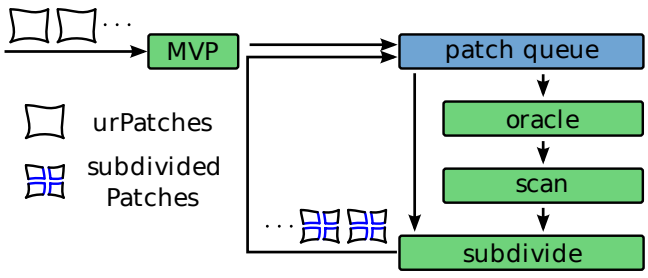

Figure 1: Bound and split: Patches are examined and subdivided breadth first until they meet all our criteria.

subdivide patches in a 1:4 split. This requires fewer iterations of bound and split, and simplifies the oracle logic [4]. The oracle outputs its decision in the form of storage requirements: 0 slots for cull, 1 slot for keep, 4 slots for subdivide. A parallel scan [5] is then used to compute the prefix sums of this storage decision array. This directly generates the storage locations where the subdivision kernel will store the kept patches and the four children of the subdivided patches. This is iterated until all patches in the queue fulfill our criteria.

4.2 Stopping Criteria

While the original REYES algorithm subdivides until the screen space bounding box of the projected patch is smaller than a given threshold, e.g. 8×8 pixels, we place additional constraints on the sub-patches. They must be:

- *Small Enough*: This is the traditional REYES criterion but we require that this threshold is smaller than the tile size. This is not a strong limitation, but guarantees that a sub-patch covers at most four adjacent tiles, and simplifies bucketing considerably.
- *Flat Enough*: The projected screen space control points deviate from the bilinear interpolated patch corners by no more than a predetermined tolerance [4]. This guarantees smooth silhouettes and bounds the overlap we introduce with our crack hiding algorithm (Section 6.3).
- *Single Sided*: The patch does not contain a silhouette edge. This simplifies crack hiding and sampling (Section 6.4), and allows for true back*patch* culling (Section 4.4). Note that subdividing a patch with a silhouette usually generates at least one sub-patch still containing a silhouette. We waive this criterion if the patch is smaller than the threshold of the flatness metric, i.e. the error we are willing to accept.

4.3 Adaptive Subdivision in the Parameter Domain

Previous implementations of adaptive subdivision on the GPU [4, 11] explicitly store the control points of the sub-patches in the patch queue. This requires considerable storage, e.g. 256 bytes per rational bicubic Bézier patch, and hence limits the amount of patches that can be managed. Also, loading control points into the kernels consumes precious memory bandwidth, potentially harming performance.

We observe that computational power grows faster on current GPUs than memory and memory bandwidth. Furthermore, a 1:4 split essentially creates a quadtree hierarchy in the parameter domain of each urPatch and our organization of the bound and split loop naturally clusters sub-patches by urPatch, hinting at the possibility to share memory transfers for nearby sub-patches.

Hence, rather than explicitly computing the control points of subdivided patches, we initially store the transformed urPatches in a separate buffer and instead subdivide in the parameter domain using *quadtree coordinates*. In other

words, the actual control points of subdivided patches are computed on the fly whenever they are needed, never stored. We represent this with the following data structure:

```
struct QuadTreeCoord {
  unsigned int urPatchIDX;
  float u;
  float v;
  float length;
};
```

`urPatchIDX` is the index of the urPatch to which the sub-patch belongs; `u` and `v` are the 2D coordinates of the lower left corner of the sub domain; and `length` represents the side length of the sub domain.

We *reconstitute* a sub-patch using four trim operations. Each of them evaluates sixteen 1D DeCasteljau steps in parallel (4 components $\times$ 4 rows/columns) to trim a boundary of the urPatch until the final sub-patch is obtained. Note that all intermediate values are stored in registers, the transformed urPatch is accessed using the texture cache and reconstitution runs at full SIMD efficiency reconstituting two patches per CUDA warp. Figure 2 shows the first two trim operations.

4.4 Back Patch Culling

The idea of *back patch culling* is a higher order analog of back face culling for triangles in the traditional rasterization pipeline to avoid unnecessary computations and save memory resources. Obviously, this savings will be offset by the expense of the test itself: Unlike the popular heuristic of analyzing only the normals at patch corners, *all* surface normals must point away from the viewer to guarantee no patch will be culled from the queue in error.

Correct patch culling is not the only benefit derived from such a procedure; unlike the 2-state outcome of the triangle back face test (*front facing*, or *back facing*), a real back patch test has a third outcome: If some normals point to the viewer and some point away, since polynomials are continuous, the patch projection must fold over on itself, meaning it contains a *silhouette*. Otherwise no two distinct uv domain points can project to the same screen space point. This means they are *bijective*, and the derived micropolygon grids must be well behaved; that is, they cannot contain overlapping or concave quads. This observation has important implications for our coverage test and crack hiding scheme.

