A High-Performance Software Graphics Pipeline Architecture for the GPU

MICHAEL KENZEL, BERNHARD KERBL, and DIETER SCHMALSTIEG, Graz University of Technology, Austria
MARKUS STEINBERGER, Graz University of Technology, Austria and Max Planck Institute for Informatics, Germany

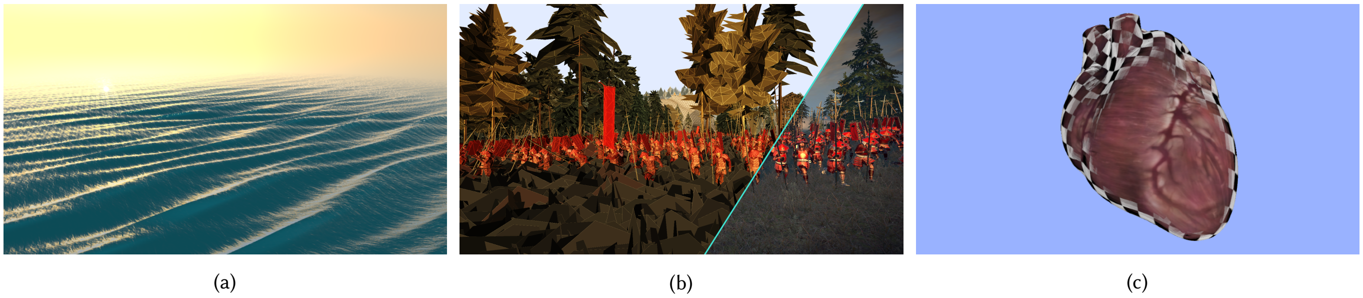


Fig. 1. Various scenes rendered by our software graphics pipeline in real-time on a GPU. (a) A smooth triangulation of the water surface in an animated ocean scene is achieved via a custom pipeline extension that allows the mesh topology to dynamically adapt to the underlying heightfield. (b) Scene geometry captured from video games like this still frame from *Total War: Shogun 2* is used to evaluate the performance of our approach on real-world triangle distributions. (c) Many techniques such as mipmapping rely on the ability to compute screen-space derivatives during fragment shading. Our pipeline architecture can support derivative estimation based on pixel quad shading, used here to render a textured model of a heart with trilinear filtering; lower mipmap levels are filled with a checkerboard pattern to visualize the effect. *Total War: Shogun 2 screenshot courtesy of The Creative Assembly; used with permission.*

In this paper, we present a real-time graphics pipeline implemented entirely in software on a modern GPU. As opposed to previous work, our approach features a fully-concurrent, multi-stage, streaming design with dynamic load balancing, capable of operating efficiently within bounded memory. We address issues such as primitive order, vertex reuse, and screen-space derivatives of dependent variables, which are essential to real-world applications, but have largely been ignored by comparable work in the past. The power of a software approach lies in the ability to tailor the graphics pipeline to any given application. In exploration of this potential, we design and implement four novel pipeline modifications. Evaluation of the performance of our approach on more than 100 real-world scenes collected from video games shows rendering speeds within one order of magnitude of the hardware graphics pipeline as well as significant improvements over previous work, not only in terms of capabilities and performance, but also robustness.

CCS Concepts: • **Computing methodologies → Rasterization**; **Graphics processors**; *Massively parallel algorithms*;

Additional Key Words and Phrases: Software Rendering, GPU, Graphics Pipeline, Rasterization, CUDA

ACM Reference Format:
Michael Kenzel, Bernhard Kerbl, Dieter Schmalstieg, and Markus Steinberger. 2018. A High-Performance Software Graphics Pipeline Architecture for the GPU. *ACM Trans. Graph.* 37, 4, Article 140 (August 2018), 15 pages. https://doi.org/10.1145/3197517.3201374

1 INTRODUCTION

For a long time now, the hardware graphics pipeline has been the backbone of real-time rendering. However, while a hardware implementation can achieve high performance and power efficiency, flexibility is sacrificed. Driven by the need to support an ever growing spectrum of ever more sophisticated applications, the *graphics processing unit* (GPU) evolved as a tight compromise between flexibility and performance. The graphics pipeline on a modern GPU is implemented by special-purpose hardware on top of a large, freely-programmable, massively-parallel processor. More and more programmable stages have been added over the years. However, the overall structure of the pipeline and the underlying rendering algorithm have essentially remained unchanged for decades.

While evolution of the graphics pipeline proceeds slowly, GPU compute power continues to increase exponentially. In addition to the graphics pipeline, modern *application programming interfaces* (API) such as Vulkan [Khronos 2016b], OpenGL [Khronos 2016a], or Direct3D [Blythe 2006], as well as specialized interfaces like CUDA [NVIDIA 2016] and OpenCL [Stone et al. 2010] also allow the GPU to be operated in *compute mode*, which exposes the programmable cores of the GPU as a massively-parallel general-purpose co-processor. Although the hardware graphics pipeline remains at the core of real-time rendering, cutting-edge graphics applications increasingly rely on compute mode to implement major parts of sophisticated graphics algorithms that would not easily map to the traditional graphics pipeline such as, e.g., tiled deferred rendering [Andersson 2009], geometry processing (cloth simulation) [Vaisse 2014], or texel shading [Hillesland and Yang 2016].

Authors' addresses: Michael Kenzel, michael.kenzel@icg.tugraz.at; Bernhard Kerbl, bernhard.kerbl@icg.tugraz.at; Dieter Schmalstieg, dieter.schmalstieg@icg.tugraz.at, Graz University of Technology, Institute of Computer Graphics and Vision, Inffeldgasse 16, Graz, 8010, Austria; Markus Steinberger, markus.steinberger@icg.tugraz.at, Graz University of Technology, Institute of Computer Graphics and Vision, Inffeldgasse 16, Graz, 8010, Austria, Max Planck Institute for Informatics, Saarland Informatics Campus Building E1 4, Saarbrücken, 66123, Germany.

Observing this trend of rapidly growing compute performance and more and more complex rendering work being pushed to compute mode, we ask the question: What level of performance could be achieved by a software graphics pipeline on a current GPU?

As an alternative to a hardware pipeline, software pipelines can provide similar or identical feature sets, while inherently supporting full programmability. For systems lacking a dedicated GPU, CPU implementations, such as the widespread Mesa3D library [1993] or the highly-optimized (now discontinued) Pixomatic SDK [RAD 2002] have served as a fallback as well as a secondary rendering engine, for example, for visibility culling [Andersson 2009].

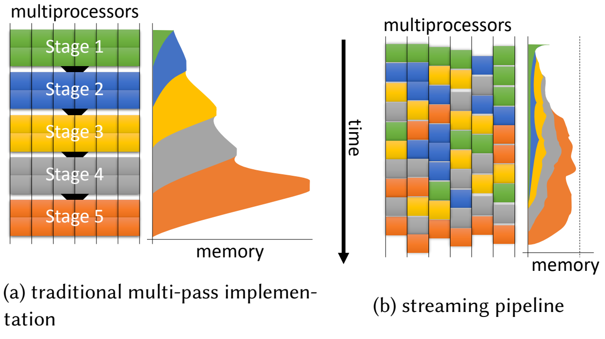
We are not the first to raise the question of a software graphics pipeline for the GPU. Previous work such as CUDARaster [Laine and Karras 2011] has demonstrated that software approaches can achieve high-performance. However, to the best of our knowledge, our CUDA Rendering Engine (cuRE) is the first end-to-end Direct3D9-class software graphics pipeline architecture able to deliver real-time rendering performance for real-world scenes on a current GPU. Unlike previous approaches, it encompasses both geometry processing and rasterization in a *streaming fashion* and operates within *bounded memory* while preserving *primitive order* until the blending stage.

The main contribution of this work is a design and a set of algorithms for achieving a high-utilization software implementation of a streaming graphics pipeline on the GPU. Extensive benchmarks demonstrate real-time performance at a low overhead factor (usually within one order of magnitude) compared to a reference implementation running on OpenGL. For demanding scenarios, where the workload is dominated by shading, cuRE can reach about 40% of the OpenGL performance. Moreover, relative performance is generally highly-consistent with OpenGL across a variety of scenes and across all individual stages of the pipeline. Finally, we present four experimental pipeline extensions that demonstrate the versatility of a software approach.

2 RELATED WORK

For an introduction to GPU architecture and programming model, we refer the reader to the available literature such as, e.g., Fatahalian and Houston [2008], or Nickolls et al. [2008]. In this paper, we will follow CUDA [NVIDIA 2016] terminology: The GPU consists of *multiprocessors* that combine single instruction, multiple data (*SIMD*) cores with fast *shared memory*. A *kernel* function is launched to a large number of parallel threads, grouped into *warps* (assigned to one SIMD core) and further into *blocks* (assigned to one multiprocessor).

On the CPU, where multi-threaded applications are explicitly controlled by the application programmer, each thread runs a unique function. In contrast, on the GPU, a single kernel function is shared by all threads. A typical way of implementing a software pipeline on the GPU is launching one kernel per stage. Unfortunately, such an approach requires all data to be passed from one stage to the next through slow global memory. Exploiting producer-consumer locality to redistribute some data through fast on-chip buffers becomes difficult to impossible. Additionally, as illustrated in Fig. 2a, if the pipeline is expanding, intermediate data can consume excessive amounts of memory as it must be buffered in its entirety. If a stage cannot fully occupy the GPU, processors are left to idle.



(a) traditional multi-pass implementation

(b) streaming pipeline

Fig. 2. Traditional multi-pass versus streaming pipeline processing. (a) Implementing a pipeline as a sequence of kernel launches (typically one for each pipeline stage) leads to suboptimal GPU utilization and excessive memory requirements. If a single stage cannot fully occupy the GPU, cores will be left to idle until processing of the next stage begins. Furthermore, intermediate data between kernel launches must be buffered in its entirety before it is consumed by the next stage. (b) A persistent megakernel with dynamic scheduling enables a streaming pipeline implementation. Work can be consumed as soon as it becomes available, leading to much better GPU utilization. By prioritizing the processing of elements towards the end of the pipeline over elements that may generate new work, the system can operate within bounded memory requirements.

These restrictions can be avoided by using a *persistent threads* architecture [Aila and Laine 2009], which fully occupies the GPU with threads that keep drawing items from a global work queue until the queue is empty. New work items can be dynamically inserted into the queue [Tzeng et al. 2010]. Combining persistent threads with a *megakernel*—an approach popularized for raytracing [Parker et al. 2010]—allows load balancing of different stages by dispatching work items to individual multiprocessors. All threads share a single, monolithic kernel function which contains branches with the code for each stage. One limitation of such a *persistent megakernel* design is that all multiprocessors must share the same resource configuration, which is determined by the most expensive stage [Laine et al. 2013]. Additionally, the global work queue can become a bottleneck. However, the advantage of being able to locally pass data between stages through shared memory can outweigh the cost of suboptimal occupancy, and the bottleneck of the global work queue can be alleviated by using a separate work queue for each stage [Steinberger et al. 2014]. Furthermore, by prioritizing later stages over earlier ones, buffer fill levels can be kept below a threshold (Fig. 2b).

