Parallel Reyes-style Adaptive Subdivision with Bounded Memory Usage

Thomas Weber*
Vienna University of Technology

Michael Wimmer†
Vienna University of Technology

John D. Owens‡
UC Davis

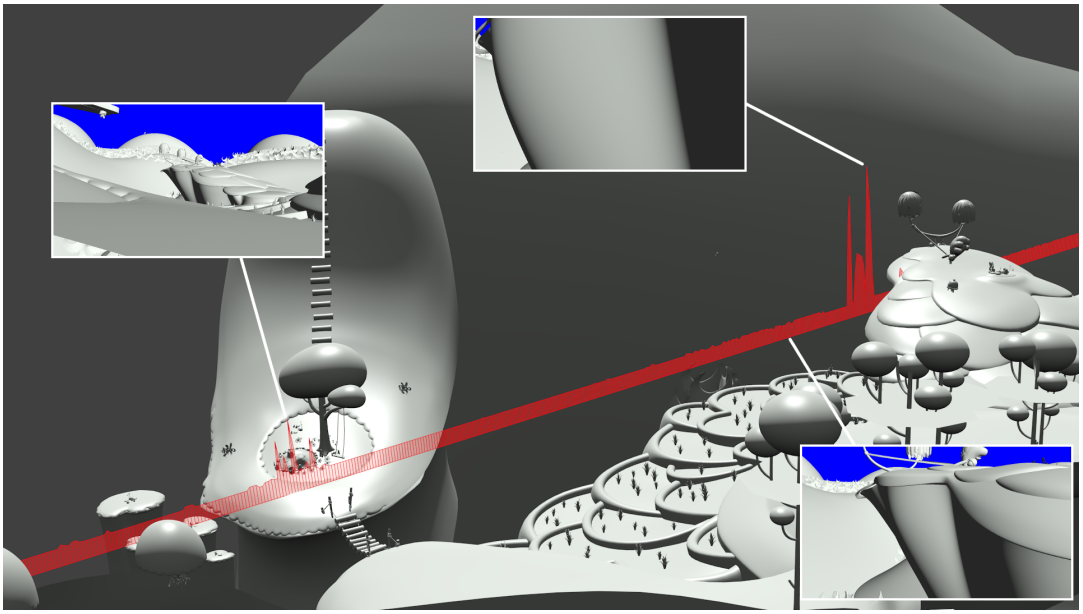

Figure 1: *Illustration of the memory usage for breadth-first adaptive subdivision as a camera moves through a scene. The thumbnails in the scene show the view from the camera at the highlighted positions. While the overall memory consumption of breadth-first remains mostly constant, there are locations where significantly more memory can be necessary. Scene courtesy of Zinkia Entertainment, S.A.*

Abstract

Recent advances in graphics hardware have made it a desirable goal to implement the Reyes algorithm on current graphics cards. One key component in this algorithm is the bound-and-split phase, where surface patches are recursively split until they are smaller than a given screen-space bound. While this operation has been successfully parallelized for execution on the GPU using a breadth-first traversal, the resulting implementations are limited by their unpredictable worst-case memory consumption and high global memory bandwidth utilization. In this paper, we propose an alternate strategy that allows limiting the amount of necessary memory by controlling the number of assigned worker threads. The result is an implementation that scales to the performance of the breadth-first approach while offering three advantages: significantly decreased memory usage, a smooth and predictable tradeoff between memory usage and performance, and increased locality for surface processing. This allows us to render scenes that would require too much memory to be processed by the breadth-first method.

*e-mail:t.weber@cg.tuwien.ac.at
†e-mail:wimmer@cg.tuwien.ac.at
‡e-mail:jowens@ece.ucdavis.edu

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms

Keywords: GPGPU, Reyes, surface rendering, parallel rendering

1 Introduction

The steady rise in the flexibility and performance of graphics hardware over the years has made it feasible to implement increasingly sophisticated rendering algorithms in real time. Among these is the Reyes rendering architecture [Cook et al. 1987], which was developed during the 1980s for production rendering.

Using Reyes for real-time rendering is desirable because it allows scenes composed of displaced higher-order surfaces to be rendered directly without any visible geometry artifacts. Surfaces are tessellated into sub-pixel sized polygons during rendering and shaded on a per-vertex basis. This also allows high-quality motion-blur and depth-of-field effects using stochastic rasterization.

Even though each stage of Reyes rendering has been successfully mapped to the programmable features of the GPU, the adoption of Reyes for real-time graphics applications has so far been hampered by practical considerations. While image quality and rendering performance are quite relevant, one of the most important aspects in this regard is robustness. For instance, it is unacceptable that a graphics pipeline can run out of memory for some unfortunate placement of the camera. Production rendering systems can fall back to pages swapping to disk, but a GPU-based Reyes pipeline must guarantee a peak memory bound for all components in order to be useful.

