A Sort-based Deferred Shading Architecture for Decoupled Sampling

Petrik Clarberg Robert Toth Jacob Munkberg

Intel Corporation

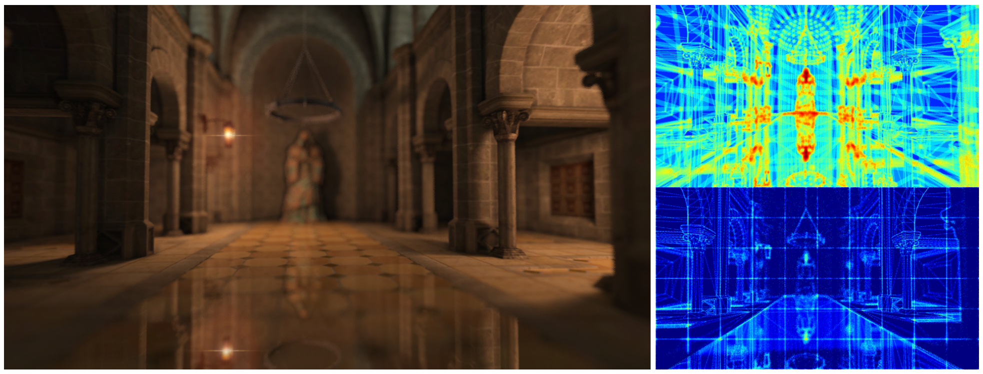


Figure 1: *The leftmost image shows the benefits of using stochastic rasterization for motion and defocus blur – notice, for example, the correctly blurred/sharp reflections. However, the cost of pixel shading using multisampled antialiasing (MSAA) in current GPUs rises substantially when blur is added (as visualized on the top right). We propose a novel graphics architecture that combines decoupled sampling [Ragan-Kelley et al. 2011] with a new tiled deferred shading approach to provide very low and stable shading rates (see bottom right). The scale ranges from 1–64 shader executions per pixel, and 16 samples/pixel were used. The scene is courtesy of Epic Games, Inc.*

Abstract

Stochastic sampling in time and over the lens is essential to produce photo-realistic images, and it has the potential to revolutionize real-time graphics. In this paper, we take an architectural view of the problem and propose a novel hardware architecture for efficient shading in the context of stochastic rendering. We replace previous caching mechanisms by a sorting step to extract coherence, thereby ensuring that only non-occluded samples are shaded. The memory bandwidth is kept at a minimum by operating on tiles and using new buffer compression methods. Our architecture has several unique benefits not traditionally associated with deferred shading. First, shading is performed in primitive order, which enables late shading of vertex attributes and avoids the need to generate a G-buffer of pre-interpolated vertex attributes. Second, we support state changes, e.g., change of shaders and resources in the deferred shading pass, avoiding the need for a single über-shader. We perform an extensive architectural simulation to quantify the benefits of our algorithm on real workloads.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: decoupled sampling, tiled deferred shading, stochastic rasterization, graphics processors

Links: DL PDF

ACM Reference Format
Clarberg, P., Toth, R., Munkberg, J. 2013. A Sort-based Deferred Shading Architecture for Decoupled Sampling. ACM Trans. Graph. 32, 4, Article 141 (July 2013), 10 pages. DOI = 10.1145/2461912.2462022 http://doi.acm.org/10.1145/2461912.2462022.

1 Introduction

In recent years, there has been a strong focus on research and development of more power efficient graphics processors. This is driven by the popularity of battery-operated devices (e.g., phones, tablets, and laptops) and the introduction of very high resolution displays. At the same time, to fulfill the ever-present goal of better visual quality and realism, it is desirable to evolve the hardware rasterization pipeline to natively support stochastic sampling. This would enable accurate motion blur and depth of field [Akenine-Möller et al. 2007] (see Figure 1), as well as many other applications [Nilsson et al. 2012].

To make stochastic rasterization possible under the constraints of low power consumption, it is critical to keep the rasterization, shading, and memory bandwidth costs at a minimum, for example, through efficient hierarchical traversal [Laine et al. 2011; Munkberg et al. 2011; Munkberg and Akenine-Möller 2012], decoupled sampling of shading and visibility [Ragan-Kelley et al. 2011], and the use of efficient buffer compression methods [Hasselgren and Akenine-Möller 2006; Andersson et al. 2011], respectively. Nevertheless, despite these recent innovations, supporting general five-dimensional stochastic rasterization in a power-constrained device is challenging due to its higher complexity and less coherent memory accesses than traditional non-stochastic rendering.

We propose a novel hardware architecture for efficient shading that supports decoupled sampling [Ragan-Kelley et al. 2011]. Decoupling the shading from visibility is a necessity for efficient stochastic rendering, as conventional multisampling antialiasing (MSAA) [Akeley 1993] breaks down with increased blur [McGuire et al. 2010; Munkberg et al. 2011]. Previous state-of-the-art solutions [Burns et al. 2010; Ragan-Kelley et al. 2011] have used caching mechanisms to enable shading reuse between visibility samples. Instead, we store a compact *shading point identifier* (SPID) alongside each visibility sample, and replace the caching by an explicit tiled sorting step to extract coherence, while keeping all data on chip. This reduces hardware complexity and allows shading to be implicitly deferred until after rasterization, ensuring

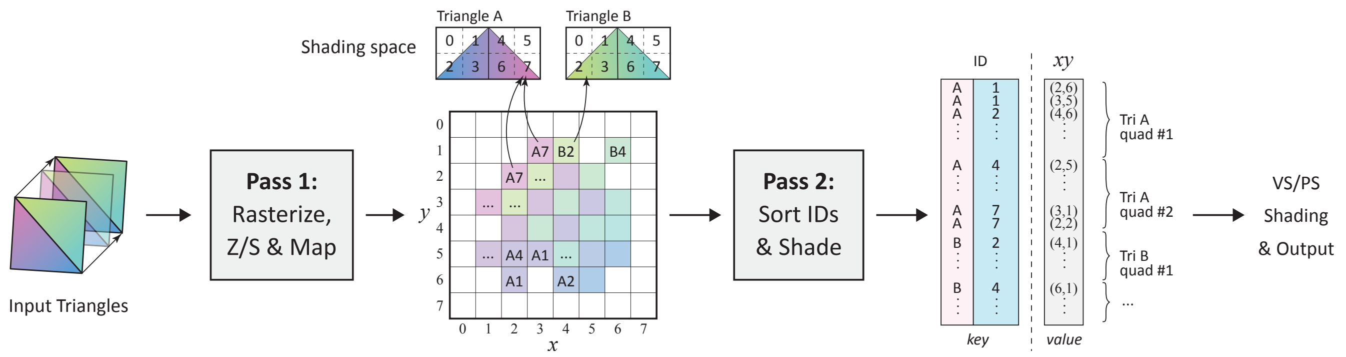


Figure 2: *Our algorithm operates in two main passes. First, all opaque geometry is stochastically rasterized using z-buffering. Instead of executing pixel shading, each visibility sample is mapped to shading space to compute a shading point identifier (SPID), which is stored to the frame buffer. Second, all SPIDs are sorted, with the shading point as key and the frame buffer sub-pixel position (x, y) as value. This generates a long list of coherent shading work, where triangles appear in submission order and shading quads are dispatched according to a Morton order space-filling curve in shading space (due to our encoding of SPIDs, shown in Figure 4) to improve texture locality. Pixel shading is performed on 2×2 pixel quads in order to allow finite differences to be used for texture derivatives, as usual. The result is written out to the list of sub-pixel positions associated with each shading quad. In practice, this second pass is tiled to keep the data on chip while sorting and shading. In this pass, we also render alpha tested or blended geometry, and perform fullscreen passes.*

that only samples visible in the final image are shaded. Figure 2 shows an overview of our algorithm. As we will see, the SPID data is highly coherent. Therefore, we devise two simple lossless buffer compression methods that drastically reduce the bandwidth required to store/load SPIDs. The SPIDs have also been designed so that our sorting step generates shading work in primitive submission order with good spatial locality. Hence, even if shading is deferred, it appears to the application as tiled forward rendering, where state and shader changes are allowed.

