Deferred Attribute Interpolation for Memory-Efficient Deferred Shading

Christoph Schied* Carsten Dachsbacher†
Karlsruhe Institute of Technology

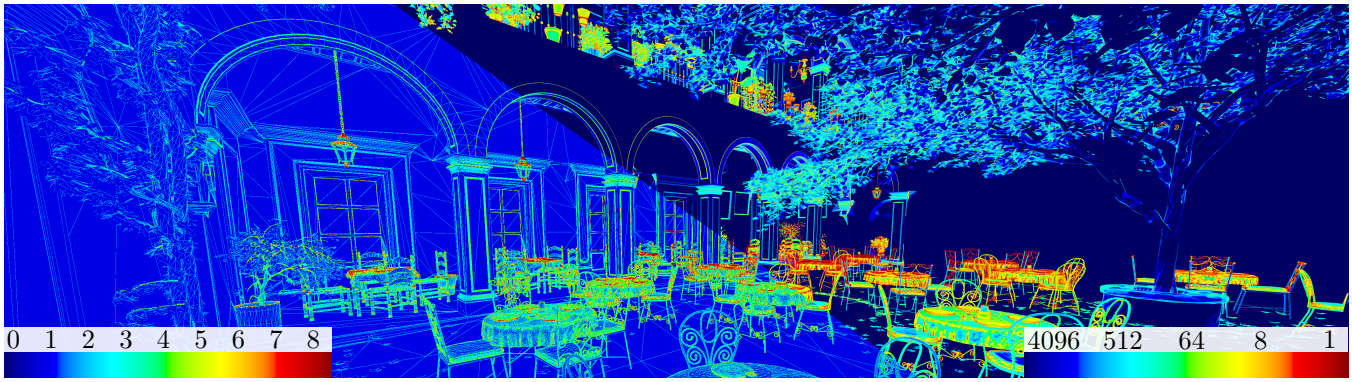


Figure 1: *Crop of an image rendered at a resolution of* 3840×2160 *with 8 times MSAA. Our method is able to compress the size of the geometry buffer to* 30% *of the deferred shading G-Buffer using* 20 *bytes per sample. We store a visibility buffer which references triangles stored in a triangle buffer. The image on the left shows the number of shader invocations per pixel. On the right side, the number of visibility samples referencing the same triangle data is shown.*

Abstract

In this work we present a novel approach to deferred shading suitable for high resolution displays and high visibility sampling rates. We reduce the memory costs of deferred shading by substituting the geometry buffer with a visibility buffer that stores references into a triangle buffer. The triangle buffer is populated dynamically with all visible triangles which is compatible with the use of tessellation. Stored triangles are represented by a sample point and screen-space partial derivatives. This representation allows for efficient attribute interpolation during shading and gives shaders knowledge about the partial derivatives of all attributes. We show that the size of the visibility buffer can be further decreased by storing a linked list of visibility samples per pixel. For high-resolution displays we propose an extension of our algorithm to perform shading at reduced frequency, allowing us to reduce the sampling rate for computationally expensive, but low-frequency signals such as indirect illumination.

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

Keywords: deferred shading, real time rendering, anti aliasing

1 Introduction

A traditional rasterization-based rendering architecture shades surface points directly when the visibility of a triangle for a certain pixel is determined. This approach, however, has several drawbacks. As visibility is determined sequentially per triangle, the previously computed shading for a pixel may be replaced by a triangle processed later on. If several shading passes are needed, the geometry needs to be processed multiple times. Deferred shading [Saito and Takahashi 1990] splits the process into two passes where first the data needed for the shading computation is stored in a buffer. In the second pass, the shading is computed for every pixel using this information. While deferred shading can be a solution for the aforementioned problems, it has its drawbacks as well. Most importantly with today's screen resolutions, the geometry buffer (G-Buffer) uses large amounts of memory.

