Decoupled Deferred Shading for Hardware Rasterization

Gábor Liktor* Carsten Dachsbacher†
Computer Graphics Group / Karlsruhe Institute of Technology

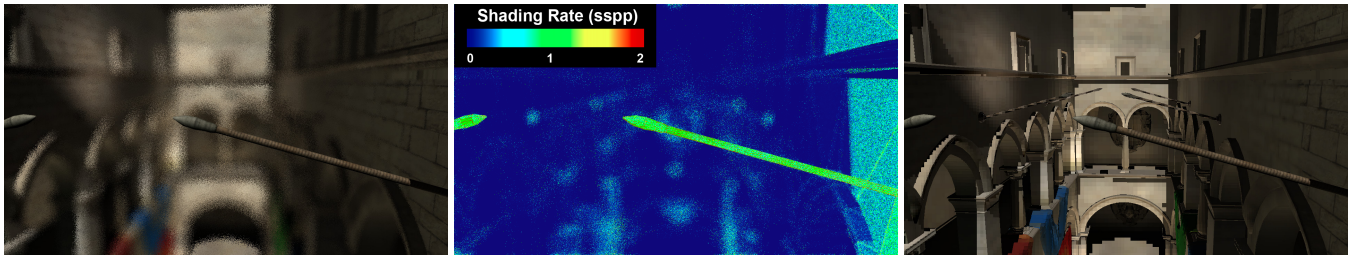


Figure 1: *Decoupled deferred shading enables efficient shading reuse for stochastic rasterization. These images show depth of field rendering with 4× visibility supersampling (left), a visualization of the shading rate (sspp–shading samples per pixel, center), and the same shading as seen from a pin-hole camera (right). Our adaptive scheme reduces the shading frequency of defocused regions.*

Abstract

In this paper we present decoupled deferred shading: a rendering technique based on a new data structure called *compact geometry buffer*, which stores shading samples independently from the visibility. This enables caching and efficient reuse of shading computation, e.g. for stochastic rasterization techniques. In contrast to previous methods, our decoupled shading can be efficiently implemented on current graphics hardware. We describe two variants which differ in the way the shading samples are cached: the first maintains a single cache for the entire image in global memory, while the second pursues a tile-based approach leveraging local memory of the GPU's multiprocessors. We demonstrate the application of decoupled deferred shading to speed up the rendering in applications with stochastic supersampling, depth of field, and motion blur.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: decoupled sampling, deferred shading, stochastic rasterization

1 Introduction

In modern rendering applications, shading is generally the most expensive part of the image synthesis process. To shade only surfaces that contribute to the final image, deferred shading techniques [Saito and Takahashi 1990] store surface data in the geometry buffer (*G-buffer*), and postpone shader evaluation after the visible surfaces are already found at each image sample location. Furthermore, unlike in forward shading methods, all the information is available in the *G-buffer* to execute multiple shading passes without recomputing visibility.

*e-mail: gabor.liktor@kit.edu
†e-mail: dachsbacher@kit.edu

Cinematic effects, like motion blur and depth of field, require stochastic visibility sampling in 5 dimensions (image and lens area, shutter time), using high number of samples to avoid spatial and temporal aliasing. We can generally assume that shading does not vary much during the shutter interval, thus multiple samples might share the same shaded color. The idea of decoupled sampling is to save computations by separating *visibility samples* from *shading samples* and to create a many-to-one correspondence between them [Ragan-Kelley et al. 2011]. Shading is only evaluated when triggered by visibility, but at a lower frequency. As a result, the number of shader evaluations does not scale linearly with the number of visibility samples.

In terms of shading reuse, the former strength of deferred shading becomes its weakness. In the *G-buffer* it is no longer trivial to decide, which visibility samples belong to the same surface, and the memory footprint grows linearly with the number of visibility samples. Consequently, current deferred real-time renderers have reduced antialiasing quality and apply post-processing effects to mimic realistic camera models.

In this paper we extend the deferred shading algorithm in the manner of decoupled sampling to group equivalent visibility samples together and evaluate the shading only once per such group. Our primary contribution is the introduction of a novel data structure, which stores shading samples in a compact form, eliminating duplicates, but keeps the full supersampled visibility information in the framebuffer.

We present two implementations for current rasterization hardware: a method that augments the rasterization process with a global shader cache, and an alternative that eliminates costly global synchronization using multiple rendering passes. We conclude with a performance analysis showing that depending on the relative cost of rasterization and shading, our method can be beneficial on GPUs for high-quality deferred shading.

2 Related Work

Deering et al. [1988] introduced a deferred shading architecture, later generalized by Saito and Takahashi [1990], where "G-buffers" are filled with per-pixel geometric properties such as depth, normals, material information, in a first render pass for later shading. Deferred shading is easy to implement on modern GPUs and nowadays widely used in real-time applications.

However, antialiasing is intricate with deferred shading, as shading inputs must not be filtered. The trivial solution of supersampling the G-buffer leads to tremendous growth in memory consumption and shading costs. Post-processed antialiasing methods aim for the reconstruction of smooth image-space boundaries, while avoiding supersampled shading. Morphological Antialiasing (*MLAA*, [Reshetov 2009]) takes the shaded image as a color buffer and essentially blurs detected edges. Subpixel Reconstruction Antialiasing (*SRAA*, [Chajdas et al. 2011]) also post-processes an image, but uses superresolution depth and normal buffers which enable a more precise detection of geometric boundaries.

