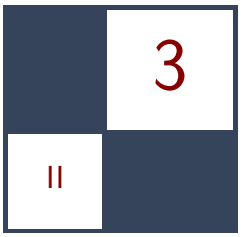

Decoupled Deferred Shading on the GPU

Gábor Liktor and Carsten Dachsbacher

Deferred shading provides an efficient solution to reduce the complexity of image synthesis by separating the shading process itself from the visibility computations. This technique is widely used in real-time rendering pipelines to evaluate complex lighting, and recently gained increasing focus of research with the advent of computational rendering.

The core idea of the technique is to presample visible surfaces into a *G-buffer* prior to shading. However, antialiasing is complicated with deferred shading, as supersampling the G-buffer leads to tremendous growth in memory bandwidth and shading cost. There are several post-processing methods that are mostly based on smoothing discontinuities in the G-buffer [Reshetov 09, Chajdas et al. 11], but these result in inferior antialiasing quality compared to forward rendering with multisample antialiasing (MSAA) or do not address the problem of memory requirements.

3.1 Introduction

In this article we discuss decoupled deferred shading, a technique that uses a novel G-buffer structure to reduce the number of shading computations while keeping the antialiasing quality high. Our edge antialiasing is an exact match of hardware MSAA, while shading is evaluated at a per-pixel (or application-controlled) frequency, as shown in Figure 3.1.

Our G-buffer implementation stores visibility samples and shading samples in independent memory locations, where a visibility sample corresponds to a subsample tested by the rasterizer, while shading samples contain surface information, which has been previously stored on a subsample level. Using decoupled sampling, several visibility samples can refer to a single shading sample. We do not seek to skip the shading of G-buffer samples in order to reduce shading costs,

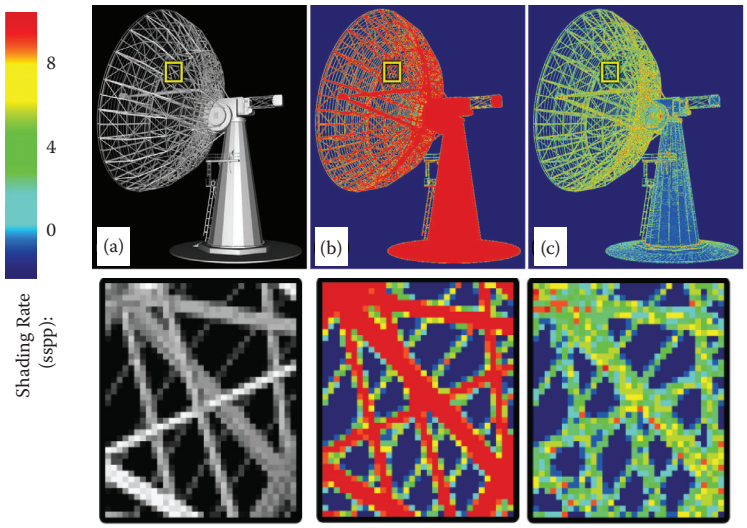

Figure 3.1. In this example our deferred shading method (a) achieves equivalent antialiasing quality to 8× MSAA, but (c) significantly reduces the number of shader evaluations. (b) To the same antialiasing quality, classic deferred shading needs a supersampled G-buffer.

instead we *deduplicate* the data itself, ensuring that a visible surface is shaded only once, regardless of the number of subsamples it covers.

This article is based on our recent research paper, presented at the 2012 ACM Symposium on Interactive 3D Graphics and Games [Liktor and Dachsbacher 12]. We cover the basic theory of decoupled sampling, and then focus on the implementation details of our new G-buffer in the OpenGL pipeline.

3.2 Decoupled Sampling in a Rasterization Pipeline

3.2.1 The Nature of Aliasing

To understand the motivation of decoupled sampling, let us consider the rendering of a 2D image as a signal-processing problem. Rasterization uses point sampling to capture visible surfaces that causes problems if the sampled signal is not band-limited: frequencies higher than the sampling frequency lead to aliasing in the rendered image. Antialiasing methods can prefilter the signal to eliminate frequencies above the sampling limit, increase the frequency of sampling, or alternatively apply reconstruction filters to supress aliasing artifacts.

Any rendering method using point sampling must first solve the *visibility problem* to find the surface points that determine the colors at each sample.

Discontinuities, such as surface silhouettes, are the primary sources of aliasing. The second type of aliasing is the possible undersampling of surface shading. Unlike visibility, shading is often treated as a continuous signal on a given surface, thus it can be prefiltered (e.g., by using texture mipmaps). It is therefore a tempting idea to save computations by sampling visibility and shading information at different granularities.

3.2.2 Decoupled Sampling

In a modern rasterization pipeline this problem is addressed by MSAA. The rasterizer invokes a single fragment shader for each covered pixel; however, there are multiple subsample locations per pixel, which are tested for primitive coverage. Shading results are then copied into covered locations. This is an elegant solution for supersampling visibility without increasing the shading cost.

Decoupled sampling [Ragan-Kelley et al. 11] is a generalization of this idea. Shading and visibility are sampled in separate domains. In rasterization, the *visibility domain* is equivalent to subsamples used for coverage testing, while the *shading domain* can be any parameterization over the sampled primitive itself, such as screen-space coordinates, 2D patch-parameters, or even texture coordinates. A *decoupling map* assigns each visibility sample to a coordinate in the shading domain. If this mapping is a many-to-one projection, the shading can be reused over visibility samples.