Multi-sample anti-aliasing (MSAA) [Akeley 1993] decouples the frequency used for visibility sampling from the shading frequency, but it cannot be used to its full potential with deferred shading. Since the shading stage does not know about the visibility samples, it cannot determine which stored samples need to be shaded. The shading therefore needs to be computed redundantly at the frequency of the visibility sampling, thus mitigating the main performance benefit of MSAA. In the G-Buffer each visibility sample comprises all geometric attributes which results in a high storage overhead.

In this work, we improve deferred shading and, in the same spirit as the method by Burns et al. [2013], break up the relation between geometry data and visibility samples. In contrast to classic deferred shading, only a reference to a triangle is stored per visibility sample. Since several visibility samples *can* reference the same triangle, it is possible to have a more compact representation of the G-Buffer, especially with high visibility sampling rates. Because current rasterization hardware cannot handle small triangles efficiently [Fatahalian et al. 2009], it is safe to assume that in practice most projected triangles will cover several visibility samples and even pixels on average. We store visible triangles represented using a sample point and screen-space derivatives only (refer to Figure 2). This allows for efficient interpolation of attributes during the shading phase. Furthermore, we use two different storage formats for the visibility buffer, the first using a multi-sampled render target whereas the second one employs a linked list for the visibility samples, which can reduce the storage costs even further. During shading we can use the triangle references to identify visibility samples corresponding to the same triangle and thereby avoid redundant shading.

*e-mail:schied@kit.edu
†e-mail:dachsbacher@kit.edu

High Performance Graphics 2015, August 7 – 9, 2015, Los Angeles, CA.

ACM 978-1-4503-3707-6/15/08 $15.00
http://dx.doi.org/10.1145/2790060.2790066

2 Related Work

The storage and computational overhead of using high visibility sampling rates in conjunction with deferred shading has been a long-standing problem. Current practice is to use low visibility sampling rates in combination with morphological anti-aliasing filters [Reshetov 2009; Jimenez et al. 2012] in a post-processing pass. These filters, however, cannot reconstruct subpixel features that are missed due to low visibility sampling rates.

Burns et al. [2013] proposed to replace the G-Buffer in deferred shading by a visibility buffer which stores triangle and instance IDs per visibility sample. Since the first render pass is solely for determining visibility, they can more clearly separate the geometry phase from the shading phase. They show that the representation using the visibility buffer can make use of caches very efficiently and thus the required memory bandwidth can be reduced significantly. Since the IDs stored in the visibility buffer reference untransformed triangles they can support instancing very efficiently. Our work takes a similar approach by storing triangle references per visibility sample. However, we store transformed triangles in a buffer in contrast to referencing untransformed triangles. Storing the triangles creates more memory traffic, but makes our method compatible with tessellation and allows for a more efficient attribute interpolation, since referencing untransformed triangles requires the vertex transformations to be performed at per pixel frequency.

With the goal of reusing shading in the context of stochastic rasterization with high visibility sampling rates, Ragan-Kelley et al. [2011] proposed *Decoupled Sampling*. Shading reuse is implemented by assigning a unique ID to shading samples using a shading grid and storing the result in a memoization cache. Liktor et al. [2012] adapted their approach for deferred shading. They break up the coupling between visibility and shading samples by referencing a shading sample in a compact G-Buffer per visibility sample and showed that this approach can be used to compress the G-Buffer in case of high visibility sampling rates. Furthermore, their approach allows them to control the shading frequency per primitive and therefore to adjust it in areas of lower interest or stronger blur. They propose a deduplication method using a memoization cache, suitable to be implemented on modern GPUs. We use this memoization cache for storing triangles.

Clarberg et al. [2013] proposed a novel deferred shading architecture for decoupled sampling. The pipeline is split into two phases. First, all primitives are rasterized and shading samples are mapped into a shading space. In the second phase the shading samples are sorted inside of tiles by their IDs. The sorted list is used to reissue the vertex shading pipeline once per triangle and finally to spawn shading quads, which are evaluated using a work queue. In contrast to our method, their pipeline requires only one rasterization pass, and furthermore they only need to repeat the vertex shading for visible triangles, whereas our pipeline requires a geometry pass. However their pipeline cannot be implemented efficiently on current graphics hardware.