2.1 Parallel rendering

Parallelization is key to achieving the level of performance that real-time rendering demands. However, while the operation of each stage of the graphics pipeline is inherently parallel, the nature of parallelism changes drastically throughout the pipeline. A stream of input primitives entering the pipeline can be processed in parallel on a per-primitive level. However, the number of pixels that will be covered by each primitive varies and cannot easily be predicted.

At the most fundamental level, work in the graphics pipeline can be parallelized either by distributing work associated with individual primitives, or by distributing work associated with individual

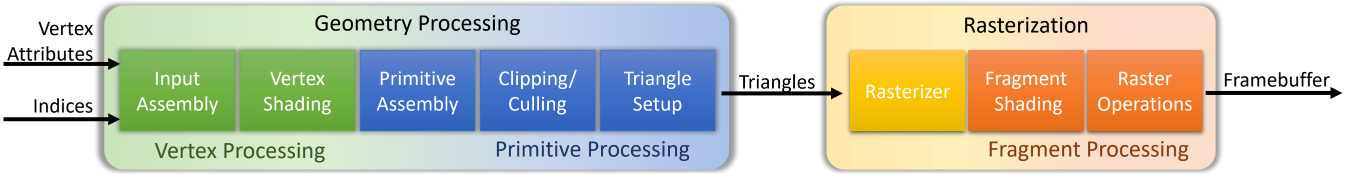


Fig. 3. Graphics pipeline for rendering indexed triangle lists. The pipeline consists of a *geometry processing* section where input triangles are projected to the image plane and a *rasterization* section where the pixels covered by each projected triangle are subsequently filled in. The input assembly stage starts by forming vertices from input vertex attributes. Each vertex is transformed by the vertex shader into a projected position and a set of output attributes to be interpolated across triangles. Primitive assembly then constructs triangles from transformed vertices. Triangles facing away or outside the viewport are culled before the triangle setup stage prepares the remaining triangles for rasterization. The rasterizer determines which pixels are covered by each triangle. For each covered pixel, a *fragment* consisting of depth and other interpolated attributes is fed into fragment shading to compute the fragment color. Finally, the raster operations stage combines the shaded fragments for each pixel by perfoming the depth test and blending fragment colors into the framebuffer.

screen regions among available processors. When distributing work per primitive, we speak of object-space parallelism; distributing work per screen region is typically referred to as screen-space parallelism. Molnar et al. [1994] classified parallel rendering approaches based on the "sorting" point in the pipeline, where the system redistributes work from object-space to screen-space: sort-first, sort-last, or sort-middle. Eldridge et al. [2000] expanded the sorting taxonomy by introducing the sort-everywhere approach. Our approach can be classified as sort-everywhere, but aligns the sorting steps with the computational hierarchy on the GPU: We use sort-middle between multiprocessors, and sort-everywhere between cores inside a multiprocessor. We discuss the reasoning behind this approach in the following section.

2.2 Software graphics pipelines

Software pipelines on the CPU were commonly used in industry prior to the introduction of the consumer GPU [Mesa 3D 1993; RAD 2002]. Later, the Intel Larrabee project [Seiler et al. 2008] demonstrated a software rendering pipeline on a multicore system with wide vector units. Unfortunately, Larrabee was ultimately not deemed competitive enough to yield a commercially viable product and repurposed as a high-performance computing appliance. However, Larabee sparked a lot of enthusiasm in the graphics research community and was an important inspiration for this paper.

A hardware-agnostic simulation environment for arbitrary graphics pipelines (GRAMPS) has been introduced by Sugerman et al. [2009]. However, only a CPU target for the GRAMPS architecture [Sanchez et al. 2011] has been published to date.

One of the first works demonstrating an implementation of a standard graphics pipeline in software on a GPU was FreePipe [Liu et al. 2010]. While FreePipe is a complete end-to-end software pipeline, its architecture follows a sort-last approach and suffers from poor GPU utilization in typical scenes due to a lack of load-balancing. For standard depth-buffering, they only present an approximate approach based on an interleaved depth and color buffer managed with atomic operations.

Probably the fastest software rendering pipeline implementation to date is CUDARaster [Laine and Karras 2011], a highly-optimized sort-everywhere design using separate kernel launches to implement each pipeline stage on the GPU. Due to its reliance on low-level optimizations, it is tightly coupled to the original hardware it was designed for. A more recent approach with a similar aim is Piko [Patney et al. 2015], a compiler for programmable graphics pipelines with a CPU and GPU backend. Like CUDARaster, Piko uses a sort-everywhere approach on top of a sequential kernel architecture. Both approaches separate pipeline stages into multiple kernels. Therefore, they do not support a streaming pipeline model and can be subject to the excessive memory requirements and bandwidth issues discussed before. CUDARaster reports a memory overhead of 2–4× the size of the input data for internal buffering. Both CUDARaster and Piko focus on rasterization and move the formation of the input stream into a preprocessing step. Piko does not address clipping, concurrent framebuffer access, and ordering constraints.

3 GRAPHICS PIPELINE ARCHITECTURE

Fig. 3 shows the logical graphics pipeline implemented in cuRE. Without loss of generality, we focus on rendering of indexed triangle lists. Other primitive types follow analogously. To limit the scope of this discussion, the treatment of a tessellation stage will be deferred as future work. Furthermore, we shall mention geometry shaders only by noting that such a stage is a straight-forward extension to the primitive processing stage of the pipeline architecture presented in the following.

3.1 Design considerations

Our design aims to provide a level of functionality that is comparable to Direct3D9-class hardware. This implies a stateful pipeline, and, in particular, preservation of primitive order. Fragments must be blended in the same order in which their generating primitives were specified, at least when using a non-associative blend function. Moreover, we require that fragment shaders must be able to compute screen-space derivatives for dependent variables, e.g., to support mipmapping [Williams 1983]. In addition to these functional requirements, a practical real-time rendering pipeline must operate within bounded memory.

Given the dynamic workload presented by a graphics pipeline, the main challenge for a competitive software implementation is to ensure near-peak utilization of all available processing cores. Contemporary hardware implementations typically follow a sort-middle or sort-everywhere approach [Purcell 2010], sort-first and sort-last approaches have been shown to exhibit poor scalability [Eldridge et al. 2000]. Guaranteeing blending in primitive order would be expensive in a sort-last approach. A sort-first approach would likely be susceptible to load imbalances.

In contrast, sort-everywhere benefits from exposing multiple points in the pipeline that provide opportunities for optimization. However, a software pipeline executing on an actual hardware must not only consider logical communication cost, but also actual communication cost. On a contemporary GPU, communication across multiprocessors (via global memory) is orders of magnitude slower than communication across SIMD cores within a multiprocessor (via shared memory.) This circumstance requires to make a trade-off: One the one hand, the cost of global communication is likely going to dominate all other design decisions performance-wise. On the other hand, some global communication is necessary for effective load balancing, in order to ensure high utilization of the device.

Consequently, we found a tiled rendering approach with sort-middle in global memory, between geometry processing and rasterization, to be the optimal design point. Allowing no global redistribution at all (as in sort-first or sort-last) would not facilitate effective load balancing, while allowing more than one global redistribution step (as in full sort-everywhere) becomes too costly in terms of memory bandwidth to remain competitive. We characterize the resulting strategy as "global sort-middle, local sort-everywhere-else."

Fig. 4 illustrates our pipeline architecture. The two main stages, geometry processing and rasterization, redistribute via global memory. The sub-stages of each of the main stages move much larger amounts of data than the main stages, but also offer a large degree of producer-consumer locality. Taking advantage of this producer-consumer locality, we redistribute between sub-stages only locally through fast on-chip shared memory without having to sacrifice significant load-balancing potential.

In the geometry stage, the most important local redistribution takes place from vertex processing to primitive processing. Vertex processing loads batches of input indices and sorts out duplicates to avoid redundant vertex shader invocations. Shared memory is used to pass vertex shader outputs on to primitive processing.

In the rasterization stage, the viewport is subdivided into bins of configurable size. Each bin is assigned a *rasterizer*; multiple bins can map to the same rasterizer. Each rasterizer reads triangles covering its bins from an input queue in global memory. During primitive processing, each triangle is inserted into the input queues of all rasterizers assigned to bins covered by the bounding rectangle of the triangle.

Every rasterizer's input queue is exclusively assigned to one thread block. As a consequence, each such rasterizer thread block gains exclusive access to the corresponding framebuffer regions. This exclusive access supports a coarse-to-fine rasterization strategy with communication only in shared memory: Per bin and triangle, a thread determines coverage of *tiles* in a bin. For each covered tile, pixel-wise coverage masks are determined. Pixel coverage masks are further compacted, and rasterizer threads are reassigned to pixel quads or individual fragments for fragment shading. Finally, fragment shading threads perform in-order framebuffer blending using local synchronization.

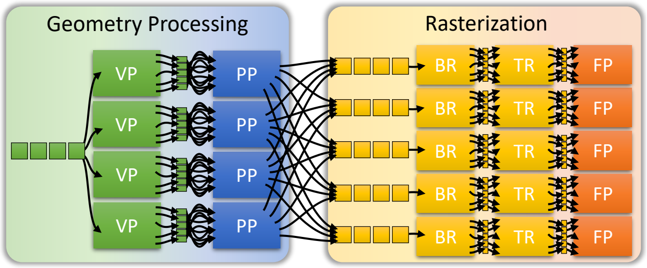


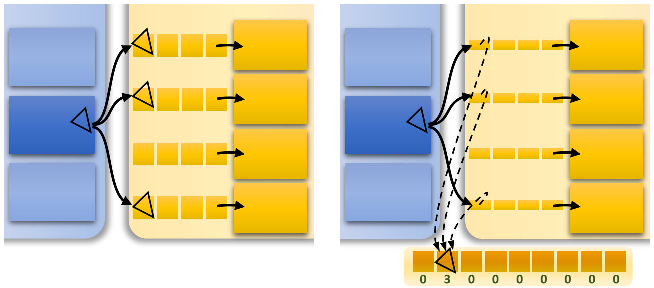
Fig. 4. Overview of our pipeline architecture. An entire thread block is either running geometry processing or rasterization in parallel for a batch of triangles. We rely on global work redistribution between geometry processing and rasterization. Producer-consumer locality allows us to limit redistribution between all other stages in ways that enable the use of faster forms of local communication. Geometry processing uses on-chip memory and the register file to locally redistribute data between vertex processing (VP) and primitive processing (PP). Rasterization uses on-chip memory to locally redistribute data between bin rasterizer (BR), tile rasterizer (TR), and fragment processing (FP).