Several recent papers exploit tiled forward rendering to cull shading work [Olsson and Assarsson 2011; Olsson et al. 2012; Harada et al. 2012]. These methods should be seen as complements to our technique as they reduce the cost of executing the pixel shader, while we focus on reducing the number of shader executions. Other methods for reducing the shading cost, such as adaptive shading rates [Vaidyanathan et al. 2012] are also well-suited to integrate in our architecture.

A key difference to previous decoupling techniques [Ragan-Kelley et al. 2011; Liktor and Dachsbacher 2012] is that we do not use a memoization cache during the main rendering pass, but instead sort the shading points afterwards. This avoids the variability and buffering requirements introduced by a cache. We also do not need to execute a pixel shader to generate data for deferred shading; our main rendering pass can be implemented entirely in fixed-function hardware and proceeds without delays. To make sorting and shading efficient, we operate on tiles of SPIDs, which are loaded into on-chip memory. This gives many of the benefits of sort-middle binning/tiling architectures, while avoiding the complexities and variable memory requirements of triangle binning [Seiler et al. 2008]. Indeed, our architecture has a fixed memory footprint and predictable performance, as the number of SPIDs is constant and no additional data is stored. We also support late shading of vertex attributes only for visible triangles.

To evaluate the performance, we have built a detailed architectural simulator that can analyze complete Direct3D 11 workloads. The simulator includes a state-of-the-art hierarchical stochastic rasterizer, decoupled sampling with adaptive shading rates [Vaidyanathan et al. 2012], and cache simulators to measure depth and color buffer bandwidth usage, including compression. To the best of our knowledge, this represents the most accurate architectural simulation of stochastic rasterization and decoupled sampling to date.

Our contributions can be succinctly summarized as:

- A novel tiled deferred shading architecture that natively supports decoupled sampling.
- An extensive architectural simulation of several different pipelines for efficient stochastic rasterization and shading.

The rest of the paper is organized as follows. In Section 3, we will give a more thorough overview of the system, followed by details in Section 4. Then, the architectural simulation and results are discussed in Section 5 and 6, respectively, with some concluding remarks in Section 7.

2 Related Work

We focus on graphics hardware pipelines that support workloads of varying triangle sizes and perform shading during or after visibility determination. As such, we will not discuss Reyes-style micropolygon pipelines [Cook et al. 1987], where shading reuse is supported by shading at the vertices prior to rasterization.

In traditional forward rendering pipelines, low shading rates can be achieved by inserting an extra z-prepass, i.e., all opaque geometry is rasterized first to prime the z-buffer and then a second time to shade non-occluded samples. However, this comes at a high cost as it nearly doubles the already high cost of stochastic rasterization, and the geometry pipeline has to be executed twice. On the other hand, conventional deferred shading [Deering et al. 1988; Saito and Takahashi 1990] naturally avoids shading occluded geometry. However, those methods rely on storing a large G-buffer (with normals, texture coordinates etc.) that is shaded by a single über-shader, and they do not support decoupled sampling. Liktor and Dachsbacher [2012] remove this limitation by storing an indirect G-buffer. Similar to us, they store per-sample SPIDs, but their method has the additional overhead of generating G-buffer entries. In addition, they rely on a z-prepass or a compaction step to remove occluded samples, which we do not need; our SPIDs are self-contained, i.e., they uniquely identify the primitive and location in shading space.

Tiled rendering [Fuchs et al. 1989] has been used in many previous systems to keep data on chip and reduce expensive memory bandwidth. To determine what work needs to be performed for each tile, a binning (sort-middle) step is often used, where the geometry

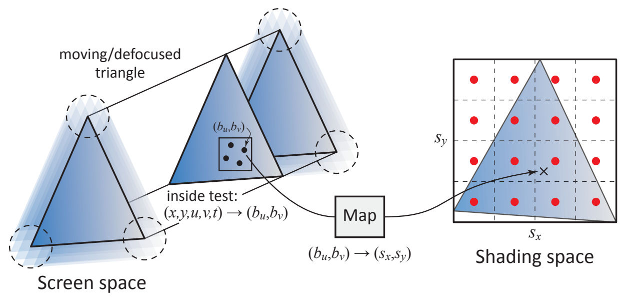

moving/defocused triangle
(b_u,b_v)
inside test: $(x,y,u,v,t) \rightarrow (b_u,b_v)$
Map
$(b_u,b_v) \rightarrow (s_x,s_y)$
s_y
s_x
Screen space
Shading space

Figure 3: *We use decoupled sampling [Ragan-Kelley et al. 2011] to transform visibility samples (left) from barycentric space to shading space (right), where the shading coordinates, (s_x, s_y), are quantized to the nearest shading point (red). Many visibility samples map to the same shading point, enabling efficient shading reuse.*

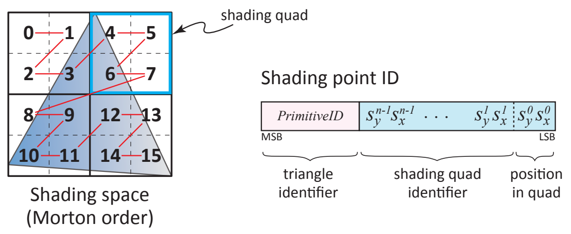


Figure 4: *The higher part (MSB) of a shading point identifier indicates which primitive the shading point belongs to, while the lower part stores the quantized position in shading space, (s_x, s_y), using n bits per x, y-component encoded in Morton (Z-)order, i.e., their bits are interleaved, where s_x^i denotes the i^{th} bit of s_x. In this encoding, the two LSBs represent the position in the 2×2 shading quad, while the higher bits uniquely identify the shading quad. By sequentially sorting all SPIDs, shading quads are found in primitive order with all samples that belong to the same quad packed together, as shown on the right in Figure 2.*

is preprocessed and stored to variable length per-tile work queues. Our architecture differs in this regard, as our main rendering pass is *not* tiled and we do not need binning, while the shading pass is tiled to give the same benefits; our sorting step produces a work queue, but in our case consisting of shading quads rather than triangles. In commercially available graphics processors, the PowerVR architecture appears to be the one that is most closely related to our work as it performs tiled deferred rendering [Imagination Technologies Ltd. 2011], although it does not support decoupled sampling.

There are few architectural studies of hardware systems for stochastic rasterization and shading. Akenine-Möller et al. [2007] present simple bandwidth measurements, but at the time, no efficient shading or rasterization methods were known. Ragan-Kelley et al. [2011] present the most complete study for decoupled sampling, which focuses on its shading cost, but also includes bandwidth simulations. Munkberg et al. [2011] measure depth buffer bandwidth, but their method only supports motion blur. None of the papers have used buffer compression, which is important to reduce the bandwidth for stochastic rendering [Andersson et al. 2011]. Our measurements include both depth and color buffer bandwidth, with occlusion culling and lossless buffer compression, and we use a state-of-the-art hierarchical 5D rasterizer to get a coherent screen-space traversal order.

3 Overview

Figure 2 summarizes our algorithm (see also Figure 7 for details). The input is triangles with per-vertex motion (t) and/or lens shear (u, v) parameters. For each sample that hits the moving/defocused triangle, the rasterizer computes barycentric coordinates, (b_u, b_v), and depth, z. Samples that pass the depth test are mapped to shading space exactly as in previous work [Ragan-Kelley et al. 2011]. This is illustrated in Figure 3. The choice of shading space is orthogonal to our algorithm. We generally use either a canonical screen-space aligned pixel grid over the triangle at the center of lens and time, which gives results equivalent to MSAA for non-blurred geometry, or a more advanced method for adapting the shading grid to the local frequency content [Vaidyanathan et al. 2012].

For each mapped visibility sample, a shading point identifier (SPID) that globally and uniquely identifies the combination of primitive and shading coordinate is computed, and written to the frame buffer. Hence, when the first pass finishes, the SPIDs stored in the frame buffer represent the sparse set of primitives and associated shading quads that will be needed to compute the color of each sample that is visible in the final image. Note that the same identifier will (hopefully) be assigned to many visibility samples, as this ensures the same shaded value will be reused.

