

# Memory-efficient Adaptive Subdivision for Software Rendering on the GPU

Marshall Plan Scholarship End Report

Thomas Weber  
Vienna University of Technology

April 30, 2014

## **Abstract**

The adaptive subdivision step for surface tessellation is a key component of the Reyes rendering pipeline. While this operation has been successfully parallelized for execution on the GPU using a breadth-first traversal, the resulting implementations are limited by their high worst-case memory consumption and high global memory bandwidth utilization. This report proposes an alternate strategy that allows limiting the amount of necessary memory by controlling the number of assigned worker threads. This makes it possible to severely reduce the amount of necessary memory and place all intermediate data in work-group local memory. The result is an implementation that scales to the performance of the breadth-first approach while offering three new advantages: significantly decreased memory usage, a smooth and predictable tradeoff between memory usage and performance, and increased locality for patch processing.

The implementation and evaluation of this work has been performed at University of California, Davis. It was funded through a scholarship from the Austrian Marshall Plan Foundation.

# 1 Introduction

The steady increase 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 [1], which is commonly used in 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. It is therefore important that all components of a Reyes pipeline can guarantee a peak memory bound.

In this report, we present a method for performing high-performance adaptive surface subdivision in parallel on the GPU while still guaranteeing a constant memory consumption. Our approach preserves locality and allows us to store intermediate surface data in work-group local memory.

This work is the result of a cooperation of the Vienna University of Technology and University of California, Davis. The implementation and evaluation phase were done at the UC Davis Department of Electrical and Computer Engineering under supervision of professor John Owens. The funding for this research stay was provided by the Austrian Marshall Plan Foundation.

## 2 Background & Previous Work

Reyes tessellates surfaces into micropolygons using a two-stage approach [1]. 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. 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 [3].

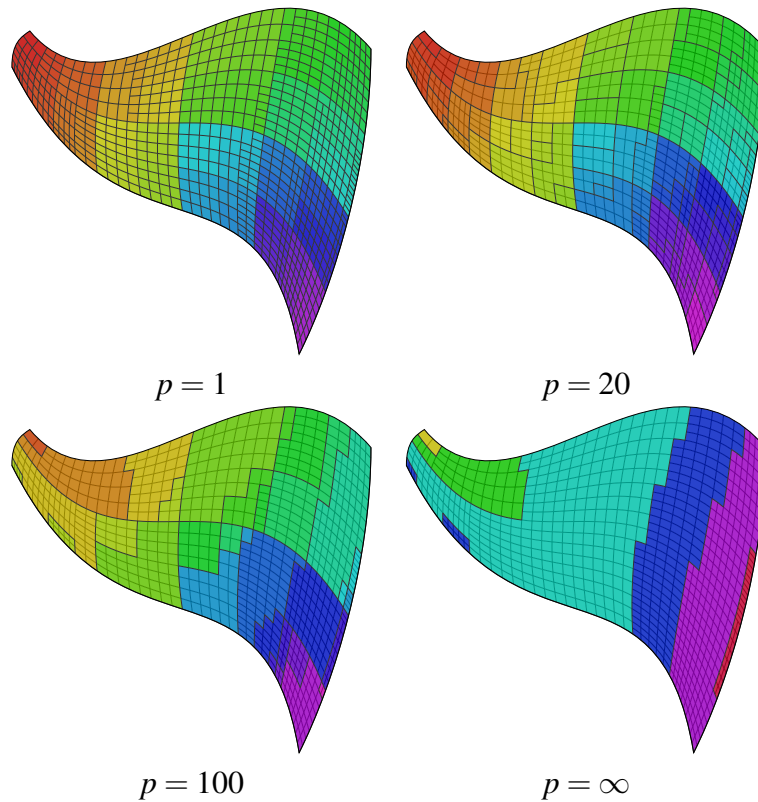


Figure 1: Comparison of evaluation order of patches for different batch sizes. Surfaces that are created in the same iteration are shaded in the same color. This shows the locality-preserving property of our subdivision algorithm: surfaces that are spatially close together are evaluated in the same iteration.

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. While doing full subdivision up to the micropolygon level is possible, this leads to unnecessary over-tessellation since surfaces can only be halved. 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.

The dicing phase maps to hardware tessellation supported on recent graphics APIs and GPUs [8]. 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.

**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 transformed the typical depth-first recursive traversal of split patches into a breadth-first operation [12]. 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 [3, 7, 14]. Nevertheless several papers build on this method.

Zhou et al. use breadth-first adaptive subdivision as part of a full GPU-based interactive Reyes renderer called *RenderAnts* [14]. *RenderAnts* uses dynamic scheduling to ensure bounded memory usage for fragment processing. However, no such bound is given for adaptive subdivision. Patney et al. use the breadth-first approach for crack-free view-dependent tessellation of Catmull-Clark subdivision surfaces [11], and Eisenacher et al. adopt the same breadth-first approach for parametric patch subdivision, but also consider surface curvature, resulting in considerably fewer patches being created [2].