3.2 Implementation strategy

We implement the streaming graphics pipeline using the persistent megakernel approach of Steinberger et al. [2014]. The stages of the rendering pipeline are branches within the kernel fuction, which are called from the megakernel scheduler, which is itself a branch of the kernel function. The scheduler dynamically assigns thread blocks the role of a rasterizer or a geometry processor on demand for optimal load balancing. In a streaming approach, the global queues must operate within limited storage requirements, so the scheduling of each block must take these limits into consideration:

(1) If there is sufficient work available for rasterization, the block executes rasterization.
(2) Otherwise, if there are input primitives for geometry processing available and all rasterizer queues can accept sufficient additional triangles, the block executes geometry processing.
(3) If there is no input geometry available, the block consumes any rasterization input data that is left in its queue.
(4) If there is neither input to geometry processing nor rasterization available and no other block is still processing geometry, the block terminates.

For the sort-middle step, we must provision sufficient space in the rasterizer queues (as mentioned in section 2) to accommodate the worst case of all blocks starting geometry processing at the same time with all their triangles ending up in the same rasterizer queue. To avoid these buffering requirements dominating the memory usage, we store only references in the rasterizer input queue and allocate the triangle data in a separate triangle buffer (Fig. 5). This indirect memory layout not only removes duplicates when a triangle covers multiple bins, but also reduces the data output of geometry processing, which can make up a significant part of the overall memory transfer. The triangle buffer is implemented as an atomically-operated ring buffer with reference counting to make sure data is not overwritten before it is consumed by rasterization. The rasterizer input queues use similar ring buffers, consisting of



(a) duplication

(b) triangle buffer

Fig. 5. (a) Triangles may be duplicated into multiple rasterizer queues. (b) By using a separate triangle buffer and storing only indices in the rasterizer queues, we can significantly reduce the memory requirements.

entries guarded by atomically operated flags, to avoid that entries are read before they have been written.

The way bins are assigned to rasterizers influences the load put on individual rasterizers. Choosing a good bin-to-rasterizer mapping is, therefore, important for performance. A detailed analysis of different rasterizer patterns has been presented by Kerbl et al. [2017]. We rely on a pattern based on their *X-shift* design, which has proven to be most effective in practice.

In the following sections, we describe geometry processing (Section 4) and rasterization (Section 5) in more detail.

4 GEOMETRY PROCESSING

Geometry processing reads the indices for each primitive, fetches the vertex attributes and runs the vertex shader. Since vertex shader invocations are costly, identical vertices indexed multiple times should ideally be processed only once. Such vertex reuse was not considered in previous work, since the irregular accesses to cached shader results is difficult to handle in parallel. Naïve approaches, such as simply executing the vertex shader once for each index [Liu et al. 2010] or running the vertex shader as a preprocessing step [Laine and Karras 2011], ignore the fact that only a small subset of the vertex data may actually be referenced by a draw call.

Vertex shading and vertex reuse. Our geometry stage processes input geometry in batches of primitives. We identify reoccurring indices within a batch and invoke the vertex shader only once per batch for each vertex index in three steps: (1) Geometry processing first loads indices for a batch of primitives, deduplicates indices, and fetches vertex attributes for each distinct vertex. (2) The vertex shader is executed exactly once for each distinct vertex. (3) We reassemble the original triangles using the transformed vertex data. As batch size, we use a common multiple of warp size and primitive size—for example $3 \cdot 32 = 96$ for a triangle list—which allows us to process each batch using one warp and, thus, take advantage of efficient intra-warp communication primitives. A more detailed description of this approach can be found in [Kenzel et al. 2018].

Primitive assembly. With our vertex reuse implementation, every thread executes the vertex shader for a single vertex. Consequently, the attributes for a triangle are distributed among threads. We use one thread per triangle to assemble the primitive data. We ensure that we have enough threads for all the triangles by limiting the number of indices loaded in the previous stage. We utilize the mapping from indices to threads created during vertex reuse detection to access the vertex shader results from other threads using register shuffles.

Clipping and culling. With the entire triangle data available, we begin by computing the clip-space bounding rectangle for each triangle. Backfacing triangles or triangles that do not intersect the view frustum are simply skipped. Traditional pipeline designs would clip triangles at the near and far planes in this step, possibly generating multiple output triangles for a single input. However, our software rasterizer has to run on the general-purpose cores of the GPU which are mainly designed for floating-point arithmetic. Working in floating point, we make use of *homogeneous rasterization* [Olano and Greer 1997] which allows us to avoid explicit primitive clipping. Nonetheless, to avoid unnecessary scheduling overhead, we compute the bounding rectangle of the clipped triangle and subsequently use this tight bounding rectangle for rasterizer assignment.

Triangle setup. If a triangle is not discarded in the previous stage, it will potentially be processed by multiple rasterizers. To determine which rasterizers the triangle needs to be sent to, we compute which bins are covered by the triangle's bounding rectangle. Triangle rasterization requires a number of parameters, such as the coefficients of triangle edge equations and attribute plane equations. Since all the necessary data is available at this point, we compute these parameters and store them in the triangle buffer. For more efficient memory access, triangle parameters are packed in ways that enable the use of vector load and store instructions. Finally, we add the triangle's location to the input queues of all covered rasterizers.

Primitive order. With multiple blocks working on geometry in parallel, the order in which triangles end up in the rasterizer queues is arbitrary. However, fragments must be output in primitive order. Since we cannot afford the bandwidth for a fragment reorder buffer, we require that fragments already be generated in primitive order. To do so, we rely on the rasterizers consuming their input in the correct order. Previous approaches, which did not implement a streaming design, had multiple options to enforce primitive order. For example, when stages are executed as separate kernels, a prefix sum can be used to allocate successive memory locations for the output.

Enforcing that triangles are entered into the rasterizer queues in the right order would require expensive synchronization between geometry processing blocks. Instead, we make use of the exclusive access a rasterizer has to its input queue to restore order before triangles are consumed. This re-ordering simply corresponds to sorting the input queue according to primitive id. Note that the indirect memory layout using the triangle buffer pays off here, since we only need to move lightweight triangle references.

However, a sorted input queue alone is not sufficient. A triangle might simply not be in the queue yet because it is still going through geometry processing, or it might never arrive because it was culled. The rasterizer must not consume input past a point up to which the queue is known to be complete in the sense that no more primitives will arrive. To achieve this, we track the progress of primitives

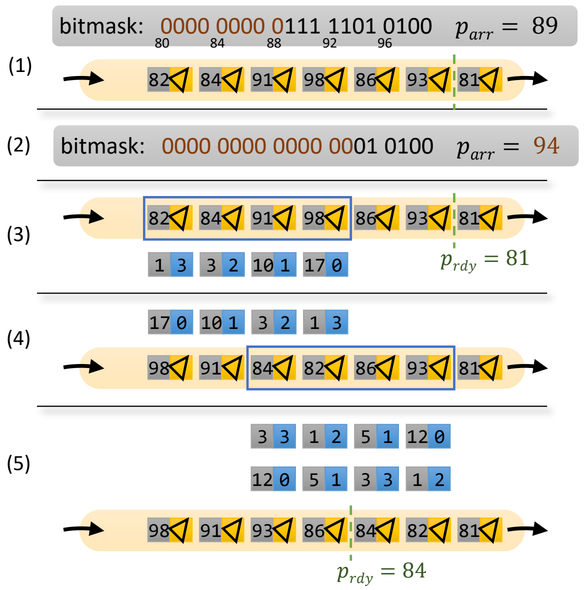


Fig. 6. A bitmask is used to track which primitives have completed geometry processing. Rasterizers use this information to restore primitive order in their input queues before consuming triangles. (1) p_{arr} = 89 indicates that all primitives up to id 89 have previously been accounted for. (2) Having determined that all bits from p_{arr} up to 94 are now set, the block updates p_{arr} and clears the bits. (3) It then starts moving a sorting window (blue rectangle) from the back of the queue to the front. p_{rdy} marks the end of the part of the queue currently known to be complete and in order. A block-wide radix sort takes the position of each element relative to the sorting window (blue) and reorders these values using the primitive id (remapped to the range $[0, p_{arr} - p_{rdy}]$) as key (grey). (4) Queue elements are then rearranged accordingly and the sorting window is moved to the front. (5) The first three elements are ready to be processed and p_{rdy} is updated.

through geometry processing using a large bitmask stored in a ring buffer, where each bit represents a primitive. When a geometry processing thread enqueues or discards a triangle, it updates the corresponding bit. Before sorting the input queue, the rasterizer searches the bitmask for the first bit not set. Note that a simple counter is not a viable alternative to this bitmask, as updating it would require serialization of geometry processing threads.

We implement an efficient search for the lowest zero bit by a parallel find-first-set operation on a word from the bitmask, followed by a reduction. To limit the size of the bitmask, we prune the lower part of the mask as soon as all bits up to a progress value p_{arr} are set. We store p_{arr} and reuse all bits below p_{arr}. A pipeline flush only needs to reset p_{arr}, as all bits are unset when all primitives have moved through the pipeline. Updating p_{arr} with atomic operations allows sharing one progress tracking buffer by all rasterizers.

For reordering, we use a block-wide radix sort. Primitives with id larger than p_{arr} are not ready for consumption yet; sorting only must deliver primitives with id less than p_{arr} to the front of the queue. We achieve this by moving a sorting window from the back of the queue to the front (Fig. 6). In our experiments, a sorting window of w = 8–10× the block size, moving along $w/2$ primitives in each step, achieved the best performance. As soon as the sorting window reaches the front of the queue, we search for the largest primitive id less than p_{arr}. This element marks the end of the part of the queue that is complete, its position is denoted as p_{rdy}. As long as the id of all sorted primitives is below p_{arr}, we continue sorting from the back. Since we are only interested in primitives with id less than or equal to p_{arr}, we can remap the primitive id from $[p_{rdy}, p_{arr}]$ to $[0, p_{arr} - p_{rdy}]$ to use as the sorting key. Doing so reduces the number of bits the radix sort needs to consider, thus avoiding the sorting time growing with absolute value of the primitive ids.

5 RASTERIZATION

Rasterization reads triangle data from the queues in global memory and outputs fragments to the framebuffer. To generate fragments, we use a hierarchical rasterization approach with three levels, similar to Greene [1996]. In our preferred configuration, every 64 × 64 bin is split into 8 × 8 tiles, and every tile covers 8 × 8 pixels. Since large triangles lead to significant data expansion during rasterization, it is challenging to keep temporary data in shared memory and ensure high occupancy. Aiming for high thread utilization, we require dynamic work balancing between the sub-stages of rasterization, The entire workload must stay within the bounds of shared memory, independent of the number of intermediate objects that might be generated, and must not interfere with primitive order.