To reduce the memory footprint of deferred shading with high visibility sampling rates, Kerzner et al. [2014] proposed a G-Buffer compression algorithm. Their compressed G-Buffer stores a fixed length array of surfaces and G-Buffer samples per pixel. During rasterization they determine per pixel if newly inserted surfaces can be merged with one of the previously stored surfaces by comparing their depth and depth derivatives. Since our method stores those attributes as well, their approach could be employed in our algorithm to identify similar surfaces during shading. Aggregate G-Buffer Anti-Aliasing [Crassin et al. 2015] follows a similar direction by prefiltering the geometry into a statistical representation which is stored per pixel. This allows them to keep the costs for shading

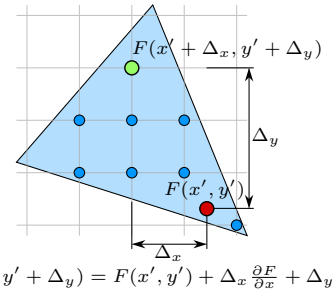


Figure 2: *A triangle attribute F is interpolated at a shading position (green) by using a sample point (red) and adding the partial derivatives of the attributes weighted by their distance to the sample point.*

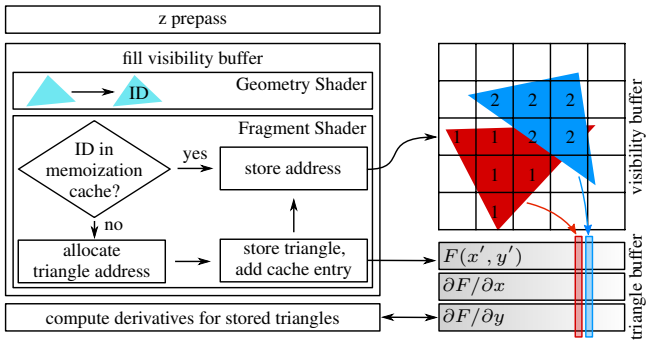


Figure 3: *Our pipeline for populating the visibility and triangle buffer. The z-prepass ensures that only visible triangles will generate fragment shader invocations. The geometry shader assigns each triangle an ID. In the fragment shader, all invocations look up the triangle ID and store the triangle address in the visibility buffer. If the cache does not store the ID yet, a new slot is allocated in the triangle buffer and one fragment shader invocation is selected to store the triangle in the triangle buffer. Finally the derivatives are computed for all stored triangles.*

constant per pixel, independent of the underlying geometry. Furthermore their representation allows them to reduce the memory costs by 30% in case of 8× MSAA.

Vaidyanathan et al. [2014] proposed a novel forward shading architecture that decouples the sampling of visibility from shading, targeting high-resolution displays. Besides reducing the overall shading rate, they also support multi-rate shading by using different shader stages, to sample low frequency shading on reduced sampling rate. Furthermore they allow the vertex shader to control the shading rate on a per- primitive base. Their approach needs modifications to the current rendering pipeline and cannot be implemented on current hardware.

3 Algorithm

Akin to deferred shading, our approach consists of two phases. First, we populate the visibility buffer with visibility samples storing references into the triangle buffer (refer to Figure 3). Section 3.1 explains our method for sampling visibility and storing the triangles. In the second phase, the shading is computed for each visibility sample. In Section 3.2 we describe our algorithm to interpolate the attributes in screen-space for shading.

3.1 Creating the Visibility and Triangle Buffers

To populate the triangle buffer, all visible triangles for the current view have to be determined. Our algorithm uses two geometry passes.