Case study: stochastic rasterization. Using stochastic sampling, rasterization can be extended to accurately render effects such as depth of field and motion blur. Each coverage sample is augmented with temporal and lens parameters. Defocused or motion blurred triangles are bounded in screen space according to their maximum circle of confusion and motion vectors. A deeper introduction of this method is outside the scope of this article, but we would like to refer the interested reader to [McGuire et al. 10] for implementation details. In short, the geometry shader is used to determine the potentially covered screen region, the fragment shader then generates a ray corresponding to each stochastic sample, and intersects the triangle.

We now illustrate decoupled sampling using the example of motion blur: if the camera samples over a finite shutter interval, a moving surface is visible at several different locations on the screen. A naïve rendering algorithm would first determine the barycentics of each stochastic sample covered by a triangle, and evaluate the shading accordingly. In many cases, we can assume that the observed color of a surface does not change significantly over time (even offline renderers often do this). MSAA or post-processing methods cannot solve this issue, as corresponding coverage samples might be scattered over several pixels of the noisy image. We can, however, rasterize a sharp image of the triangle at a fixed *shading time*, and we can find corresponding shading for each visibility sample by projecting them into the pixels of this image.

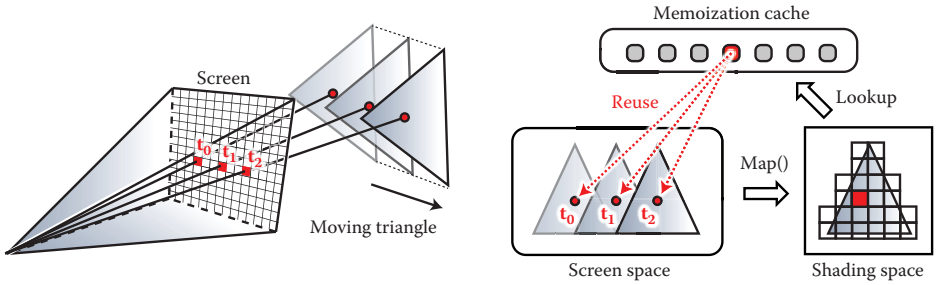

Figure 3.2. The idea of the memoization cache. Decoupled sampling uses visibility and shading samples in separate domains. Assuming constant shading over a short exposure time, multiple visibility samples can refer to the identical shading sample. Recently computed shading samples are cached during rasterization, to avoid redundant shader execution.

Memoization cache. This concept is illustrated in Figure 3.2. Note that the second rasterization step mentioned above does not actually happen, it is only used to define a *shading grid* on the triangle, a discretization of the shading domain. A shading sample corresponds to one cell of the shading grid, and we can then assign a linear index to each shading sample. Using this indexing, Ragan-Kelley et al. augmented the conventional rasterization pipeline with a *memoization cache* [Ragan-Kelley et al. 11]. In their extended pipeline, each visibility sample requests its shading sample from the cache using the decoupling map, and fragment shaders are only executed on a cache miss. Unfortunately, this method is not directly applicable to the current hardware architecture.

3.3 Shading Reuse for Deferred Shading

Conventional deferred shading methods couple visibility and surface data in the G-buffer. After the geometry sampling pass it is no longer trivial to determine which samples in the G-buffer belong to the same surface. Stochastic rasterization further increases the complexity of the problem by adding significant noise to visibility samples, preventing the use of any edge-based reconstruction.

The memory footprint is one of the most severe problems of deferred shading. As all shading data must be stored for each subsample in the G-buffer, even if one could save computation by reusing shading among these samples, the supersampling quality would still be bounded by memory limitations. Current real-time applications typically limit their deferred multisampling resolution to 2×/4× MSAA, then apply reconstruction filters. It has been demonstrated that accurate rendering of motion blur or depth of field would require an order of magnitude larger sample count with stochastic sampling [McGuire et al. 10].