In clip space, the correct back patch test becomes sign test on the coefficients of the z component of the parametric tangent plane; that is, the tangent plane at each point of the patch $P(u, v)$. We can compute it as the 4D cross product of $P(u, v)$, $\frac{\partial}{\partial u}P(u, v)$, and $\frac{\partial}{\partial v}P(u, v)$. While this sum of products of polynomials appears to result in a bidegree 8 polynomial, the actual result is only bidegree 7. We omit a formal proof, but note that this result is easily verified using symbolic algebra software.

```
  float b0, b1, b2, b3;

  { // subdivide left; threadIdx.x/y are [0..3] for a total of 16 threads
    int rowStart = urPatchIDX*64 + threadIdx.y * 16 + threadIdx.x;

    b0 = tex1Dfetch(urPatchTexRef, rowStart + 0*4 ); //first cp.[0..3]
    b1 = tex1Dfetch(urPatchTexRef, rowStart + 1*4 ); //second cp.[0..3]
    b2 = tex1Dfetch(urPatchTexRef, rowStart + 2*4 ); //third cp.[0..3]
    b3 = tex1Dfetch(urPatchTexRef, rowStart + 3*4 ); //fourth cp.[0..3]

    float ul = u; float iul = 1.0f - ul;

    float b01   = iul * b0  + ul * b1;
    float b12   = iul * b1  + ul * b2;
    float b23   = iul * b2  + ul * b3;

    float b0112 = iul * b01   + ul * b12;
    float b1223 = iul * b12   + ul * b23;

    float bm  = iul * b0112 + ul * b1223;

    b0 = bm;
    b1 = b1223;
    b2 = b23;
//  b3 = b3;
  }
  { // subdivide right
    float ur = level/(1.0f - u); float iur = 1.0f - ur;

    float b01   = iur * b0  + ur * b1;
    float b12   = iur * b1  + ur * b2;
    float b23   = iur * b2  + ur * b3;

    float b0112 = iur * b01   + ur * b12;
    float b1223 = iur * b12   + ur * b23;

    float bm  = iur * b0112 + ur * b1223;

    // *17 to avoid bank conflicts for the top-bottom subdivision
    int rowStart = patchAddrSM + threadIdx.y * 17 + threadIdx.x;

    cpsSub[rowStart +  0*4 ] = b0;
    cpsSub[rowStart +  1*4 ] = b01;
    cpsSub[rowStart +  2*4 ] = b0112;
    cpsSub[rowStart +  3*4 ] = bm;
  }
}
```

Figure 2: Reconstituting a rational bicubic Bézier patch using 4 trim operations. Only the code for the first two trimmings is shown requiring less than 40 operations including address calculation.

To avoid costly computations in the oracle, we compute only the z component of the parametric tangent plane in Bézier form, resulting in a scalar valued 8×8 control net, when loading the model. In the oracle kernel, when a bicubic patch is reconstituted from its quadtree coordinate, we also reconstitute the biseptic tangent plane z component using a degree 7 analog of the reconstitution algorithm. We count positive and negative entries within the resulting 8×8 array using a parallel reduction in shared memory. A positive count of 64 means *back facing*, a negative count of 64 means *front facing*, and a non-zero positive and negative count means *silhouette*.

5 Parallel Middle Sort

Once all patches in the patch queue satisfy our stopping criteria, we sort them into buckets, one bucket per tile. Our goal is to store a variably sized list of covering patches for each bucket.

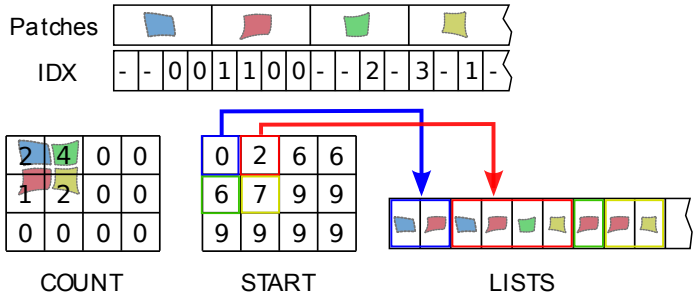

Figure 3: To bucket the subdivided patches we use CUDA's `atomicAdd()`. This counts the patches per bucket and delivers the indices of the patch inside the buckets at the same time. Using those and the prefix sum of the counters we compute the final positions of the patches in the bucket lists.

To facilitate this in parallel and with variable list lengths, we use CUDA's `atomicAdd()` and organize the lists as shown in Figure 3: $COUNT$ stores the length of the list at each bucket, $START$ contains the start indices of the lists and $LISTS$ contains the actual lists of patches. We create these lists with the following algorithm:

Init: We allocate a temporary buffer IDX with space for four integers per patch in the patch queue. Note that each patch covers at most four adjacent tiles at this point (see Section 4). Further we initialize the entries of $COUNT$ to zero.