First, a z-prepass is performed. This ensures that in the second pass that is used to populate the visibility buffer, only fragment shader invocations for visible triangles will be generated. Since the number of fragment shader invocations corresponding to the same triangle is not known beforehand, we need to mediate access to the triangle buffer and select one fragment shader invocation to store the triangle and broadcast the buffer index to the other invocations. We assign a unique ID to each triangle and use the memoization cache proposed by Liktor et al. [2012] to store mappings between triangle IDs and triangle buffer indices. Each fragment shader invocation performs a lookup into the cache using the triangle ID. If the entry is not found, the fragment shader invocation competes for exclusive write access to the corresponding cache entry. If write access is granted, a slot in the triangle buffer is allocated and the buffer index is stored in the cache. Otherwise, the fragment shader invocation repeatedly looks up in the cache and competes for write access until either the entry is found or write access is granted. The triangle is stored by the fragment shader invocation that was granted write access to the cache. Finally, all invocations store the buffer position in the visibility buffer.

3.2 Perspective-Correct Attribute Interpolation using Partial Derivatives

Since the visibility buffer only stores references to triangles, the vertex attributes need to be interpolated for the shading computation. In the following we will briefly recapitulate the interpolation in a triangle in 2D. With attributes defined over a triangle per vertex as a_i, screen-space 2D-interpolation can be performed using barycentric weighting of all attributes. An interpolated attribute $\tilde{a}$ is computed as $\tilde{a}(x,y) = \sum \lambda_i(x,y)a_i$, where $\lambda_i(x,y)$ are the barycentric coordinates for a point (x,y) in relation to a 2D-triangle. Since barycentric interpolation is linear in x and y, the partial derivatives of the interpolant are constant. This allows us to reformulate the interpolation as

$$\tilde{a}(x,y) = a_{x'y'} + (x-x')\sum_i \frac{\partial \lambda_i}{\partial x}\cdot a_i + (y-y')\sum_i \frac{\partial \lambda_i}{\partial y}\cdot a_i \ . \quad (1)$$

For a triangle projected onto the screen, where each vertex is given in 4D homogeneous coordinates (x_i, y_i, z_i, w_i), a perspective correction needs to be applied when interpolating the attributes. This correction is done by first interpolating a_i/w_i and $1/w_i$ separately and a subsequent division, see e.g. [Davidovič et al. 2012]. The perspective-correct interpolation is thus given as $a = \frac{\sum \lambda_i a_i/w_i}{\sum \lambda_i/w_i}$. We can reformulate this equation similar to Equation 1 as

$$a(x,y) = \frac{\frac{a_{x'y'}}{w_{x'y'}} + (x-x')\frac{\partial a/w}{\partial x} + (y-y')\frac{\partial a/w}{\partial y}}{\frac{1}{w_{x'y'}} + (x-x')\frac{\partial 1/w}{\partial x} + (y-y')\frac{\partial 1/w}{\partial y}} \ . \quad (2)$$

We store the partial derivatives of a/w and $1/w$ in the triangle buffer, as well as the sample point. This interpolation scheme has the benefit that the computationally expensive parts only need to be computed once per triangle, whereas other interpolation schemes that determine the barycentric coordinates, e.g. using ray-triangle intersection tests, need to be carried out at a per-pixel frequency.

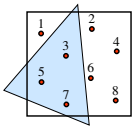
Note that the barycentric coordinates could be stored as one additional attribute, hence allowing to interpolate additional triangle data resident in GPU memory, without explicitly storing this data in the triangle buffer. However this approach only works for attributes that are not affected by vertex transformations.