Decoupled shading is often used in the context of high-quality (preview) rendering. The LightSpeed architecture for cinematic relighting [Ragan-Kelley et al. 2007] uses an indirect framebuffer for decoupling shading samples from pixels together with a deep-framebuffer to handle antialiasing, motion blur and transparency efficiently; cache compression is reduced to deep-framebuffers. In the same spirit, Hoberock et al. [2009] propose compaction of G-buffers for heterogeneous shader execution. Rendering with Reyes also yields high-quality images, however, typically at high computational cost. To this end, Burns et al. [2010] propose to uniformly sample an object's parametric domain for shading (instead of taking mesh vertices as shading samples) and resolve visibility before shading. Many previous works in this field propose changes to the hardware rendering architecture for (decoupled) shading: Fatahalian et al. [2010] suggest augmenting the GPU pipeline with functionality to gather and merge rasterized fragments from adjacent triangles to reduce overall pipeline shading work. The work of Ragan-Kelley et al. [2011] is closely related and the basis for our work: they propose to use a many-to-one hash from visibility to shading samples and use a buffer to memoize shading samples to reuse them across visibility samples. However, their method is not directly applicable to current graphics hardware.

We demonstrate our method in the context of stochastic rasterization. Cook et al. [1984] introduced distributed ray tracing, which renders effects such as depth of field and motion blur by stochastically generating samples in the spatio-temporal domain. It has always been obvious that this blurring should require less samples. Far later, Egan et al. [2009] described a rendering algorithm for motion blurred scenes, based on a frequency analysis, using adaptive space-time sampling and sheared reconstruction filters to accelerate rendering. Soler et al. [2009] analyzed depth of field to derive the necessary sampling rate for rendering images. Stochastic rasterization [Akenine-Möller et al. 2007; McGuire et al. 2010] transfers this idea from ray tracing to rasterization on graphics hardware. To our knowledge, our method is the first implementation of deferred shading on GPUs which can efficiently shade stochastically rasterized images.

3 Overview of Decoupled Shading

Our algorithm for decoupling shading and visibility samples builds on the work of Ragan-Kelley et al. [2011]. The major difference compared to their approach is that we do not reuse evaluated shading samples during rasterization, but data in the *G-buffer*, which eliminates redundancy before the shading takes place.

We first provide a brief overview of decoupled sampling in a rasterization architecture. The main motivation is to separate the domain of visibility from the domain of shading, so that they can be sampled independently. In case of rasterization, visibility samples are typically fragments covered by triangles in screen space. Instead of per-fragment shading, each visibility sample is mapped to the shading domain and assigned to a shading sample (see Figure 2). If this mapping is a many-to-one projection, shading is efficiently reused among multiple visibility samples. The shading domain can be any parameterization over rendered surfaces, such as screen space coordinates, 2D patch parameters or even the texture domain.

As an example, consider a fast-moving triangle, which is sampled over a finite time interval using stochastic rasterization. A naive algorithm would determine the barycentric coordinates in the triangle for each visibility sample, and evaluate the shading accordingly. In many cases, however, we can assume that the visible color of a surface does not change relevantly over time (e.g. a moderately glossy material). Therefore we can use a rasterized image of the triangle as shading domain at a fixed *shading time*, and by using the barycentrics of the stochastic rays' intersection points, we can find corresponding shading locations for each visibility sample by mapping them to "pixels" of this image.

4 Compact Geometry Buffer

In hardware rasterization pipelines, multisampling generates subpixel samples to supersample the visibility function and avoid image space aliasing. However, shading is usually evaluated at a lower frequency, as it is expected to be already prefiltered on a pixel level. In the terminology used in this paper, a *visibility sample* is where the visibility test happens without shading, and a *shading sample* is a location in shading space where the final shaded color is evaluated. The number of shading samples is determined by the *shading rate*, the density of shader executions per pixel. For example *MSAA* invokes a single shader per pixel, thus has a shading rate of 1. As we demonstrate later, our method also supports adaptive shading rates, where the number of shader evaluations depends on the frequency-content of the image.

Conventional deferred shading methods trivially couple visibility and surface data in the *G-buffer*, resulting in a memory footprint proportional to the number of visibility samples. We introduce the compact geometry buffer (in the following: *CG-buffer*), a decoupled storage for deferred shading. This data structure has the same functionality as the *G-buffer*, storing the necessary inputs of shaders for delayed evaluation. However, it avoids storing redundant data that corresponds to the same surface and results in equivalent shader output.

Instead of storing the shading information in the framebuffer, a visibility sample stores a reference to a shading sample in a compact linear buffer (Figure 4). Multiple visibility samples can reference the same shading sample to allow shading reuse. The size of the geometry buffer does not grow with the supersampling density, but is governed by the shading rate. This eliminates all redundant shader executions that have been the major overhead in image space deferred shading.

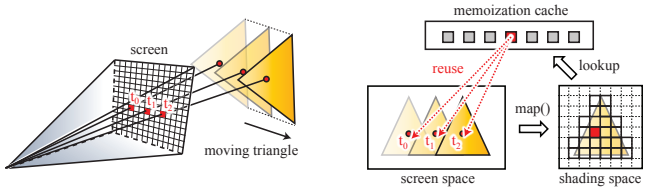


Figure 2: ***(a)*** *A moving surface point is visible at multiple locations over time on the screen. Assuming the time $t_2 - t_0$ is small enough, the same shaded color can be used for all screen samples.* ***(b)*** *Decoupled sampling samples visibility and shading in different domains. The method reuses recent shading samples from a* memoization cache*, mapping multiple visibility samples to a single sample in shading space.*