Calculate Addresses: For each patch we perform an

$$\texttt{atomicAdd}(\&COUNT[tileID_{0..3}],\ 1)$$

on the tiles it overlaps. That way we count the number of patches that will be stored in each bucket correctly despite the possibility of multiple threads

accessing the same bucket simultaneously. The values returned by the call are the indices in the patch list of the buckets and we store them in IDX. Starting from the lower left corner we subsequently store the indices for the top-left, top-right, bottom-left, bottom-right bucket or -1 for not covering a bucket.

Sort Patches into Buckets: After the addresses are calculated, we compute $START$ as the prefix sum of $COUNT$ using a parallel scan [5]. Combining $START$ and IDX we can now sort the patches into the buckets: For each patch in the patch queue we simply store its index at $LISTS[START[tileID_{0..3}] + IDX_{0..3}]$.

6 Tile Based Rasterizer

Using $COUNT$ and $START$ we know for each tile, how many and which patches we need to rasterize from $LISTS$. Using *thread pooling* [4] we switch between per patch parallelism during the grid setup and shading, and per pixel parallelism during sampling inside the kernel. For ease of implementation we launch one CUDA tread block per tile, allocate one lightweight thread per pixel and let the CUDA scheduler handle the load balancing.

6.1 Dicing

For each sub-patch, we evaluate a 4×4 grid of surface points. This is relatively small compared to the grid sizes used in traditional REYES, but demanded by the limited amount of shared memory available. However, as we will show in Section 7.2, sorting is more efficient with smaller tiles. Using smaller tiles limits the maximum screen space extent of the sub-patches (see Section 4.2), so a 4×4 grid is sufficient in many cases. This also puts grid points in a one-to-one correspondence with control points, enabling us to compute grids efficiently, and with few registers, using a modified form of our reconstitution algorithm.

This modification is based on the relationship between control points and samples in the curve case, see Figure 4. By evaluating the cubic Bernstein basis functions at parameter values $0, \frac{1}{3}, \frac{2}{3}$, and 1, we see that samples on the curve are related to the control points by

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \frac{1}{27} \begin{bmatrix} 27 & 0 & 0 & 0 \\ 8 & 12 & 6 & 1 \\ 1 & 6 & 12 & 8 \\ 0 & 0 & 0 & 27 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}. \tag{1}$$

At the end of each 1D DeCastlejau step, we replace the Bézier points with curve points using Equation (1). Due the separability of tensor product surfaces, this will result in a 4×4 grid of surface samples.

6.2 Gridshading

We flat shade the micropolygons using an estimated surface normal for each quad. Conceptually, we find the plane that contains the centroid of the quad

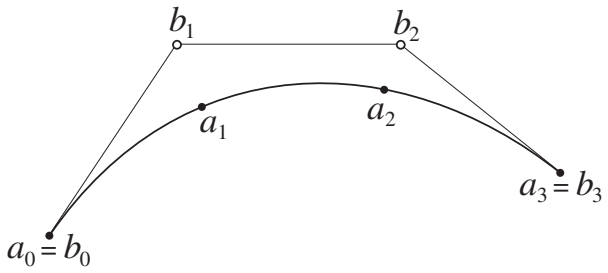

Figure 4: A Cubic Bézier curve with control points b_i and corresponding points on the curve a_i is illustrated.

and the difference of the diagonal vertices using the $4D$ cross product. In practice however, this requires more than are necessary to get a convinving result. Instead, we perform the homogeneous division, take the $3D$ cross product of the diagonal vectors, solve for the plane containing the centroid, and back project using the sparse perspective matrix. If the quad contains a degenerate edge, this will yeild the tangent plane of the corresponding triangle and produce a valid normal.

We found flat shading to be more stable and requiring considerably fewer registers than shading at vertex positions and using Gouraud or Phong interpolation, especially when degenerated patches are encountered. This is not an integral part of the overall algorithm and interpolation could be used as well. However Phong shading would require some sort of scoreboarding to archive reasonable SIMD efficiency with tiny triangles. After shading, we adjust grid points to hide cracks.

6.3 Crack Hiding

Our strategy to hide cracks caused by adaptive subdivision (Figure 5, left) is to extend the grids to the Bézier convex hull of the patch (Figure 5, right). This guarantees that all sample points covered by the parametric surface will be set.