5.1 Bin raster work distribution

There are multiple options to implement the required work distribution. First, we could consider threads as individual workers which can take on any stage at any point and store intermediate data in atomically operated queues. However, this would lead to increased thread divergence, increased overhead from atomic operations, and keeping memory usage bounded would be difficult. Second, we could use entire warps as workers, allowing them to cooperatively work on individual objects. However, this approach would still require atomically operated queues and leave the problem of unbounded memory requirements unresolved. Instead, we chose to coordinate all threads within a block, assigning all of them to the same stage at once. This design decision allows using prefix sums (c.f. “Allocating Processors” in Blelloch [1990]) for work assignment, and we can assign an arbitrary number of threads to work cooperatively on one object. Using a prefix sum also automatically ensures primitive order is preserved.

Bounded memory consumption is tackled with an *encode only, expand in next stage* strategy. We limit every thread to generate a single fixed-size output. This output must encode the number of threads required to be completed in the next stage and a hook to decode the actual data. This strategy is similar to how hardware tessellation subdivides work between the hull shader (only sets tesselation factors) and the domain shader (generates vertices before commencing shader code). Threads working on the same stage and producing a fixed-size output allow temporary data to be stored as a simple array in shared memory.

The rasterization work distribution (Fig. 7) works as follows: Triangles are fetched from the input queue. Binning computes the number of potentially covered bins. A first work distribution assigns

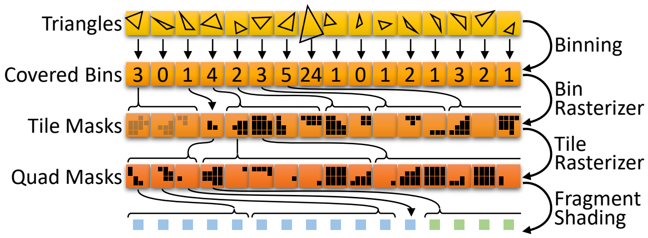


Fig. 7. Our rasterization work distribution using shared memory. All threads work concurrently on the same stage, generating output for the next stage. The following stages only return to the previous one when all data from that stage has been consumed, e.g., the sixteen grayed-out tile mask bits have been consumed in a previous step.

bins to bin rasterizer threads, which compute tile bit masks and store them in the next work distribution. After tiles are assigned to threads, they compute fragment coverage and quad masks, which go through work distribution again before entering fragment shading. After all threads are done with fragment shading and blending, they consume the next set of quad masks from the work distribution. The block continues with fragment shading and blending, until the quad mask work distribution is empty. Then, the tile rasterizer continues work on the next set of available tiles. This approach ensures that the block only steps back one stage when all elements from the current set of work have been processed. It can never happen that more output is generated than can be held in the work distributions. Note that all information from the previous stages is readily available at any point, e.g., the fragment shader can use information from the bin and tile rasterizer to determine the fragment coordinates.

5.2 Hierarchical rasterization

One thread per triangle computes binning information from the triangle's bounding box. Binning determines how many bins owned by the active rasterizer may be covered by the input triangle. Efficient handling of large triangles demands that this computation must not iterate over bins. In accordance with the encode-only strategy, we compute the actual coordinates of the hit bin right before launching the bin rasterizer.

The bin rasterizer uses one thread to determine which tiles in a given bin are hit by the triangle. Using the concept of coverage masks, the hit tiles can be determined without explicitly checking each tile. To avoid the memory traffic incurred by precomputed coverage masks [Laine and Karras 2011], we use an approach similar to Kainz et al. [2009] to construct a conservative coverage mask in scanline order directly from the triangle's edge equations (Fig. 8).

The tile rasterizer determines the actual pixels inside its tile that are covered by the triangle. We follow the same approach as for bin rasterization, however, instead of constructing a conservative mask, the tile rasterizer computes an exact rasterization. Our coverage rasterizer always works along positive y steps, flipping the vertices for edges with negative slope. This approach makes sure that edges shared between triangles exhibit the same sequence of floating point operations and deliver the same intersection results. This establishes a robust fill convention for edges that exactly intersect pixels.

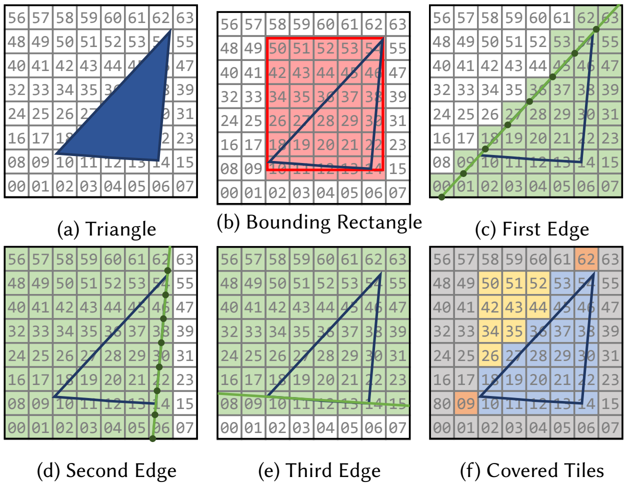

(a) Triangle
(b) Bounding Rectangle
(c) First Edge
(d) Second Edge
(e) Third Edge
(f) Covered Tiles

Fig. 8. The bin rasterizer computes the tile coverage mask using the bounding rectangle and edge equations of a triangle (a) and stores it as a bit mask. The bounding rectangle (b) creates a rectangular pattern and can be generated by duplicating the pattern of a single row. For every edge (c-e), we iterate over all rows, compute the intersection points between the edge and the row (dark dots) and construct the bit mask for the current row. The combination of all masks yields the tile mask (f, blue). Note that the edge equations discard most of the not-hit tiles (grey), possibly a significant number that would not be discarded by the bounding rectangle (yellow). Due to conservative tile rasterization, however, some tiles around vertices are only discarded by the bounding rectangle (orange, bit 09 and 62), justifying the use of bounding rectangle and edge equations.

5.3 Fragment shading

The coverage mask already contains all the information necessary to perform fragment shading. We could simply assign one thread for every set bit in the coverage mask to execute the fragment shader.

However, one crucial feature we must support in the fragment shader is the computation of screen-space derivatives of dependent variables (demonstrated in Fig. 1c). While derivatives can be estimated from differences to neighboring fragments, neighbors do not exist in all cases. To solve this problem, we rely on *quad fragment shading*: the fragment shader is always invoked for quads of 2×2 pixels, *helper threads* are spawned to run the shader for pixels outside the triangle if necessary, all writes being discarded.

To implement quad fragment shading, we compute another bitmask in which bits do not correspond to individual pixels but to quads of pixels. A bit in this *quad coverage mask* is set iff any of the pixels of the quad were covered. The quad coverage mask can be computed from the original coverage mask using a series of inexpensive bit operations. For every set bit in the quad coverage mask, we start four consecutive threads within the same warp to execute the fragment shader. Neighboring values of a variable can then be accessed using register shuffle instructions to estimate screen-space derivatives via finite differences. We keep the original coverage mask to identify helper threads and disable writes.

The rasterizer delivers all pixels that are inside the triangle and within the bounding rectangle, which has been shrunk during geometry processing to consider clipping. However, as triangles are in general not oriented in parallel to the image plane, additional fragments may be generated in front or behind the near or far plane. These fragments should be discarded before launching the fragment shader. When it is safe to do so, one can also perform an early z-test at this point. The fragment's interpolated coordinates, depth, and attributes required by the fragment shader are computed using the information stored in the triangle buffer by the triangle setup stage.

5.4 Raster operations

Previous software rendering pipelines used very small tiles [Laine and Karras 2011]. This allows keeping data for blending and depth testing in shared memory. Small tiles are efficient, if the entire input data is collected before working on a tile, so that sufficient parallel workload per tile is available. In contrast, data in a streaming approach only arrives gradually. Setting up a large number of rasterizers, each working on a very small tile, would, therefore, leave many rasterizers waiting for input. Larger input buffers become necessary, stripping the streaming approach of all its advantages. Thus, our streaming design makes each rasterizer responsible for many bins, with framebuffer and depth buffer kept in global memory accessed during blending operations. Note that the hardware graphics pipeline relies on special-purpose hardware for efficient blending, which is, unfortunately, not available in compute mode.

Since multiple fragments are processed in parallel within each rasterizer, different threads might potentially access the same pixel for blend operations. To resolve this issue, we have to detect and resolve contention within the rasterizer before blending. According to our experiments, delaying the conflict avoidance logic to the end of the pipeline works best.

Before launching the fragment shader, we detect possible access conflicts by searching from each coverage mask to the front, looking for a tile with the same coordinates and an overlapping mask (in the original mask, ignoring helper threads) using simple bit operations. If a conflict is detected, we store the index of the closest coverage mask involved in a conflict (the *predecessor index*). If there are multiple conflicts, this approach implicitly generates a linked list of conflicts. After fragment shading, every thread with an active conflict holds back on blending. After the other threads completed their blending operation, we synchronize all threads in the block. Threads whose predecessor does not have a store conflict are now first in line and can complete blending. Afterwards, they set their predecessor index to zero, indicating to threads that are waiting on them that it is safe to perform blending now. We continue this loop, until all threads complete their blend operations. Note that, if there are no conflicts, this approach allows blending to complete without any serialization.

6 PROGRAMMING INTERFACE

Our software implementation is based on the assumption that target hardware and execution configuration are known at compile time. This allows decisions based on architectural properties such as the number of multiprocessors, SIMD width, or the number of threads per block to be made during compilation and, thus, avoids unnecessary run-time overhead and makes sure that optimal code is generated for every GPU.

The C++ implementation can be divided into two main parts: the graphics pipeline itself, and a module layer. The graphics pipeline is built as a C++ class template that expects the vertex buffer, input layout definition, primitive topology, blend function, and shaders to use as parameters.

The module layer lets the user instantiate different pipeline configurations, and takes care of generating the necessary CUDA kernel functions to invoke the pipeline and manage pipeline state. A module definition containing one or more pipeline configurations can be compiled into a binary image using the standard CUDA toolchain. A matching host library is provided, which offers utility functions for accessing pipeline configurations inside a CUDA binary, finding a suitable configuration for a given GPU, and launching draw calls from the CPU side.