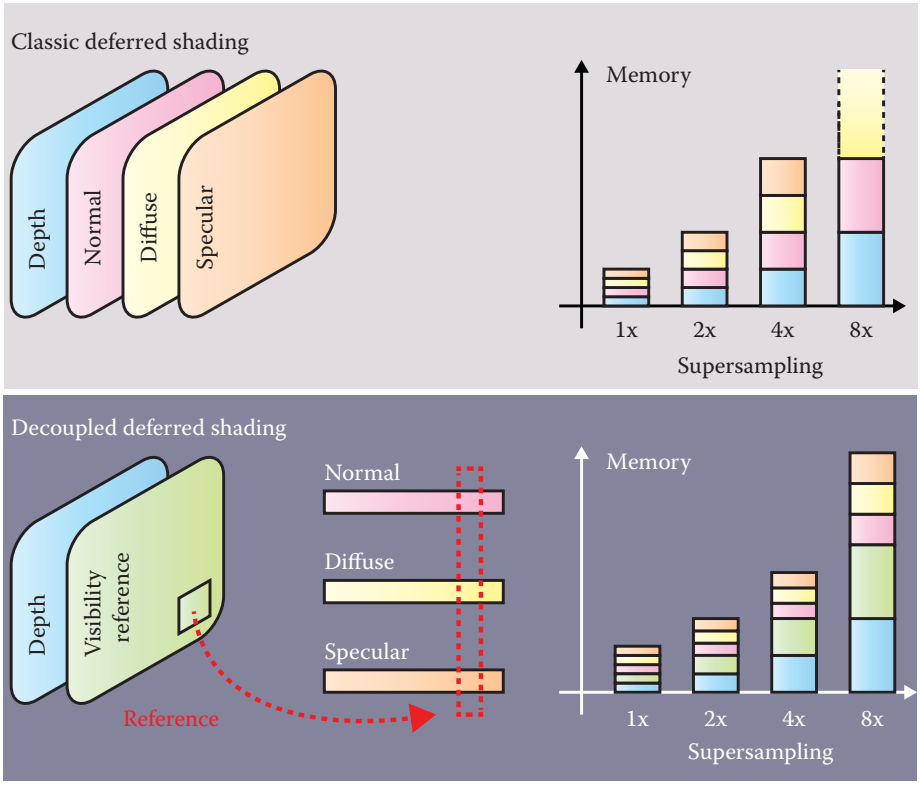

Figure 3.3. The G-buffer stores shading data at full supersampled resolution before shading and resolving. We introduce a visibility buffer that references shading data in compact linear buffers. Due to our shading reuse scheme, the size of the compact buffers does not scale with the supersampling density.

Compact geometry buffer. Instead of trying to use reconstruction filters or sparse shading of the supersampled G-buffer, we can avoid any shading and memory consumption overhead by not storing redundant shading data in the first place. We address this problem with a novel data structure, the *compact G-buffer*, a decoupled storage for deferred shading. It has the same functionality as the G-buffer, storing the inputs of shaders for delayed evaluation. However, instead of storing this information in the framebuffer, we collect *shading samples* in compact linear buffers. The contents of the framebuffer are purely *visibility samples*, each sample storing its depth value and a reference to a shading sample in the linear buffers. We compare this data layout to the conventional G-buffer in Figure 3.3. Akin to classic deferred shading, our methods can render images in three main stages.

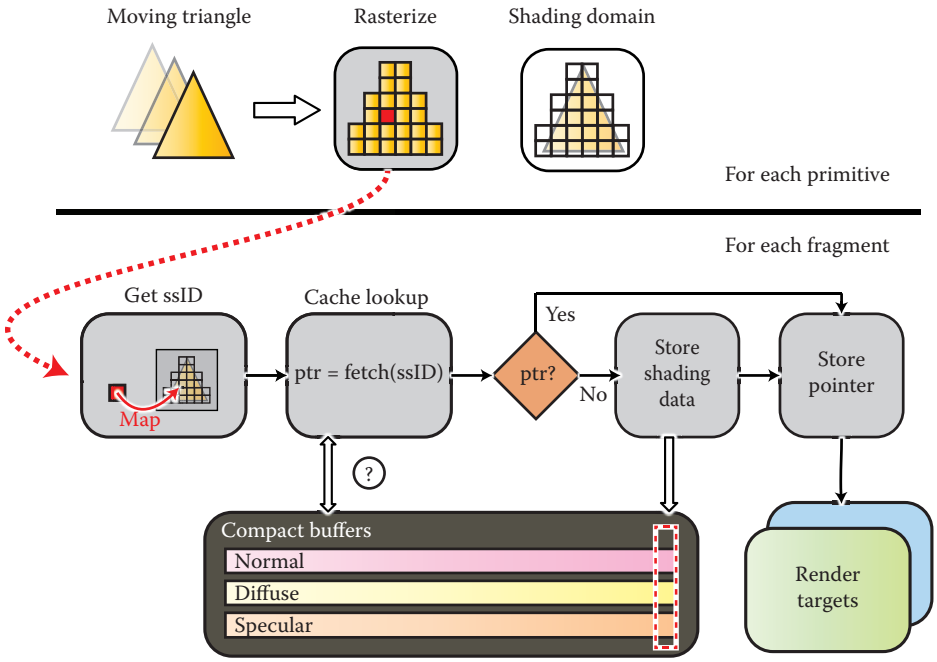

Figure 3.4. The outline of decoupled deferred shading in a rasterization pipeline. Prior to rasterization, each primitive is bound and projected to a shading grid. During fragment processing, the fragments are mapped to their corresponding cells on the shading grid. Shading reuse is implemented by referencing the same data from multiple samples in the render targets.

3.3.1 Algorithm Outline

Sampling stage. We rasterize all surfaces into the compact geometry buffer (*CG-buffer*). Figure 3.4 shows the outline of this sampling stage. During rasterization each fragment is assigned to a shading sample ID (*ssID*), which is searched in the cache. 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 in addition to referencing it.

In Section 3.2.2 we have already introduced the concept of a shading grid. In our pipeline, we use this grid to allocate an ssID range for each primitive. This virtual address space ensures that shading sample keys of concurrently rasterized primitives do not overlap, and the sampler can use these ssIDs to uniquely reference a cached shading sample entry.

We provide further implementation details in the next section. In fact, we only made a small modification in the decoupled sampling pipeline. While the

CG-buffer itself could be directly used as a global memoization cache, it would be very inefficient to search for shading samples directly in it, especially that a cached entry is only relevant for the currently rasterized primitives in flight.

Shading and resolving stages. The collected samples in the compact buffers are then shaded using GPU compute kernels. These kernels only execute for shading samples that are marked visible (see the next section). Finally each visibility sample can gather its final color value in a full-screen pass. This method trivially extends to an arbitrary number of render targets, supporting efficient shading reuse for multiview rasterization as well.