Figure 1 demonstrates how the memory consumption for breadth-first bound-and-split can look as a camera moves through a scene. (Figure 6 shows the same data as a regular graph with axis labels.) Note how this value stays at a mostly constant level for most of the path with a small number of sharp spikes in memory consumption at certain locations. Properly rendering from views such as these can easily exceed the memory budget of an application or even the available physical memory.

In this paper, we present a method that allows choosing the memory budget for parallel bound-and-split. When there is enough available memory, the performance and behavior of the algorithm is the same as the breadth-first approach. In case the breadth-first memory requirements exceed our memory budget, we can get a smooth, asymptotic tradeoff between memory usage and performance. This should make it possible to write rendering systems that perform well in the general case, while still being robust enough to render arbitrary scenes and viewpoints with reasonable performance.

2 Previous Work

Reyes tessellates surfaces into micropolygons using a two-stage approach [Cook et al. 1987]. In the first phase, surfaces are recursively subdivided until they are smaller than a given screen-space bound. After this, the surfaces are uniformly evaluated to create grids of polygons. This phase is also called *dicing*. The reason for separating tessellation into these two steps is that it results in more uniformly sized polygons and better vectorization than either step could achieve on its own [Fisher et al. 2009].

Applying only dicing would lead to problematic over- or under-tessellation for parts of surfaces that are strongly distorted, for instance, due to perspective projection. On the other hand, while doing full subdivision up to the micropolygon level is possible, this leads to unnecessary over-tessellation since surfaces can only be halved, effectively limiting the dicing rates to powers of two. Having dicing as a separate phase avoids this, since the optimal dicing rate for every bounded surface can be chosen. Performing shading and rasterization on grids instead of single polygons is also desirable for parallelization, since vertex and face operations can be vectorized.

The dicing phase maps to hardware tessellation supported on recent graphics APIs and GPUs [Loop and Schaefer 2008]. This feature works well and is commonly used in current games. Hardware tessellation also allows the selection of separate tessellation levels for the inside and each boundary edge of a surface in order to avoid surface cracks. We will not go into detail about our implementation of the dicing phase, since we feel that this topic is already well explored. Hardware tessellation has been around for several years and is supported by most recent graphics processors and APIs. Instead, we will focus on the memory-efficient implementation of the far less predictable subdivision phase. The output of this is a flat array of parametric ranges on the 2D surfaces, which can easily be used as input for hardware tessellation.

Programmable Tessellation on the GPU Over the past five years, many researchers have used the programmable features of the GPU to implement high-quality tessellation. Patney and Owens's adaptive subdivision on the GPU [2008] transformed the typical depth-first recursive traversal of split surfaces into a breadth-first operation. While this performs well on the GPU, using a breadth-first traversal means that the peak memory consumption of this algorithm rises exponentially with the number of splits [Zhou et al. 2009; Loop and Eisenacher 2009; Fisher et al. 2009]. Sanchez et al. [2011] also note the disadvantages of breadth-first scheduling (compared to other scheduling strategies) with respect to memory usage and locality. Nevertheless, several papers build on this method.

Zhou et al. [2009] use breadth-first adaptive subdivision as part of a full GPU-based interactive Reyes renderer called *RenderAnts*. RenderAnts uses dynamic scheduling to ensure bounded memory usage for fragment processing. However, no such bound is given for adaptive subdivision. Patney et al. [2009] use the breadth-first approach for crack-free view-dependent tessellation of Catmull-Clark subdivision surfaces, and Eisenacher et al. [2009] adopt the same breadth-first approach for parametric surface subdivision, but also consider surface curvature, resulting in considerably fewer surfaces being created.

Fisher et al. [2009] present a method for efficiently avoiding surface cracks during subdivision by applying the scheme used in hardware tessellation. They allow surfaces to be split along nonisoparametric edges to ensure integer tessellation factors at all times. Their paper also discusses the scalability issues of breadth-first subdivision and gives this as a reason for their decision to implement their adaptive subdivision on the CPU using multithreading and balanced stacks. This gives excellent memory scalability and good locality, but does not scale well beyond a relatively small number of concurrent threads.

Tzeng et al. [2010] consider adaptive subdivision from a scheduling point of view. They make use of persistent kernels and distribute the total work over many work-groups. To ensure load balance, they advocate a scheduling strategy based on work-stealing and work-donation. This approach has the advantage of avoiding host-device interaction for enqueueing additional iterations. However, while general memory consumption is greatly reduced with their approach, the peak memory usage remains unpredictable.

A method for the real-time tessellation of Catmull-Clark surfaces on the GPU was presented by Nießner et al. [2012a]. They avoid having to fully subdivide all surfaces by directly tessellating regular faces as B-Spline surfaces and only applying further subdivisions to faces containing an extraordinary vertex. This allows them to greatly reduce the memory consumption. In a follow-up paper, they discuss how semi-sharp creases can be handled efficiently [Nießner et al. 2012b]. While their presented methods work well, their approach is essentially an efficient implementation of dicing Catmull-Clark surfaces, since the subdivision level for a single model has to be constant.