Although there are many duplicates, matching SPIDs will generally be spatially spread out depending on the amount of blur and the geometry. Therefore, to generate a compact list of coherent shading work in the second pass, we start by sequentially *sorting* the SPIDs using radix sort. Sorting is somewhat of a brute force solution, but attractive from a hardware point of view. Since the working set is a fixed, constant number of SPIDs, the sort algorithm can be efficiently implemented as a hardware unit operating against a dedicated on-chip memory buffer. The resolution and target frame rate determine its necessary throughput. In our simulations, we assume screen-space aligned tiles of a certain size, e.g., 128×128 pixels, although other partitionings are possible.

Due to the sorting step, the encoding of the SPIDs is a critical component — it determines in which order shading work is generated. With the encoding shown in Figure 4, primitives are shaded in submission order using a space-filling curve in shading space to maximize texture and shading locality, which has been our goal. The algorithm proceeds by scanning the sorted list of SPIDs using a simple state machine; whenever a new primitive is found, its vertex attributes are shaded and interpolation setup performed (only the positions were needed before). This is very similar to how the standard 3D pipeline normally operates. For each unique shading quad found, the pixel shader is dispatched to compute its colors. This step also reuses existing hardware. The main difference is that the rasterizer is inactive (or free to work on other tasks), since we obtain our shading work from the sorted list of SPIDs instead.

Whenever a quad finishes pixel shading, its result is scattered to the list of visibility samples associated with the shading quad. These are found in order as a payload to the sorted shading points (c.f., the right side of Figure 2). The hardware thus only needs to keep track of a starting pointer and a counter for each shading quad in flight. Also note that since all visibility samples are known prior to shading, it would be possible to let the shader access this information in order to perform per-sample operations. This is something that is not possible in previous cache-based decoupled sampling methods, as they dispatch shading immediately upon cache misses.

To keep external memory bandwidth usage at a minimum, the tile populated with colors is kept on chip until it is ready to be resolved to a single-sampled surface and written out to main memory. This summarizes the key ideas of our architecture. In the following sections, the implementation and simulation of the complete system is described, including some extensions.

4 System Details

In this section, we will discuss the details of how the ideas described in Section 3 are transformed into a practical system that supports real workloads. To help the reader, Figure 7 shows an architectural comparison between the standard forward rendering pipeline, and both cache-based and our sort-based architecture for decoupled sampling. We denote the first phase of our algorithm the *frontend*, and the second phase the *backend*.

4.1 Frontend Pipeline

All three pipelines share nearly the same geometry pipeline with a standard input assembler (IA) and vertex shader (VS). They also share the same rasterizer and depth/stencil unit (Z/S). It is also possible to support a geometry shader and tessellation (not shown in the figure). The only difference is that our architecture only needs to compute the vertex *positions* in the frontend. It may therefore be worthwhile to compile a specialized position-only vertex shader for this purpose, where the computations of other vertex attributes have been removed. In the decoupled pipelines, each visibility sample is mapped to shading space, but after this, the functionality diverges; we compute and store SPIDs for all samples. The bandwidth usage for this, although writing multisampled data, is not particularly high as the SPIDs are highly coherent and therefore easy to compress.

4.2 Buffer Compression for Shading Point IDs

Lossless buffer compression is commonly used to reduce bandwidth usage in graphics processors (see the survey by Hasselgren et al. [2006] for an introduction). The data is typically arranged in blocks of a certain size, which are stored uncompressed in the L1 cache. When a block is evicted, it is speculatively compressed, or stored uncompressed to memory (or to the next level cache) if the compression fails to reduce its size. By operating over blocks that are a multiple N of the bus width, it is thus possible to achieve a maximum compression ratio of N:1. The configurations we use are listed in Section 5.3. Note that the buffer is always allocated at its uncompressed size, but bandwidth usage to it is reduced.

We encode SPIDs using 32 bits each for the primitive IDs and Morton order shading coordinates, which allows up to $4.3 \cdot 10^9$ primitives per frame (assuming a sequential assignment of IDs) and a $64\text{k} \times 64\text{k}$ pixel shading grid. The exact widths of these fields are not critical as it is the entropy of the data, not its size, that determines the actual bandwidth usage. We store the two values in separate 32-bit uint buffers, which are individually compressed.

The characteristics of the data differ a bit; the primitive IDs are by nature very coherent, as a block often contains samples from adjacent primitives, in which case the primitive IDs span only a small range. An exception is when a block contains a surface that only partially covers the background. In this case, there are two (or more) distinct ranges of IDs. Inspired by the success of *offset* compression for both depth and color data [Hasselgren and Akenine-Möller 2006; Rasmusson et al. 2007], we found that a simple min/max offset codec (see Figure 5 left) often works very well. The shading coordinates are also quite coherent thanks to our Morton order encoding, but there is a risk of larger gaps. However, as decoupled sampling maps many visibility samples to the same shading coordinate, there are in many cases a limited set of unique shading coordinates within a block, especially if adaptive sampling [Vaidyanathan et al. 2012] is used to reduce shading in blurry regions. To handle this, we have designed a codec that stores a *palette* of the unique values in the block, and uses a few per-sample index bits to address into this. The palette itself is offset-encoded to save bits (see Figure 5 right).

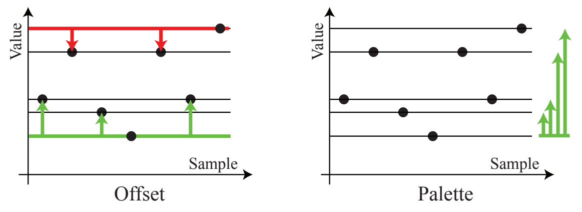


Figure 5: *The left figure illustrates a min/max offset codec, which handles compression of the primitive IDs particularly well. For shading coordinates, we have designed a simple palette-based offset codec, which stores per-sample indices into a small palette of offset-encoded unique values.*

The two codecs complement each other well, as the palette codec handles blocks with few values but large ranges, while the min/max offset codec handles blocks with many values in small ranges. In our suite of test scenes, their combination reduces the bandwidth usage for SPID data to 35–40% of the uncompressed bandwidth, i.e., effectively transferring 22–26 bits per SPID, on average.

4.3 Tiled Shading Backend

Our tiled shading backend performs the operations described in the lower half of Figure 7c. The input is a tile of the fullscreen SPID buffer generated in phase one, which is sorted, scanned, and shaded.

Sort The sort algorithm is intended to be implemented as a fixed-function unit that operates against a dedicated on-chip buffer. Each key-value pair consists of the SPID as key (64 bits) and the sample position within the tile as value (m bits). For example, with 128^2 pixel tiles and 16 samples per pixel, $m = 18$. The tile buffer thus needs to hold 2.6 MB if sorting is done in-place, and the total amount of data that needs to be sorted at 1280×720 pixels resolution is 144 MB (uncompressed). Our algorithm is not sensitive to the choice of sort algorithm, i.e., it does not have to be comparison-based or stable, as the order of samples with matching SPIDs does not matter. The memory access patterns are more important, as it is desirable to minimize the *on-chip* bandwidth usage to the tile buffer.

In a prototype implementation, we use least-significant digit (LSD) radix sort with digits of $k = 8$ bits. The streaming nature of radix sort allows buffer compression to be used also in this step. For each k bits of the sorting key, the algorithm makes a first pass to build a histogram, and a second pass to shuffle the data according to a prefix sum over the histogram. By choosing a memory layout partitioned into k-bit planes, the first pass can thus be implemented to only read the specific data it needs for histogram computation. As a further optimization, when loading a tile from memory, it is possible to detect the number of significant bits in the primitive ID and shading coordinate fields. This is determined by the number of primitives and their size in shading space. Very often the higher order bits are not used. In practice, we therefore need 4–5 sort passes, which with compression consume in the order of 0.5 GB of on-chip memory bandwidth (read+write) per frame. The main drawback of this approach is that it requires a second on-chip tile buffer to ping-pong the data during sorting. In the future, it would be interesting to explore in-place sort algorithms suitable for hardware.