3.4 Implementation

In this section we focus on how to implement decoupled deferred shading on a modern GPU. In our examples we provide OpenGL Shading Language (GLSL) source code snippets. We use global atomics and scatter operations, therefore a minimum version of OpenGL 4.2 is required for our application. The implementation could also be done in DirectX 11.1, which supports unordered access binding to all shader stages.

The primary problem for our example is the lack of hardware support for decoupled shading reuse, which is an architectural limitation. The hardware version of the memoization cache, as described in Section 3.2.2, is a fast on-chip least recently used (LRU) cache assigned to each rasterizer unit. Of course, every component of the pipeline (ultimately even rasterization) can be implemented in software, but only with reduced performance compared to dedicated hardware. From now on we assume that our renderer still uses the hardware rasterizer, though this technique could be also integrated into a full software implementation, such as [Laine and Karras 11].

3.4.1 Architectural Considerations

Note that the implementation of decoupled sampling for a forward renderer would be very inefficient on current GPUs. First, using hardware rasterization, we can only simulate the caching behavior from fragment shaders. Unfortunately we cannot prevent the execution of redundant shaders, like the proposed architecture of [Ragan-Kelley et al. 11] does. The rasterizer will launch fragment shaders for each covered pixel or subsample and we can only terminate redundant instances afterwards. This introduces at least one new code path into the shading code, breaking its coherency.

The second problem is how to avoid redundant shading. Shading reuse can be regarded as an *election problem*: shader instances corresponding to the same shading sample must elect one instance that will evaluate the shading, the others need to wait for the result. This can only be solved using global synchronization,

as current hardware does not allow local on-chip memory usage in rasterization mode, and the execution of fragment shaders is nondeterministic. Furthermore, waiting for the result would mean a significant delay for a complex shader.

With our modification we can move the shader evaluation into a deferred stage, which results in a more coherent fragment shader execution. While we cannot avoid using the global memory to simulate the memoization cache, the overhead of decoupled sampling is independent from the shading complexity. This is the key difference that makes our algorithm feasible even for current GPUs: if the shading computation is "expensive enough," the constant overhead of our caching implementation will be less than the performance gain of reduced shading. Furthermore, we can utilize our CG-buffer to keep the memory footprint of the shading data minimal.

3.4.2 Decoupling Shading Samples

We now discuss a method that implements the sampling stage of decoupled deferred shading in a single rasterization pass. The first problem we need to solve is how to assign shading samples to fragments. Prior to rasterization, each primitive needs to be processed to determine its shading domain (see Section 3.3.1

```
in vec2 in_scrPos[]; // screen-space positions
flat out ivec4 domain; // shading grid of the triangle
flat out uint startID; // ID of the first sample in the sh. grid

uniform float shadingRate;

// global SSID counter array
layout(size1x32) uniform uimageBuffer uCtrSSID;

void main(){
    // project screen position to the shading grid
    vec2 gridPos0 = scrPos[0] * shadingRate; [...]

    vec2 minCorner = min(gridPos0, min(gridPos1, gridPos2));
    vec2 maxCorner = max(gridPos0, max(gridPos1, gridPos2));

    // shading grid: xy-top left corner, zw-grid size
    domain.x = int(minCorner.x) - 1;
    domain.y = int(minCorner.y) - 1;
    domain.z = int((maxCorner.x)) - domain.x + 1;
    domain.w = int((maxCorner.y)) - domain.y + 1;

    // we allocate the ssID range with an atomic counter.
    uint reserved = uint((domain.z) * (domain.w));
    startID = imageAtomicAdd(uCtrSSID, 0, reserved);
}
```

Listing 3.1. The geometry shader generates a shading grid for each triangle, and ensures globally unique ssIDs using an atomic counter

for details). As we only consider triangles, we can conveniently implement this functionality in a geometry shader.

Listing 3.1 is an extract from the geometry shader code that assigns a shading grid for each rasterized triangle. In our implementation, the geometry shader might also set up conservative bounds for stochastically rasterized triangles.

Listing 3.2 shows the pseudocode of the fragment shader, implementing the remainder of our pipeline. As we described before, the output of this shader is only a pointer to the corresponding shader data. Note that due to driver limitations on integer multisampling, we need to store the references in floating point, using the `intBitsToFloat` GLSL function. The shading samples are stored using image buffers.

We omit the details of visibility testing, which might be standard multisampled rasterization, or the implementation of stochastic rasterization, which casts randomly distributed rays inside the conservative screen space bounds of the triangles. We only assume that the visibility method returned the barycentrics of the intersection point. The visibility sample is then assigned to a shading sample, using the grid provided by the geometry shader.

```
layout(location = 0, index = 0) out float FragPtr;

// shader inputs: position, normal, texcoords
flat in vec3 vpos0...

// packed CG-buffer data
layout(rg32ui) uniform uimageBuffer uColorNormalBuffer;
layout(rgba32f) uniform uimageBuffer uViewPosBuffer;

void main(){
    // hw-interpolation or stochastic ray casting...
    vec3 baryCoords = getViewSamplePos();

    // get nearest shading sample
    uint localID = projectToGrid(baryCoords, shadingRate);
    uint globalID = startID + localID;
    bool needStore = false;

    int address = getCachedAddress(globalID, needStore);
    FragPtr = intBitsToFloat(address);

    if(needStore){
        // for each texture...
        textureGradsInShadingSpace(localID, dx, dy);
        vec4 diffuse = textureGrad(texDiffuse, texCoord, dx, dy);
        [...]
        // pack color, normal, view positions into the CGbuffer
        imageStore(uColorNormalBuffer, address, ...);
    }
}
```

Listing 3.2. The fragment shader implements the decoupling map and the chaching mechanism for shading samples.

```
uint projectToGrid(vec3 baryCoords, float shadingRate){
    vec3 vpos = coords.x * vpos0 + coords.y * vpos1 + coords.z * vpos2;
    vec2 screenPos = projToScreen(vpos);
    ivec2 gridPos = ivec2(screenPos * shadingRate + vec2(0.5f)) - domain
        .xy;
    return uint(domain.z * gridPos.y + gridPos.x);
}
```

Listing 3.3. The decoupling map is a simple projection to a regular grid. The density of this grid is determined by the shading rate.