In a different application domain, Hou et al. consider the problem of memory-efficient parallel tree traversal during k-d tree construction [2011]. With similar motivation to this work, they propose a partial breadth-first search traversal scheme that only evaluates a limited number of leaves in a tree.

3 Adaptive Subdivision on the GPU

The classic Reyes pipeline implements adaptive subdivision as a recursive operation. Reyes estimates the screen-space bound of a surface to decide whether the surface needs further subdivision or can be sent to the next pipeline stage for dicing. If further subdivisions are necessary, Reyes splits the surface and recursively calls bound-and-split on the new sub-surfaces. This process can be thought of as the depth-first traversal of a tree ("split tree"). While this is easy to implement on regular CPUs and requires minimal memory ($O(N + k)$, where N is the number of input surfaces and k is the maximum depth of the split tree), this approach is not suitable for the GPU since it is inherently sequential. Due to this exponential growth in memory consumption, the static preallocation of memory for this operation quickly becomes unfeasible.

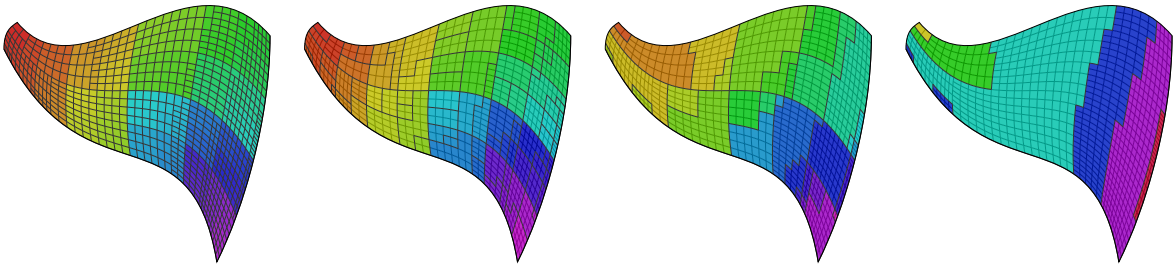

Figure 2: *Comparison of evaluation order of surfaces for different batch sizes (p is the number of surfaces in a batch). Surfaces that are created in the same iteration are shaded in the same color. This shows the locality-preserving property of our subdivision algorithm: surfaces that are spatially close together are evaluated in the same iteration.*

Patney and Owens [2008] parallelize the Reyes split phase by transforming this depth-first operation into a breadth-first traversal of the split tree. This way, a single iteration of the adaptive subdivision can be implemented using a parallel bound kernel, prefix sums, and a copy kernel. These are then iterated until all surfaces have been successfully bounded. Figure 3 gives an overview on how this approach works.

While this is simple to implement and yields excellent speedup, this approach suffers from high peak memory usage. Since all nodes of a single depth in the split-tree have to be held in memory, the worst-case memory consumption is the number of possible leaves of a binary tree of maximum depth k. This is $O(N \cdot 2^k)$, where N is the number of input surfaces processed at once. Due to this exponential growth in memory consumption, the static preallocation of memory for this operation quickly becomes unfeasible.

It is possible to split the input surfaces into several batches that are subdivided separately. This slightly reduces the worst-case memory consumption, but the overall memory consumption can still be very high, especially since the total memory consumption of the individual input surfaces varies highly due to perspective projection. The results section presents the test scene EYESPLIT, which has a very high memory requirement despite only containing a single surface. Furthermore, reducing the batch size also reduces the overall performance especially during the first few iterations.

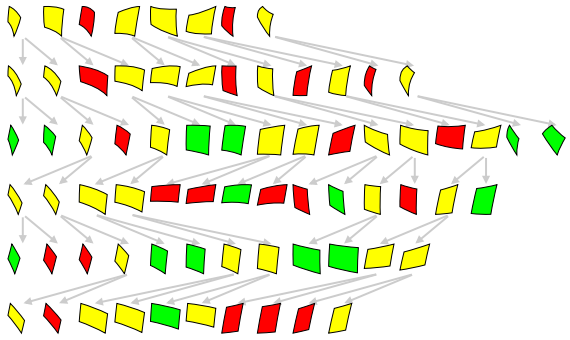

Figure 3: *Schematic overview of breadth-first subdivision. Each row represents the state of the surface buffer during one iteration. Each surface can either be culled (red), split (yellow), or drawn (green). For each split surface in the previous iteration, two new surfaces are generated in the following iteration. This always happens for all surfaces in the surface buffer.*

3.1 Adaptive Subdivision with Bounded Memory

Instead, we propose an adaptation of this approach where the number of surfaces processed at a given iteration is limited by a constant value p. The buffer of surfaces is used as a parallel last-in-first-out data structure where surfaces are read from the end of the buffer, and any generated sub-surfaces are appended back to the end. By using this approach, we can bound the peak memory consumption by $O(N + p \cdot k)$. Figure 4 illustrates how this approach works.