Fisher et al. present a method for efficiently avoiding surface cracks during subdivision by using the tessellation scheme used in hardware tessellation [3]. 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. consider adaptive subdivision from a scheduling point of view [13]. 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. [9]. 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 work, they discuss how semi-sharp creases can be handled efficiently [10]. 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 [5]. 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.

Patney and Owens [12] parallelize Reyes’s 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 patches have been successfully bounded. Figure 2 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. Due to this exponential growth in memory consumption, the static preallocation of memory for this operation quickly becomes unfeasible.

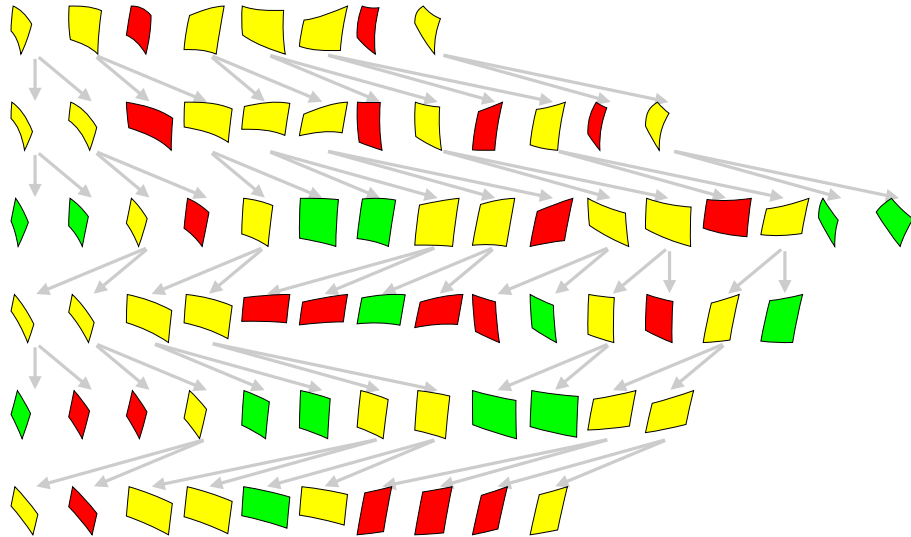


Figure 2: Schematic overview of breadth-first subdivision. Each row represents the state of the surface buffer during one iteration. Each patch can either be culled (red), split (yellow), or drawn (green). For each split patch in the previous iteration, two new patches are generated in the following iteration. This always happens for all patches in the surface buffer.

### 3.1 Adaptive Subdivision with Bounded Memory

Instead, we propose an adaption 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 3 illustrates how this approach works, and Section 3.2 proves this bound.

Adding the batch size  $p$  as a tweakable parameter in the subdivision process allows us to balance between memory consumption and performance. Figure ?? in the Results section 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 1.

In our implementation, a *bound* kernel first copies the last  $p$  surfaces into a

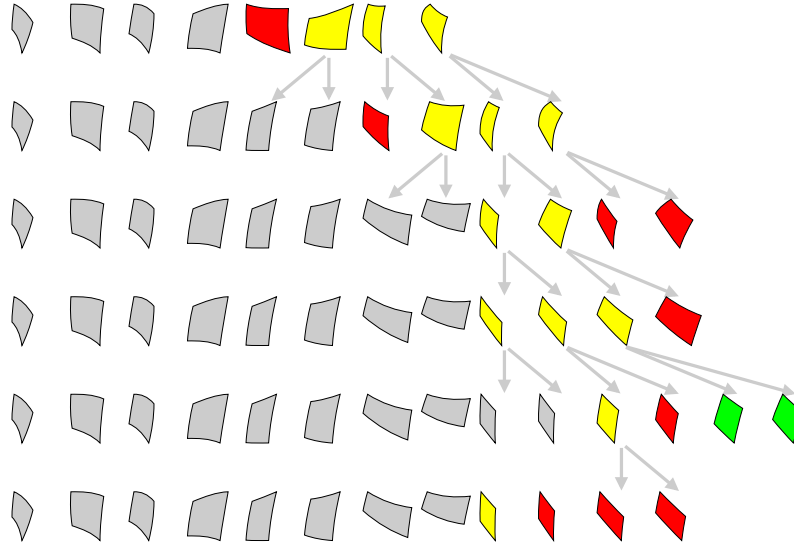


Figure 3: Schematic overview of how our memory-bounded subdivision operates. Unlike in figure 2, the number of active surfaces at each iteration is constant (in this case,  $p = 4$ ). The other surfaces are inactive and shaded in gray.

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.

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. 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. The accumulated flags from the prefix sum are used to find the correct location in the output and surface buffers.

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.

Keeping the children of a surface that has been split close together also improves locality. Figure 4 shows the difference between placing the sub-surfaces

in the order described by Patney and Owens [12] (NONINTERLEAVED) with our approach (INTERLEAVED).