The method `projectToGrid` assigns the fragment to a shading sample, as we show in Listing 3.3. The local index of the shading sample is the linearized index of the closest shading grid cell to the visibility sample. Later, when the shading data is interpolated, some shading samples might fall outside the triangle. These are snapped to the edges (by clamping the barycentrics to 0 or 1, respectively), otherwise some shading values would be extrapolated.

The computation of the texture mip levels also needs special attention. Normally, this is done by the hardware, generating texture gradients of 2×2 fragment blocks. Depending on the shading rate, the shading space gradients can be different. For example, a shading rate of 0.5 would mean that 2×2 fragments might use the same shading sample, which would be detected (incorrectly) as the most detailed mip level by the hardware. Therefore we manually compute the mip level, using the `textureGrad` function.

In Listing 3.2 we have also tried to minimize the divergence of fragment shader threads. The method `getCachedAddress` returns the location of a shading sample in the global memory. In case of a cache miss, a new slot is reserved in the CG-buffer (see below), but the shading data is only written later, if the `needStore` boolean was set.

3.4.3 Global Shading Cache

For a moment let us consider the cache as a "black box" and focus on the implementation of the CG-buffer. If a shading sample is not found in the cache, we need to append a new entry to the compact linear buffers, as shown in Figure 3.4. The CG-buffer linearly grows as more samples are being stored. We can implement this behavior using an atomic counter that references the last shading data element:

```
address = int(atomicCounterIncrement(bufferTail));
```

The streaming nature of the GPU suggests that even a simple first-in, first-out (FIFO) cache could be quite efficient as only the recently touched shading samples

are "interesting" for the fragments. We therefore did not attempt to simulate an LRU cache, as suggested by Ragan-Kelley et al. On a current GPU there can be several thousand fragments rasterized in parallel, thus the global cache should also be able to hold a similar magnitude of samples to achieve a good hit rate. In a naïve implementation, a thread could query the buffer tail, and check the last N items in the CG-buffer. Of course the latency of the iterative memory accesses would be prohibitively high. We now show a cache implementation that performs cache lookups *with only one buffer load* and an atomic lock.

The concept of the shading grid already creates indices for shading samples that are growing approximately linearly with the rasterized fragments. Thus the FIFO cache could also be implemented by simply storing the last N *ssID* values. Consequently, instead of linearly searching in a global buffer, we could introduce a bucketed hash array. The full implementation of our optimized algorithm is shown in Listing 3.4.

The hash function (`hashSSID`) is a simple modulo operation with the number of buckets. This evenly distributes queries from rasterized fragments over the buckets, which is important to minimize cache collisions (when threads with different ssIDs compete for the same bucket). In case of a cache miss, multiple threads compete for storing the shading samples in the same bucket, therefore we use a per-bucket locking mechanism (`uBucketLocks`). Note that we try to minimize the number of instructions between obtaining and releasing a lock: the computation of a shading sample does not happen inside the critical section, but we only set the `needStore` flag to perform the storage later.

As the execution order of fragments is nondeterministic, there is no guarantee that all threads obtain the lock in a given number of steps. In practice we have very rarely experienced cases when the fragment shader execution froze for starving fragment shaders. While we hope this will change on future architectures, we have limited the spinlock iterations, and in case of failure the fragment shader falls back to storing the shading sample without shading reuse. In our experiments this only happened to a negligible fraction of fragments.

One other interesting observation was that if the number of cache buckets is high enough, we can really severely limit the bucket size. As the reader can see in the source code, a bucket stores only a single `uvec4` element, which corresponds to two shading samples: a cache entry is a tuple of an ssID and a memory address. This is a very important optimization, because instead of searching inside the cache, we can look up any shading sample with a single load operation using its hash value.

In our first implementation, each bucket in the cache has been stored as a linked list of shading sample addresses, similarly to the per-pixel linked list algorithm of [Yang et al. 10]. When we have made experiments to measure the necessary length of this list, we have found that in most cases even a single element per bucket is sufficient, and we did not have any cache misses when we considered only two elements per bucket. This is why we could discard the expensive linked

```
layout(rgba32ui) uniform uimageBuffer uShaderCache;
layout(r32ui) uniform volatile uimageBuffer uBucketLocks;

int getCachedAddress(uint ssID, inout bool needStore){
    int hash = hashSSID(ssID);
    uvec4 bucket = imageLoad(uShaderCache, hash);
    int address = searchBucket(ssID, bucket);

    // cache miss
    while(address < 0 && iAttempt++ < MAX_ATTEMPTS){
        // this thread is competing for storing a sample
        uint lock = imageAtomicCompSwap(uBucketLocks, hash, FREE, LOCKED
            );
        if(lock == FREE){
            address = int(atomicCounterIncrement(bufferTail));

            // update the cache
            bucket = storeBucket(ssID, hash, bucket);
            imageStore(uShaderCache, hash, bucket);
            needStore = true;

            memoryBarrier(); // release the lock
            imageStore(uBucketLocks, hash, FREE);
        }

        if(lock == LOCKED){
            while(lock == LOCKED && lockAttempt++ < MAX_LOCK_ATTEMPTS)
                lock = imageLoad(uBucketLocks, hash).x;

            // now try to get the address again
            bucket = imageLoad(uShaderCache, hash);
            address = searchBucket(ssID, bucket);
        }

        // if everything failed, store a the data redundantly
        if(address < 0){
            address = int(atomicCounterIncrement(bufferTail));
            needStore = true;
        }
    }
}
```

