Data-parallel Micropolygon Rasterization

C. Eisenacher [1] and C. Loop [2]

[1]University of Erlangen-Nuremberg, [2]Microsoft Research Redmond

Abstract

We implement a tile based sort-middle rasterizer in CUDA and study its performance characteristics when used as a backend for adaptive tessellation down to micropolygons. Tessellation and bucketing map very well to the data-parallel paradigm of CUDA, and the majority of time is spent with rasterization. Despite this, our fastest implementation is able to reach 30-50% of the hardware rasterization performance of an Nvidia GTX 280. Overall we are able to rasterize 4 M textured and Phong shaded microquads into a 1600x1200 framebuffer at 10-12 fps.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Graphics processors, Parallel processing

1. Introduction

Modern GPUs have evolved into general purpose floating point processing devices containing many parallel cores with wide SIMD units. They are fast and flexible enough that tile based sort-middle triangle rasterization, implemented in software on such a GPU, has been proposed as alternative to fixed-function rasterization units [SCS*08].

At the same time real-time rendering with micropolygons became feasible. Higher-order surfaces are tessellated directly on the GPU and the resulting triangles are processed by the traditional pipeline. However, this creates many tiny triangles that are difficult to process as memory access patterns and SIMD efficiency degrade with small polygons.

Fatahalian et al. [FLB*09] analyze the percentage of point-in-polygon tests that result in hits, or *sample test efficiency* (STE), of several data-parallel rasterization algorithms. To understand how these results translate onto existing hardware we implement and analyze a sort-middle rasterizer for rational bicubic Bézier patches in CUDA.

Interestingly enough a simple 8x8 stamp, for which Fatahalian et al. predict a STE of only 2%, translates to 30-50% of the hardware rasterizing performance in our setting.

2. Previous Work

The concept of a software sort-middle [MCEF94] rasterizer regained a lot of attention with Larrabee [SCS*08]. The authors argue that, while it probably cannot outperform custom silicon, it is “fast enough”, undoubtedly more flexible and should scale well with an increasing number of cores.

Many rasterizers use *stamps*: Multiple samples test in parallel whether they are inside the half-spaces defined by the edges of a polygon [Pin88]. This requires one dot product per edge and sample. A sample is covered if all values are positive. While a hierarchical approach [Gre96, SCS*08] is generally attractive, we only use tile sized stamps.

REYES was developed as an off-line renderer at Pixar in the mid 1980's [CCC87]. It adaptively subdivides higher order surfaces until their screen space size is small (*bound and split*). Then they are *diced* (tessellated) into regular *grids* of *micropolygons*, i.e. polygons with an area on the order of one pixel [AG99]. The grid is shaded and pixel coverage is determined by stochastic super-sampling of the micropolygons. Our overall algorithm is inspired by the REYES pipeline.

Owens et al. [OKTD02] 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 and generate fragments that are composited into the final image. They note that over 80% of the total time is spent subdividing patches.

In 2008 Patney and Owens demonstrate a hybrid GPU approach [PO08]: They subdivide Bézier patches breadth-first and dice them using CUDA. Then they transfer micropolygons to the hardware rasterizer for sampling. Eisenacher et al. [EML09] refine the implementation of Patney and Owens and try to generate as few polygons as possible to minimize transfer overheads. We use a modified version of their algorithm as front end for our rasterizer.

3. Data-Parallel Micropolyon Rasterization

We divide the screen into equally sized *tiles* with associated buckets and perform the following four steps:

Bound and Split: First we adaptively subdivide bicubic Bézier patches until the screen space extent of the sub-patches is smaller than a given threshold.

Dice: Then we evaluate a 4x4 grid for each sub-patch and shade the vertices. Position and color are written to a VBO.

Parallel Bucketing: Then we sort the index of each sub-patch into buckets. A single bucket contains the indices of all patches that cover its associated screen space tile.

Tile Based Rasterization: Finally we load the grids for each tile from the VBO and sample the micropolygons.