Scan After sorting the SPIDs, the list is sequentially scanned to find shading work. Conceptually, this is implemented as a simple state machine that looks for new primitives and/or shading quads. For each new primitive, the hardware needs to lookup the associated vertices in order to perform vertex position and attribute shading. If there was only a single draw call, this would simply be a matter of accessing the corresponding vertex and index buffers at the right position. However, it is our goal to support complete modern graphics APIs, e.g., Direct3D 11, where a command buffer specifies all rendering operations and their associated state.

We propose a simple solution, where the command buffer is reprocessed for each tile, but the hardware skips state changes and draw calls that are not needed for the current tile. For example, if no SPIDs with primitive IDs within the range of a draw call are found, it is skipped, otherwise the relevant vertices are shaded (via a standard vertex cache to enable reuse between primitives). See the flowchart in Figure 6. Another option is that the application/driver creates optimized per-tile command buffers. In this case, the application is aware of the fact that the hardware internally performs tiled rendering and can exploit this by providing specialized shaders to cull shading work etc., similar to recent tiled forward rendering methods [Olsson and Assarsson 2011; Harada et al. 2012].

Things get slightly more complex if the frontend contains geometry expansion, e.g., tessellation. In this case, a single global primitive ID is not appropriate, and it is better to define the SPIDs based on draw call, patch, and triangle-in-patch IDs. This enables retessellation of patches with one or more visible samples in the backend. We include one test scene with tessellation to demonstrate the feasibility of this. However, there is room for improving the process by using more advanced bookkeeping mechanisms.

Pixel Shading All pixel shading is dispatched as 2×2 shading quads to support shader derivatives through finite differences, which is standard in current GPUs. In the standard pipeline, shading quads directly map to quads on the screen, while with decoupled sampling they exist in shading space. In the latter case, the shading coordinates first have to be mapped back to barycentrics before attribute interpolation [Ragan-Kelley et al. 2011]. With our algorithm, shading quads will be dispatched according to a Morton order space-filling curve in shading space, while with cache-based decoupling, they appear in a more unpredictable fashion. Also, since we are already operating on tiles, it is natural to keep the pixel shader output in an on-chip buffer until we are done with rendering the tile. This drastically reduces the memory bandwidth usage for our architecture. The size of this buffer depends on the tile size, sampling rate, and color buffer format. For example, with a 16-bit RGBA float format and 128^2 pixel tiles at $16\times$, the output buffer is 2.0 MB, which seems reasonable given the large savings it provides. In our implementation, this fits within the extra tile buffer that was needed for radix sort, which is no longer used after sorting.

Note that, since each shading quad has an associated list of visibility samples, we can support *per-sample* operations even with decoupled sampling enabled. One example would be to access per-sample/pixel data structures using general read/write accesses or atomic operations, i.e., unordered access views (UAVs) in Direct3D 11. This feature would be difficult to support with cache-based decoupled sampling [Ragan-Kelley et al. 2011] since the exact set of visibility samples is not known at the time of shading.

Alpha Blending and Post-Processing After the backend completes shading of the SPIDs, the output buffer holds the color of all non-transparent geometry. Note that since frame buffer blending operations cannot be supported with deferred shading (as only the frontmost visible surface is shaded), any additional passes that require alpha blending must be rendered last. This is done tiled, on top of the output buffer, using existing forward rendering methods [Ragan-Kelley et al. 2011]. These passes may include, for example, foliage, decals, lens flare, and other post-processing effects. In this category, we also include any passes that change the output coverage mask, e.g., through alpha testing or alpha-to-coverage.

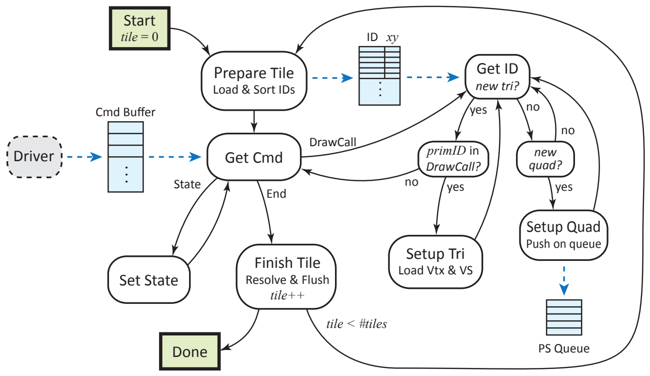


Figure 6: *The flowchart describes the operations performed in the shading backend. Black arrows represent control flow, while blue denotes data. Since all shading work (within a tile) follows in primitive submission order, we support state changes (e.g., shaders, resources etc.), unlike traditional deferred rendering. For each draw call, we proceed with shading its sparse set of vertices and shading quads, before switching back to looking for new commands. The command buffer is provided by the graphics driver, as usual.*

4.4 Discussion

An alternative to our tiled deferred shading would be to perform a z-prepass before rendering the main pass with shading enabled. However, as stochastic rasterization is still quite computationally expensive, both due to the high sampling rates and the relatively high cost of 5D sample tests, we believe it is desirable to avoid any extra rasterization passes. To reach the same low bandwidth usage, the complexity and variable memory footprint of triangle binning also has to be taken into account. Another alternative would be to use a shading cache instead of an explicit sorting step in a deferred architecture like ours. The drawback would be that shading quads are generated in an arbitrary order from many triangles in parallel, making state changes (e.g., swapping shaders), late vertex attribute shading, and interpolation setup difficult or impossible.

Conceptually, our deferred shading pass operates very similar to the current graphics pipeline, with the difference that the triangle traversal is replaced by sequentially scanning a sorted list of shading points. Both vertex shading, interpolation setup, and pixel shader dispatch operate nearly identically to the current pipeline. In some aspects, our architecture may even simplify the pipeline. For example, during rasterization of the main pass, we do not have to worry about pixel shader execution, making a streamlined implementation easier. The cost of adding hardware for the fixed-function sorting unit and associated buffers is offset by the large decrease in off-chip bandwidth. In summary, we believe that our sort-based tiled deferred architecture provides many unique benefits that are otherwise hard to get.

5 Architectural Simulator

We have implemented an architectural simulator based on a feature-complete Direct3D 11 compliant software driver, which has been extended to support hierarchical stochastic rasterization and decoupled sampling. The simulator has been instrumented to measure the rasterization and shading costs, as well as extended with cache simulators for frame buffer memory bandwidth measurements. We can simulate all three graphics pipelines shown in Figure 7. The simulator also supports adding an automatic z-prepass and tiled rendering. In the following, we will discuss the different parts of the simulator, in order to make our results reproducible.