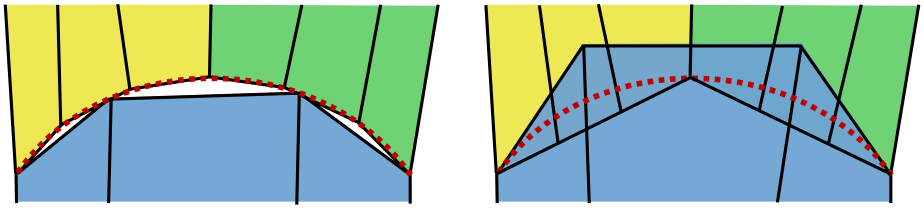

Figure 5: Subdividing to different levels results in holes along edges (left). We cover these by adjusting the patch boundaries to coincide with its Bézier convex hull (right) to hide them.

Of course this will create overlap and samples not covered by the parametric surface might be set at silhouettes. However the bias introduced is bounded by the flatness metric so the convex cull can not deviate from the real surface much. A one pixel threshold works very well in practice. Also note that this manipulation of the grid vertices does not introduce shading errors, as we have shaded the grid before adjusting vertices.

Recall that the sub-patches that are bucketed are either smaller than the flatness threshold, generally less than one pixel, or do not contain silhouettes; the latter means they are bijective and have well behaved grids. Hence the Bézier convex hull is completely determined by the four boundary curves. For a 4×4 grid it is even enough to adjust the two interior points of the boundary curves to make the footprint convex.

Figure 6 illustrates the three possible configurations for such an interior point a_1. We first determine where its control point b_1 is located relative to the

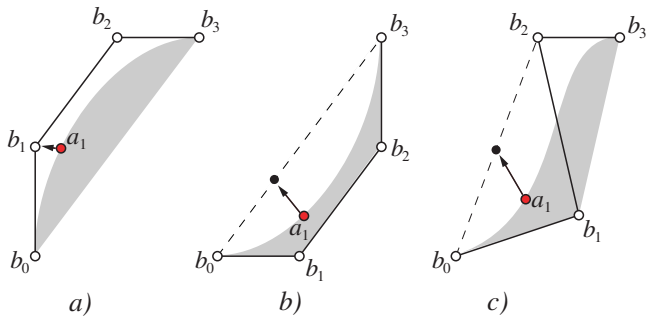

Figure 6: To cover cracks we adjust the boundary curve interior points according to 1 of 3 possible configurations. The gray region indicates the side of the curve with the patch.

boundary edge defined by b_0 and b_3 by looking at the signed area of the triangle b_0, b_3, b_1. If its determinant is negative then b_1 is outside (case a)) and we move the interior grid point to b_1.

Otherwise we need to interpolate the new position depending on the position of b_2. If b_2 is inside the boundary edge (determined by the signed area of the triangle b_0, b_3, b_2, case b)) we interpolate a_1 as $\frac{2}{3}b_0 + \frac{1}{3}b_3$. Otherwise a_1 is interpolated as $\frac{1}{2}b_0 + \frac{1}{2}b_2$. Note that we can recover the Bézier control points of the boundary curves from the previously computed grid by inverting Equation (1) to conserve shared memory and registers.

Due to the overlapping of convex hulls, pixel samples in the overlap region will count two hits corresponding to the same surface layer. For opaque surfaces without layering, the z-buffer test resolves this issue without noticeable artifacts. In case we need multiple layers we must merge the hits to get correct layering.

We do this with a *fuzzy z-test* [3]: If the difference between the new z value and the existing z-buffer value is less than a tolerance and the sub-patches have the same `urPatchIDX`, they are considered to belong to the same layer. Examples using Order Independent Transparency and Constructive Solid Geometry are shown in Figure 7.

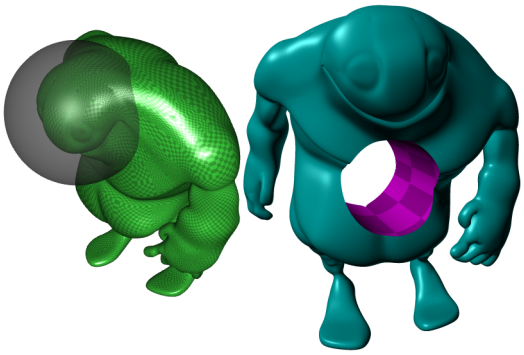

Figure 7: For algorithms requiring sorted layers like Order Independent Transparency and Constructive Solid Geometry, a *fuzzy z-test* prevents artifacts from double hits caused by our crack hiding strategy.

When the projection of a patch edge is (to high precision) linear, adjacent patches with different subdivision levels will form the equivalent of T-junctions that lead to pixel dropouts in traditional rasterization. We occasionally see similar artifacts with our rasterizer if we test against edge equations without tolerance (see Section 6.4). However, when testing against the machine epsilon for floats we do not observe them in our experiments.

6.4 Sampling

Once the grid boundaries have been adjusted to hide cracks, we switch to a per pixel parallelism for coverage testing. One thread is responsible for all subsamples of its pixel.