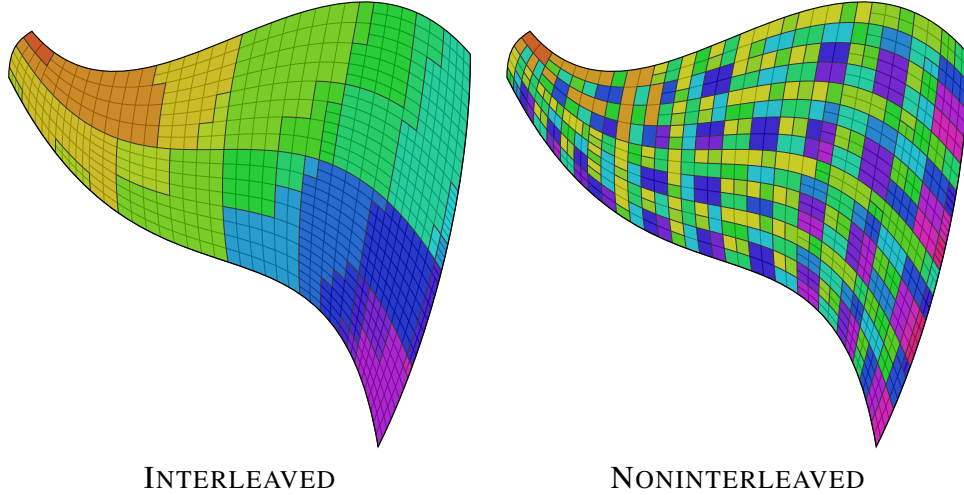


Figure 4: Illustration of the effect the placement order after split has on the locality of generated patches. Surfaces created during the same iteration share the same color. INTERLEAVED is the order described in this section while NONINTERLEAVED uses the Patney-Owens order [12].

These operations are then repeated until the surface buffer is empty.

### 3.2 Proof: Peak Memory Bound

To prove our asserted peak memory consumption of  $O(N + p \cdot k)$ , we consider the state of the ordered surface buffer  $S_t = \{s_0, \dots, s_n\}$  at every iteration  $t \in \mathbb{N}$ . If we can show that

$$\forall t \in \mathbb{N}_0: \|S_t\| < N + p \cdot k \quad (1)$$

then the general memory consumption must also be bounded, since the size of all other buffers apart from  $S_t$  remains constant throughout execution.

The split level  $d: S_t \rightarrow \mathbb{N}_0$  gives the number of subdivisions that have so far been applied to a given surface. For example, for a fresh input surface  $s$ ,  $d(s) = 0$ . If we split this surface into two sub-surfaces  $s_l$  and  $s_r$ , then  $d(s_l) = d(s_r) = 1$ . If we split  $s_l$  further into  $s_{ll}$  and  $s_{lr}$ , then  $d(s_{ll}) = d(s_{lr}) = 2$  and so on. We also define the operator  $D_i(S_t) = \|\{s \in S_t | d(s) = i\}\|$ , which gives the number of surfaces in  $S_t$  that have subdivision level  $i$ .



We will prove this memory bound via induction. To do so we will show that the following invariants are fulfilled at every iteration  $t$ :

$$\forall s \in S_t: d(s) < k \quad (2)$$

$$\forall s_i, s_j \in S_t, i < j: d(s_i) \leq d(s_j) \quad (3)$$

$$D_0(S_t) \leq N \quad (4)$$

$$\exists m \in \mathbb{N}_1: \forall i \in \mathbb{N}_1, i \neq m: D_i(S_t) \leq p, \quad (5)$$

$$D_m(S_t) \leq 2p - \sum_{j>k} D_j(S_t)$$

(2) asserts that no surface in the surface buffer may have been divided more than  $k$  times. This can be ensured by always culling or drawing surfaces that have already received  $k - 1$  subdivisions. (3) requires that the surfaces in the buffer are ordered by their subdivision level. (4) ensures that there can never be more than the initial number of surfaces with subdivision level 0 in  $S$ .

Invariant (5) is the most important one. It requires that there always exists one positive subdivision level  $m$  so that all other positive subdivision levels  $i \neq m$  must occur at most  $p$  times in  $S$ .  $m$  may occur up to  $2p$  times, but only if there exist no surfaces with a higher subdivision level in the buffer. Otherwise the number of these is subtracted from  $2p$  to give the number of allowed occurrences of  $m$ .

The memory invariant (1) must be fulfilled when (2), (4), and (5) are fulfilled, since there can be at most  $k$  subdivision levels which occur at least once; among those, one can occur at most  $N$  times, one can occur at most  $2p$  times, and the rest can occur at most  $p$  times. It should be easy to see that all invariants are fulfilled for  $S_0$ .

Now we take a look at an arbitrary iteration  $t$  so that  $S_t$  fulfills all invariants. When we take the last  $p$  items from  $S_t$ , potentially split them if they have a subdivision level below  $k + 1$ , and append the split patches back at the end of the surface buffer, we get  $S_{t+1}$ , the state of the surface buffer in the next iteration.