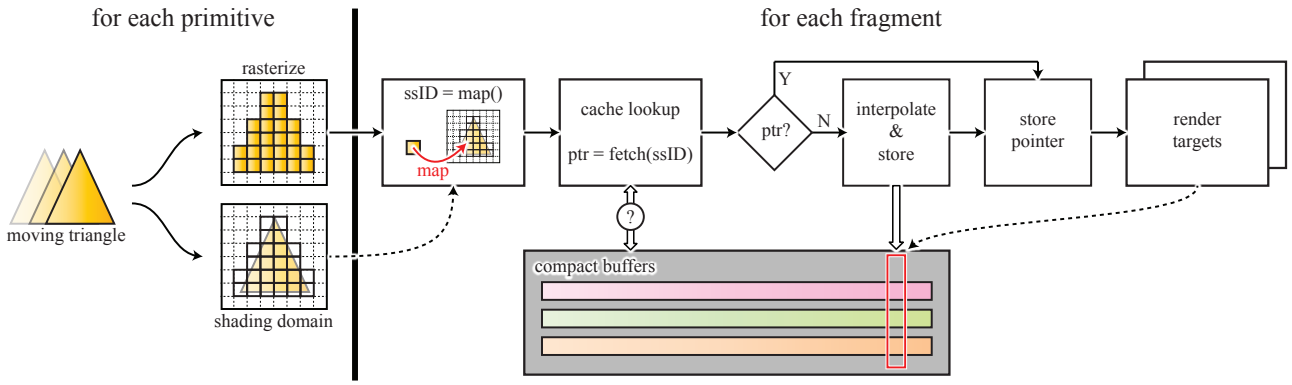


Figure 3: *The outline of decoupled deferred shading in a rasterization pipeline. Prior to rasterization, each primitive is bound and projected to a shading grid. During fragment processing, the fragments are mapped to their corresponding cells on the shading grid. The shading domain and its cell gives a unique key for the shading sample, which we use to ensure that the same shading data is not stored multiple times in the compact buffers. Shading reuse is implemented by referencing the same data from multiple samples in the render targets.*

4.1 Usage

In the manner of deferred shading, the usage of the *CG-Buffer* consists of three main stages. In the *sampling* stage, the visible surfaces are identified, while the necessary shader inputs are filled into the data buffers. Our sampling stage is more intricate, but serves as a basis to accelerate the *shading* stage which is typically the most expensive part of the pipeline. Finally, shaded colors are loaded to visibility samples and filtered at each pixel in the *resolve* pass.

In order to reuse shading samples, we need to identify them with uniqe indices (*ssID*). During rendering, each shading primitive allocates an *ssID range*: a virtual address space that ensures that shading sample indices of different shading domains do not overlap during the frame. This address space does not have any relation to physical memory, but allows unique search keys for the sampler to find existing shading samples in the *CG-buffer*. Staying at the former example of the moving triangle, the *ssID range* would be the number of pixels the triangle would cover in the shading domain. This is an upper bound on the number of potentially generated samples.

If the shading data was found, we only store a pointer to its address in the memory. In case of a miss, we also need to allocate a new slot in the compact buffers and store the data besides referencing. The efficiency of the searching method is crucial for the entire pipeline, as it is executed at every visibility sample. Figure 3 shows the outline of the decoupled sampling process in a rasterization pipeline.

4.2 Integration into Rasterization Pipelines

Modern GPUs use a parallel *sort-last-fragment* architecture for rasterization [Molnar et al. 1994]. Considering fragments as potential shading samples, the *ssID* has to be unique during the entire rendering pass, as fragments are processed out-of-order, and multiple triangles can be rasterized at the same time. On the other hand, the hardware contains a very limited number of rasterizer units and one such unit completely processes a given triangle before taking the next one.

Ragan-Kelley et al. [2011] augment the pipeline with a fast on-chip LRU cache assigned to each rasterizer unit. This limits shading reuse within the rasterized primitive, but naturally fits to the GPU architecture, and avoids any explicit synchronization between fragment processing units. Our method would also benefit from a rasterizer-level hardware cache, but we can demonstrate significant performance improvement even by using current GPU architectures, above a certain shading complexity and sample count.

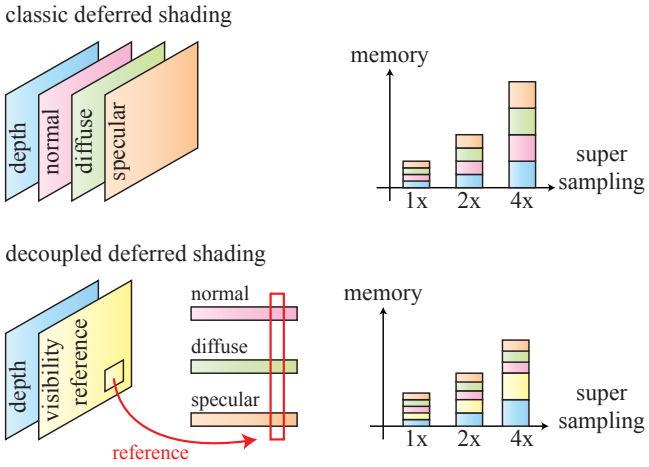


Figure 4: *While the* G-buffer *stores all shading data at full supersampled resolution, we introduce a visibility buffer which references compact shading data buffers. Due to shading reuse the size of the compact buffers does not scale with the supersampling density.*

During rendering, we need to incrementally assign addresses to shading samples in the compact buffers, in order to keep the memory footprint minimal. We use atomic counters similarly to the linked list generation method of Yang et al. [2010]. We also need to synchronize processed visibility samples, otherwise multiple threads would store the same shading data at different locations in the memory (and reduce the efficiency of shading reuse).

Current hardware does not allow local on-chip memory usage in rasterization mode (e.g. in a fragment shader), which restricts possible implementations to use the global memory for synchronization. Our first algorithm uses a global cache for fetching shading samples, and performs decoupled sampling in a single rasterization pass. We also present a multi-pass method which trades off bandwidth for synchronization speed, and uses GPU compute kernels to implement the memoization cache in the local memory.

5 Algorithm

Note that while the remainder of this paper focuses on a rasterization pipeline, the idea of the *CG-buffer* is more general, and can also be applied to alternative sampling methods, like ray tracing (e.g. generating a shared secondary ray for multiple primary rays).