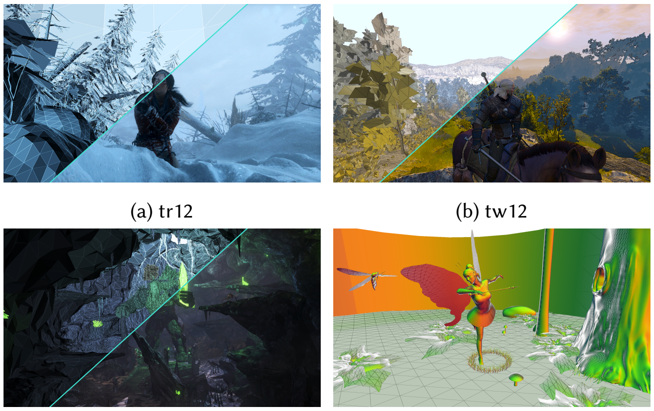
This design offers a convenient high-level interface to a low-level pipeline implementation, in certain ways similar to how a graphics pipeline is exposed through a modern graphics API. Shader programs are represented as C++ function objects. Shader input and output parameters are inferred from the function signature. Using template metaprogramming techniques inspired by SH [McCool et al. 2002], the pipeline implementation automatically matches shader signatures to the inputs and outputs of the next and previous stages, allocates inter-stage buffers, and generates code to pack and unpack shader parameters to and from vector slots for efficient memory access.

7 PERFORMANCE EVALUATION

To evaluate the performance of our approach on real-world data, we collected more than 100 test scenes captured from recent video games. These include *Deus Ex: Human Revolution* (abbrev. de) with 0.3 M–1.2 M triangles, *Rise of the Tomb Raider* (tr) with 0.4 M–3.6 M triangles, *The Stone Giant Demo* (sg) with 2.9 M–8.0 M triangles, *The Witcher 3: Wild Hunt* (tw) 0.3 M–4.6 M triangles, and *Total War: Shogun 2* (sh) with 0.6 M–3.2 M triangles. The scenes were captured by injecting a geometry shader through a Direct3D 11 hook to write out clip-space geometry for each draw call using the stream output stage. The resulting triangle soup was turned into an indexed triangle mesh by fusing vertices based on their position. To account for the fact that some applications switch backface culling modes between drawcalls, all triangles were duplicated with reversed winding order. Example scenes are shown in Fig. 9 and also Fig. 1b.

7.1 Experimental setup

As test platform, we used an Intel Core i7-4930K CPU @ 3.4 GHz with 32 GiB of RAM running Windows 10. Experiments were performed on an NVIDIA GeForce GTX 780 Ti, GTX 980 Ti, GTX 1080 Ti, and GTX 1080. Timings were obtained as average over 30 frames after a burn-in period. The full set of results for all combinations of GPU, renderer, and scene is included as supplemental material. In the interest of brevity, we will limit our discussion here to a representative subset.



(a) tr12

(b) tw12

(c) sg13

(d) fry

Fig. 9. A selection of images representative of the test scenes used: (a) Rise of the Tomb Raider, (b) The Witcher 3: Wild Hunt, (c) NVIDIA's Stone Giant demo, (d) Fairy Forest. (a, b, c) The wireframe renderings on the left hand side were generated by our pipeline with per-vertex colors extracted from the original scene; the right hand side shows the source frames as they were captured from each application. To allow for a meaningful comparison between all the different approaches, our actual tests use very simple shaders and, thus, essentially evaluate the cost of the pipeline itself. *Rise of the Tomb Raider screenshot courtesy of Crystal Dynamics; The Witcher 3: Wild Hunt screenshot courtesy of CD PROJEKT S.A.; Stone Giant screenshot courtesy of NVIDIA Corporation. All images used with permission.*

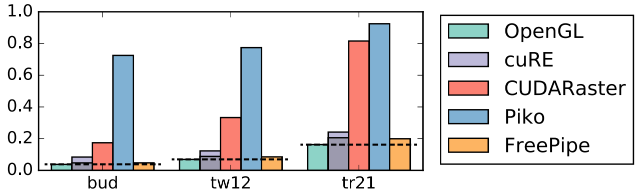


Fig. 10. Memory consumption (as fraction of total device memory) of three scenes rendered on the GTX 780 Ti at a resolution of 1024×768. CUDARaster and Piko require excessive amounts of memory to run. The baseline is formed by OpenGL (dashed line) and FreePipe, which only hold the input data in memory. The light part of the bar shown for cuRE indicates allocated but unused queue memory.

We compare against the three most important previous works, namely, FreePipe [Liu et al. 2010], CUDARaster [Laine and Karras 2011], and Piko [Patney et al. 2015], as well as the hardware graphics pipeline using OpenGL. An additional version of the OpenGL renderer denoted by the subscript fi uses the ARB_fragment_shader_interlock extension to force the hardware pipeline to perform interlocked framebuffer access in a way more similar to our software pipeline, cuRE. For tables and plots, w/o indicates that primitive order and quad shading is deactivated, w/p is with primitive order, w/q with quad shading, and no subscript indicates that both are on. We set the index queue size to 32 000 indices, and our triangle buffer can hold up to one million entries.

Due to the fundamental differences between all the approaches to be compared—not only in terms of their mechanisms, but particularly also their capabilities—we settled on a minimum set of features that can be supported by all methods to allow for a meaningful comparison. Not all approaches are capable of supporting primitive-order blending. Some methods utilize an early-depth-test optimization, while others do not. FreePipe only implements an approximate depth-buffering scheme. We, thus, use a simple rendering of interpolated vertex normals without blending or depth buffering as the lowest common denominator for a performance evaluation in terms of pure drawing speed.

All methods, except Piko, are implemented as plugins for our testbed application. Thus, all these methods run in the exact same environment on the exact same input data. For CUDARaster, we simply wrap the available open source implementation for our framework. Since there is no such reference implementation for FreePipe, we provide our own implementation of their algorithm. Our OpenGL reference renderer is based on OpenGL 4.5 core profile and follows common best practices such as using immutable storage for static data and uniform buffers to manage shader parameters. OpenGL drawing time is measured using GL_TIME_ELAPSED queries; the CUDA-based approaches use CUDA events. Since the overhead for setting up a new frame varies among the different algorithms, we compare only time spent on drawing primitives. Each scene is drawn in a single call as one large mesh using a single shader. No coarse primitive sorting or other optimizations commonly found, e.g., in game engines were used.

Due to its dependencies, we, unfortunately, were unable to successfully integrate Piko with our testbed. Instead, we evaluated their open source reference implementation separately on a suitably configured Linux system running on the same machine. Experiments were carried out using the same test scenes. However, since the Piko rasterization pipeline does not support clipping and will not handle triangles that cross into the negative w space correctly, we provide pre-clipped versions of all our test scenes to Piko. To not give Piko an unfair advantage over other methods by having to process fewer triangles, these pre-clipped scenes were constructed such that clipped parts of each original scene triangle become separate input triangles, which Piko can discard before rasterization.

7.2 Results

CUDARaster and Piko both focus on the rasterization part of the pipeline and run the vertex shader on the entire vertex buffer in advance, which reduces complexity of geometry processing significantly. However, if only parts of the vertex buffer are referenced by a draw call—which is common practice in video games—this may result in considerable performance issues and even malfunctions in practice. To enable a comparison, we merged all draw calls into a single render call, pre-transforming all vertices and removing all non-referenced vertices. All vertex shaders simply multiply the input data with the identity matrix. A demonstration of the influence of multiple draw calls is shown in Table 1. The performance results summarized for each game, for a subset of resolutions and tested architectures are shown in Table 3; selected scenes and standard common test cases are given in Table 2.

Table 1. Performance comparison for different number of draw calls in ms for three specific test scenes on the GTX 780 Ti at 1920 × 1080. As CUDARaster always runs the vertex shader on the whole vertex buffer, it can achieve high performance when draw calls are merged into one. When multiple draw calls reference subsets of the vertex buffer, performance can quickly deter. The performance loss of our approach is only due to separate draw calls leading to underutilization.

	tw17	sh19	tr16
Triangles	0.3 M	2.0 M	5.2 M
Draw Calls	81	512	446
CUDARaster single call	1.9	7.7	13.2
cuRE single call	9.3	44.9	87.2
CUDARaster multiple calls	31.0	707.1	1383.1
cuRE multiple calls	33.3	180.0	262.3

Table 2. Frame time for selected scenes in ms at 1024×768 for Buddha (bud), Fairy Forest (fry), San-Miguel (san), Sibenik (sib), and four game scenes.

		bud	fry	san	sib	tr12	tw12	sh23	sg13
GTX 1080	OpenGL	1.1	0.2	7.5	0.1	2.9	2.3	0.8	0.4
	OpenGL$_{fi}$	1.2	0.3	7.5	0.1	4.0	3.9	2.1	0.9
	CUDARaster								
	cuRE	7.8	2.8	35.8	1.6	28.4	39.9	16.8	6.4
	cuRE$_{w/q}$	5.4	2.4	29.2	1.4	24.0	21.5	14.8	6.1
	cuRE$_{w/p}$	8.3	2.8	35.5	1.4	25.3	37.9	14.2	5.8
	cuRE$_{w/o}$	5.5	2.3	28.0	1.4	20.9	19.4	12.1	5.4
	Piko	8.4	3.6	44.0	2.9	37.1	25.1	12.5	7.7
	FreePipe	0.8	72.3	292.7	68.1	261.3	66.2	141.4	145.0
GTX 780 Ti	OpenGL	1.8	0.4	12.3	0.2	5.1	3.4	1.4	0.8
	OpenGL$_{fi}$								
	CUDARaster	4.2	2.1		1.5	20.1	14.3	7.1	4.6
	cuRE	23.1	8.0	143.3	4.8	88.9	95.2	37.4	18.3
	cuRE$_{w/q}$	17.1	7.4	105.7	4.1	70.7	56.9	33.8	16.7
	cuRE$_{w/p}$	23.6	7.8	143.7	4.3	82.8	93.5	31.1	16.7
	cuRE$_{w/o}$	16.9	7.0	95.6	3.6	63.5	50.4	28.3	16.7
	Piko	19.4	8.5		7.5	89.9	62.3	28.9	19.8
	FreePipe	2.1	156.3		149.1	903.6	182.5	463.8	492.4

As expected, none of the software approaches reach the performance of the hardware pipeline. CUDARaster stays within a factor of 4–6, while cuRE manages to stay within about one order of magnitude. Depending on the test case, restoring primitive order costs up to 20% of performance. Quad shading may add little cost for scenes dominated by big triangles, like in *Sibenik*, but may also add up to 20% of overhead for scenes with many small triangles, like more complex game scenes. The performance edge of CUDARaster over our approach is not surprising, as it follows a sequential kernel approach, which takes away the burden of scheduling and allows to store data for blending and depth testing in shared memory. However, recall that a sequential kernel approach may lead to increased memory consumption, as all intermediate data needs to be stored in global memory (Fig. 10), leading to failure for large test cases, like *San-Miguel*. Memory issues also quickly become evident with larger resolutions; e.g., CUDARaster did not run in 4K (3840 × 2160). CUDARaster is furthermore dependent on optimizations targeting the outdated Fermi architecture, execution on architectures more recent than Kepler fails.