Listing 3.4. Implementation of the global shading cache.

list behavior and pack all buffers in a single vector. However, this optimization only works if the hash function uniformly distributes cache requests (like ours), and the number of buckets is high. In our examples we use a bucket count of 32,768.

3.4.4 Shading and Resolving

While the sampling stage described above successfully eliminates all duplicate shading samples, the resulting CG-buffer might still hold redundant information. Depending on the rasterization order of triangles, several samples in the compact

buffers might not belong to any visible surfaces. Even filling up the z-buffer in a depth prepass might not solve the problem: if early z-testing is disabled a z-culled fragment can still write data into a uniform image buffer. We therefore execute another pass that marks visible shading samples, and optionally removes invisible data from the CG-buffer.

Visibility. Marking visible samples is surprisingly easy. After the sampling stage is finished, we render a full-screen quad with subsample fragment shader execution, and each fragment shader stores a visibility flag corresponding to its shading sample. There is no synchronization needed, as each thread stores the same value. To evaluate the quality of shading reuse, we used a variant of this technique, which counts visibility samples per-shading sample. In the diagnostics code we atomically increment a per-shading sample counter for each subsample in the framebuffer. The heatmap visualizations in this article were generated using this method.

Compaction. Because of the rasterization order, there is no explicit bound on the size of the compact buffers. Using the visibility flags, we can perform a stream compaction on the shading data before shading. Besides efficient memory footprint this also increases the execution coherence during the shading process.

In this article we do not provide implementation details for shading, as it is orthogonal to our decoupling method. The final pixel colors are evaluated by rendering a full-screen quad and gathering all shaded colors for each visibility sample. This is the same behavior as the resolve pass of a standard multisampled framebuffer, except for the location of subsample colors.

3.5 Results

In this section we discuss possible application of our method in deferred rendering. While current GPU architectures do not have hardware support for decoupled sampling, the overhead of our global cache management can be amortized by the reduction of shader evaluations. We focus on stochastic sampling, a rendering problem especially challenging for deferred shading.

While the software overhead of decoupled sampling makes our method rather interactive than real time, we demonstrate significant speedup for scenes with complex shading. All images in this article were rendered at $1{,}280 \times 720$ pixels on an Nvidia GTX580 GPU and Intel Core i7 920 CPU.

Adaptive shading. We have computed the average shading rate of these images, to roughly estimate the shading speedup compared to supersampled deferred shading. We save further computation by reducing the density of the shading grid of blurry surfaces. Our adaptive shading rate implementation is only a proof-of-concept based entirely on empirically chosen factors. For better quality,

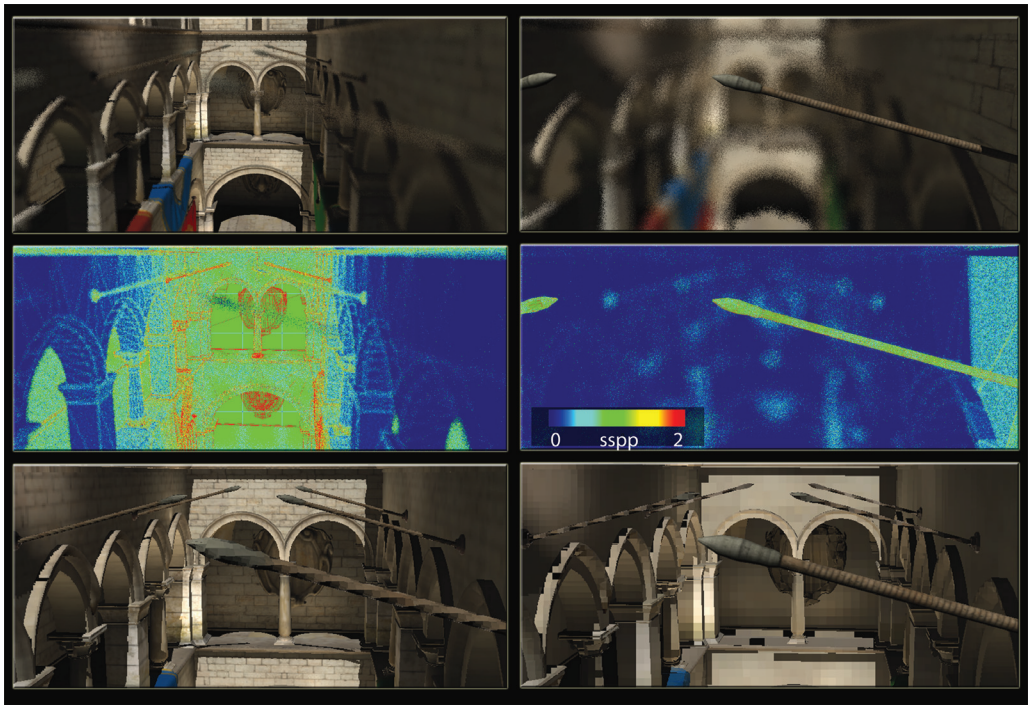

Figure 3.5. Focusing from the background (left column) to a foreground object (right column), our adaptive method concentrates shading samples on sharp surfaces. The motivation is to prefilter shading more agressively, as defocus is similar to a low-pass filter over the image. The middle row visualizes the shading rate. In the bottom row we show how the same surface shading would appear from a pinhole camera. The texture filtering matches the shading resolution.