5.1 Global Shading Cache

First, we discuss a method that implements the sampling stage of decoupled deferred shading in a single rasterization pass. It can replace the *G-buffer* generation pass of any existing deferred renderer, adding decoupled functionality to the shader stages.

Prior to rasterization, the shading primitives are processed, and their *ssID range* is determined. By simply adding this value to a global atomic counter, our method guarantees that all shading samples will have a unique value during the frame. This step can be implemented prior to hardware tessellation, or in a geometry shader, where the viewport dimensions of the primitives (thus the maximum number of shading samples) are known. Fragment shaders are responsible for mapping visibility samples to shading samples and eliminating redundancy from the shading data.

An important observation is that due to the streaming nature of GPUs, only the recently stored shading samples are "interesting" for a given fragment, which allows efficient shading reuse even with a simple FIFO cache strategy: instead of searching in the entire *CG-buffer*, we can store the recent *ssIDs* together with their memory addresses in a cache.

On current GPUs there can be several thousand fragments rasterized in parallel, which implies that the global cache should also be able to hold a similar magnitude of samples to allow a good hit-rate. This leads us to the classical dilemma of cache-design: the need for a large cache, and the desire to minimize the overhead of a cache lookup are opposing. Our primary goal is to design a data structure that reduces the interaction with the global memory as much as possible. Instead of using classical FIFO or LRU (least recently used) strategy, we apply a hash function, a standard technique for accelerating searching algorithms. The hash function $h()$ is a many-to-n mapping of the *ssID* values queried by the fragment shaders.

The data in the global shading cache is organized in a bucketed hash array, which stores addresses of recently used shading samples in each bucket. As there can be multiple shading samples in use with the same hash, each bucket stores a small FIFO queue, where each element is a pair of an *ssID* and a pointer to the shading data. To determine if the given shading sample is already stored in the memory, the fragment shader computes *h(ssID)* and fetches the data from the cache. As shading sample indices are incrementally assigned, a simple modulo N function is a suitable hash, where N is the number of buckets. This ensures that on average, queries are uniformly distributed among the buckets, and within the same triangle, the number of collisions (different *ssIDs* with the same hash) are minimal.

As the compact geometry buffer is constructed, the fragment shaders write references to their corresponding shading data into the render targets. For opaque objects, the z-buffer algorithm trivially solves the occlusion problem, while transparent surfaces are expected to be rendered later. Our method is compatible with existing order independent transparency methods ([Yang et al. 2010], [Enderton et al. 2010]). In the end of the sampling pass, only visible references are present in the frame buffer.

Race conditions When a sample is not found, the fragment shader has to evaluate the shading data at the location of the shading sample and store it in the buffer. This creates a race condition among shader threads, as multiple of them might decide to insert the same *ssID* to the cache, a behavior we have to avoid. Therefore modifying data in a bucket requires mutual exclusion. In our implementation each bucket has a binary semaphore, which indicates if it is under modification. Algorithm 1 details the global cache management in a pixel shader.

Algorithm 1 For each rasterized fragment $frag$ in parallel

```
 1: ssID ← map(frag, domain)
 2: hash ← h(ssID)
 3: bucket ← load(shaderCache, hash)
 4: address ← getCachedAddress(bucket, hash, ssID)
 5: needStore ← FALSE
 6:
 7: {Found in cache?}
 8: while address is NULL do
 9:    lock ← atomicExchange(locks[hash], LOCKED)
10:    if lock is FREE then
11:       {We have exclusive access to this bucket}
12:       bucket ← load(shaderCache, hash)
13:       address ← getCachedAddress(bucket, hash, ssID)
14:       {new shading sample}
15:       if address is NULL then
16:          address ← atomicInc(bufferTail)
17:          insertFIFO(bucket, ssID, address)
18:          needStore ← TRUE
19:       end if
20:       locks[hash] ← FREE {release bucket}
21:    end if
22: end while
23:
24: if needStore then
25:    storeShadingData(frag, address)
26: end if
27:
28: return address
```

A thread can acquire the semaphore by using an atomic exchange operation (line 9). An atomic counter is used to allocate the first available memory slot in the shading data buffers (line 16). When the list header is updated, the semaphore can be released (line 20). Storing the shading data might involve multiple memory transactions, therefore we move it outside the loop, and use the flag $needStore$ to mark fragments that need to evaluate shading data as well (line 24). Note, that after entering the critical section, the fragment needs to refresh its information about the bucket, as other threads might have modified it before (line 12). The above pseudocode is simplified, in practice we avoid deadlocks, and store multiple addresses in a single word of the cache to minimize the number of global memory transactions.

Compaction Depending on the rasterization order of triangles, there can be several entries in the *CG-buffer*, which do not belong to any visible surface. A fragment with a smaller depth value overwrites the references of formerly rasterized ones (also, when there is no early z-test, shading data is stored regardless visibility). We can of course fill up the z-buffer in a prepass of decoupled shading, but in order to provide a general solution, we clean up the *CG-buffer* after the sampling stage (before shading) from such irrelevant data in an optional pass.

By rendering a full-screen quad, we iterate over all samples in the visibility buffer and set a flag for the referenced shading samples. This time no synchronization is needed as write-write conflicts always store the same values. These flags can then be used to ignore

invisible samples, or even to remove them from the memory using stream compaction before shading, like in [Hoberock et al. 2009].

5.2 Per-Tile Shading Cache

The above algorithm reuses shading globally, finding corresponding visibility samples during rasterization. However, it applies global memory atomics, which becomes its main bottleneck. We now present an alternative approach, which is motivated by the fact that in many cases, a shading sample is not visible at arbitrary locations over the entire screen, but rather in relatively small regions. In most practical applications, the amount of motion blur or defocus fits to this assumption.