We use a precomputed 2D Halton sequence [7] for the subsample locations and test whether they are inside the four half-spaces defined by the four polygon edges [12]. We compute four dot products and check the signs. If all four values are positive the pixel sample is covered. We do not need to consider non convex quads, since our back patch testing guarantees that grids cannot contain them.

For covered samples we interpolate the z value from the the quad vertices, perform a z-test and store the quad color if necessary. For effects that require sorted layers, we use a bubble sort with fuzzy z-test to insert the sample. Due to similar memory access patterns for neighboring threads it performs rather well for the number of layers we use. Finally we apply whatever logic is needed on the layers to obtain a desired effect, average the subsamples and write the final pixel color to the frame buffer.

Sampling suffers the most from the limited amount of shared memory: Each thread has to compute edge equations separately from the grid vertices. To avoid unnecessary coverage tests we use a few simple bounding box tests. While the edge equations are reused for the subsamples, the registers holding the edge equations cannot be used for the z or color buffer entries further harming performance.

7 Results and Discussion

To test our REYES-like renderer on the GPU we render the familiar Utah *teapot*, the *bigguy* and the *killeroo* model with the view points shown in Figure 8. All results use a 1 pixel threshold for the flatness metric and are measured on an nVidia GTX 280 using CUDA 2.1 on Windows XP.

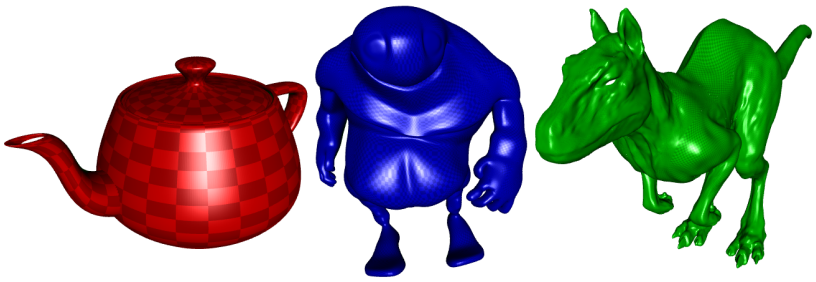

Figure 8: The models used for analysis: The Utah teapot, the bigguy and the killeroo; 32, 3570 and 11532 urPatches.

7.1 Bound and Split

Subdivision in the Parameter Domain: We subdivide the killeroo model to 51k patches. Storing control points in the patch queue requires two buffers of size 13 MB each and we measure 3.6 ms for the oracle and 1.0 ms for the split. Subdividing in the parameter domain requires two buffers with 0.8 MB for the patch queue and 3 MB for the transformed urPatches. The oracle takes 4.1 ms and subdivision 0.3 ms.

Reconstituting the sub-patch makes the oracle computationally more expensive, but subdivision becomes trivial and needs little bandwidth. Overall, subdividing in the parameter domain saves 0.2 ms in this case and requires 4.6 MB for the patch queue instead of 26 MB. With computational density increasing faster than memory bandwidth we expect the trade off compute for memory bandwidth to be even more beneficial with future hardware.

Stopping Criteria: We perform more complicated bounding tests than previous approaches. However we use very little bandwidth and avoid repetitive tests: For finished patches we set a flag and copy the previous decision instead of testing again.

Overall, we can bound and split 51 k / 4.5 = 11.6 M patches per second for the killeroo model. This is highly competitive with previous implementations: Patney and Owens [11] use a bounding box test and report 14 k / 6.99 = 2.0 M sub-patches per second on a GTX 8800. Eisenacher et al. [4] use a flatness test and report 45 k / 3.4 = 13.2 M patches per second on a GTX 280.

7.2 Parallel Middle Sort

Sorting performance is directly related to the size of the screen space tiles: Smaller tiles mean more buckets and lower probability of conflicting parallel writes to the same bucket that need to be synchronized. On the other hand, smaller tiles mean more patches cover multiple tiles and we need to reconstitute, shade and rasterize multiple copies.

Table 1 shows both effects for a 1600x1200 rendering of the killeroo. It it subdivided until all patches have a screen space bounding box of less than 4x4 pixels. Increasing the tile size reduces the overlap from 2.45 for 4x4 tiles to 1.14 for 32x32 tiles. However, despite having to sort over twice as many patches, sorting into 4x4 tiles is faster overall, as it deals with considerably less conflicting writes on average.

tile size	4x4	8x8	16x16	32x32
sorted patches	662 k	445 k	352 k	309 k
overlap factor	2.45x	1.65x	1.30x	1.14x
address calc. [ms]	8.28	8.57	10.45	15.38
bucket write [ms]	1.45	0.91	0.73	0.69

Table 1: Rendering the killeroo using different tile sizes. The model is subdivided to 270k sub-patches, but due to overlap multiple copies are stored. Larger buckets require more costly synchronization. Time in ms on a GTX 280 card.