our method could be easily extended with the recent results of [Vaidyanathan et al. 12], who presented a novel anisotropic sampling algorithm, based on image space frequency analysis.

Depth of field. Figure 3.5 shows two renderings of the Crytek Sponza Atrium scene from the same viewing angle, but different focusing distance. In this example the most expensive component of rendering is the computation of the single-bounce global illumination, using 256 virtual point lights (VPLs), generated from a reflective shadow map (RSM) [Dachsbacher and Stamminger 05].

We do not only avoid supersampling the G-buffer, but also reduce the shading frequency of surfaces using the minimum circle of confusion inside each primitive. This approach prefilters shading of defocused triangles, causing slight overblurring of textures, however, we found this effect even desirable if the number of visibility

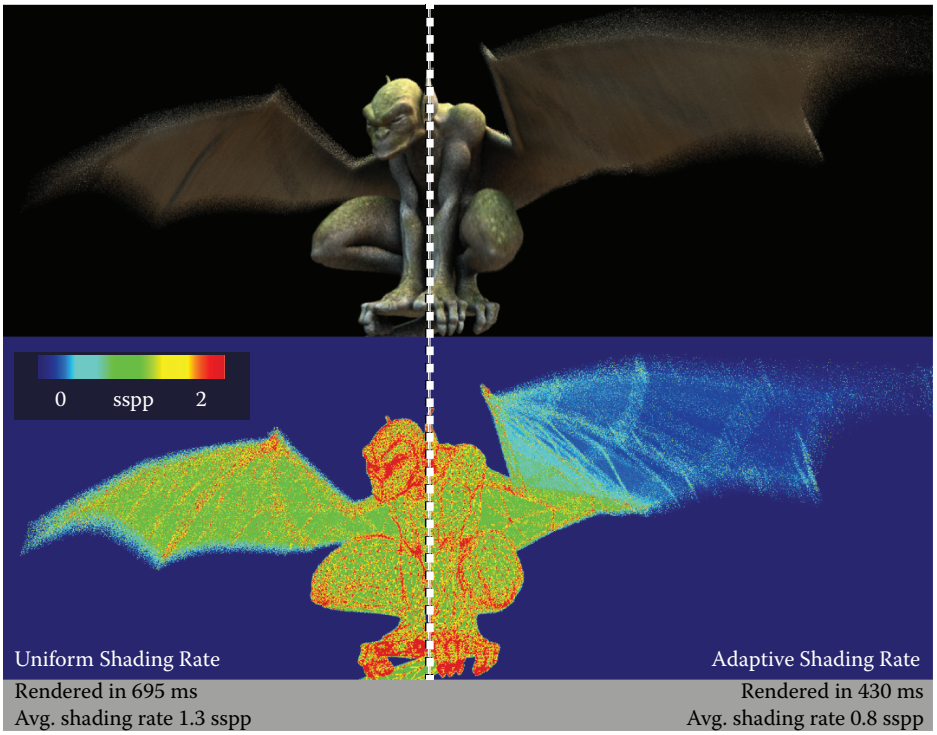

Figure 3.6. A motion blurred character rendered with eight times stochastic supersampling. Deferred shading is computed using 36 ambient occlusion samples per shading sample. The shading rate stays close to one shading sample per pixel (sspp) despite the supersampling density (left side). We can save further ∼30% of the rendering time by adaptively reducing sampling of fast-moving surfaces (right side).

samples is small (it effectively reduces the apparent noise of surfaces). The images were rendered using four times supersampling, the stochastic sampling stage took 90 ms, and the shading with 256 VPLs took 160 ms.

Motion blur. Figure 3.6 presents an animated character, rendered with motion blur. This example features ray-traced ambient occlusion and image-based lighting, using the Nvidia OptiX raytracing engine. When using hardware rasterization, high-performance ray tracing is only possible in a deferred computational shading pass. Here we demonstrate adaptive shading again, by reducing the shading rate of fast-moving triangles. We scale the shading grid based on the x and y component of the triangle motion vectors. Our results (and the reduction of shading) can be significantly improved by using the anisotropic shading grid of [Vaidyanathan et al. 12].