We can be sure that (2) is fulfilled for  $S_{t+1}$  since we take special care to never split surfaces that have already been split  $k + 1$  times. Similarly, (4) should be easy to see, since we can only increase the subdivision level of surfaces, so  $D_0(S_{t+1})$  can only be smaller than  $D_0(S_t)$ .

Since we know that  $S_t$  is sorted by subdivision level, we know that the  $p$  surfaces we took for subdivision must have the maximal amount of subdivisions for  $S_t$ . By splitting them, we can only increase this, so the split surfaces we put back at the end are guaranteed to be larger than all of the remaining ones. And thanks to

the specific order described in section 3.1, we can ensure that if two split surfaces are in a specific order relative to each other, then the split products must also be in that order, only with their subdivision level incremented by one. Thus (3) is fulfilled for  $S_{t+1}$ .

To see that (5) is fulfilled, we must consider the special subdivision level  $m$  for  $S_t$ . The range of surfaces with subdivision level  $m$  can start at most  $2p$  elements away from the end of the buffer. At most  $p$  surfaces of this range can be outside the last  $p$  elements of the buffer. If we now remove the last  $p$  surfaces  $P$  and split them in any way, the resulting subsurfaces must have a subdivision level greater than  $m$ . This means that the number of remaining surfaces with tessellation level  $m$  must now be  $\leq p$ .  $m$  is no longer a special depth in  $S_{t+1}$ .

What remains to be seen is that there can be at most one new level  $m'$  for which there are more than  $p$  surfaces in  $S_{t+1}$ , and that it must be at the end of the buffer. Since what remains of the old level  $m$  has  $\leq p$  surfaces after the iteration, and the other surfaces of  $S_t$  are known to have  $\leq p$ , the new  $m'$  can only belong to the surfaces created from  $P$ . For any combination of non-overlapping ranges in  $P$ , there can be at most one subset that contains more than  $p/2$  surfaces. If this range is  $x$  surfaces away from the end, then it can contain at most  $p - x$  surfaces. If we now subdivide this range and maintain the order of the split products for this range, they must form the subdivision level  $m'$ , fulfilling invariant (5).  $\square$

### 3.3 Storing Intermediate Surfaces in Work-Group Local Storage

Section 3.1 describes an implementation that stores all surfaces (including intermediate surfaces) in GPU global memory. We can improve the efficiency of our implementation by storing intermediate surfaces in work-group local memory so that all coordination of individual threads can be done on the work-group level.

Instead of having a single surface buffer in global memory, each work-group keeps its own surface buffer in work-group local memory. Single iterations of the subdivision algorithm described in section 3.1 are performed in a loop within the work-group. Communication and flow control of threads within a work-group is done using local memory, work-group-local prefix sums, and barriers. Surfaces that have been successfully bounded are transferred to an output buffer in global memory.

With this approach, no explicit host intervention is necessary to start another iteration of the subdivision algorithm, thus global memory bandwidth is reduced.

Instead of having to read and write surface data to and from global memory, the only times when surfaces need to be transferred out of local memory is during the initial reading of input surfaces and when writing the final bounded surfaces to the destination buffer.

We can deterministically store the entire intermediate surface buffer in work-group local memory, since the amount of necessary memory for this derives from our memory bound  $O(p \cdot k)$ , where  $p$  is the work-group size of this kernel. In practice, this should be set to the native SIMD width.

As an example, for recent AMD GPUs, this is 64. Let's say we allow a maximum subdivision level of 15. We would have to allocate enough shared memory to store  $64 \cdot 15 = 960$  surfaces. If one surface requires 24 B of storage, the total amount of necessary shared memory would be just under 23 KiB. If this exceeds the amount of available work-group local memory, then a small amount of global memory can be allocated for spill buffers.

Our implementation uses a single global input queue to store work and performs well for our test cases. While no load-balancing is necessary between threads within a work-group, it is still possible that some work-groups run out of work more quickly than others. In this case, a load balancing strategy similar to the one described by Tzeng et al. [13] could be used, but this has not yet been necessary in our implementation.

If the output buffer is full and there is still work to be done, then the kernel must stop operation and return control back to the host so it can render the generated patches. The content of the work-group local surface buffer needs to be backed up to global memory so that it can be recovered once operation is resumed.

## 4 Results

We have implemented a simple Reyes renderer in OpenCL called *Micropolis* that implements adaptive subdivision, dicing, shading, and micropolygon sampling as kernels on the GPU.

The renderer supports several different methods for adaptive subdivision for comparison. BREADTH uses the breadth-first approach of Patney and Owens [12]. BOUNDED is our base implementation with bounded memory. LOCAL is our variant that stores intermediate patches in work-group local memory.

In case BREADTH runs out of memory, it allocates 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 work. The necessary time-overhead for this is not part of the measured subdivision times because we allow for a certain number of rendered frames before measurement.