7.3 Tile Based Rasterization

Crack Hiding: Cracks caused by adaptive subdivision are very disturbing, especially for moving objects. The left side of Figure 9 shows a cut-out featuring three visible subdivision cracks. The right side shows that rendering the Bézier convex hull removes the holes and causes no visible artifacts at the silhouette.

Tile Size: Similar to parallel sorting, the performance of tile based rasterization is dependent on the tile size. We render a 1600×1200 image of the killeroo model with 4x4, 8x8 and 16x16 tiles at different scales. Figure 10 visualizes the ms per frame including overlap. Smaller tiles perform less unnecessary coverage tests and more tiles can be scheduled on a multiprocessor concurrently to hide latency. However, very small tiles require more concurrent thread blocks than the scheduler can handle, leaving resources unused. 8x8 tiles seem to hit a sweet spot, outperforming 4x4 tiles by a factor of more than two and being faster than 16x16 tiles despite more overlap.

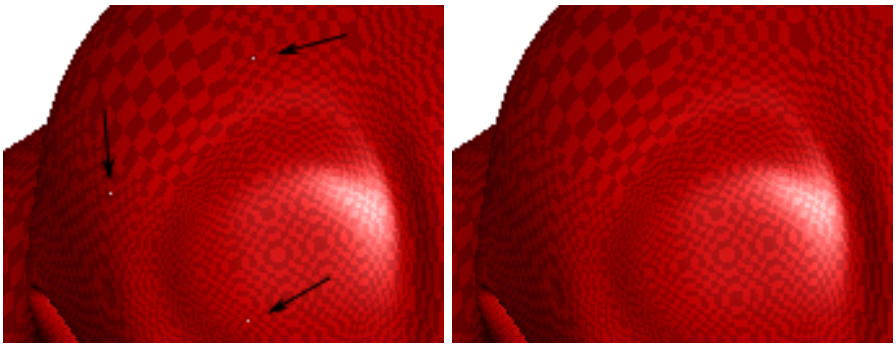

Figure 9: Crop of a rendering of the bigguy model with subdivision cracks (left). Our algorithm removes them (right).

Overall performance: Table 2 shows timings for a few select screen sizes. We use 8x8 pixel sized screen tiles and subdivide until the screen space bounding box (BB) is smaller than 8x8 or 4x4 pixels. We list the number of patches rasterized and the complete time needed to render and display a frame with one or four samples per pixel.

screen	BB		teapot	big guy	killeroo
512 x 512	8x8	#	25 k	32 k	38 k
		1 spp	11.0 ms	12.4 ms	16.0 ms
		4 spp	18.1 ms	20.8 ms	27.0 ms
	4x4	#	61 k	76 k	84 k
		1 spp	27.1 ms	29.4 ms	31.6 ms
		4 spp	44.8 ms	48.7 ms	51.9 ms
1600 x 1200	8x8	#	138 k	165 k	184 k
		1 spp	52.5 ms (19 fps)	71.2 ms (14 fps)	83.8 ms (12 fps)
		4 spp	95.2 ms	130 ms	149 ms
	4x4	#	348 k	404 k	445 k
		1 spp	140 ms	184 ms	213 ms
		4 spp	253 ms	337 ms	381 ms

Table 2: Total time per frame using 8x8 screen tiles and 1 or 4 samples per pixel (spp). Patches are subdivided until their bounding box (BB) is less than 8x8 or 4x4 pixel. We use CUDA 2.1 and an nVidia GTX 280 on Windows XP.

To show where time is spent, we present a breakdown for the most complex settings in Table 3. While overlap varied considerably with differently tile sizes (see Table 1), the overlap factor seems to be relatively constant with different models. The dominating cost is the actual rasterization which scales fairly linearly with the number of patches being rasterized.

The teapot is a noteworthy exception due to its simple geometry: First, fewer urPatches mean the probability for a cache hit during reconstitution is

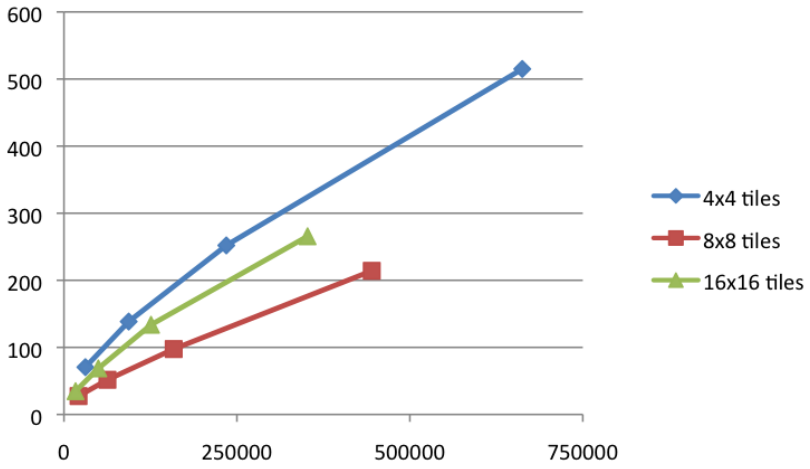

Figure 10: Milliseconds per frame for rendering a 1600×1200 image of the killeroo at different zoom levels. Using 8x8 tiles performs best.