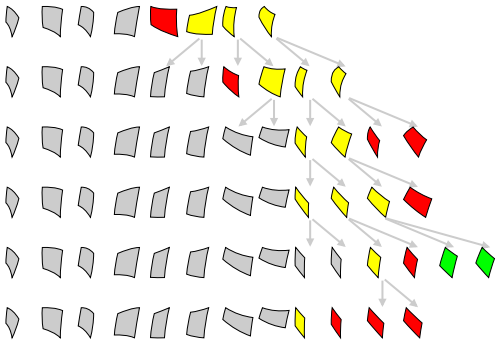

Figure 4: *Schematic overview of how our memory-bounded subdivision operates. Unlike in Figure 3, the number of active surfaces at each iteration is constant (in this case, $p = 4$). The other surfaces are inactive and shaded in gray.*

Adding the batch size p as a tweakable parameter in the subdivision process allows us to balance between memory consumption and performance. Figure 7 shows the impact the chosen batch size and the amount of assigned memory have on the overall subdivision time. As the batch size increases, the subdivision time asymptotically approaches that of breadth-first subdivision. Our approach also preserves locality, as can be seen in Figure 2.

In our implementation, a *bound* kernel first copies the last p surfaces into a temporary buffer and estimates the screen-space bound for each of them. Depending on this bound, the kernel decides an action to be taken on this surface (*draw*, *split*, or *cull*), which is stored as a flag value in a separate buffer.

Whether a surface is ready to be drawn depends on the size of its screen-space bound, which is estimated by the kernel. Surfaces are culled when they are outside of the camera frustum or a surface has been split the maximum number of times. More advanced systems may also support occlusion culling, for instance by accessing a hierarchical depth buffer in GPU memory; however, our implementation does not at the moment.

Procedural displacement also affects the screen-space bound of a surface. While there exist methods to efficiently estimate the bounds of displaced surfaces [Munkberg et al. 2010; Nießner and Loop 2013], our renderer is limited to a configurable safety margin to avoid erroneous culling of displaced surfaces near the screen edge.

The temporary storage of surfaces is necessary to avoid surfaces being overwritten by split surfaces before they have been read. This is not necessary in breadth-first subdivision, which uses a ping-pong buffer approach. While our temporary storage requires one additional write operation, the performance cost is minimal.

We then apply a prefix-sum operation to these flag buffers to calculate write locations. The *split* kernel checks the flag buffer and either copies the bounded surface into the output buffer or applies a split operation and places the resulting sub-surfaces at the end of the surface buffer.

For a surface P, the split-results P'_0 and P'_1 are placed at address $a_0 = S + f_c \cdot 2 + 0$ and $a_1 = S + f_c \cdot 2 + 1$ respectively, where S is the current size of the surface buffer and f_c is the prefix sum of the split flags. Using this particular order is necessary to prove the memory bound of our algorithm.

The flags accumulated by the prefix-sum operator are then used in a subsequent copy kernel to find the correct location for writing in the global-output and surface buffers. Surfaces remaining in the surface buffer will be further split by subsequent iterations of our subdivision algorithm, until the surface buffer is empty. The output surfaces of a single iteration are copied to an output buffer from where they are ready to be used by subsequent dicing and rasterization kernels.

In our implementation, the output surfaces are immediately consumed by subsequent pipeline stages. This way, we can make sure that the maximum number of surfaces that have to be processed in later stages is p. It is also possible to collect the output of several iterations before passing it on. However, collecting the entire output of the algorithm before passing it along further is not recommended, since this might once again lead to unbounded memory consumption due to the unpredictable amount of output surfaces.

Keeping the children of a surface that has been split close together also improves locality. Figure 5 shows the difference between placing the sub-surfaces in the order described by Patney and Owens [2008] (NONINTERLEAVED) with our approach (INTERLEAVED).

We chose these names due to the order in which the split results are written into the output buffer. NONINTERLEAVED separates the left-hand and right-hand split products of $\{a, b, c\}$ in the order $\{a_0, b_0, c_0, a_1, b_1, c_1\}$, while INTERLEAVED places left-hand and right-hand products in the order $\{a_0, a_1, b_0, b_1, c_0, c_1\}$.

Active surfaces are always read from the end of the surface buffer, and their potential children in the subdivision tree are always put back at that end again. As a result, and since the local order of the split products mirrors that of their parents, we can always expect that the surface buffer is sorted by subdivision level. This means that surfaces closer to the beginning of the buffer have had fewer subdivisions applied to them than those at the end.