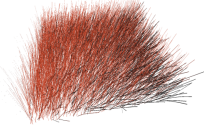
				
	TEAPOT	BIGGUY	KILLEROO	HAIR
$N_{in}$	32	3 570	11 532	10 000
$N_{out}$	25 234	39 691	33 945	304 474
$S$	25 202	36 121	22 413	294 474

Table 1: Overview of the different test scenes used for performance analysis.  $N_{in}$  and  $N_{out}$  are the numbers of surface patches before and after adaptive subdivision.  $S$  is the number of surface splits performed during subdivision. The Big Guy model was created by Bay Raitt. Killeroo NURBS model supplied by headus 3D tools.

Table 1 shows the models we used for evaluating our renderer. The model TEAPOT has a small number of very large patches. HAIR has a large amount of very small patches. BIGGUY and KILLEROO are in between in terms of patch count and size.

All benchmarks have been measured on a system with an AMD Radeon HD 7970 GPU and a 2.6GHz Intel Core i7 920 CPU. The graphics driver used was Catalyst 14.3.

Table 2 lists the execution times for various combinations of adaptive subdivision methods and test models. The scenes are rendered at a resolution of  $1280 \times 1024$  and surfaces are split until they are smaller than 8 pixels along each dimension. Table 1 shows the number of input and output patches as well as the number of performed split operations for each model. The chosen batch size  $p$  for BOUNDED is 10000. For LOCAL 32 work-groups of size 64 are used. This is the number and size of the multiprocessors available in the HD 7970. The maximum number of recursive subdivisions  $k$  has been set to 15.

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:

	BREADTH	BOUNDED	LOCAL
TEAPOT	3.89	3.89	3.69
BIGGUY	2.87	3.75	2.21
KILLEROO	2.41	2.70	1.87
HAIR	3.73	16.72	10.60

Table 2: Execution times of adaptive subdivision for different algorithms and data sets. All values are in milliseconds. The batch size for BOUNDED is 10000.

- Patney and Owens give times for the adaptive subdivision of TEAPOT (6.99 ms) and KILLEROO (3.46 ms) [12]. 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.
- Tzeng et al. [13] give overall frame render times 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 with that configuration (TEAPOT: 5.88 ms, BIGGUY: 5.54 ms, KILLEROO: 5.94), although this is once again hard to compare since Tzeng et al.’s renderer uses complex transparency and 16× multisampling.

BOUNDED performs reasonably well compared to BREADTH for all scenes except HAIR. Figure 5 and figure 6 show the impact the chosen batch size has on the subdivision performance of BOUNDED when being applied to the models HAIR and TEAPOT respectively. To achieve performance similar to BREADTH for HAIR, a batch size of about 250000 would need to be chosen. This would require about 85 MiB of memory. But even 20 MiB of assigned memory gives a reasonable speedup.

The memory consumption of BOUNDED with a batch size of 10000 is about 3 MiB. By comparison, the worst-case memory requirement for storing all patches with breadth-first subdivision for a scene the size of HAIR would be  $2^{15} \cdot 10000 \cdot 24 \text{ B} = 7.3 \text{ GiB}$  of memory. This is just for one of the two necessary ping-pong buffers. Additional buffers of the same dimension are necessary for flags and prefix sums.

The performance of LOCAL generally exceeds both BOUNDED and BREADTH for most scenes, despite only requiring about 1 MiB of global memory. The amount of necessary shared memory per work-group is about 20 KiB, well within the capabilities of recent GPUs.

Figure 7 shows the impact the number of assigned work-groups has on the subdivision time of HAIR with LOCAL. The reason for the dip in performance just under the 100 work group mark isn't entirely clear. We believe that this is due to a change in how the graphics driver schedules work groups.

LOCAL's chief performance advantage stems from reduced interactions with the host: both BREADTH and BOUNDED incur significant overhead in enqueueing new iterations. The only time LOCAL needs to return to the host is when the output buffer for fully bounded surfaces has filled up and needs processing. In our case, this happens every 10000 patches, so about 3–4 times in total for TEAPOT, BIGGUY, and KILLEROO. HAIR requires 31 iterations of LOCAL, which is the reason why HAIR performs worse on LOCAL than on BREADTH.

If LOCAL didn't have to return control to the host whenever the output buffer is full, it could subdivide HAIR within 4.07 ms. This is close to the performance of BREADTH. We believe the reason why BREADTH is slightly faster on HAIR is because the GPU's hardware scheduler is doing a better job at occupying cores and hiding memory latency, as it has more in-flight threads than in our persistent kernel approach [6].

We could most easily improve the performance of all presented methods by avoiding a return to the host for enqueueing additional kernels. In particular, BOUNDED would benefit if the overhead of additional iterations were negligible. This is because for a GPU that can keep 20480 concurrent threads in flight, it should make no difference whether it performs five consecutive 40000 item kernels or one 200000 item kernel of the same operation.

The upcoming OpenCL version 2.0 supports device-side enqueueing of kernels, which should allow reducing the overhead per iteration significantly. However there were no available implementations for OpenCL 2.0 on the GPU at the time of writing. Another approach that should work with OpenCL 1.2 hardware would be to enqueue consecutive iterations speculatively.