3.3 Multi-Rate Shading

Some shading signals, such as indirect diffuse illumination, are low-frequency and therefore can be sampled with reduced shading rate.

```
// mask stores which samples need to be shaded
uint mask = (1 << NUM_VISIBILITY_SAMPLES) - 1;
vec3 accum = vec3(0.0);
while(mask > 0) {
  int i = findLSB(mask); // next sample index to shade
  uint vs = read_visibility_sample(i);
  if(vs != ~0) { // is there a triangle referenced?
    uint sample_mask = vs >> 24; // extract coverage
    uint t = vs & 0x00ffffff; // extract triangle idx
    accum += bitCount(sample_mask) * compute_shading(t);
    mask &= ~sample_mask; // mark shaded samples
  } else { // no triangle referenced
    accum += background_color;
    mask &= ~(1 << i); // mark shaded sample
  }
}
return accum / float(NUM_VISIBILITY_SAMPLES);
```

Listing 1: *Shade-and-resolve pass for the multi-sampled visibility buffer. A bitmask tracks the samples that need to be shaded (2). In conjunction with the triangle ID (6), a coverage mask for the triangle inside of the pixel is stored in the visibility buffer. The sample coverage mask is used to determine the multiplicity of the triangle sample (10). Whenever the shading is computed for a triangle, the coverage bits are removed from the mask that holds the samples that need to be shaded (11).*



Coverage Mask:

1	2	3	4	5	6	7	8
0	0	1	0	1	0	1	0

Figure 4: *Sample coverage mask. For each visibility sample inside of one pixel, the coverage mask contains a bit indicating if the visibility sample is inside of the primitive.*

We create shading samples using the method as proposed by Liktor et al. [2012]. In contrast to their method, i.e. storing G-Buffer samples, our method creates shading samples storing a triangle reference as well as its screen-space position. Each shading sample is assigned a unique ID using a screen-space shading grid [Ragan-Kelley et al. 2011]. For deduplicating the shading samples across all visibility samples, we make use of a memoization cache. The shading sample address is stored in an additional visibility buffer. A compute pass is performed that evaluates all shading samples. These samples are then read during the shading phase and combined with the shading computed at full shading rate.

In contrast to bilateral upsampling [Kopf et al. 2007], our method does not suffer from artifacts due to undersampling on high-frequency geometry since the shading rate is controlled per primitive and not uniformly for the whole frame buffer. However, undersampling the shading signal can obviously result in temporal artifacts when the assumption of a low-frequency shading signal is violated.

4 Implementation

In the following section we will discuss our GPU implementation that uses OpenGL 4.5.

4.1 Visibility Buffer

For each triangle, we store the material ID, vertex normals, texture coordinates, $1/w$ for one sample point, as well as the screen-space partial derivatives of all attributes in x and y. From this sample point, we extrapolate the attribute to the center of the screen, which allows us to omit storing the actual sample position. Using the fragment shader, we store all vertices of the triangle in the triangle buffer. The derivatives are computed in a separate compute pass which overwrites the vertices in the triangle buffer with their corresponding

derivatives. Since we do not store indices into the triangle buffer but offsets, the storage format can be different for each of the triangles, which may be needed for implementing different materials. We issue draw calls sorted by material which ensures that triangles sharing the same material are stored in compact ranges. A compute pass is performed per material to compute the partial derivatives of all attributes. In case that tessellation is used, triangle IDs are assigned in the geometry shader by incrementing an atomic counter, otherwise the builtin GLSL variable `gl_PrimitiveID` is used.

We use a size of 2^{13} entries for the memoization cache, which gives us almost perfect hit rates, and low overhead due to low contention between the fragment shader invocations. While lower cache sizes are still sufficient for a good hit rate, the overhead due to contention becomes larger. We did not measure any benefits for the hit rate or the performance by further increasing the cache size. We use a 1D-texture for the cache buckets storing two triangle IDs with their corresponding triangle buffer offset. Write access to the memoization cache is mediated using a spinlock per cache bucket. If a fragment shader invocation does not find a cache entry for the given triangle ID, it tries to gain exclusive write access by performing an atomic exchange operation on the lock texture. If access is granted, a slot in the triangle buffer is allocated, the cache entry is updated and the lock is released. Otherwise the lock is polled repeatedly until it is released, or a maximum number of iterations is exceeded. In the rare event that neither the triangle ID is found in the cache, nor write access to the cache is granted, the triangle is stored redundantly.