Although Piko follows a similar approach to CUDARaster and does not consider clipping, blending, primitive order, or quad shading, their performance is significantly lower than CUDARaster. Comparing it to our approach without quad shading and without primitive order, it can be seen that cuRE$_{w/o}$ is usually between 1.2× and 2× faster. Interestingly, our speedup on more recent architectures is higher than on the GTX 780 Ti. Piko only achieves a better average performance for *Stone Giant*, which we attribute to the uneven distribution of small triangles in these scenes, which complicates our load balancing. Piko also requires significant amounts of memory and thus has issues with larger test scenes and resolutions above 1024 × 786. Also, their approach requires the screen resolution and maximum number of input primitives to be hard-coded, which allows for additional compiler optimizations, at the expense of practical usability.

FreePipe always assigns one thread to each triangle and runs into performance issues when triangles become larger. Moreover, it does not support blending, primitive order or quad shading. The only scene that achieved good performance is *Buddha*, which consists of many evenly small triangles. In all other cases the performance gap is 1–2 orders of magnitude, and, for larger test scenes, FreePipe just times out. As FreePipe does not store any intermediate data, it uses the minimal amount of memory.

Overall, cuRE achieved the most consistent performance, being the only approach apart from OpenGL that completed all test cases in all resolutions on all architectures. It is the only approach that implements an entire pipeline with full geometry processing, primitive order, quad shading and appropriate blending. cuRE automatically works with low memory requirements, even if larger buffers are allocated (Fig. 10). In comparison to CUDARaster and Piko, which need to store all intermediate data in global memory, the streaming design of cuRE operates with very small intermediate buffers, demonstrating only a slight overhead compared to OpenGL. Our queue sizes can—to a certain extent—be chosen freely. For example, in tr21, the index queues reached a fill-level of 97%, but never overflowed. When queues fill up, our approach simply does not run geometry processing, leading to idle blocks waiting for other rasterizers to complete their workload. Obviously, this can reduce performance, but guarantees correctness. Note that Piko must be pre-configured to a fixed buffer size for all data, which we chose at 75% of the GPU memory to allow a maximum number of scenes to complete. However, the amount of memory was not enough for many scenes, like, *San-Miguel*. Piko has no feature to track fill levels.

To quantify how similar in behavior the different approaches are to the hardware graphics pipeline, we computed the Pearson correlation coefficient between OpenGL and the tested methods over the entire test body. On the GTX 1080 Ti, cuRE achieved correlations of .80 and .94, Piko .69 and 0.70, and FreePipe .50 and .67 with OpenGL and OpenGL$_{fi}$, respectively. On the GTX 780 Ti,

Table 3. Summary results in ms for our entire test body of more than 100 test cases. Next to OpenGL, cuRE is the only approach that completed all test scenes on all architectures and resolutions. In most cases, there is a clear ordering among the software approaches with same capabilities: CUDARaster is faster than cuRE, and cuRE$_{w/o}$ is faster than Piko (with the exception of sg on the 780 Ti); Piko is faster than FreePipe. Comparing relative performance, our approach performs better on newer architectures and even achieves real-time framerates in 4K resolution for most games.

		1024 × 768					1920 × 1080					3840 × 2160				
		de	tr	sg	sh	tw	de	tr	sg	sh	tw	de	tr	sg	sh	tw
GTX 1080	OpenGL	0.4	1.4	0.2	0.8	0.9	0.5	1.8	0.5	0.9	1.0	1.3	3.5	1.7	1.4	1.7
	OpenGL$_{fi}$	0.6	2.1	0.7	1.2	1.4	1.3	3.5	1.6	1.6	2.1	4.0	10.5	5.5	3.6	4.8
	CUDARaster															
	cuRE	5.5	16.8	5.5	10.2	13.0	9.8	27.2	11.3	13.3	16.8	25.1	59.7	33.7	27.3	33.8
	cuRE$_{w/q}$	5.0	14.5	5.3	8.2	10.3	9.2	25.1	11.1	11.4	14.4	24.4	57.4	33.3	25.3	31.4
	cuRE$_{w/p}$	5.1	15.3	5.1	9.3	11.6	9.0	25.1	10.5	12.0	14.7	23.5	54.9	32.0	24.3	29.6
	cuRE$_{w/o}$	4.5	13.0	4.8	7.3	8.8	8.4	22.6	10.2	10.0	12.2	22.5	51.8	31.3	22.1	26.6
	Piko	13.2	21.8	5.8	9.3	12.3										
	FreePipe	146.4	197.3	143.6	136.9	74.4	386.1	522.6	376.7	354.3	196.4					
GTX 780 Ti	OpenGL	0.7	2.5	0.6	1.3	1.6	1.2	3.6	1.3	1.7	2.0	3.8	9.2	4.9	3.4	4.1
	OpenGL$_{fi}$															
	CUDARaster	3.7	10.7	3.6	5.9	7.4	6.8		7.5	8.3	10.3					
	cuRE	15.8	48.0	16.1	27.0	34.3	24.7	65.9	27.8	36.5	46.3	67.1	179.8	92.3	75.8	95.1
	cuRE$_{w/q}$	14.1	40.4	15.5	21.1	27.5	22.4	57.5	26.6	30.5	38.0	64.0	168.9	90.5	67.9	84.9
	cuRE$_{w/p}$	15.0	44.6	14.8	25.1	32.0	22.9	59.9	25.5	33.8	41.0	61.6	162.8	85.4	68.0	83.5
	cuRE$_{w/o}$	13.2	36.3	13.9	18.8	24.3	20.4	50.6	24.5	27.5	32.8	58.7	150.5	84.1	59.0	72.3
	Piko	28.2		13.0	22.1	28.1										
	FreePipe	440.2	635.1	389.4	412.7	200.0										

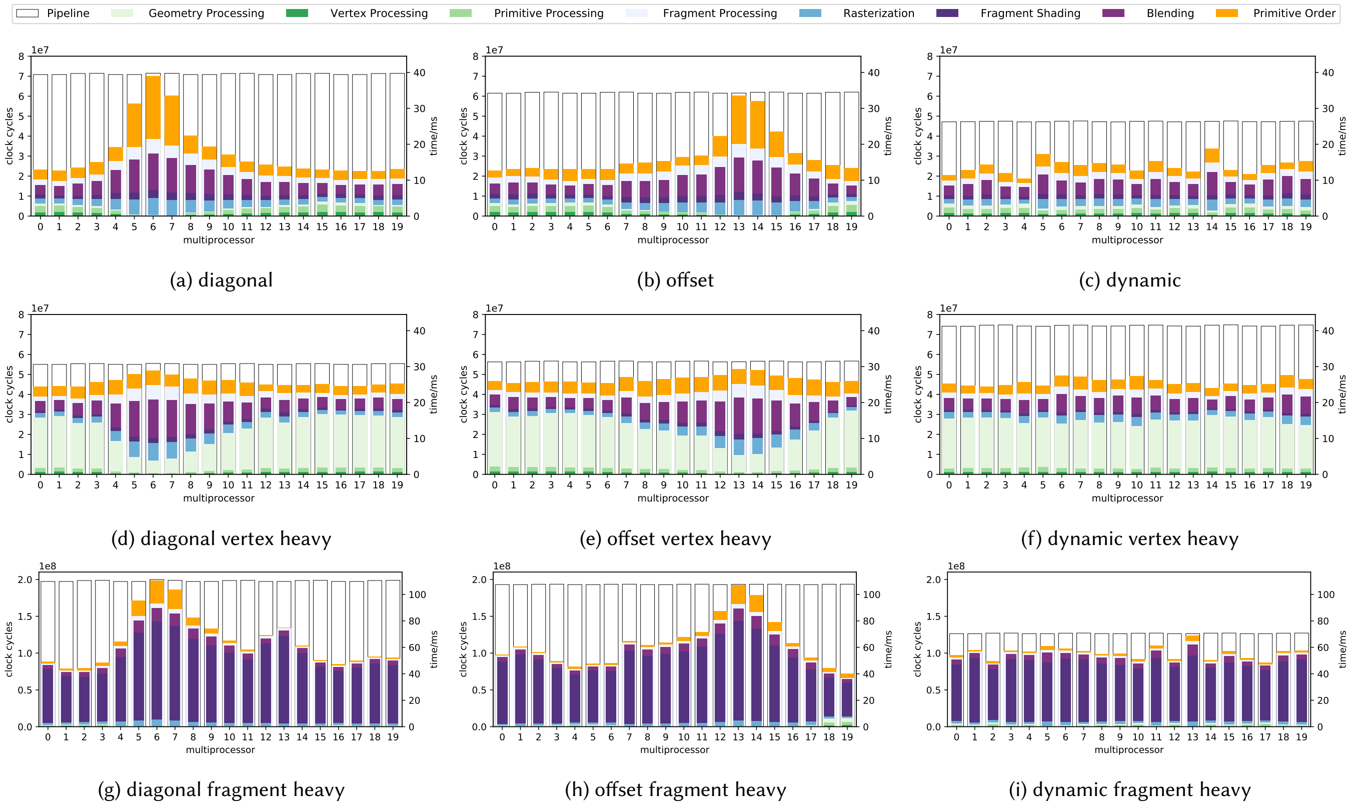
the correlation coefficients with OpenGL are significantly higher, with cuRE achieving .94, CUDARaster .92, Piko .97 and FreePipe .31. OpenGL$_{fi}$ is not supported on the GTX 780 Ti. These results show that, especially for more recent architectures, the performance of cuRE is predictably similar to the hardware pipeline. In case the hardware is forced to use interlocked framebuffer access, the relative performance matches particularly closely (up to 0.95), while other approaches stay below 0.70.

8 PERFORMANCE ANALYSIS

Fig. 11 shows a detailed performance breakdown of how much processing time is spent in each stage on each multiprocessor for a typical scene on a GeForce GTX 1080. We compare the load balancing capabilities of different methods of rasterizer assignment for three different workloads: simple lambert shading, an expensive vertex shader with a trivial fragment shader, and a trivial vertex shader with an expensive fragment shader. As simulated shader load, we use 16 octaves of simplex noise [Perlin 2001] to compute a per-vertex color or fragment color respectively. In addition to static rasterizer assignment with a diagonal or offset pattern according to Kerbl et al. [2017], we also implemented an alternative version of the megakernel in which rasterizer blocks are dynamically assigned to rasterizer queues. Exclusive ownership is guaranteed by requiring each block to acquire ownership of the queue it intends to process through a locking mechanism. The scheduling algorithm checks the number of elements available in all rasterizer queues and attempts to acquire a queue starting from the largest one in descending order. This approach will potentially yield better load balancing at increased scheduling overhead due to the need to search multiple queues to find work.