## 5 Conclusion and Future Work

This report 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. Our highest-performance implementation stores intermediate patches in work-group local memory and uses persistent kernels for control flow. With it, we achieve speedups over the traditional approach with bounded (and substantially less) memory usage. We believe the performance

and 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.

The performance of our iterative version could be greatly improved by using device-side enqueue, as supported in the upcoming version 2.0 of OpenCL. This might be competitive with the work-group local version. Integrating adaptive subdivision in a larger GPU graphics pipeline would also allow for interesting optimizations that cull occluded surfaces during subdivision.

Work-group local subdivision would be very well suited as a component in a concurrent producer-consumer pipeline. Instead of having to abort and recover when the output buffer is full, the subdivision kernel can just idle until the pipeline consuming stage is available again. Since later stages like shading and sampling usually require considerably more computing time, even a small fraction of the hardware resources dedicated for subdivision should be enough to keep the pipeline going without it becoming the bottleneck.

Robust adaptive subdivision has many possible uses. Hanika et al. present a method for ray-tracing polygons using a two-level approach with ray reordering [4]. This method may be well-suited for implementation on the GPU using our described method for geometry generation.

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

## 6 Acknowledgments

We would like to thank Anjul Patney, Stanley Tzeng, and Tim Foley for giving valuable input on adaptive subdivision and efficient task scheduling on the GPU. Another thank you goes to Nuwan Jayasena of AMD for providing us with testing hardware and support. This work was made possible thanks to a generous scholarship from the Austrian Marshall Plan Foundation.

## References

- [1] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, pages 95–102, July 1987.
- [2] Christian Eisenacher, Quirin Meyer, and Charles Loop. Real-time view-dependent rendering of parametric surfaces. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, I3D '09*, pages 137–143, 2009.
- [3] Matthew Fisher, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, William R. Mark, and Pat Hanrahan. DiagSplit: Parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics*, 28(5):150:1–150:10, December 2009.
- [4] Johannes Hanika, Alexander Keller, and Hendrik P. A. Lensch. Two-level ray tracing with reordering for highly complex scenes. In *Proceedings of Graphics Interface 2010, GI '10*, pages 145–152, 2010.
- [5] Qiming Hou, Xin Sun, Kun Zhou, Christian Lauterbach, and Dinesh Manocha. Memory-scalable GPU spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):466–474, April 2011.
- [6] Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference, HPG '13*, pages 137–143, 2013.
- [7] Charles Loop and Christian Eisenacher. Real-time patch-based sort-middle rendering on massively parallel hardware. Technical Report MSR-TR-2009-83, Microsoft Research, May 2009.
- [8] Charles Loop and Scott Schaefer. Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Transactions on Graphics*, 27(1):8:1–8:11, March 2008.
- [9] Matthias Nießner, Charles Loop, Mark Meyer, and Tony Deroose. Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces. *ACM Transactions on Graphics*, 31(1):6:1–6:11, February 2012.



- [10] Matthias Nießner, Charles T. Loop, and Günther Greiner. Efficient evaluation of semi-smooth creases in Catmull-Clark subdivision surfaces. In *Eurographics (Short Papers)*, pages 41–44, 2012.
- [11] Anjul Patney, Mohamed S. Ebeida, and John D. Owens. Parallel view-dependent tessellation of Catmull-Clark subdivision surfaces. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 99–108, 2009.
- [12] Anjul Patney and John D. Owens. Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics*, 27(5):143:1–143:8, December 2008.
- [13] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 29–37, 2010.
- [14] Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. RenderAnts: Interactive Reyes rendering on GPUs. *ACM Transactions on Graphics*, 28(5):155:1–155:11, December 2009.