3.1. Bound and Split

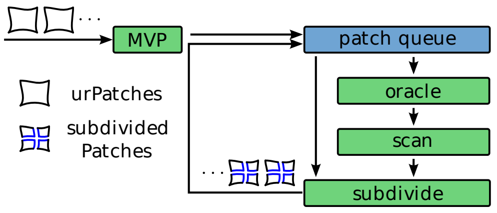

Figure 1: *Bound and split: Patches are examined and subdivided breadth-first until they are small in screen space.*

Our subdivider, visualized in Figure 1, follows the implementation of [EML09] closely: We start by transforming the original 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 whether to *cull*, *keep* or *subdivide* them in a 1:4 split. The decision is written in the form of storage requirements: 0 slots for cull, 1 slot for keep, 4 slots for subdivide.

A parallel scan [HOS*08] is used to compute the prefix sum of this storage decision array. This directly generates the storage locations where the subdivision kernel will store patches. For details we refer the reader to [EML09].

We subdivide in the parameter domain to save memory and bandwidth, and iterate until all patches are smaller than a certain screen space size, e.g. 4x4 pixel. A threshold smaller than the tile size guarantees that a sub-patch covers at most four adjacent tiles, and simplifies bucketing considerably.

3.2. Parallel Bucketing

Once all patches in the patch queue satisfy our criteria, we dice them into 4x4 grids and sort the index of each grid into the buckets of the tiles it overlaps. Our goal is to store a variably sized list of covering patches for each bucket.

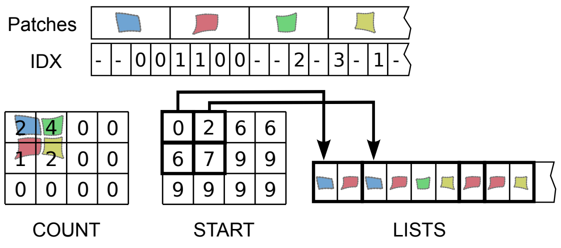

Figure 2: *To bucket the subdivided patches we use CUDA's* `atomicAdd()`*. This counts the patches per bucket and delivers the indices of the patch inside the buckets at the same time. Using those and the prefix sum of the counters we compute the final positions of the patches in the bucket lists.*

To facilitate this in parallel and with variable list lengths, we organize the lists as shown in Figure 2: $COUNT$ stores the length of the list at each bucket, $LISTS$ contains the actual lists and $START$ contains the start indices for each sub-list. We create these lists with the following algorithm:

Init: We initialize the entries of $COUNT$ to zero and allocate a temporary buffer IDX. The latter stores the indices of each patch inside the per bucket lists of the tiles it overlaps. Note that each patch covers at most four adjacent tiles (top-left, top-right, bottom-left, bottom-right) at this point.

Address Calculation: For each patch we perform an `atomicAdd(&`$COUNT[tileID_{0..3}]$`, 1)` on the up to four tiles it overlaps. That way we count the number of patches that will be stored in each bucket correctly, despite multiple threads accessing the same bucket simultaneously. The values returned by the call are the indices of the patch in the sub-list of each bucket and we store them in IDX.

Bucket Write: After the addresses are calculated, we compute $START$ as the prefix sum of $COUNT$ using a parallel scan [HOS*08]. Combining $START$ and IDX we can now sort the patches into the buckets: For each patch in the patch queue we store the index of its grid at $LISTS[START[tileID_{0..3}] + IDX_{0..3}]$.

3.3. Tile Based Rasterization

Using $COUNT$ and $START$ we know for each tile, how many and which grids we need to rasterize from $LISTS$. We launch one CUDA thread block per tile, allocate one lightweight thread per pixel and let the CUDA scheduler handle the load balancing. After loading grid vertices and colors into shared memory, each thread loops over all quads in the grid. It computes edge equations for each quad and tests for coverage. This requires four dot products and a sign test [Pin88]. For covered samples we interpolate z from the quad vertices, perform a z-test and store the interpolated color if necessary. Z- and color-buffer values are stored in one register each.

For multisampling, having each thread test its subsamples leads to poor SIMD efficiency. Instead, we use same thread block size and divide our tile into sub-tiles, so that neighboring threads test neighboring sub-samples for coverage, and resolve the final pixel color via shared memory. This improves SIMD efficiency and is significantly faster. To avoid unnecessary computation we use a few simple bounding box tests, saving about 30% in total time.