With this loosening of requirements, we can limit shading reuse to only certain blocks of pixels. Our second method splits the image to uniform tiles, and searches for duplicate shading samples within these tiles, processing each of them independently in parallel. A surface is shaded redundantly if visible in multiple tiles, but this is only a marginal reduction in efficiency, provided that the tiles are large enough compared to the amount of blur.

This algorithm uses the same *CG-buffer* data structure as presented in Section 4, but the sampling stage is split into two rasterization passes and a compute pass (Figure 5). In the first pass, the fragment shaders do not store shading data in the compact buffers, but only the *ssID* of the visibility sample, which is computed identically to the previous method. In parallel, the z-buffer is filled up with the depth values of the closest visible surface at each sample. This step can be combined with the standard *z pre-pass* of many deferred rendering systems.

At the end of the first pass, only the *ssIDs* of the visible surfaces are stored. We can now implement the same caching mechanism, *but in the local on-chip memory*. The main advantage of this approach is that this step can be implemented as a fully computational pass, where the programmer controls thread execution. We describe this pass using the terminology of the OpenCL programming language.

In our implementation, each tile is processed by a *work-group*, a set of threads that execute on the same multiprocessor. These threads can share data with each other through the *local memory*, which also supports fast atomic operations. The implementation of the caching is identical to the single-pass method, but all memory accesses are a magnitude faster. As a drawback, the size of the cache is highly limited compared to the global approach. The visibility samples in a tile are not processed linearly, but in smaller blocks to maximize the propability that the data in the shading cache is still "hot".

The output of the caching pass is a memory address for each visibility sample where the shading data can be stored. Ideally, samples with the same *ssIDs* receive the same address. We now have to physically store the data for shading, and that is performed in the second rasterization pass. This pass does not write anything to the frame buffer, but interpolates the parameters at shading samples, and stores them in the global memory. The z-buffer filled up by the first pass is used to make sure that only the visible fragments will execute this step. This is very important, as there is no synchronization between shaders, which causes write-write conflicts on the same address. After the execution of this pass, the *CG-buffer* is constructed the same way as in the first algorithm.

5.3 Shade and Resolve

The shading samples are evaluated using GPU compute kernels. These kernels only execute for samples that are marked as visible, and their cost is independent from the number of visibility samples.

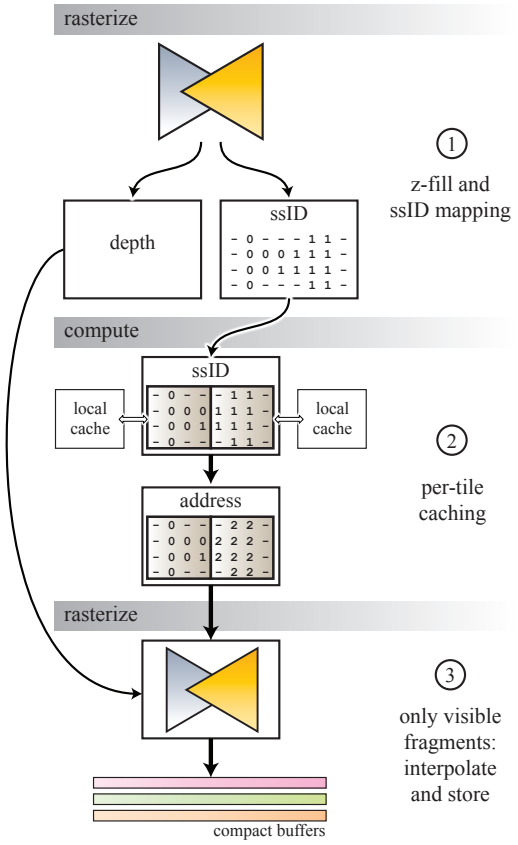


Figure 5: *The outline of the multi-pass decoupled shading algorithm. Note, that for simplicity we use assume flat-shaded triangles (no interpolation). The example also shows that shading samples visible at multiple tiles are redundantly stored. The major benefit of this method is the speed of the local cache.*

Finally each sample can gather its final color value using another full-screen pass. Note that this method trivially extends to arbitrary number of render targets, supporting efficient shading reuse for multi-view rasterization as well.

6 Deferred Shading for Stochastic Sampling

In the following we provide the details how we used decoupled deferred shading within a stochastic rasterization pipeline. Describing the entire pipeline is outside the scope of this paper and we thus refer to Akenine-Möller et al. [2007] and McGuire et al. [2010] for more details.

Stochastic Rasterization Stochastic rasterization first uses the geometry shader to compute a conservative bound for each defocused or motion blurred triangle, extending its bounding box with the respective circle of confusion or screen space motion vector. In the screen region covered by the bounding box, the fragment shader generates stochastically sampled rays on the camera lens and time, and intersects the triangle using ray casting (for more details see [McGuire et al. 2010]). If the intersection test was successful, we use the obtained barycentric coordinates to compute the *ssID* of the nearest shading sample.