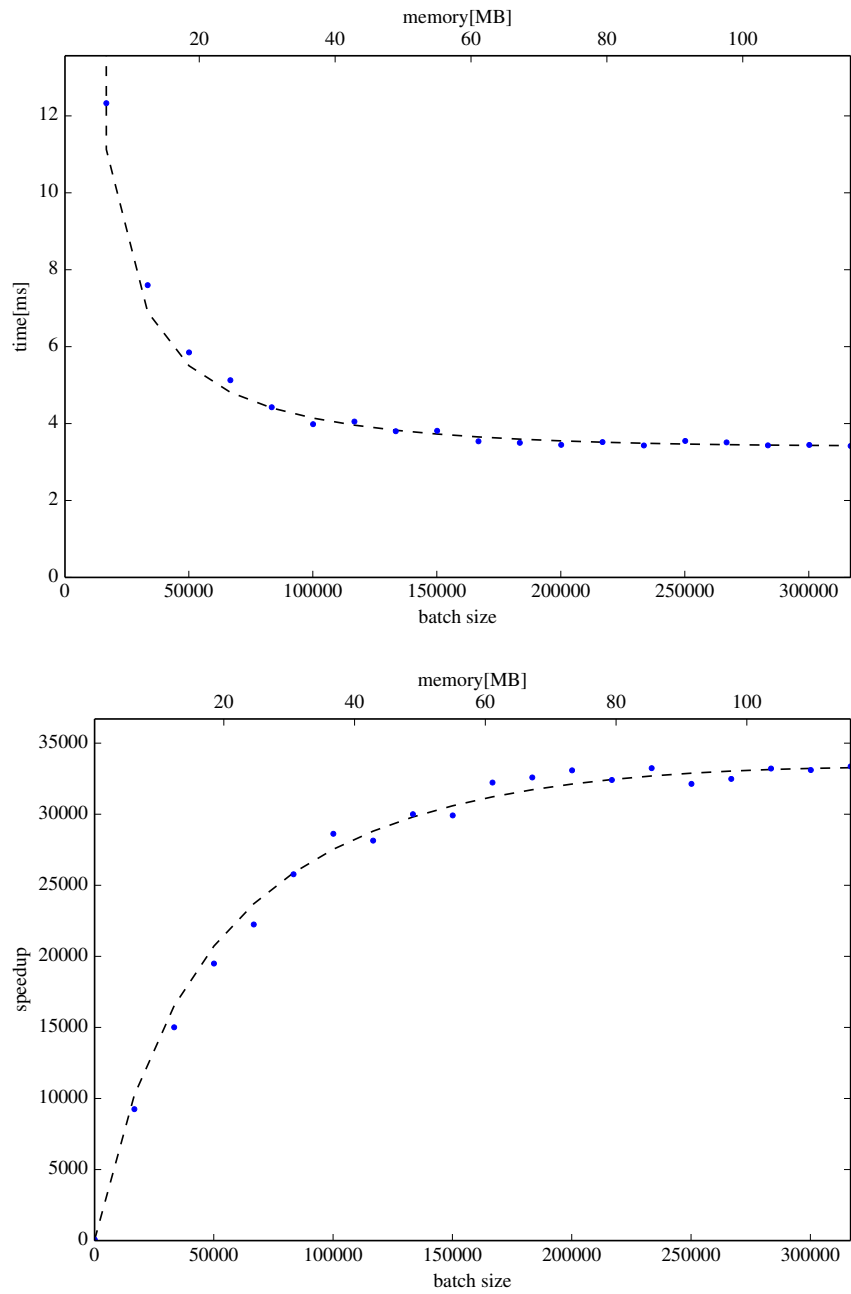


Figure 5: Achievable subdivision time and speedup for algorithm BOUNDED depending on number of assigned processors and memory. The data set used for testing was HAIR.