At every iteration, the subdivision algorithm consumes p surfaces and appends at most $2p$ surfaces back to the buffer. These new surfaces are guaranteed to have a higher subdivision level than the ones that were consumed. This has the effect that for each intermediate subdivision level, there can be at most p surfaces in the buffer (safe for the root and top levels). Since we are actively limiting the maximum allowed subdivision level to k and there can be at most p surfaces per subdivision level, we can make sure that there are at most $O(N + p \cdot k)$ surfaces in the buffer at any point in time.

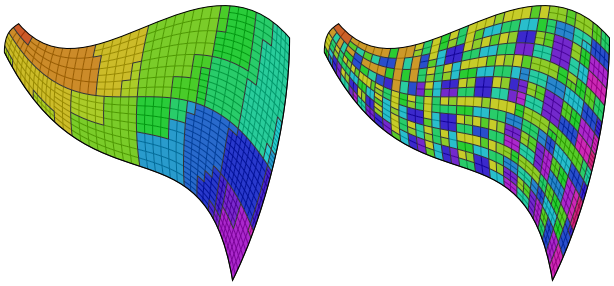

Figure 5: *Illustration of the effect the placement order after split has on the locality of generated surfaces. Surfaces created during the same iteration share the same color.* INTERLEAVED *is the order described in this section while* NONINTERLEAVED *uses the order of Patney and Owens.*

4 Results

We have implemented a simple Reyes renderer (called *Micropolis*) in OpenCL that implements adaptive subdivision, dicing, shading, and micropolygon sampling as kernels on the GPU. It supports both the breadth-first adaptive subdivision approach and our memory-bounded method.

BREADTH implements the breadth-first approach of Patney and Owens [2008]. In case this algorithm runs out of memory, it will allocate further memory on-the-fly. This is necessary since the worst-case memory consumption of breadth-first subdivision is so high that preallocation is not possible. This exact situation is what we want to avoid with this paper. Since we allow for a certain number of rendered frames before measurement, the necessary time-overhead for this does not affect the measured subdivision times.

BOUNDED implements adaptive subdivision with bounded memory as described in the previous section.

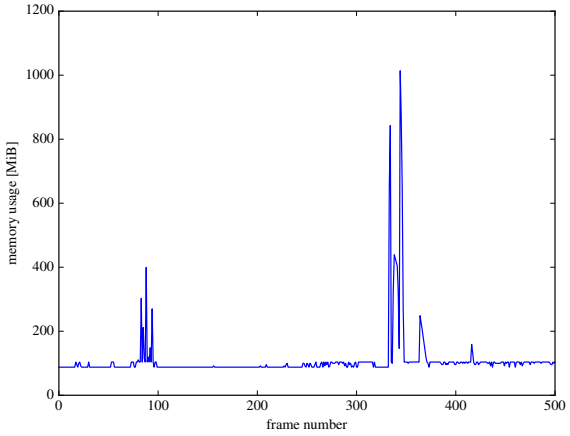

Figure 6: *Memory usage of* BREADTH *as the camera moves along a straight path through the* ZINKIA *scene. Figure 1 gives the position and local context for the features in this graph.*

Table 1 shows the test scenes we used for evaluating our renderer.

	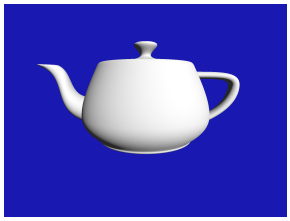	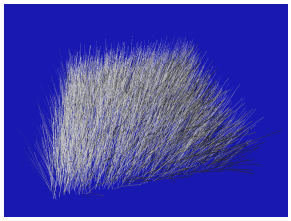	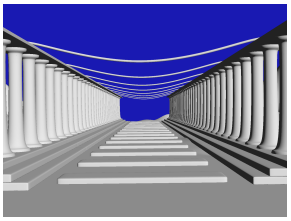	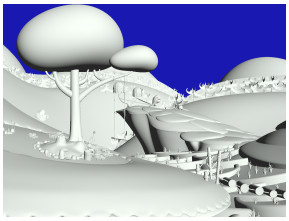
	TEAPOT	HAIR	COLUMNS	ZINKIA
N	32	10 000	12 850	999 812

Table 1: *Overview of the different test scenes used for performance analysis. N is the number of surface patches in a scene before applying adaptive subdivision. Not pictured is the synthetic test scene* EYESPLIT*, because all that can be seen is a white rectangle over the entirety of the frame buffer.* ZINKIA *scene courtesy of Zinkia Entertainment, S.A.*

TEAPOT contains a single large object composed of a small number of surfaces. HAIR is a single mesh with a large number of surfaces and moderate depth complexity. COLUMNS contains about the same number of surfaces as HAIR, but has a lower depth complexity. The ZINKIA scene is very detailed and contains almost a million surfaces.