We repeatedly and redundantly compute edge equations in each thread and for each sub-tile. This obviously wastes considerable amounts of computation but avoids complicated access patterns into shared memory. Computing edge equations during dicing and loading edge-equations instead of grid positions seems like a promising tradeoff to explore.

4. Results and Discussion

To test our rasterizer we render the familiar Utah *teapot*, the *bigguy* and the *killeroo* model with the view points shown in Figure 3. All results are measured on an nVidia GTX 280 using CUDA 2.1 on Windows XP.

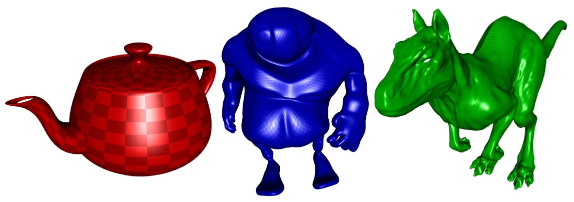
Figure 3: *The Utah teapot, the bigguy and the killeroo rendered with our system; 32, 3570 and 11532 urPatches.*

4.1. Bound and Split

Subdivision in the parameter domain makes the oracle computationally more expensive, but subdivision becomes trivial and we need considerably less bandwidth. Overall, we can bound and split 29.4 M patches per second for the killeroo model. This is highly competitive: Patney and Owens [PO08] report 2.0 M/s on a GTX 8800, Eisenacher et al. [EML09] report 13.2 M/s on a GTX 280.

4.2. Parallel Bucketing

Small 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 store (and rasterize) multiple copies.

We render a 1600x1200 image of the killeroo with a screen space bounding box less than 4x4 pixels. Table 1 shows the effect of different tile sizes when sorting the resulting 270k sub-patches. Increasing the tile size reduces the tiles per patch from 2.45 for 4x4 tiles to 1.14 for 32x32 tiles. However, despite having to write over twice as many indices, sorting into 4x4 tiles is faster overall, as we have to deal with considerably less conflicting writes on average.

tile size	4x4	8x8	16x16	32x32
address calc. [ms]	2.48	2.55	2.94	4.05
bucket write [ms]	0.34	0.34	0.33	0.33
patches written	662 k	445 k	352 k	309 k
tiles per patch	2.45x	1.65x	1.30x	1.14x

Table 1: *Bucketing 270k sub-patches using different tile sizes. Large tiles require more synchronization during address calculation, even though less writes occur.*

4.3. Tile Based Rasterization

Similar to parallel bucketing, the performance of tile based rasterization is dependent on the tile size. We render the same 1600*x*1200 image of the killeroo model with 4x4, 8x8 and 16x16 tiles and different scales. As a different tile sizes have different amounts of overlap, Figure 4 visualizes the ms per frame vs. the number of patches actually rasterized.

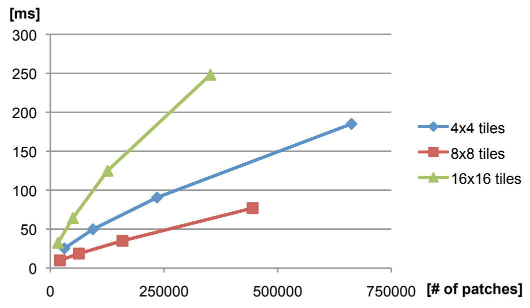

Figure 4: *Milliseconds per frame for rendering the killeroo at four different zoom levels. Patches that cover multiple tiles are rasterized multiple times. Using 8x8 tiles performs best.*

For small tiles less unnecessary coverage tests are performed for pixel sized polygons and more tiles are 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 and being considerably faster than 16x16 tiles despite more overlap.

4.4. Overall performance

Table 2 shows timings for a few selected screen sizes and settings. We use 8x8 pixel sized screen tiles and subdivide until the screen space bounding box (BB) of a patch 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 (spp).