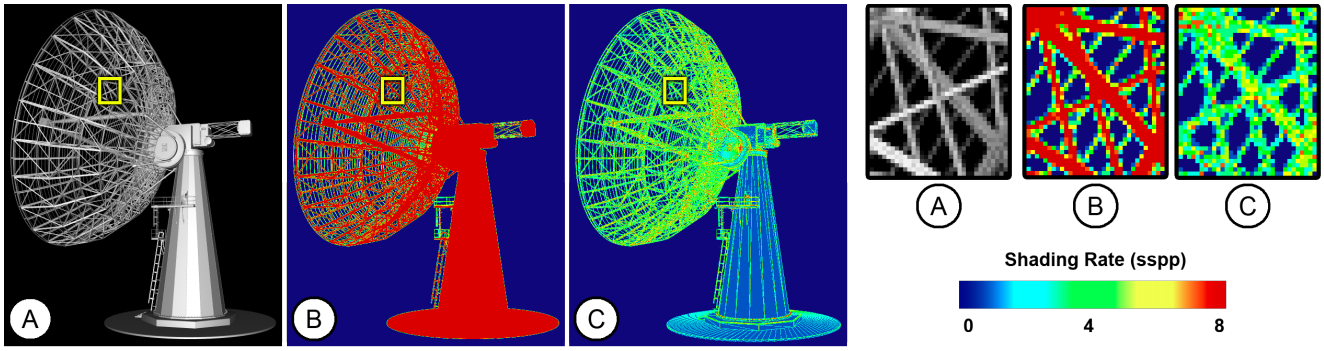


Figure 6: *In order to achieve the same antialiasing quality as with 8× MSAA (A), standard deferred shading needs to evaluate shading at 8× supersampling (B). Our method stores and evaluates shading at significantly lower rates (C), but keeps the antialiasing quality by supersampling visibility. The heat maps visualize the number of shading samples per pixel.*

Decoupling Scheme We adopted the decoupling mapping proposed by Ragan-Kelley et al. [2011]. This scheme uses a rasterized image of every individual triangle as seen from the center of the camera lens at the beginning of the shutter interval to generate shading samples within the triangle. The main advantage of this method is its simplicity: the shading rate can be adjusted on a per-triangle level by increasing or decreasing the resolution of the respective rasterized image. The nearest shading sample can be found by projecting the ray-triangle intersection points to the same grid. As a drawback, it does not provide uniform sampling density near to triangle edges: some shading samples might fall outside the triangle in the grid (having negative barycentric coordinates). We can solve this problem by snapping these shading samples to the edges, but this locally increases the shading frequency.

Adaptive Shading Rate In our implementation, the resolution of the shading grid is derived from the minimum amount of defocus blur in the triangle area for depth of field rendering, and from the screen space motion vectors when rendering motion blur. As the geometry shader needs to compute this information to generate the bounding volume, the additional cost of making the shading adaptive is negligible.

7 Applications

In this section we discuss the applications which we tested with our decoupled deferred shading method.

Depth of Field We have integrated the compact geometry buffer into a conventional real-time deferred shading pipeline, where the workload is dominated by non-subpixel-sized polygons. In this case reusing shading samples within a single triangle already results in reasonable speedup, as it typically can be used for several pixels on the screen. To demonstrate the limitations of previous deferred shading techniques, we use stochastic rasterization to render depth of field in a scene with high depth complexity. Stochastic rasterization generates a noisy *G-buffer*, where edge-based reconstruction filters, like *MLAA* or *SRAA* are not usable. This forces classical deferred shading to store the shading data at full supersampled resolution, while with our method the size of the *CG-buffer* does not increase significantly.

Figure 1 shows multiple renderings of the *Crytek Sponza Atrium* scene using decoupled deferred shading. In this example the most expensive component of shading is the computation of the single-bounce indirect illumination, using 256 virtual point lights (VPLs) generated from a reflective shadow map [Dachsbacher and Stamminger 2005]. Using OpenCL kernels for processing this many VPLs is a practical solution, as multiple threads can cooperate on loading chunks of the VPL data into the on-chip local memory.

During adaptive shading, we apply an empirically chosen factor to reduce the shading frequency of surfaces with a large cicrle of confusion. If the number of visibility samples is low, it is even desired to pre-filter surface data, such as textures, prior to sampling. With high number of visibility samples, however, it can result in slight overblurring of textures.

Antialiasing We also demonstrate that without blur, our technique is an exact match of hardware *MSAA* in quality. In fact, we use *MSAA* during the *CG-buffer* generation: a single fragment shader is called to store the shading reference to all subsamples. Figure 6 shows the shading rate used by decoupled deferred shading to compute 8× *MSAA* antialiasing, in comparison with true supersampling. All three methods resulted in identical images, but the costs of shading are different.

Motion Blur Figure 7 presents snapshots of an animated character rendered with motion blur, using stochastic rasterization. This example features ray traced ambient occlusion and image-based lighting. We use the *OptiX* ray tracing engine [Parker et al. 2010] to process the contents of the compact geometry buffer. When using hardware rasterization, high-performance ray tracing is only possible in a deferred shading pass.

Adaptive shading degrades the shading rate of fast-moving triangles, based on their screen space motion vectors. In this case, we do not have shading rate per triangle, but we use the x and y components of the motion vector in the shading grid to determine the speed of the surface in the shading domain. We then apply empirical factors again to reduce the frequency of shader evaluation along the motion.

8 Evaluation

While current GPU architectures do not have hardware support for decoupled shading, the overhead of our global cache management method is amortized by the reduction of shader evaluations. Figure 8 shows detailed timings for the *Sponza* and *Gargoyle* scenes on an Nvidia GTX 580 GPU. All the images in this paper were rendered at 1280 × 720 pixels. We have also computed the average shading rate of these images, to roughly estimate the reduction of shading cost compared to supersampled deferred shading.

As global synchronization is our major bottleneck on current GPUs, we have expected the multi-pass sampling approach to outperform the first method in several cases.