We have prepared three different viewpoints to evaluate ZINKIA. These three views are extracted from a straight path that has the camera move along a line through the ZINKIA scene as shown in Figure 1. Figure 6 shows the breadth-first memory usage at each position of this path. The views we chose are one representing the average case (ZINKIA1), one for the highest memory spike near the tree (ZINKIA2), and one for the 1 GiB spike close to the cliff (ZINKIA3).

In addition, we have also prepared a synthetic test scene called EYESPLIT, which cannot be reasonably pictured. This is intended to demonstrate the possible worst-case behavior of our subdivision algorithms. EYESPLIT contains a single planar surface patch with the camera placed in such a way that the split axis of the surface falls onto the camera's eye plane. This has the effect that the subdivision of the surface does not terminate before the allowed number of recursive splits has been exhausted and the surface gets culled. The eye-split problem is an intrinsic property of the Reyes pipeline, and artists have learned to avoid it in production rendering. Nevertheless, it is important that such a configuration can be evaluated without the subdivision pipeline stage of a renderer exceeding its memory budget.

All benchmarks have been measured on a system with an AMD Radeon R9 290 GPU and a 3.4GHz Intel Core i5-4670K CPU. The graphics driver used was Catalyst 14.9 on a 64-bit Linux system.

Table 2 lists the execution results for various combinations of adaptive subdivision methods and test models. The scenes are rendered at a resolution of 1280×720 and surfaces are split until they are smaller than 8 pixels along each dimension. For BOUNDED, three different batch sizes (low: 10000, medium: 40000, medium: 200000) are evaluated. The batch size of BREADTH is defined by the scene and view itself. The maximum number of recursive subdivisions k has been set to 23.

Note that the memory consumption of BREADTH is the actual amount of necessary memory, while BOUNDED is configured to allocate enough memory for the worst-case possible memory consumption. Especially for simple scenes, this can mean that the conservative amount of memory allocated by BOUNDED exceeds the amount of memory actually needed by both BREADTH and BOUNDED. The average case is usually a lot better. A good example for this is HAIR, which actually only requires at most 7 subdivisions to any surface in the scene. This can also be seen from the *max patches* value in table 1, where the actual amount of stored patches for BOUNDED always remains lower than for BREADTH.

A variant of BOUNDED that reallocates memory buffers on-the-fly like BREADTH does could significantly reduce the amount of necessary memory for these scenes. Our own focus was more on handling extreme cases gracefully while accepting a constant memory budget for anything lower. This is why we have not implemented this.

For configurations where the view-inherent batch size of BREADTH does not exceed the configured batch size, we can achieve a similar performance with BOUNDED. This is expected, since the exact same amount of computation kernels with the same dimensions are executed. In case the assigned batch size of BOUNDED is lower than that of BREADTH, we get a smooth transition from low to high depending on the amount of assigned memory. Especially for scenes with high memory demand like ZINKIA3, assigning just 11% of the memory necessary for BREADTH can give 66% of the overall performance.

The 1 GiB spike of ZINKIA3 shows that doing naive breadth-first subdivision is not feasible for real-world graphics applications. The Zinkia scene is in no way extreme in what is to be expected of Reyes rendering for interactive applications, and the render settings we have chosen should be reasonable for the scene at hand. 1 GiB of memory is 25% of the total physical memory of a top-of-the line desktop GPU, and considering we are only rendering at 720p, this value would grow for higher resolutions. Figures like these seem especially prohibitive in the mobile space where such a memory consumption can easily exceed the total available memory on current devices.

Figure 7 demonstrates the impact of the chosen batch size on the performance of BOUNDED. The achievable processing rate depends highly on the intrinsic parallelism of a scene, with simpler scenes very quickly reaching a plateau. The performance of complex scenes like ZINKIA2/3 and EYESPLIT asymptotically approaches that of BREADTH when more memory is assigned. The curve of HAIR shows how the processing rate quickly rises with more assigned resources, starts to go flat, and then remains almost constant past a certain point. This is the point at which the batch size is large enough to keep all surfaces active at all times. It can be seen that the other curves mirror this behavior at different scales.

Note that the memory values used for the horizontal axis in figure 7 don't include the constant memory requirement for the initial number of patches. This is done to make the memory usage and batch-size axes align. If we didn't do this, the plot would be shifted on the x axis, with the ZINKIA plot being the only one with a clearly