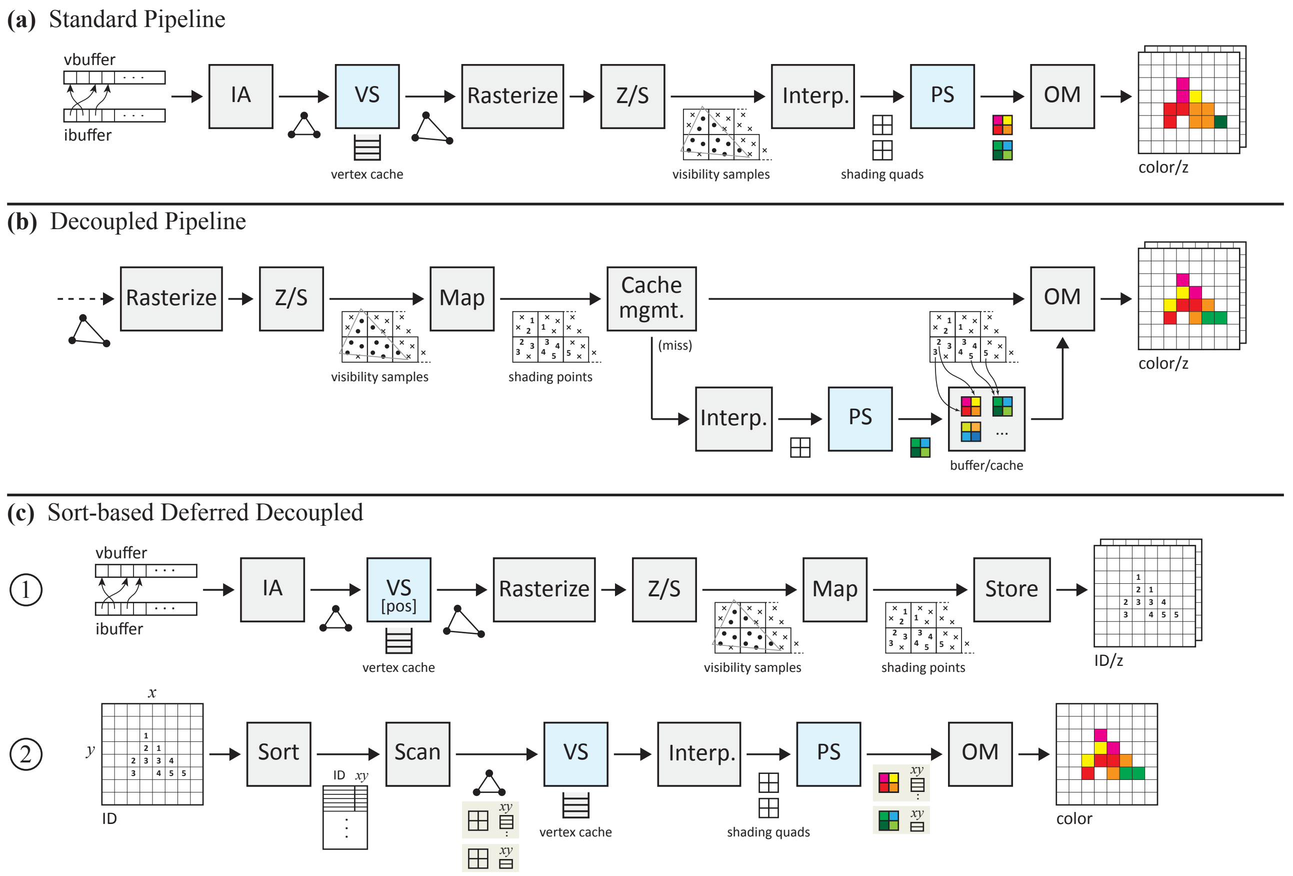


Figure 7: *In **(a)** the standard forward rendering pipeline with MSAA of current GPUs is shown. This is extended in **(b)** with cache-based decoupled sampling [Ragan-Kelley et al. 2011], which shares the same geometry pipeline, but adds the ability to map visibility samples to shading space, and cache logic to lazily shade and reuse shading in the backend. Our architecture in **(c)** splits the work in two phases, where all rasterization and mapping of samples takes place in the first phase, and all pixel shading (PS) in the second. In our simulations, we run the first phase only once (fullscreen) and the second phase tiled to save off-chip bandwidth for the color buffer, but other configurations are possible. The key innovation is the ability to sort and scan the generated SPIDs, which enables deferring vertex attribute shading, triangle interpolation setup, and most importantly, pixel shading to after visibility determination. Note that, thanks to the sorting step, each shading quad has an associated array of unique visibility sample positions, so the output merger (OM) does not have to deal with write conflicts.*

5.1 Rasterization

We use a state-of-the-art hierarchical stochastic rasterizer that efficiently reduces the number of sample coverage tests and provides cache coherent traversal. As part of the triangle setup, we perform motion and defocus aware view frustum culling [Laine and Karras 2011b] and backface culling [Munkberg and Akenine-Möller 2011]. During traversal, we cull moving and defocused triangles against each screen space tile using using the two tests introduced by Laine et al. [2011] and a 5D linearized edge equation test [Munkberg and Akenine-Möller 2012]. Tiles are traversed in Morton order for good spatial locality, and we use two hierarchical levels: 8×8 pixels and 2×2 pixels. The rasterizer uses an optimized 5D sample test [Laine and Karras 2011a] and supports arbitrary sampling patterns in $xyuvt$. Currently, a 5D Sobol pattern [Joe and Kuo 2008] is used.

As an example, averaged over the 3000 frames of the CITADEL animation (see Section 6) rendered at 16 samples/pixel, the operations are divided between triangle setup (2%), tile tests (15%), sample coverage tests (68%), and barycentric and depth interpolation for sample hits (15%). The rasterizer has an average sample test efficiency (STE) of 56% in this example, and the average total cost is 4.5 Gops/frame.

5.2 Pixel Shading

In our simulator, shaders are specified as Direct3D 11 bytecode (compiled from HLSL by the application). For the purpose of gathering statistics, each shaded quad is counted as four shader executions, i.e., four shaded pixels, and we measure all costs at this granularity as the hardware cannot shade partial quads. To generate false color images to illustrate the shading cost, we divide the cost by the number of visibility samples each shading quad is used for, and scatter the resulting cost to those samples.

5.3 Bandwidth Simulation

We focus on measuring the total *frame buffer* memory bandwidth, i.e., the bandwidth to all depth and color buffers. This is done by tracking all memory accesses to those resources, including accesses due to non-stochastically rendered geometry and fullscreen post-processing passes. The SPID data in our algorithm is stored as two color buffers in 32-bit uint format, as discussed in Section 4.2.

The simulator uses separate L1 caches for depth and color data, as is common practice. To get realistic bandwidth estimates, both the depth and color pipelines include lossless buffer compression. For this purpose, all depth/color data is arranged in blocks of 256 bytes

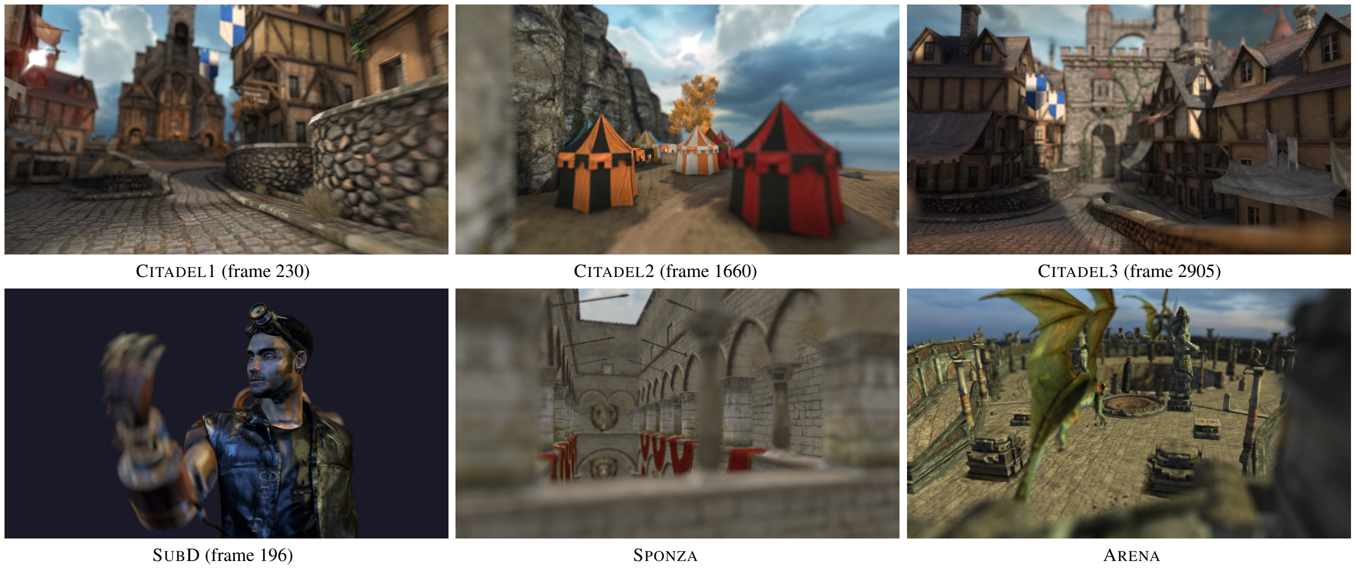


Figure 8: *Our selection of test scenes, to which we have added motion and defocus blur;* CITADEL1–3 *are three different frames from a benchmark for Unreal Engine 3 for mobile devices, courtesy of Epic Games, Inc.* SUBD *is an example from the Microsoft DirectX SDK (June 2010) that uses tessellation,* SPONZA *is the well-known Atrium Sponza Palace by Marko Dabrovic, and* ARENA *is a test scene courtesy of Intel Corporation, featuring animated dragons and more complex shaders.*

and stored uncompressed in the L1 caches. The memory bus is assumed to be 64 bytes wide in all measurements, which enables a maximum compression of 4:1. All parameters are configurable in our simulator, but these numbers are in line with typical hardware architectures and published studies. Note that the size of a block in pixels depends on the memory layout, data format, and number of samples per pixel used for a resource.