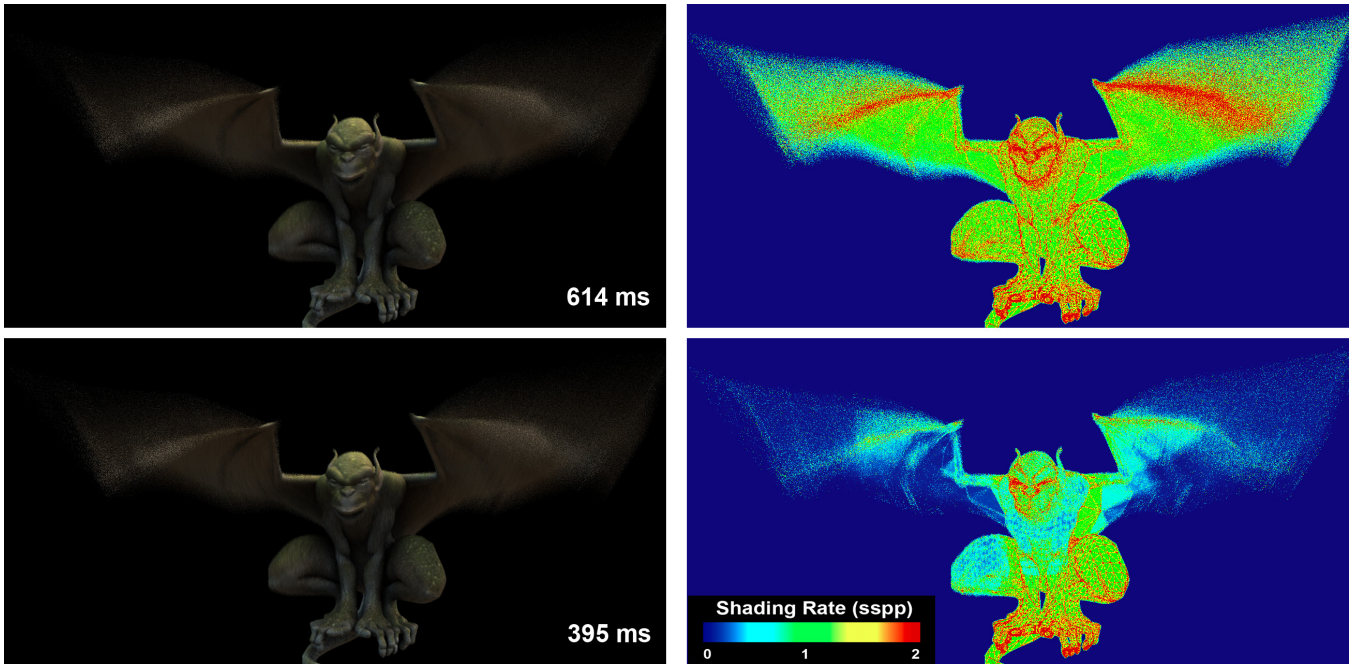


Figure 7: *A motion blurred character rasterized with 8× stochastic supersampling. Deferred shading is computed using 36 ambient occlusion rays per shading sample (using OptiX). Due to decoupled sampling, the shading rate stays close to 1 sspp despite the high sample count (top row). Adaptive shading (bottom row) saves ∼ 30% of the rendering time by further decreasing the shading rate of fast-moving surfaces.*

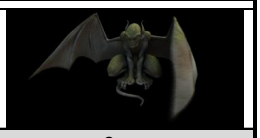
SPONZA	Defocused	Sharp	
Avg. Shading	0.4 sspp	1.11 sspp	
Sampling	90 ms	24.1 ms	
Shading	158 ms	330 ms	
Frame	251 ms	357 ms	4 spp
GARGOYLE	Blurry	Sharp	
Avg. Shading	0.8 sspp	1.3 sspp	
Sampling	24.6 ms	24.8 ms	
Shading	438 ms	608.3 ms	
Frame	465.3 ms	635.7 ms	8 spp

Figure 8: *Performance measurements with complex shading and stochastic rasterization. Note that despite the more expensive sampling, the blurry objects rendered faster due to the reduced shading costs.*

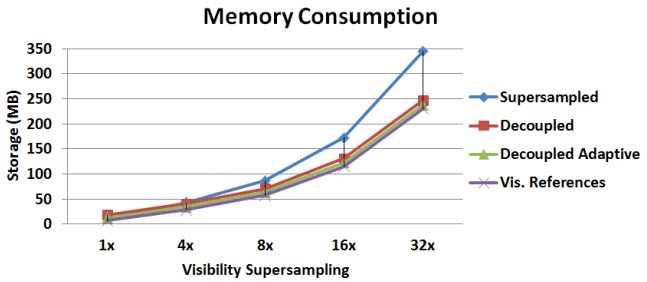


Figure 9: *Storage requirements of the CG-buffer compared to standard G-buffers. Our memory footprint consists of visibility and shading samples. The lightweight visibility data saves significant space at high supersampling resolutions, and note that the footprint of shading samples becomes negligible (we also plotted the space used for storing visibility refereences only). The Sponza scene was rendered at 1280×720 pixels.*

During our evaluation, though, it proved to be inferior in almost all situations. We can explain these negative results with multiple reasons. First, the bandwidth consumption of the multi-pass sampling stage increases proportionally with the number of visibility samples. In the second rasterization pass, fragment shaders have to execute on a per-subsample level to write the shading data to the CG-buffer, and as there is no synchronization, every visibility sample needs to interpolate and store its data independently.

In the global caching method, we could also exploit that *ssID ranges* are generated incrementally, so we could distribute queries uniformly over the hash buckets. The cache size in the local memory is very limited, and as a tile stores *ssIDs* from several triangles, the number of hash collisions (and therefore lock-spinning iterations) is very high. The problem is essentially to deduplicate numbers in a 2D array on the GPU, which is algorithmically expensive.

In this section we limit our measurements to the global method, and we believe that in the future the performance of our multi-pass sampling can be improved by using a different caching strategy, or by integrating it into a tile-based sort middle architecture.

We analyzed of memory consumption of our method in comparion with supersampled deferred shading. We save storage by essentially *deduplicating* shading data in the *G-buffer*. However, this only takes effect at higher sample counts, as we need to store additional information, which existing techniques do not require. We assume that the ground truth supersampled deferred method uses 12 bytes per subsample in the *G-buffer*: 32 bits for depth-stencil, 2× RGBA8 textures for normals and material information. In fact, state-of-the-art deferred rendering engines use even more.