scene	method	batch size	time [ms]	memory [MiB]	max patches	processed	processing rate [M patches/s]
TEAPOT	BREADTH	5030	1.72	0.52	5030	22172	12.92
TEAPOT	BOUNDED	10000	1.69	4.88	5030	22172	13.11
TEAPOT	BOUNDED	40000	1.70	19.53	5030	22172	13.03
TEAPOT	BOUNDED	200000	1.69	97.66	5030	22172	13.11
HAIR	BREADTH	150958	1.79	16.27	150958	430958	240.37
HAIR	BOUNDED	10000	6.98	5.08	49000	430958	61.71
HAIR	BOUNDED	40000	2.98	19.73	115488	430958	144.51
HAIR	BOUNDED	200000	1.80	97.86	150958	430958	239.91
COLUMNS	BREADTH	38712	3.02	4.14	38712	293178	96.98
COLUMNS	BOUNDED	10000	5.98	5.15	22326	293178	49.00
COLUMNS	BOUNDED	40000	3.02	19.80	38712	293178	97.03
COLUMNS	BOUNDED	200000	3.01	97.93	38712	293178	97.25
ZINKIA1	BREADTH	999812	6.21	107.78	999812	1402768	225.78
ZINKIA1	BOUNDED	10000	29.25	24.91	999812	1402768	47.95
ZINKIA1	BOUNDED	40000	12.50	39.56	999812	1402768	112.20
ZINKIA1	BOUNDED	200000	7.48	117.69	999812	1402768	187.50
ZINKIA2	BREADTH	3847162	18.92	414.74	3847162	9284930	490.81
ZINKIA2	BOUNDED	10000	138.89	24.91	999812	9284930	66.85
ZINKIA2	BOUNDED	40000	55.51	39.56	999812	9284930	167.26
ZINKIA2	BOUNDED	200000	25.62	117.69	1040464	9284930	362.44
ZINKIA3	BREADTH	9766796	33.90	1052.81	9766796	20946484	617.89
ZINKIA3	BOUNDED	10000	315.62	24.91	999812	20946484	66.37
ZINKIA3	BOUNDED	40000	120.62	39.56	999812	20946484	173.66
ZINKIA3	BOUNDED	200000	51.05	117.69	1305212	20946484	410.33
EYESPLIT	BREADTH	1950752	9.25	210.28	1950752	4024029	434.89
EYESPLIT	BOUNDED	10000	62.99	4.88	85236	4024029	63.89
EYESPLIT	BOUNDED	40000	23.68	19.53	260960	4024029	169.90
EYESPLIT	BOUNDED	200000	11.43	97.66	843844	4024029	352.01

Table 2: *Test results for various combinations of test scenes and subdivision method.* max surfaces *is the maximum amount of surfaces stored in memory at any given point in time.* processed *is the total number of surfaces processed during subdivision including intermediate surfaces. The processing rate is the number of processed surfaces divided by the subdivision time.*

visible shift by about 20 MiB to the right. The constant offsets of the other scenes are relatively small with at most 0.27 MiB for COLUMNS.

Exact performance comparisons against previous implementations are difficult because of different rendering parameters, but our overall performance appears competitive modulo differences in hardware and rendering parameters:

- Patney and Owens [2008] give times for the adaptive subdivision of TEAPOT (6.99 ms) and KILLEROO (3.46 ms). They perform fewer split operations (512×512 resolution with a 16-pixel bound) and use a significantly less powerful NVIDIA GeForce 8800 GTX for measurement. Under this configuration our subdivision times are 1.43 ms for TEAPOT and 0.30 ms for KILLEROO with BREADTH. The subdivision times for BOUNDED are essentially the same.

- Tzeng et al. [2010] give overall frame render times including shading and rasterization for TEAPOT (51.81 ms), BIGGUY (90.50 ms), and KILLEROO (54.11). They render at resolution 800×800 and use a 16-pixel bound. Micropolis is considerably faster in this configuration (TEAPOT: 3.08 ms, BIGGUY: 3.11 ms, KILLEROO: 5.94 ms). However this is once again hard to compare since Tzeng et al.'s renderer uses complex transparency and 16× multisampling.

5 Conclusion and Future Work

This paper has presented a method for implementing adaptive surface subdivision on the GPU with a bounded peak memory consumption. The output order of generated surfaces also preserves locality. We believe the memory advantages of our algorithm over previous GPU implementations of bound-and-split may make adaptive surface subdivision more tractable for real-time usage, in particular for constrained rendering environments like mobile platforms.

One aspect not discussed so far is the best choice for the batch size. As can be seen in Figures 6 and 7, this is highly dependent on the chosen scene and viewpoint. One possible heuristic for this would be counting the intermediate surfaces per subdivision level to estimate the necessary breadth-first batch size and runtime behavior. The preceding frame could be used for this.

The performance of BOUNDED could be greatly improved by using device-side enqueue, as supported in version OpenCL 2.0. This is because a lot of the overhead of performing more iterations comes from the necessary host-device interactions. If this overhead were negligible, then even relatively small batch sizes should be able to fully utilize all available parallelism for a given graphics processor. However, AMD only released a preliminary driver supporting OpenCL 2.0 just weeks before submission of this paper, which is why we weren't able to fully explore this.