Depth Buffer Bandwidth We model a typical depth/stencil testing unit (denoted Z/S in Figure 7) that resembles those of modern graphics processors. For all measurements, we use hierarchical occlusion culling [Morein 2000] with one $z_{\max}$-value per block. This data is kept in a block header, which is stored in a separate small control surface. The header also includes a few control bits and a *clear mask*, i.e., a bit mask that indicates with one bit per sample if the depth value is cleared or has been written. Both of these optimizations significantly reduce the memory read bandwidth as many depth tests can be answered without accessing the main depth buffer. The bandwidth to the control surface is modeled through a separate smaller cache, where multiple block headers are packed into 64-byte cache lines and written uncompressed to memory.

For the main depth data, we use *depth offset* compression [Hasselgren and Akenine-Möller 2006], which has recently been found to work very well for stochastic rendering [Andersson et al. 2011]. Our findings confirm this result. In our implementation, we also take advantage of the clear mask to only encode offsets for non-cleared samples, which helps improve the compression ratios in the beginning of the rendering. The depth data cache was chosen to be 64 kB, following Andersson et al. [2011], while the cache for control surfaces is 32 kB to allow a larger region of the screen to be cached (our headers are also larger due to the clear mask).

Color Buffer Bandwidth In our simulations, render targets are backed by a 128 kB color cache. While even larger caches would be beneficial, the cost of die area, current leakage, and latency tolerance limits the capacity in practice. For each block of color data, two bits of control data are stored in a separate control surface, indicating whether the block is cleared, compressed, or uncompressed. Indicating cleared blocks reduces bandwidth usage at the beginning of rendering to each render target. The control surface constitutes a negligible part of bandwidth usage, and is therefore omitted.

Color data can be compressed with either of three simulated codecs, and compressed blocks begin with four bits indicating which codec was used as well as the compressed size. The primary codec employs offset compression [Rasmusson et al. 2007] using a single reference point. In addition to this, the two codecs described in Section 4.2 are applied to scalar integer buffers. For each evicted block, all applicable encoders try to compress the data, and the smallest of the resulting data blocks is written to memory. This approach is standard practice in buffer compression.

5.4 Limitations and Extensions

Our simulator runs sequentially and does not model the concurrency issues of actual graphics hardware pipelines, e.g., there would normally be multiple shading quads in flight. For this reason, we only simulate the first level of caches, which are accessed by a single unit at the time. However, a real implementation would likely include a memory hierarchy with multiple levels of shared caches. To model this in a meaningful way, we would have to perform memory simulations at all steps of the pipeline, i.e., include vertex, index, constant buffers, and so on. This was left out to limit scope, which seems to be a common tradeoff. In fact, we have not seen any published full system simulations of modern graphics pipelines.

Last, it is unclear how different system costs relate, i.e., rasterization, shading, and bandwidth, as it depends very much on the specific hardware implementation and workloads. Hence, we present the numbers separately to let the reader make their own interpretations. It should be noted that it may be possible to further reduce the bandwidth by, e.g., using time-dependent occlusion culling [Akenine-Möller et al. 2007; Boulos et al. 2010] and better color buffer compression schemes [Ström et al. 2008]. We expect such improvements to have a positive impact on all algorithms.

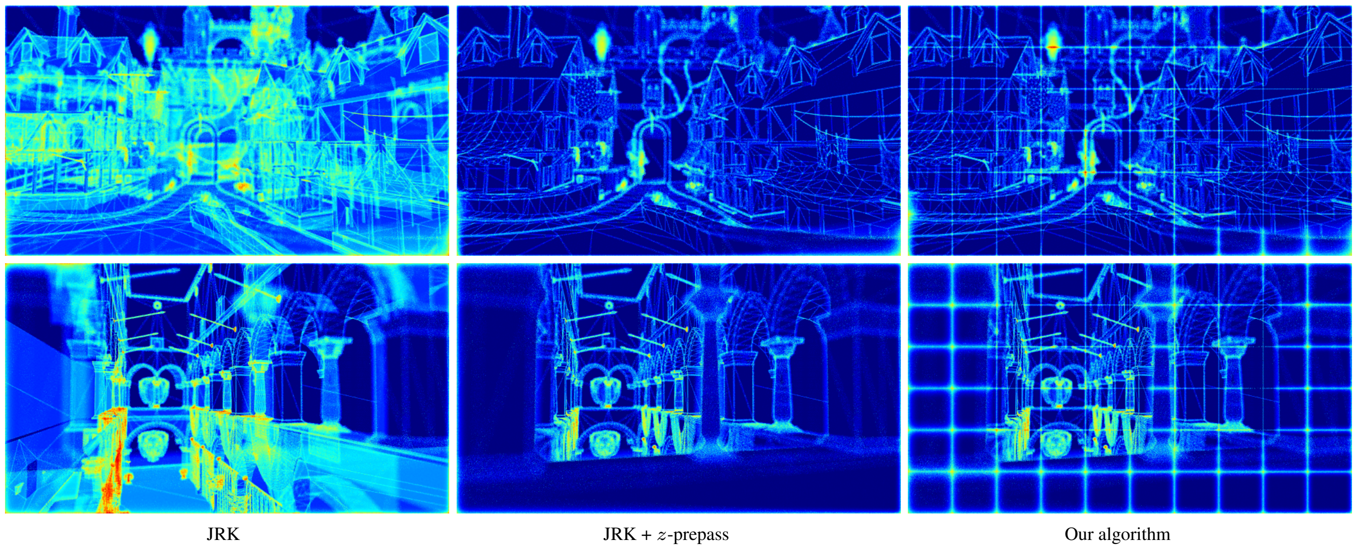


Figure 9: *The average per pixel shading rate for* CITADEL3 *and* SPONZA *using (left) forward rendering with cache-based decoupled sampling (JRK) [Ragan-Kelley et al. 2011], and (middle) the same approach but with a z-prepass added. Our algorithm (right) achieves nearly the same low shading rate, but without the extra rasterization cost and while consuming much less off-chip bandwidth.*

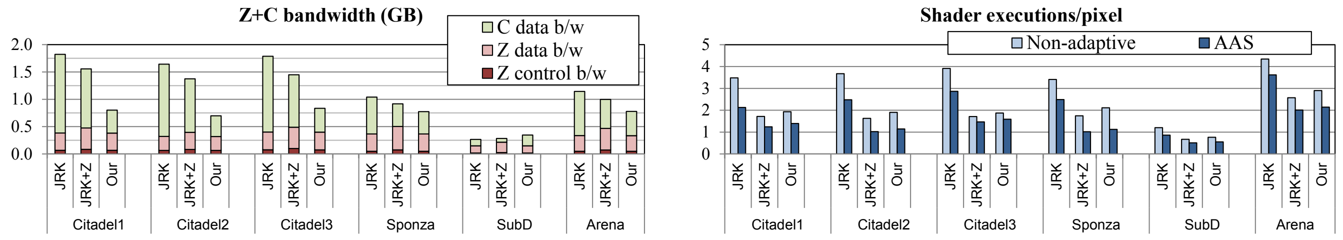


Figure 10: *The average bandwidth usage (left) and per pixel shading rate (right) for our test scenes. Our algorithm consumes significantly less bandwidth by keeping the data on chip and achieves nearly the same low shading rate as decoupled sampling with a z-prepass (JRK+Z).*

6 Results

To quantify the benefits of our algorithm compared to the alternatives, we have simulated several different Direct3D 11 workloads. Figure 8 shows our selection of test scenes. In particular, we emphasize the CITADEL benchmark scene, which during its animation features several interesting environments, both indoors and outdoors, with varying amounts of blur. It also includes multiple post-processing passes for lens flare, fog, ambient occlusion etc., which have been included in our bandwidth measurements.[1]

All results were generated at 1280×720 pixels resolution using 16 stochastic samples in $xyuvt$ per pixel. This sampling rate was chosen to provide acceptable image quality for both motion blur and depth of field, assuming the simple box filter of current GPUs. Better reconstruction filters are expected to lower this requirement, but robust real-time filters for the combined 5D problem is still an area of active research [Shirley et al. 2011; Lehtinen et al. 2011].