higher, for the teapot it is even 100%. Hence we manage to bound and split almost 17 M patches per second for the teapot, compared to 14 M for the bigguy and 13 M for the killeroo. Similar effects occur during setup and shade. Second, only few sub-patches are long and thin (in contrast to e.g. the killeroo legs) so our bounding box tests work well and we avoid unnecessary coverage tests.

7.4 Summary

The breadth-first approach for subdivision maps very well onto the data-parallel paradigm behind CUDA and only around 10% of the total time per frame are spent subdividing patches. Also subdividing in the parameter domain and reconstituting sub-patches from quad tree coordinates works very well for the existing imbalance between memory bandwidth and computational power on current GPUs: Reconstitution is already faster than loading patches. We expect this to become even more important, as we expect the imbalance between bandwidth and compute to widen in future hardware generations.

As we sort sub-patches, sorting is not the limiting factor in our algorithm. Only about 5% of the total time per frame is spent sorting patches. It scales linearily with the number of primitives that get sorted. Sorting one patch is equivalent to sorting to 9 mircoquads. Even when we assume that microquads cause zero overlap, sorting patches would be 5x faster than sorting microquads and 10x faster than sorting triangles, which would make sorting a significant part of the total time per frame.

Subdividing in the parameter domain prevents the use of linearization methods to hide subdivision cracks. Our overlap approach can still be used and works surprising well for solid objects. The fuzzy z-test allows it to be used with trans-

	teapot	bigguy	killeroo
subdivided patches	218k	240k	270k
rasterized patches	348k	404k	445k
overlap	1.60x	1.68x	1.65x
system overhead		2.5	
bound	12.8	16.3	19.4
split	1.1	1.2	1.5
address calc.	8.9	7.5	8.6
bucket write	0.7	0.8	0.9
setup and shade	92.9	121.0	143.0
crackfix	9.0	12.0	13.7
sample	4*37.8	4*50.8	4*56.3
total	253.0	336.4	381.4

Table 3: Time in ms for the individual steps, rendering a 1600x1200 image with 8x8 screen space tiles. Sub-patches have a bounding box of less than 4x4 pixels.

parent objects and CSG. However around 7% per frame are spend calculating the convex hull.

Most time per frame is spent shading and rasterizing. A major part of this is caused by the tiny on-chip caches: We have to compute edge equations per pixel and store both the z-buffer and the visible point samples in the comparably slow global memory. Further we get poor SIMD efficiency when we sample the small micropolygons sequentially. As, due to our back patch test, micropolygons of the same sub-patch cannot overlap, i.e. each pixel can only be hit once by a given sub-patch, there might great potential to improve efficiency.

8 Known Limitations

There are three main limitations of our approach. First, while our approach hides cracks caused by adaptive subdivision, it cannot fix cracks that are already in the model; so we require the initial patch model to be watertight.

Second, reconstituting a small sub-patch directly is numerically more challenging than using a traditional DeCastlejau pyramid, where errors tend to even out. With a subdivision depth of around ten to twelve we should reach the limit for single precision arithmetic. However, even with the teapot adaptively subdivided to 1 M sub-patches we do not observe related artifacts.

Third, similar to existing local crack avoidance techniques like edge linearization [4,9], hiding cracks using overlap does not work with displacement mapping as displaced T-vertices will open holes along subdivision boundaries.

9 Conclusions and Future Work

We have presented a software bucketing rasterizer written in CUDA, that implements an imaging pipeline for rational bicubic surface primitives. The performance of our system depends on many factors and is comparable to existing hybrid approaches. We can write shaders in simple C without special shader languages and do not need large transfer buffers between CUDA and the hardware pipeline. This and the fact that we subdivide in the parameter domain allows us to manage considerably more sub-patches then previous hybrid approaches.

An interesting open question is which shading model is best suited for massively parallel SIMD hardware. Grid shading uses SIMD vectors very efficiently but might shade surfaces that are not visible in the final image. Overall it might be beneficial to first rasterize all micropolygons of a sub-patch to calculate *potential* coverage and then perform the z-test (and possibly shading) in parallel.

One of the interesting attributes of a REYES-style renderer is the natural integration of displacement shaders, where one just displaces vertices of the grid of micropolygons. However, displaced micropolygons might be displaced into neighboring tiles, so usually a maximum displacement is considered during bucketing to avoid this. This a very serious challenge for the small tiles favored by our renderer on existing CUDA cards.