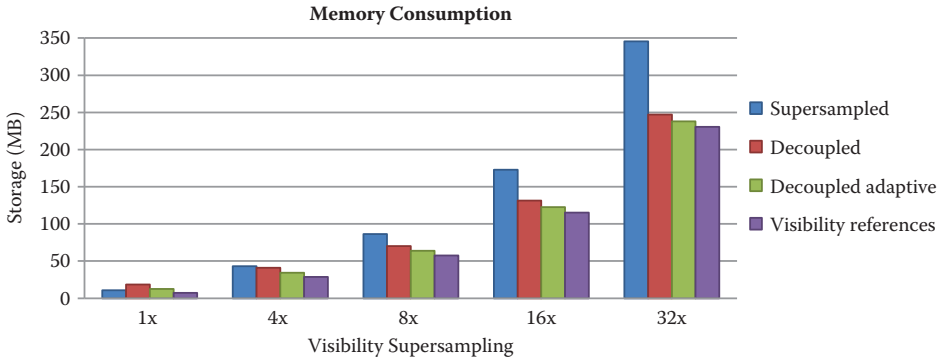

Figure 3.7. Storage requirements of the CG-buffer compared to a standard deferred G-buffer. Only the size of the visibility data grows with supersampling. We rendered the Sponza scene at 1,280 × 720 pixels.

3.5.1 Memory Consumption

We have analyzed the memory consumption of our method, compared to supersampled deferred shading. We save storage by essentially deduplicating shading data in the G-buffer. However, as a shading sample might not coincide with any visibility samples on the surface, we cannot reconstruct the surface positions based on a subpixel depth value. While other solutions are possible, we assume that we are forced to store the view space position of each shading sample.

We assume that the ground truth deferred method uses 12 bytes per subsample in the G-buffer: 32 bits for depth-stencil and two RGBA8 textures for normals and material information. In fact, most modern deferred renderers use typically more bytes per subsample. The memory footprint of our CG-buffer can be divided into per-visibility and per-shading sample costs. In the former we need to store an integer pointer besides the 32-bit depth-stencil. We need 16 bytes per shading sample: the view positions are packed into 8 bytes (16 bits for $x - y$ and 32 bits for z), and we store the same normal and material information.

If the shading rate is one 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× MSAA, our memory consumption matches the supersampled G-buffer's, and we save significant storage above this sample count. Our measurements on the Sponza scene are summarized in Figure 3.7.

3.5.2 Conclusion

In this chapter we presented a decoupled deferred shading method for high-quality antialiased rendering. To our knowledge this is the first deferred shading method

designed for stochastic rasterization. Unfortunately on current GPUs we need to implement stochastic rasterization and the shading cache using shaders, to overcome the limitations of the hardware pipeline. We consider our results beneficial for interactive applications, where shading cost dominates the rendering, however, the overhead of the global cache implementation is generally too high for real-time rendering.

We expect that the major synchronization bottleneck will disappear in future rendering architectures. While we cannot predict whether future GPUs would have a hardware-accelerated version of the memoization cache, some way of local synchronization among fragement shaders would already remove most of the overhead. Using a tile-based rendering architecture instead of sort-last-fragment would allow us to use a more efficient, per-tile on-chip shading cache.

In our examples we have assumed that the visible color of surfaces remains constant in a single frame, and shading can be prefiltered. This might cause artifacts on fast-moving surfaces, therefore we could extend our method to support interpolation among temporal shading samples. In the future it will be interesting to separate the frequency content of shading itself: a hard shadow edge in fact cannot be prefiltered, but there are low-frequency components of shading, e.g., diffuse indirect illumination, where sparse shading can bring relevant speedup.

3.6 Acknowledgments

We would like to thank Anton Kaplanyan and Balázs Tóth for the helpful discussions during the development of this project. Gabor Liktor is funded by Crytek GmbH.

Bibliography

[Chajdas et al. 11] Matthäus G. Chajdas, Morgan McGuire, and David Luebke. "Subpixel Reconstruction Antialiasing for Deferred Shading." In *Proceedings of Symposium on Interactive 3D Graphics and Games*, pp. 15–22. New York: ACM, 2011.

[Dachsbacher and Stamminger 05] Carsten Dachsbacher and Marc Stamminger. "Reflective Shadow Maps." In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pp. 203–231. New York: ACM, 2005.

[Laine and Karras 11] Samuli Laine and Tero Karras. "High-Performance Software Rasterization on GPUs." In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pp. 79–88. New York: ACM, 2011.

[Liktor and Dachsbacher 12] Gábor Liktor and Carsten Dachsbacher. "Decoupled Deferred Shading for Hardware Rasterization." In *Proceedings of the*

ACM Symposium on Interactive 3D Graphics and Games, pp. 143–150. New York: ACM, 2012.

[McGuire et al. 10] M. McGuire, E. Enderton, P. Shirley, and D. Luebke. "Real-Time Stochastic Rasterization on Conventional GPU Architectures." In *Proceedings of the Conference on High Performance Graphics*, pp. 173–182. Aire-la-Ville, Switzerland: Eurographics Association, 2010.

[Ragan-Kelley et al. 11] J. Ragan-Kelley, J. Lehtinen, J. Chen, M. Doggett, and F. Durand. "Decoupled Sampling for Graphics Pipelines." *ACM Transactions on Graphics* 30:3 (2011), article no. 17.

[Reshetov 09] Alexander Reshetov. "Morphological Antialiasing." In *Proceedings of the Conference on High Performance Graphics 2009*, pp. 109–116. New York: ACM, 2009.

[Vaidyanathan et al. 12] Karthik Vaidyanathan, Robert Toth, Marco Salvi, Solomon Boulos, and Aaron E. Lefohn. "Adaptive Image Space Shading for Motion and Defocus Blur." In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics COnference on High Performance Graphics*, pp. 13–21. Aire-la-Ville, Switzerland: Eurographics Association, 2012.

[Yang et al. 10] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. "Real-Time Concurrent Linked List Construction on the GPU." *Computer Graphics Forum* 29:4 (2010), 1297–1304.