Overall, runtime is clearly dominated by blending unless very expensive shaders are used, which is one area in which CUDARaster gains an advantage, since it can perform blending in shared memory. The offset pattern yields slightly better performance over the simple diagonal pattern in the scene used here. Increasing the vertex load causes the relative significance of the slightly better load-balancing of the offset pattern to vanish. Interestingly, both static approaches turn out to be faster under higher vertex load, which we can only explain as a result of the increased latency-hiding potential and decreased resource contention due to higher arithmetic load leading to an overall reduced pipeline overhead. In cases where a different pattern cannot yield much improvement for static rasterizer assignment anymore, dynamic rasterizer assignment can sometimes still achieve a significant increase in performance. However, given a heavy vertex load, dynamic rasterizer assignment loses to both static variants as diminishing returns from better load balancing cannot outweigh the inherently larger scheduling overhead of the dynamic method.

As a general observation, the difference between pipeline implementations tends to shrink with increased shading load as the relative system overhead becomes more and more insignificant compared to the shading work. This holds true even when comparing our approach to OpenGL drawing the same scene with equivalent shaders. With very heavy fragment load (32 octaves of simplex noise), we are only a factor of about 2.5× slower than OpenGL from initially a factor of more than 10 with trivial shaders. Both, cuRE



(a) diagonal

(b) offset

(c) dynamic

(d) diagonal vertex heavy

(e) offset vertex heavy

(f) dynamic vertex heavy

(g) diagonal fragment heavy

(h) offset fragment heavy

(i) dynamic fragment heavy

Fig. 11. Detailed performance breakdown for the scene sh08 on the GTX 1080, showing the time spent in each stage on each multiprocessor. We compare static rasterizer assignment with diagonal and offset patterns as well as dynamic rasterizer assignment for simple lambert shading as well as vertex and fragment-heavy workloads.

as well as OpenGL behave in similar ways as the shading load is moved around. We could not identify an individual bottleneck that could be attributed with being responsible for most of the the performance gap between software and hardware. Considering all of the above, we come to the conclusion that the main limiting factor for a software graphics pipeline is simply a generally higher base overhead across all parts of the system.

Unsurprisingly, the rasterizer queues and, especially, the triangle buffer are responsible for a significant portion of the global memory traffic. It is notable that at 4K resolution, they still contribute about 70% of the total memory transactions in typical scenes. However, more than 70% of the queueing traffic and more than 95% of queue management atomics hit the L2 cache. We have seen indication that there might be potential to improve the triangle buffer memory access pattern, which currently seems to be a main source of scattered writes. Heavier shading loads somewhat relief pressure on the queueing system and achieve slightly improved cache hit rates.

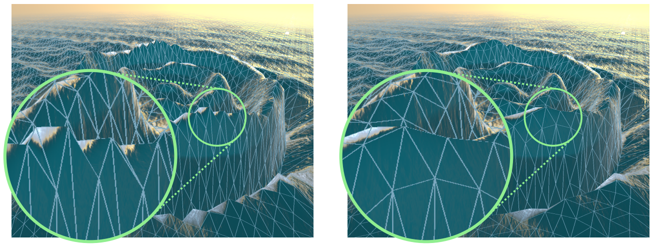
On the Pascal GPU, the best configuration we could find was running at 50% occupancy with two blocks per multiprocessor and 16 warps per block. We achieve an issue efficiency of 25–30%, with the majority of stalls (about 40%) caused by inter-warp synchronization. Almost all of these stalls are observed in the scheduling logic; only a few percent are owed to rasterizer queue sorting and framebuffer access. Memory latency is responsible for only about 20% of stalls.

9 EXPERIMENTAL PIPELINE MODIFICATIONS

To demonstrate the merits of having a fully-programmable graphics pipeline that can be completely customized for an individual application, we implemented a number of pipeline modifications. First, we extended the pipeline by a custom shader stage to enable programmable primitive topology. Further examples consider programmable blending and its many applications, ranging from medical visualizations to 2D graphics and document display. We also show how some minor modifications to the rasterizer stage can enable direct wireframe rendering as well as efficient rendering of adaptively subsampled images.

9.1 Custom primitive types

A water surface, as depicted in Fig. 1a, is commonly created from a regular grid of vertices animated by a wave function. To draw the surface, these vertices are typically triangulated in a regular fashion. Such an approach can lead to artifacts like the ones seen in Fig. 12a, where some edges significantly deviate from the contour of the height field. To solve this problem, we simply make our input and primitive assembly stages programmable by two new types of



(a) regular triangulation

(b) adaptive triangulation

Fig. 12. By making the primitive assembly stage of our pipeline programmable, we enable an application to define its own primitive types. (a) A height field such as the water wave depicted here would traditionally be drawn using a static index buffer, forming triangles according to a regular pattern which can lead to artifacts, such as the "zigzag" shape along the crest of the wave, where edges are forced to go against the contour of the underlying wave function. (b) By introducing a new, adaptively triangulated quad primitive type, we achieve a much better mesh that dynamically accommodates to an animated height field at almost negligible cost.

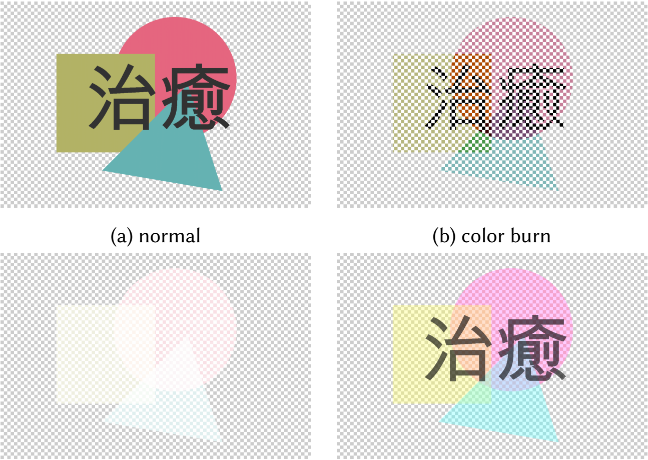
shader: a *primitive assembly shader* and a *primitive triangulation shader*. The primitive assembly shader enables custom primitive types by defining which vertices of the input stream form a primitive. By adding this stage, the distinction between indexed and non-indexed primitives essentially becomes obsolete—the shader programmer can decide whether to load vertex indices from a memory buffer, evaluate a sequence based on the primitive index, or perform some even more general computation. Once vertices have been fetched and passed through the vertex shader, the primitive triangulation shader is used to determine how a primitive is split up into triangles for rasterization.

We require the primitive assembly shader to statically declare how many output triangles a primitive will generate for our geometry processing to continue to work using one thread per triangle. The primitive triangulation shader is run in parallel by all the threads allocated for the processing of the primitive's output triangles. We use these parallel invocations to compute a score for each potential triangulation of the primitive. The primitive assembly stage simply chooses the triangulation which returns the smallest score.

To render the water surface, we define a primitive assembly shader for a new *adaptive quad* primitive type, which takes four vertices from the input stream to produce two triangles. The corresponding primitive triangulation shader computes an energy term based on the deviation of the face normals from the vertex normals, which are computed from the wave function in the vertex shader. The pipeline chooses the triangulation that minimizes this energy term, leading to dynamic mesh adaptation (Fig. 12b) at only a very small additional cost (≈ 0.1 ms on a GeForce GTX 1080). To achieve the same effect with the hardware pipeline would require updating the index buffer in an additional pass before rendering each frame.

9.2 Direct wireframe rendering

Another example of the power of a pure software approach can be seen in the wireframe overlays in Fig. 12. Rendering such images



(a) normal

(b) color burn

(c) screen

(d) hard light

Fig. 13. Many of the numerous different blend functions commonly used in 2D graphics are not supported in hardware. Our software graphics pipeline can be modified to implement any desired blend function. This example shows a vector graphic rendered by our software pipeline using (a) normal, (b) color burn, (c) screen, and (d) hard light blending.

using a conventional hardware graphics pipeline would either require two passes with state changes in-between, where additional measures such as a slope-scaled depth bias have to be taken to avoid artifacts, or rely on fragile tricks involving the use of geometry shaders and additional interpolators [NVIDIA 2007]—all just to reconstruct information in the fragment shader that is already available in the rasterizer. Since we have full control over all aspects of the pipeline, we can modify the rasterizer to pass information like barycentric coordinates and the screen-space distance from the fragment to each edge to the fragment shader. With this information, the wireframe can be incorporated directly into the fragment shader at virtually no overhead.

9.3 Programmable blending

Since our graphics pipeline already performs blending in software, it is trivial to add another shader stage through which an application can define its own blend function instead of being limited to a set of predefined hardware functions. Fig. 13 depicts results of a vector graphics renderer based on the method described by Loop et al. [2005] with support for the full set of blend functions specified in the *portable document format* (PDF) standard [ISO 2008].

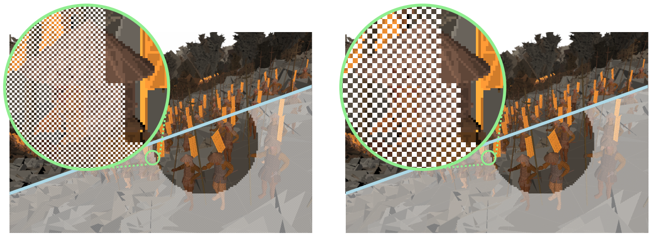
9.4 Adaptive subsampling

As display resolutions keep increasing, it gets harder for modern applications to render high-quality content quickly enough to maintain a minimum desired frame rate at full resolution. This problem becomes even more pronounced in Virtual Reality, which requires very high resolutions, while, at the same time, motion-to-photon latency must be kept low. A popular strategy to cope with this issue is *checkerboard rendering*. To save computation, rendering of every other pixel is skipped following a checkerboard pattern, thus producing a subsampled image at ideally twice the speed. A reconstruction filter is then applied to upsample the result to full display resolution. A more sophisticated, adaptive approach for VR applications combines checkerboard rendering with foveated rendering [Vlachos 2016]. Based on eye-tracking information, a small area around the user's current gaze point is rendered in full resolution, while resorting to checkerboard rendering for the remaining areas.