10 Update

After writing the previous Sections we made some minor changes to our implementation. Overall we observe around a 2x performance increase but the relationship between the individual steps stayed the same. Instead of measuring everything again, we just give the new numbers for the killeroo model on the same hardware (see Table 4) and briefly describe the changes.

Oracle: We reordered the computation and avoided some unnecessary ones. For example if a patch is already classified as front facing we do not reconstitute the tangent patch or perform the back patch test again. Understanding that shared memory needs to be declared `volatile` explicitly, allowed us to save most calls to `__syncthreads()`, which improved performance considerably. Despite doing more tests, our oracle is now faster than the bounding box test of Patney Owens and the flattness test of Eisenacher et al. Overall we bound and split 269 k / (6.6+1.5) = 33.2 M patches per second for the killeroo model.

Sort Middle: Our subdivision scheme naturally clusters sub-patches spatially. This is beneficial for caching during reconstitution, but harmful for sorting: Neighboring patches in the patch queue have a very high probability of being sorted into the same bucket, which requires costly synchronization. By sorting patches with a large stride using `int globalPatchIdx = blockIdx.x + patchIdxInBlock*gridDim.x` for each thread we loose coalasced memory access but avoid many of the much more costly synchronizations. As a result sorting becomes almost 3x faster.

Rasterizer: The rasterizer has been optimized in several ways. We use a 3D cross product instead of the 4D cross product (this is already described in Section 6.2). The bounding boxes are now computed avoiding redundant computations, cracks are fixed with less warp serializes and one thread is responsible for one sample instead of one pixel. The latter requires the pixel color to be resolved using shared memory but improves SIMD efficiency with multisampling considerably: Four samples per pixel take only about 2x longer than one sample and 16 samples per pixel take only about 6x longer.

For comparison we also give the timings for a comparable hybrid approach using our bound and split framework. For each sub-patch we compute 16 surface vertices, hide cracks using our overlap strategy, and store them with normals and texture coordinates. We map them to the hardware pipeline and render them using a pre-computed index buffer and Phong shading. For this example our software rasterizer is about 2.5x slower than the hybrid approach.

	Hybrid	1 spp	4 spp	16 spp
System overhead	5.4 ms			
Init	0.2 ms			
Oracle	7.7 ms			
Splits	1.5 ms			
Compute vertices & attributes	7.8 ms			
Count		2.5 ms		
Sort		0.3 ms		
Reconstitute sub-patches		8.4 ms		
Perspective divide		1.8 ms		
Phong-Shade (& nrml comp.)		5.6 ms		
Hide cracks		5.3 ms		
Bounding boxes		3.6 ms		
Sampling	16.1 ms	45.8 ms	109.2 ms	254.2 ms
Resolve pixel color		0.7 ms		
Total	38.7 ms	88.8 ms	152.2 ms	297.2 ms

Table 4: Rendering the killeroo at 1600x1200 with the latest implementation. Using 8x8 tiles and 4x4 bounding boxes, the model is subdivided to 270k subpatches. For the hybrid approach 4.3 M vertices for 4.9 M triangles and their vertex attributes (164 MB) get mapped. Including overlap our approach renders 445k patches, diced into 4 M microquads. A 7.3 MB texture with the final image is mapped and rendered using a screen sized quad.

References

[1] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Picture*, chapter 6, pages 153–154. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[2] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. *SIGGRAPH Comput. Graph.*, 21(4):95–102, 1987.

[3] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. *ACM Trans. Graph.*, 22(3):657–662, 2003.

[4] Christian Eisenacher, Quirin Meyer, and Charles Loop. Real-time view-dependent rendering of parametric surfaces. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 137–143, New York, NY, USA, 2009. ACM.

[5] M Harris, J. D. Owens, S. Sengupta, Y. Zhang, and A. Davidson. CUDPP: CUDA data parallel primitives library. April 2008.

[6] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[7] Harald Niederreiter. *Random number generation and quasi-Monte Carlo methods.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[8] Nvidia Corporation. *NVIDIA CUDA: Compute unified device architecture*, June 2008.

[9] John D. Owens, Brucek Khailany, Brian Towles, and William J. Dally. Comparing Reyes and OpenGL on a stream architecture. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 47–56, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[10] Anjul Patney and John D. Owens. Real-time REYES: Programmable pipeline and research challenges. In *ACM SIGGRAPH Asia Course Notes*, 2008.

[11] Anjul Patney and John D. Owens. Real-time REYES-style adaptive surface subdivision. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia)*, 27(5), December 2008.

[12] Juan Pineda. A parallel algorithm for polygon rasterization. *SIGGRAPH Comput. Graph.*, 22(4):17–20, 1988.

[13] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.