We implemented two visibility buffer storage formats which take the coverage information into account. The coverage mask can be accessed in the fragment shader and reports for the current primitive which visibility samples are inside (refer to Figure 4). The z-prepass allows us to compute precise coverage information per primitive for each pixel, using a post-depth-test coverage mask, available in OpenGL using the `EXT_post_depth_coverage` extension.

Multi-sampled Visibility Buffer We use a multi-sampled buffer that stores 32 bit per visibility sample in conjunction with the depth buffer. We allocate 24 bit for the triangle reference, and furthermore store a 8 bit coverage mask in the remaining bits. The coverage information allows us to skip visibility samples during shading (refer to Section 4.3).

Per Pixel Visibility Sample Linked Lists Employing a per-pixel linked list, we can implement an even more compact storage scheme for the visibility buffer. We use a multi-sampled depth buffer, but only store one 32 bit pointer to the head of the list *per pixel*. Since the z-prepass guarantees correct coverage information, we can identify the number of visibility samples covered by the triangle per pixel from the coverage mask and store the count alongside a triangle reference in the linked list. Furthermore, this information allows us to skip storing the linked list when full coverage is detected and store the triangle address directly in the per-pixel visibility list head pointer. We mark the storage of a triangle reference using a special bit. Each node in the list stores the visibility count of the triangle, a triangle address, as well as a pointer to the next node, using 64 bit in total.

4.2 Computing Derivatives

The interpolation given in Equation 2, requires to compute the partial derivatives in x and y for all vertex attributes. We tried to use the per-fragment derivatives computed by the GPU, but we found numerical issues when the derivatives are computed for fragments close to the near clip plane since we need to divide the attributes by w. This results in errors in the reconstruction of the attributes and causes temporal instabilities. We therefore compute the partial derivatives of the barycentric coordinates λ_i analytically in a separate compute pass (refer to Figure 3) and deduce the partial derivatives for the attributes thereof. Refer to Appendix A for the complete derivation. We need to project the triangle onto the screen for this computation and thus we need to ensure the triangle does not intersect the near clip plane. We use homogeneous clipping [Blinn and Newell 1978], but as we are only interested in computing the derivatives, we use a simplified algorithm and only shrink the triangle to fit inside the clipping volume instead of creating new triangles.

4.3 Shading

With MSAA, a resolve pass needs to be performed per pixel that weights shading samples by the number of corresponding visibility samples. We combine the shading and resolving step by iterating over all visibility samples and shading the corresponding samples as needed. Each of the shaded samples is weighted with its multiplicity in the visibility buffer. In case of the linked list visibility buffer, we traverse the list and shade for each visibility sample. We also store information about the number of covered samples in the linked list and use this to correctly weight all shading samples. Our shading algorithm for the visibility buffer without the linked list is shown in Listing 1. We keep track of a set of visibility samples that still need shading, and for each shaded visibility sample we remove all corresponding covered bits.

This simple approach has the benefit that it does not create any overhead for distributing the workload and for performing the resolve. However, this comes at the cost of potential branch divergence when neighboring pixels need to shade a different number of visibility samples or different materials. We experimented with different approaches using compute shaders to distribute the workload more uniformly per thread, but we were not able to achieve a speedup compared to our simple method.

Our implementation uses bindless textures for accessing the textures during shading and therefore does not require a common storage format across different materials. We avoid storing the tangent frame as an additional attribute by computing the tangent-space from the screen-space derivatives of the world-space position and normal [Schueler 2007], hence reducing the storage costs in the triangle buffer.

This approach to shading requires that all shading is performed using an uber-shader approach. However this is not an inherent limitation in our method since different schemes of dispatching shading computations may be used, such as the previous work by Burns et al. [2013]. The materials used in our evaluation are very similar and share a common triangle storage format which makes the uber-shader approach a good fit.