We compare our algorithm against cache-based decoupled sampling (JRK) and optionally include a z-prepass (JRK+Z). We have omitted statistics for standard MSAA, as in the presence of blur, MSAA is substantially more expensive than our baseline (JRK). For example, in the CITADEL trace, an MSAA-based approach shades about twice as much as JRK, with a per-frame variation of 3.0–14.1 shader executions per pixel (JRK has 2.4–4.9). The accompanying video shows per-frame shading rate heatmaps (including standard MSAA). Figure 9 shows examples for two frames.

Figure 10 presents the color and depth bandwidth usage for our test scenes. As expected, with a z-prepass (JRK+Z) there is a slight increase in depth bandwidth usage, while the color buffer bandwidth usage is reduced, for a net win. Our algorithm consumes significantly less external bandwidth for most scenes. There are several reasons; first, we avoid the extra z-prepass, and second, the SPIDs are more coherent and easier to compress than color information. Last, and very importantly, the implicit tiling allows the color data to be kept on-chip longer, until after MSAA resolve.

Figure 10 also shows the shading rate, which is almost on par with decoupled sampling with a z-prepass for all scenes. As with any tiled rendering algorithm, there is some degree of bin spread [Seiler et al. 2008], i.e., shading cannot be reused across tile boundaries. This explains the slight increase in shading rate over JRK+Z. We also include *adaptive anisotropic shading* (AAS) [Vaidyanathan et al. 2012] versions of all algorithms, and we see that with AAS enabled, the shading efficiency of JRK+Z and our algorithm is even more similar. Finally, the rasterization cost for each test scene is reported in Table 1. Note that our algorithm shades almost as efficiently at JRK+Z at about half the rasterization cost. The small increase in cost over forward rendering (JRK) is due to our tiled rasterization of alpha blended geometry, which slightly increases the rasterization setup cost.

[1]We have replaced its original post-processing based solution for motion/defocus blur by stochastic rendering.

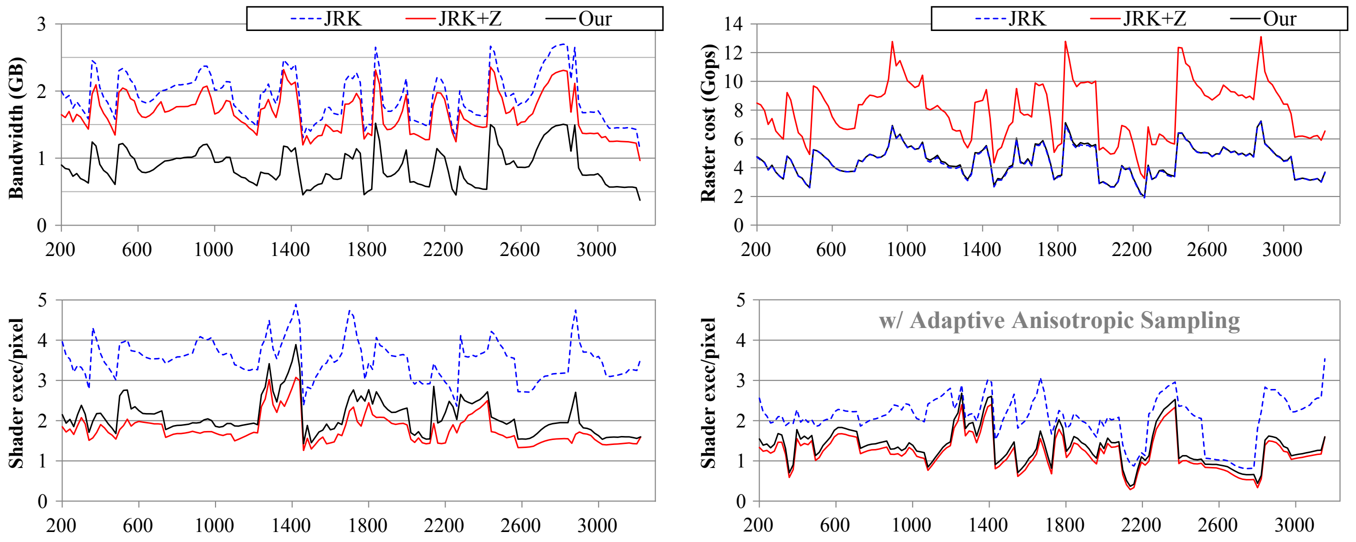


Figure 11: *The full* CITADEL *trace. For every 20th frame, we report bandwidth (C+Z) (upper left), rasterization cost (upper right) and the number of shader executions per pixel (bottom left). Additionally, we show shading statistic with adaptive anisotropic shading (AAS) for all algorithms (lower right). Our algorithm consumes less bandwidth and has low raster and shading cost. It also scales favorably with AAS.*

	CITADEL1	CITADEL2	CITADEL3	SPONZA	SUBD	ARENA
JRK	4.3	4.1	6.1	5.3	5.4	6.6
JRK+Z	7.9	7.5	11.2	10.6	10.9	12.9
Our	4.3	4.2	6.2	5.3	5.4	6.7

Table 1: *The rasterization cost for our test scenes in Gops/frame. Note that JRK+z-prepass has substantially higher cost.*

In Figure 11, we report statistics for the full CITADEL trace. As can be seen, the bandwidth, shading, and rasterization costs are low and relatively stable throughout the animation. It is also interesting to note that AAS efficiently counteracts the bin spread problem, as the surfaces contributing the most to bin spread are also the most sparsely shaded. Figure 12 illustrates the reduction in sensitivity to tile size when employing adaptive shading rates. The effect is visualized in Figure 13.

7 Conclusion

Decoupling of shading and visibility in the graphics pipeline is vital to support stochastic sampling. In this paper, we have explored a novel hardware architecture that gives the benefits of deferred shading, i.e., shading only what is visible, without inheriting the drawbacks of G-buffer based approaches. Our main motivation comes from minimizing the off-chip memory bandwidth usage, which is very expensive in terms of power consumption. We also want to keep the rasterization cost low by avoiding extra z-prepasses, since the cost of stochastic rasterization is still a significant factor (in the range of hundreds of Gops/second). Second, we want to reuse as much as possible of the existing fixed-function units.

The presented architecture reaches these goals by rendering a fullscreen buffer of shading point identifiers, and then working on tiles to defer shading (of both vertex attributes and pixels) until last in the pipeline, while still shading in primitive order. In summary, we believe that our sort-based tiled deferred architecture provides many unique benefits. One obvious next step is to focus on reducing the number of samples per pixel that is necessary for good image quality. Recent work on clever reconstruction filters show great promise in this direction.

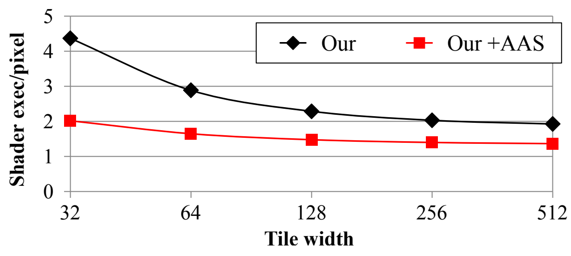


Figure 12: *The shading cost increases for small tile sizes with our algorithm due to bin spread. Note that with adaptive anisotropic shading (AAS), the bin spread effect is largely avoided.*

Acknowledgements

We thank the anonymous reviewers and Intel's Advanced Rendering Technology (ART) team for their valuable feedback. We also thank Aaron Coday, Tom Piazza, Charles Lingle, and Aaron Lefohn for supporting this research, and Nir Benty and Uzi Sarel for their help. Epic Games, Inc., kindly gave us permission to use the CITADEL scene, and SPONZA was created by Marko Dabrovic.