The memory footprint of the *CG-buffer* can be divided into per visibility and per shading sample costs. In the former we need to store an integer *ssID* besides the depth-stencil. As we have multiple references from the framebuffer to the same shading sample, the depth value is not enough to reconstruct the view space position of a surface. Therefore we are forced to store the view space position on additional 8 bytes (16 bits for $x - y$ and 32 bits for z). The total consumption is then 8 bytes per visibility sample and 16 bytes per shading sample.

If the shading rate is 1, and there is no multisampling, our method uses twice as much memory as conventional techniques. However, the number of shading samples does not scale with the supersampling resolution. At $4\times$ *MSAA* the expected footprint of our method is 48 bytes per pixel, the same as with standard deferred shading. Above this sample count, our method saves significant amount of storage.

For our benchmark, we have used the *Sponza* scene, and filled the entire screen with geometry to make our comparison fair (The scene shown in Figure 8). We have used stochastic rasterization with varying sample counts. Figure 9 shows our results. We could further decrease the required storage when using adaptive shading.

9 Conclusion and Future Work

In this paper we presented decoupled deferred shading, which uses a compact geometry buffer to store shading samples independently from visibility samples. This enables reusing of shading computation using the capabilities of the currently available GPUs. We demonstrated that above a certain amount of shader complexity, i.e. with high-quality rendering, our method can outperform standard deferred shading approaches. We expect that the major synchronization bottleneck of our method will disappear on future architectures, where more efficient cache management can be implemented due to hardware support or flexible software rasterization.

We have assumed that the shaded color of surfaces does not change relevantly in a single frame, over the lens or in the shutter interval. This might cause artifacts on fast-moving surfaces, causing smeared highlights and shadows on the screen. In the future, we will support interpolation among multiple shading samples, and increase the dimensionality of the shading domain.

An interesting direction is to extend the *ssID* generation to provide temporarily coherent IDs and thus reuse shading samples across multiple frames. We also plan to investigate better caching strategies for our tile-based method to increase its overall performance.

Acknowledgements

We would like to thank Anton Kaplanyan and Balázs Tóth for the helpful discussions and the anonymous reviewers for their valuable suggestions. The first author of this paper is funded by Crytek GmbH.

References

AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. 2007. Stochastic rasterization using time-continuous triangles. In *Proc. of Symposium on Graphics Hardware*, 7–16.

BURNS, C. A., FATAHALIAN, K., AND MARK, W. R. 2010. A lazy object-space shading architecture with decoupled sampling. In *Proc. of High Performance Graphics*, 19–28.

CHAJDAS, M. G., MCGUIRE, M., AND LUEBKE, D. 2011. Subpixel reconstruction antialiasing for deferred shading. In *Proc. of Symposium on Interactive 3D Graphics and Games*, 15–22.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. *Computer Graphics (Proc. of SIGGRAPH) 18*, 3, 137–145.

DACHSBACHER, C., AND STAMMINGER, M. 2005. Reflective shadow maps. In *Proc. of Symposium on Interactive 3D Graphics and Games*, 203–231.

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. *Computer Graphics (Proc. of SIGGRAPH) 22*, 4, 21–30.

EGAN, K. T., TSENG, Y.-T., HOLZSCHUCH, N., DURAND, F., AND RAMAMOORTHI, R. 2009. Frequency analysis and sheared reconstruction for rendering motion blur. *ACM Transactions on Graphics (Proc. of SIGGRAPH) 28(3)*, 3, 93:1–93:13.

ENDERTON, E., SINTORN, E., SHIRLEY, P., AND LUEBKE, D. 2010. Stochastic transparency. In *I3D '10: Proceedings of the 2010 symposium on Interactive 3D graphics and games*, 157–164.

FATAHALIAN, K., BOULOS, S., HEGARTY, J., AKELEY, K., MARK, W. R., MORETON, H., AND HANRAHAN, P. 2010. Reducing shading on gpus using quad-fragment merging. *ACM Transaction on Graphics 29*, 4, 67:1–67:8.

HOBEROCK, J., LU, V., JIA, Y., AND HART, J. C. 2009. Stream compaction for deferred shading. In *Proc. of High Performance Graphics*, 173–180.

MCGUIRE, M., ENDERTON, E., SHIRLEY, P., AND LUEBKE, D. 2010. Real-time stochastic rasterization on conventional gpu architectures. In *Proc. of High Performance Graphics*, 173–182.

MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. 1994. A sorting classification of parallel rendering. *IEEE Computer Graphics and Application 14*, 4, 23–32.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics 29* (July), 66:1–66:13.

RAGAN-KELLEY, J., KILPATRICK, C., SMITH, B. W., EPPS, D., GREEN, P., HERY, C., AND DURAND, F. 2007. The lightspeed automatic interactive lighting preview system. *ACM Transactions on Graphics (Proc. of SIGGRAPH) 26*, 3.

RAGAN-KELLEY, J., LEHTINEN, J., CHEN, J., DOGGETT, M., AND DURAND, F. 2011. Decoupled sampling for real-time graphics pipelines. *ACM Transactions on Graphics 30*, 3.

RESHETOV, A. 2009. Morphological antialiasing. In *Proc. of the Conference on High Performance Graphics*, 109–116.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. *Computer Graphics (Proc. of SIGGRAPH) 24*, 4, 197–206.

SOLER, C., SUBR, K., DURAND, F., HOLZSCHUCH, N., AND SILLION, F. 2009. Fourier depth of field. *ACM Transactions on Graphics 28*, 2, 18:1–18:12.

YANG, J. C., HENSLEY, J., GRN, H., AND THIBIEROZ, N. 2010. Real-time concurrent linked list construction on the gpu. *Computer Graphics Forum 29*, 4, 1297–1304.