4.4 Multi-rate Shading

By using a screen-space shading grid [Ragan-Kelley et al. 2011] and deduplicating the shading sample using a memoization cache, we are able to shade at arbitrary rates. For the special case of a shading rate $1/2$ used in our examples, we can skip the memoization cache by communicating inside of a shading quad, employing the `NV_shader_thread_group` extension. If all fragments in the shading quad are covered by the triangle, a common shading sample is spawned, which is shaded at the center of the shading quad. In this case, one of the fragment shader invocations is selected deterministically to create and store the shading sample. Otherwise, for each pixel where the triangle is visible, a sample is created that will be shaded at the pixel center.

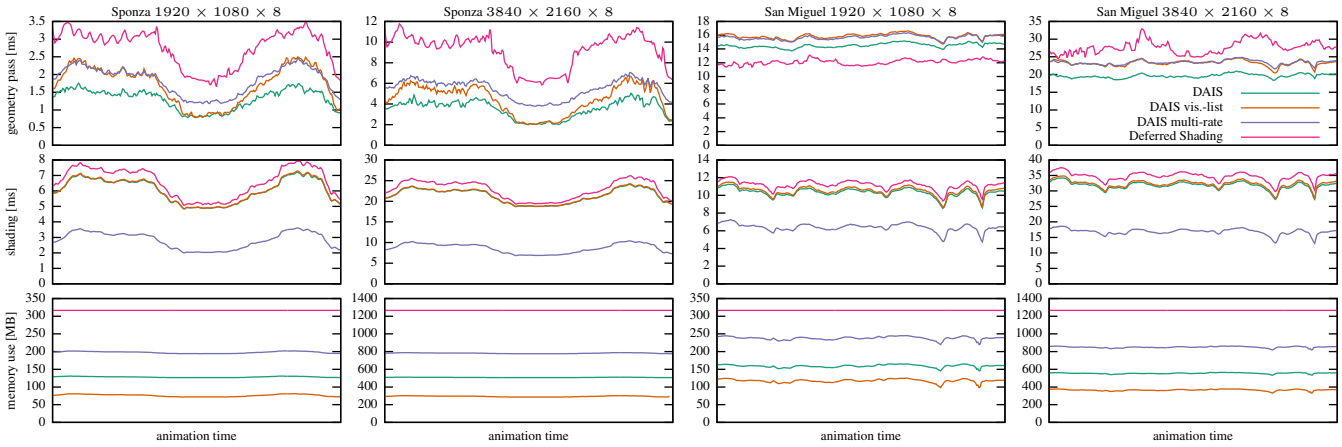


Figure 5: *Timings and total memory usage for an animation in different scenes at different resolutions. We compare our technique,* deferred attribute interpolation shading (DAIS)*, to deferred shading. DAIS vis-list uses a linked list per pixel to store visibility samples. DAIS multi-rate uses a shading rate of* $1/2$ *for computing the indirect illumination. The geometry pass corresponds to the first phase of deferred shading and includes the z-prepass for our method. Memory use includes all buffers needed for shading, i.e. triangle buffer, visibility sample list and multi-rate shading samples. Additional* 80 *kB are used for the memoization cache during the geometry pass. Note the varying y-axis scaling.*

		Timings reported in ms					Memory usage reported in MB		
	Scene	Z-Prepass	Compute Derivatives	Fill Vis.-Buffer (incl. Z-Pre.)	Shade	**Frame**	Triangle Buffer	Vis.-Sample List	**Total**
DAIS	Sponza	1.10	0.08	3.56	20.79	27.01	2.84		509.09
	San Miguel	7.17	0.65	19.75	33.00	66.97	54.69		560.93
DAIS with list	Sponza	1.09	0.08	4.26	20.80	28.18	2.84	5.89	293.50
	San Miguel	7.13	0.64	23.67	33.63	71.81	54.69	34.80	374.25
DAIS multirate	Sponza	1.18	0.08	5.59	8.28	16.51	2.84		782.00
	San Miguel	7.34	0.64	23.65	17.87	55.76	54.69		858.86
	Scene			Fill G-Buffer	Shade	**Frame**			**Size G-Buffer**
Deferred Shading	Sponza			11.35	22.03	35.98			1265.62
	San Miguel			26.87	36.07	77.14			1265.62