References

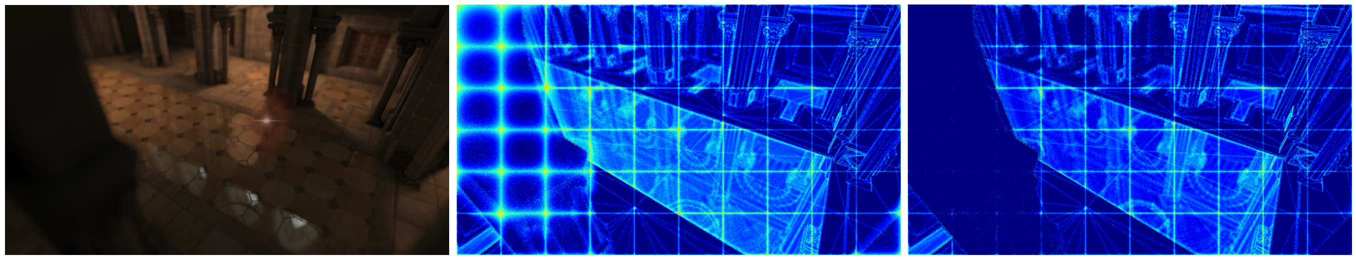
AKELEY, K. 1993. RealityEngine Graphics. In *Proceedings of SIGGRAPH 93*, ACM, 109–116.

AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. 2007. Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware*, 7–16.

ANDERSSON, M., HASSELGREN, J., AND AKENINE-MÖLLER, T. 2011. Depth Buffer Compression for Stochastic Motion Blur Rasterization. In *High Performance Graphics*, 127–134.

BOULOS, S., LUONG, E., FATAHALIAN, K., MORETON, H., AND HANRAHAN, P. 2010. Space-Time Hierarchical Occlusion Culling for Micropolygon Rendering with Motion Blur. In *High Performance Graphics*, 11–18.

BURNS, C. A., FATAHALIAN, K., AND MARK, W. R. 2010. A Lazy Object-Space Shading Architecture with Decoupled Sampling. In *High Performance Graphics*, 19–28.



CITADEL (frame 560) Our algorithm Our algorithm + AAS

Figure 13: *The problem of bin spread that is common in tiling architectures, i.e., that large blurs cause triangles to stride tile boundaries, is almost entirely avoided by using adaptive anisotropic shading (AAS) [Vaidyanathan et al. 2012]. The method combines very favorably with our architecture, and it has no discernible impact on the image quality with the conservative settings we are using. In this example, the average shading rate is reduced by 35% from 2.29 to 1.49 executions/pixel, resulting in tiles with very well-balanced shading work.*

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, ACM, vol. 21, 95–102.

DEERING, M., WINNER, S., SCHEDIWY, B., DUFFY, C., AND HUNT, N. 1988. The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, ACM, vol. 22, 21–30.

FUCHS, H., POULTON, J., EYLES, J., GREER, T., GOLDFEATHER, J., ELLSWORTH, D., MOLNAR, S., TURK, G., TEBBS, B., AND ISRAEL, L. 1989. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System using Processor-Enhanced Memories. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, ACM, vol. 23, 79–88.

HARADA, T., MCKEE, J., AND YANG, J. C. 2012. Forward+: Bringing Deferred Lighting to the Next Level. In *Eurographics 2012 – Short Papers*, 5–8.

HASSELGREN, J., AND AKENINE-MÖLLER, T. 2006. Efficient Depth Buffer Compression. In *Graphics Hardware*, 103–110.

IMAGINATION TECHNOLOGIES LTD., 2011. POWERVR Series5 Graphics – SGX architecture guide for developers.

JOE, S., AND KUO, F. Y. 2008. Constructing Sobol Sequences with Better Two-Dimensional Projections. *SIAM Journal on Scientific Computing, 30*, 5, 2635–2654.

LAINE, S., AND KARRAS, T. 2011. Efficient Triangle Coverage Tests for Stochastic Rasterization. Tech. Rep. NVR-2011-003, NVIDIA Corporation, Sep.

LAINE, S., AND KARRAS, T. 2011. Improved Dual-Space Bounds for Simultaneous Motion and Defocus Blur. Tech. Rep. NVR-2011-004, NVIDIA Corporation, Nov.

LAINE, S., AILA, T., KARRAS, T., AND LEHTINEN, J. 2011. Clipless Dual-Space Bounds for Faster Stochastic Rasterization. *ACM Transactions on Graphics, 30*, 106:1–106:6.

LEHTINEN, J., AILA, T., CHEN, J., LAINE, S., AND DURAND, F. 2011. Temporal Light Field Reconstruction for Rendering Distribution Effects. *ACM Transactions on Graphics, 30*, 55:1–55:12.

LIKTOR, G., AND DACHSBACHER, C. 2012. Decoupled Deferred Shading for Hardware Rasterization. In *Symposium on Interactive 3D Graphics and Games*, 143–150.

MCGUIRE, M., ENDERTON, E., SHIRLEY, P., AND LUEBKE, D. 2010. Real-Time Stochastic Rasterization on Conventional GPU Architectures. In *High Performance Graphics*, 173–182.

MOREIN, S. 2000. ATI Radeon HyperZ Technology. In *Graphics Hardware, Hot3D Proceedings*.

MUNKBERG, J., AND AKENINE-MÖLLER, T. 2011. Backface Culling for Motion Blur and Depth of Field. *Journal of Graphics, GPU, and Game Tools, 15*, 2, 123–139.

MUNKBERG, J., AND AKENINE-MÖLLER, T. 2012. Hyperplane Culling for Stochastic Rasterization. In *Eurographics 2012 – Short Papers*, 105–108.

MUNKBERG, J., CLARBERG, P., HASSELGREN, J., TOTH, R., SUGIHARA, M., AND AKENINE-MÖLLER, T. 2011. Hierarchical Stochastic Motion Blur Rasterization. In *High Performance Graphics*, 107–118.

NILSSON, J., CLARBERG, P., JOHNSSON, B., MUNKBERG, J., HASSELGREN, J., TOTH, R., SALVI, M., AND AKENINE-MÖLLER, T. 2012. Design and Novel Uses of Higher-Dimensional Rasterization. In *High Performance Graphics*, 1–11.

OLSSON, O., AND ASSARSSON, U. 2011. Tiled Shading. *Journal of Graphics, GPU, and Game Tools, 15*, 4, 235–251.

OLSSON, O., BILLETER, M., AND ASSARSSON, U. 2012. Clustered Deferred and Forward Shading. In *High Performance Graphics*, 87–96.

RAGAN-KELLEY, J., LEHTINEN, J., CHEN, J., DOGGETT, M., AND DURAND, F. 2011. Decoupled Sampling for Graphics Pipelines. *ACM Transactions on Graphics, 30*, 3, 17:1–17:17.

RASMUSSON, J., HASSELGREN, J., AND AKENINE-MÖLLER, T. 2007. Exact and Error-Bounded Approximate Color Buffer Compression and Decompression. In *Graphics Hardware*, 41–48.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible Rendering of 3-D Shapes. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, ACM, vol. 24, 197–206.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transactions on Graphics, 27*, 3, 18:1–18:15.

SHIRLEY, P., AILA, T., COHEN, J., ENDERTON, E., LAINE, S., LUEBKE, D., AND MCGUIRE, M. 2011. A Local Image Reconstruction Algorithm for Stochastic Rendering. In *Symposium on Interactive 3D Graphics and Games*, 9–14.

STRÖM, J., WENNERSTEN, P., RASMUSSON, J., HASSELGREN, J., MUNKBERG, J., CLARBERG, P., AND AKENINE-MÖLLER, T. 2008. Floating-Point Buffer Compression in a Unified Codec Architecture. In *Graphics Hardware*, 75–84.

VAIDYANATHAN, K., TOTH, R., SALVI, M., BOULOS, S., AND LEFOHN, A. 2012. Adaptive Image Space Shading for Motion and Defocus Blur. In *High Performance Graphics*, 13–21.