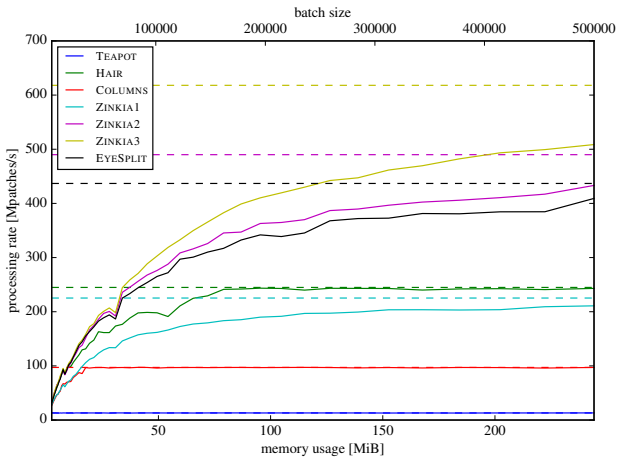

Figure 7: *Subdivision performance for our test scenes depending on the amount of assigned memory and batch size. The X axis shows the amount of used memory on the bottom axis and the batch size on top. Smaller scenes very quickly level out, while larger scenes show asymptotic growth. The dashed horizontal lines represent the processing rate achievable by* BREADTH *and the upper bound for* BOUNDED.

Robust adaptive subdivision has many possible uses beyond the classic Reyes algorithm. Hanika et al. [2010] presented a method for ray-tracing polygons using a two-level approach with ray reordering. This method may be well-suited for implementation on the GPU using our described method for geometry generation. Integrating adaptive subdivision into a larger GPU graphics pipeline would also allow for interesting optimization possibilities like culling occluded surfaces during subdivision.

The source code for *Micropolis*, the OpenCL Reyes renderer described in this paper, can be found at `https://github.com/ginkgo/micropolis`.

6 Acknowledgments

We'd like to thank Anjul Patney, Stanley Tzeng, Julian Fong, and Tim Foley for their valuable input. Another "thank you" goes to Nuwan Jayasena of AMD for supplying us with testing harware and giving support on driver issues.

This paper was supported by a scholarship from the Austrian Marshall Plan Foundation, by a generous gift from AMD, by National Science Foundation Award CCF-1017399, and by the Intel Science and Technology Center for Visual Computing.

References

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

EISENACHER, C., MEYER, Q., AND LOOP, C. 2009. Real-time view-dependent rendering of parametric surfaces. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, 137–143.

FISHER, M., FATAHALIAN, K., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. DiagSplit: Parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics 28*, 5 (Dec.), 150:1–150:10.

HANIKA, J., KELLER, A., AND LENSCH, H. P. A. 2010. Two-level ray tracing with reordering for highly complex scenes. In *Proceedings of Graphics Interface 2010*, GI '10, 145–152.

HOU, Q., SUN, X., ZHOU, K., LAUTERBACH, C., AND MANOCHA, D. 2011. Memory-scalable GPU spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics 17*, 4 (Apr.), 466–474.

LOOP, C., AND EISENACHER, C. 2009. Real-time patch-based sort-middle rendering on massively parallel hardware. Tech. Rep. MSR-TR-2009-83, Microsoft Research, May.

LOOP, C., AND SCHAEFER, S. 2008. Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Transactions on Graphics 27*, 1 (Mar.), 8:1–8:11.

MUNKBERG, J., HASSELGREN, J., TOTH, R., AND AKENINE-MÖLLER, T. 2010. Efficient bounding of displaced Bézier patches. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, 153–162.

NIESSNER, M., AND LOOP, C. 2013. Analytic displacement mapping using hardware tessellation. *ACM Transactions on Graphics 32*, 3 (July), 26:1–26:9.

NIESSNER, M., LOOP, C., MEYER, M., AND DEROSE, T. 2012. Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces. *ACM Transactions on Graphics 31*, 1 (Feb.), 6:1–6:11.

NIESSNER, M., LOOP, C. T., AND GREINER, G. 2012. Efficient evaluation of semi-smooth creases in Catmull-Clark subdivision surfaces. In *Eurographics (Short Papers)*, 41–44.

PATNEY, A., AND OWENS, J. D. 2008. Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics 27*, 5 (Dec.), 143:1–143:8.

PATNEY, A., EBEIDA, M. S., AND OWENS, J. D. 2009. Parallel view-dependent tessellation of Catmull-Clark subdivision surfaces. In *Proceedings of the Conference on High Performance Graphics*, HPG '09, 99–108.

SANCHEZ, D., LO, D., YOO, R. M., SUGERMAN, J., AND KOZYRAKIS, C. 2011. Dynamic fine-grain scheduling of pipeline parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 22–32.

TZENG, S., PATNEY, A., AND OWENS, J. D. 2010. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, 29–37.

ZHOU, K., HOU, Q., REN, Z., GONG, M., SUN, X., AND GUO, B. 2009. RenderAnts: Interactive Reyes rendering on GPUs. *ACM Transactions on Graphics 28*, 5 (Dec.), 155:1–155:11.