	512x512						1600x1200					
	8x8 BB			4x4 BB			8x8 BB			4x4 BB		
	#	1 spp	4 spp	#	1 spp	4 spp	#	1 spp	4 spp	#	1 spp	4 spp
teapot	25k	6.9	10.4	61k	12.0	19.0	138k	26.0	45.8	348k	54.9	95.6
big guy	32k	7.8	12.2	76k	13.7	22.6	165k	35.0	63.0	404k	71.1	129.4
killeroo	38k	9.1	14.4	84k	14.7	23.7	183k	38.3	70.0	445k	77.0	140.6

Table 2: *Total time per frame in ms using 8x8 screen tiles. Patches are subdivided until their bounding box (BB) is smaller than 8x8 or 4x4 pixel. Each patch is diced into a 4x4 grid and shaded. For each grid 9 microquads are rasterized.*

To show that the dominating cost is the actual rasterization, we present a breakdown in Table 3. For comparison we also give timings for a hybrid approach, where we render the VBO containing vertices and colors using the hardware pipeline and a pre-computed index buffer. For this example our software rasterizer achieves about 48% of the total fps of the hybrid approach. Note that we rasterize almost twice as many microquads as patches operlap multiple tiles.

	Hybrid	1 spp	4 spp	16 spp
System overhead	4.6			
Bound & Splits	7.7 + 1.5			
Evaluate grid	6.0			
Count		2.5		
Sort		0.3		
Setup		8.2		
Sampling	19.0	46.2	110.6	261.4
Total	37.2	77.0	140.6	290.2

Table 3: *Rendering the killeroo at 1600x1200. Using 8x8 tiles and 4x4 bounding boxes, we create 270 k sub-patches or 2.4 M microquads for the HW rasterizer. Due to patch overlap we test 445k grids, or 4 M microquads. Time in ms.*

5. Conclusion and Future Work

We have presented and analyzed a bucketing rasterizer for micropolygons written in CUDA. Tessellation and sorting map well to the data-parallel model and the dominant cost is rasterization.

Our implementation can be improved in many areas: Cracks resulting from adaptive subdivision need to be taken care of, and displacement mapping, which is a major selling point for micropolygons, is difficult with the tiny tiles favored by our current implementation. High overlap results in unnecessary coverage tests and tessellating into 4x4 grids introduces a lot of redundant shading for the edge vertices. Larger tiles and a more hierarchical approach might leverage many of those issues.

Of course a software rasterizer will always consume computational resources for an operation that is effectively free on current GPUs. While today's hardware is not built for micropolygons, it is an encouraging result, that a comparably simple implementation with theoretical STE of only 2%, can perform within the reach of highly optimized silicon.

Acknowledgements: We want to thank the anonymous reviewers for their work and detailed feedback, as well as Bay Raitt of Valve Software for the big guy model. The Killeroo model is courtesy of Headus (metamorphosis) Pty Ltd (available at `http://www.headus.com`).

References

[AG99] APODACA A. A., GRITZ L.: *Advanced RenderMan: Creating CGI for Motion Picture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999, ch. 6, pp. 153–154.

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The Reyes image rendering architecture. *SIGGRAPH Comput. Graph. 21*, 4 (1987), 95–102.

[EML09] EISENACHER C., MEYER Q., LOOP C.: Real-time view-dependent rendering of parametric surfaces. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2009), ACM, pp. 137–143.

[FLB*09] FATAHALIAN K., LUONG E., BOULOS S., AKELEY K., MARK W. R., HANRAHAN P.: Data-parallel rasterization of micropolygons with defocus and motion blur. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 59–68.

[Gre96] GREENE N.: Hierarchical polygon tiling with coverage masks. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM, pp. 65–74.

[HOS*08] HARRIS M., OWENS J. D., SENGUPTA S., ZHANG Y., DAVIDSON A.: CUDPP: CUDA data parallel primitives.

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (1994), 23–32.

[OKTD02] OWENS J. D., KHAILANY B., TOWLES B., DALLY W. J.: Comparing Reyes and OpenGL on a stream architecture. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2002), Eurographics Association, pp. 47–56.

[Pin88] PINEDA J.: A parallel algorithm for polygon rasterization. *SIGGRAPH Comput. Graph. 22*, 4 (1988), 17–20.

[PO08] PATNEY A., OWENS J. D.: Real-time REYES-style adaptive surface subdivision. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia) 27*, 5 (Dec. 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 Trans. Graph. 27*, 3 (2008), 1–15.