Implementations using the hardware graphics pipeline have to rely on discarding fragments in the fragment shader to achieve adaptive checkerboard rendering. The downside of such an approach, especially with long-running fragment shaders, is that it inherently leads to suboptimal GPU utilization, as half the threads in a warp executing fragment work are idle most of the time: Half the threads perform a discard and exit immediately, while the other half runs the fragment shader to completion. Another problem is caused by the fact that the GPU performs pixel quad shading. To get any benefit at all, a checkerboard pattern with 2 × 2 pixel squares has to be chosen, as, otherwise, any skipped pixel threads would still be forced to execute the full fragment shader branch, while acting as helper threads for their pixel quad. Using a 2 × 2 pixel pattern, however, has a diminishing effect on the quality of the upsampled image.

In our software pipeline, we could instead modify the coverage mask of each tile to skip the fragment shader before it is even scheduled. Therefore, we introduce the *coverage shader*, a new type of shader which is called for each tile after the rasterizer has computed the coverage mask. Using a coverage shader, an application can programmatically modify the coverage mask of each tile before the fragment shader is dispatched. In the case of adaptive checkerboard rendering, we use a coverage shader that simply computes the distance of each tile from the user's gaze point. If a tile falls in the area where the user is currently looking, we pass on the coverage mask computed by the rasterizer unchanged. Otherwise, we combine the coverage mask using a bitwise logical AND with a bitmask representing the checkerboard pattern to use.

By using the coverage shader, instead of discarding fragments in the fragment shader, we can avoid the overhead of invoking the fragment shader for pixels that should be skipped altogether and achieve significantly better GPU utilization, since only fragment threads that will produce an output are going to be scheduled. In addition, our software pipeline allows us to turn off quad shading when not required by the application, e.g., in case screen-space derivatives are not used or can be computed analytically. Alternatively, the rasterizer could also be modified to perform quad shading across the diamond-shaped fragment neighborhood given by the checkerboard pattern. In any of these cases, we can switch to a 1 × 1 pixel square checkerboard pattern, which achieves better image quality at the same cost. For the scene depicted in Fig. 14, adaptive checkerboard rendering using a discard in the fragment shader is more than 30% faster than rendering at full resolution. Using the coverage shader led to another performance increase of more than 30%. The 1 × 1 pixel pattern achieved a 22% smaller mean squared error to the full resolution rendering while running at the same speed as the 2 × 2 pixel pattern, when quad shading is off.



(a) 1 × 1 pixel pattern

(b) 2 × 2 pixel pattern

Fig. 14. Using our programmable coverage shader stage, we can implement adaptive checkerboard rendering without the inefficiencies of the conventional approach based on discarding fragments. The images above show a scene captured from the game *Total War: Shogun 2* rendered with adaptive checkerboard rendering using (a) 1 × 1 and (b) 2 × 2 pixel squares.

10 CONCLUSION

We have demonstrated that a standard graphics pipeline implemented entirely in software can achieve real-time rendering performance for real-world scenes on a GPU today. The flexibility of a software approach opens up interesting possibilities in the form of application-defined graphics pipelines tailored directly to a specific task. Performance results established over more than 100 test scenes show rendering speeds for our approach within one order of magnitude of the hardware graphics pipeline. We are also within a factor of approximately three of CUDARaster while enabling a bounded-memory, end-to-end streaming implementation with a fully-compliant vertex processing stage that is portable across all current CUDA hardware architectures. We are considerably faster than Piko while at the same time addressing core issues such as primitive order, clipping, and screen-space derivatives. Our streaming graphics pipeline was the only approach next to the OpenGL reference able to complete all test cases in all resolutions on all graphics cards.

While a software implementation will never replace an equivalent hardware graphics pipeline when it comes to raw speed and efficiency, we see an interesting prospect in graphics hardware gradually adding support for software pipelines alongside the traditional hardware pipeline. By doing so, future GPU architectures could cover both ends of the spectrum and give applications a choice to use the programmable hardware pipeline for maximum efficiency or a hardware-accelerated software pipeline in cases where such a more flexible approach better fits the task at hand. Given sufficient hardware support in key places, the overhead of a software rendering pipeline may ultimately be amortized in certain applications by enabling a more straightforward implementation of rendering algorithms in a custom graphics pipeline. The hardware cost of adding this support would likely be limited, as many of the desirable features such as highly-optimized framebuffer operations and high-performance scheduling logic are already present in the hardware.

An important open question is what the interfaces through which such hardware support would be exposed to software should look like. We believe that a software pipeline implementation such as ours, which is similar in design and behavior to the hardware pipeline, may proove to also be a powerful tool in attacking problems such as this one. Even with a pure software system like the one we presented, there is a need for a proper interface between pipeline and application. Particular strength of a software approach in this regard lies in the fact that the boundary between graphics pipeline and application can be moved around freely which facilitates rapid exploration of different ideas.

For future work, we plan to further develop our implementation into a Direct3D12-class graphics pipeline with support for tessellation, and experiment with more complex pipeline extensions. Ultimately, we seek to generalize our approach to allow arbitrary streaming graphics pipelines to be expressed in a way that facilitates the synthesis of an efficient GPU implementation from an abstract pipeline model.

As a service to the community, we have made our implementation available as open source: https://github.com/GPUPeople/cuRE.

ACKNOWLEDGMENTS

This research was supported by the Max Planck Center for Visual Computing and Communication, by the German Research Foundation (DFG) grant STE 2565/1-1, and the Austrian Science Fund (FWF) grant I 3007.

REFERENCES

Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)*. ACM, New York, NY, USA, 145–149.

Johan Andersson. 2009. Parallel Graphics in Frostbite – Current & Future. SIGGRAPH '09 Courses, Beyond Programmable Shading, Talk.

Guy E. Blelloch. 1990. *Prefix Sums and Their Applications.* Technical Report CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

David Blythe. 2006. The Direct3D 10 System. *ACM Trans. Graph.* 25, 3 (July 2006), 724–734.

Matthew Eldridge, Homan Igehy, and Pat Hanrahan. 2000. Pomegranate: A Fully Scalable Graphics Architecture. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00).* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 443–454.

Kayvon Fatahalian and Mike Houston. 2008. A Closer Look at GPUs. *Commun. ACM* 51, 10 (Oct. 2008), 50–57.

Ned Greene. 1996. Hierarchical Polygon Tiling with Coverage Masks. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96).* ACM, New York, NY, USA, 65–74.

K. E. Hillesland and J. C. Yang. 2016. Texel Shading. In *Proceedings of the 37th Annual Conference of the European Association for Computer Graphics: Short Papers (EG '16).* Eurographics Association, Goslar, Germany, 73–76.

ISO. 2008. *Document management – Portable document format – Part 1: PDF 1.7.* International Standard 32000-1:2008. International Organization for Standardization.

Bernhard Kainz, Markus Grabner, Alexander Bornik, Stefan Hauswiesner, Judith Muehl, and Dieter Schmalstieg. 2009. Ray Casting of Multiple Volumetric Datasets with Polyhedral Boundaries on Manycore GPUs. *ACM Trans. Graph.* 28, 5 (Dec. 2009), 152:1–152:9.

Michael Kenzel, Bernhard Kerbl, Wolfgang Tatzgern, Elena Ivanchenko, Dieter Schmalstieg, and Markus Steinberger. 2018. On-the-fly Vertex Reuse for Massively-Parallel Software Geometry Processing. (2018). arXiv:1805.08893

Bernhard Kerbl, Michael Kenzel, Dieter Schmalstieg, and Markus Steinberger. 2017. Effective Static Bin Patterns for Sort-middle Rendering. In *Proceedings of High Performance Graphics (HPG '17).* ACM, New York, NY, USA, 14:1–14:10.

Khronos. 2016a. *The OpenGL Graphics System: A Specification.* The Khronos Group Inc.

Khronos. 2016b. *Vulkan 1.0.32 - A Specification.* The Khronos Group Inc.

Samuli Laine and Tero Karras. 2011. High-performance Software Rasterization on GPUs. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG '11).* ACM, New York, NY, USA, 79–88.

Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference (HPG '13).* ACM, New York, NY, USA, 137–143.

Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. 2010. FreePipe: A Programmable Parallel Rendering Architecture for Efficient Multi-fragment Effects. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '10).* ACM, New York, NY, USA, 75–82.

Charles Loop and Jim Blinn. 2005. Resolution Independent Curve Rendering Using Programmable Graphics Hardware. *ACM Trans. Graph.* 24, 3 (July 2005), 1000–1009.

Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. 2002. Shader Metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '02).* Eurographics Association, Aire-la-Ville, Switzerland, 57–68.

Mesa 3D. 1993. The Mesa 3D Graphics Library. https://www.mesa3d.org.

Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. 1994. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.* 14, 4 (July 1994), 23–32.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (March 2008), 40–53.

NVIDIA. 2007. *Solid Wireframe.* Whitepaper WP-03014-001_v01. NVIDIA Corporation.

NVIDIA. 2016. *CUDA C Programming Guide.* NVIDIA Corporation.

Marc Olano and Trey Greer. 1997. Triangle Scan Conversion Using 2D Homogeneous Coordinates. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware (HWWS '97).* ACM, New York, NY, USA, 89–95.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Trans. Graph.* 29, 4 (July 2010), 66:1–66:13.

Anjul Patney, Stanley Tzeng, Kerry A. Seitz, Jr., and John D. Owens. 2015. Piko: A Framework for Authoring Programmable Graphics Pipelines. *ACM Trans. Graph.* 34, 4 (July 2015), 147:1–147:13.

Ken Perlin. 2001. Noise Hardware. SIGGRAPH '01 Courses, Real-Time Shading, Talk.

Tim Purcell. 2010. Fast Tessellated Rendering on the Fermi GF100. High Performance Graphics 2010, Hot 3D, Talk.

RAD. 2002. *Pixomatic SDK Features.* RAD Game Tools Inc. http://www.radgametools.com/cn/pixofeat.htm

D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis. 2011. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques.* IEEE Computer Society, Washington, D.C., USA, 22–32.

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: A Many-core x86 Architecture for Visual Computing. *ACM Trans. Graph.* 27, 3 (Aug. 2008), 18:1–18:15.

Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU. *ACM Trans. Graph.* 33, 6 (Nov. 2014), 228:1–228:11.

John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 1-3 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. 2009. GRAMPS: A Programming Model for Graphics Pipelines. *ACM Trans. Graph.* 28, 1 (Feb. 2009), 4:1–4:11.

Stanley Tzeng, Anjul Patney, and John D. Owens. 2010. Task Management for Irregular-parallel Workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics (HPG '10).* Eurographics Association, Aire-la-Ville, Switzerland, 29–37.

Alexis Vaisse. 2014. Efficient Usage of Compute Shaders on Xbox One and PS4. Game Developers Conference Europe 2014, Talk.

Alex Vlachos. 2016. Advanced VR Rendering Performance. Game Developers Conference 2016, Talk.

Lance Williams. 1983. Pyramidal Parametrics. *SIGGRAPH Comput. Graph.* 17, 3 (July 1983), 1–11.