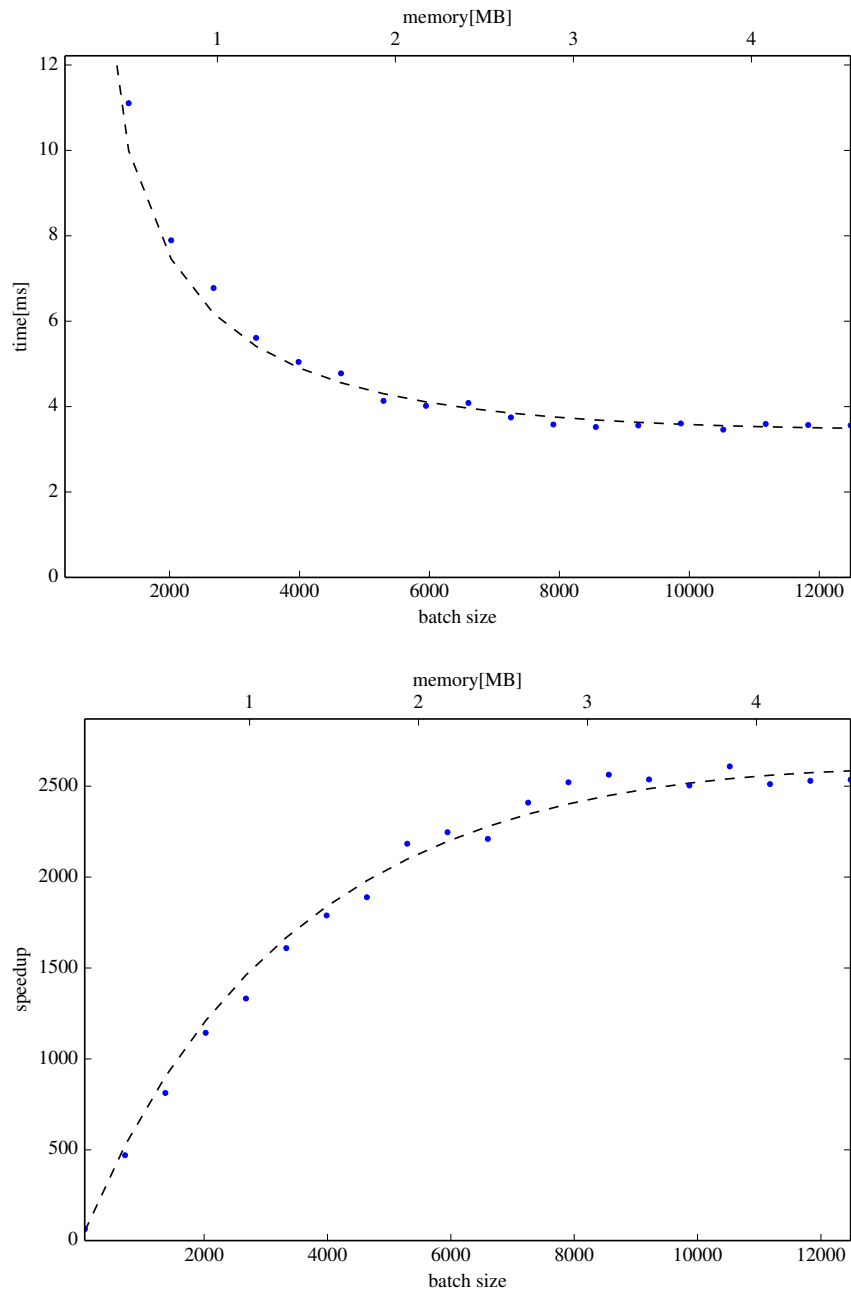


Figure 6: Achievable subdivision time and speedup for algorithm BOUNDED depending on number of assigned processors and memory. The data set used for testing was TEAPOT.

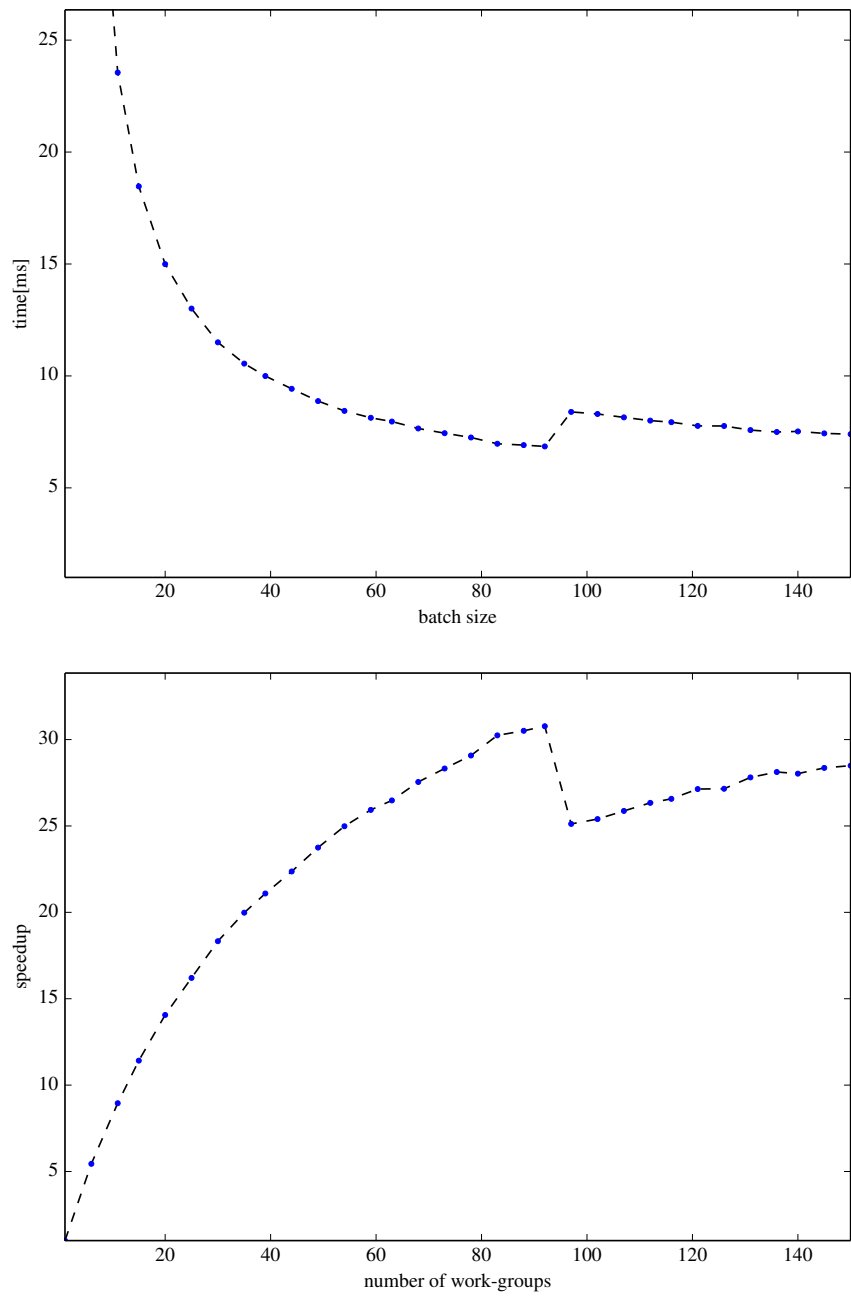


Figure 7: Achievable subdivision time and speedup for algorithm BOUNDED depending on number of assigned work groups. The data set used for testing was HAIR.