Table 1: *Timings and total memory for a static viewport in different scenes at a resolution of* $3840 \times 2160 \times 8$*. The viewports correspond to the start of the animation in Figure 5. We compare our technique,* deferred attribute interpolation shading (DAIS)*, to deferred shading. DAIS vis-list uses a linked list per pixel to store visibility samples. DAIS multi-rate uses a shading rate of* $1/2$ *for computing the indirect illumination.*

As the result of shading is typically of high dynamic range, we compress it into 4 bytes using the LogLuv representation [Larson 1998] to save memory bandwidth when reading back the shading samples during the resolve phase.

5 Results and Discussion

We evaluate three different versions of our algorithm *deferred attribute interpolation shading (DAIS)*. The version *DAIS* employs a conventional multisampled framebuffer and stores 8 bytes per sample, including the depth buffer. The multirate shading version *DAIS multirate* additionally stores 4 bytes per sample to reference the shading sample. The *DAIS list* version employing the linked list visibility buffer uses a multisampled depth buffer and stores one additional 4 bytes list-pointer *per pixel*. Per list element 8 bytes are stored. All materials share the same triangle storage format using 80 bytes per triangle. Per triangle we store $1/w$, normal and texture coordinate as well as their partial derivatives. Furthermore we store a material ID and 4 additional unused bytes for padding purposes.

We compare our method to conventional deferred shading and implemented a deferred renderer with a G-Buffer format requiring 20 bytes per visibility sample, including the depth buffer, storing surface albedo, a triangle ID, linear depth, quantized normal and a material ID. We do not perform a z-prepass for deferred shading since we found it to be detrimental for the performance. To identify visibility samples that correspond to the same triangle the triangle ID is stored as an additional attribute. We iterate over all samples and cache the shaded results per triangle ID to avoid redundant shading. Special care is taken to avoid branch divergence by first searching for the next sample requiring shading and then computing shading across the warp. In contrast to the method by Lauritzen [2010], where either only one or all samples are shaded, we can decide on a finer granularity to determine which samples require shading. This allows us to compare the performance of shading using a geometry buffer to shading using the triangle buffer, with the same shading workload per pixel. To simulate costly computation of indirect illumination we exemplarily use reflective shadow maps [Dachsbacher and Stamminger 2005] with 64 RSM-samples per pixel.

We selected the scenes *Sponza* and *San Miguel* with 262 267 and 8 145 860 triangles respectively, to evaluate the effects of the number of triangles in terms of performance and memory savings. For all measurements we used 8 visibility samples per pixel. We unconditionally draw all geometry in all scenes and do not use culling. All benchmarks were performed using a NVIDIA Geforce GTX 980.

Figure 5 depicts timings and memory consumption for a camera animation in two different scenes. The reported timings are for filling the visibility buffer or geometry buffer, as well as for shading. Refer to the supplemental material for a video showing the camera paths. Our method is able to significantly reduce the memory cost, while achieving faster render times in most cases. In the Sponza scene, our method is able to outperform deferred shading at both resolutions, while at the same time reducing the memory usage. In the San Miguel scene, our method is slower than deferred shading using a resolution of 1920×1080, but is still able to reduce the memory usage significantly. With doubled screen resolution, the time needed

to render the visibility buffer only increases by about 40%, whereas for deferred shading the time is more than doubled, leading to a performance advantage for our method. This characteristic can be explained by the two geometry passes we have to perform, leading to a bigger overhead for highly tessellated scenes. In Table 1 we show timings as well as memory statistics in more detail for rendering one frame at the resolution of 3840×2160 for both scenes. Using our multi-rate shading technique, we are able to reduce the time needed for shading significantly, however at the cost of increased memory usage and runtime for creating the visibility buffer. In total, the render time for one frame is significantly reduced. The memory needed for the multi-rate shading is still significantly lower compared to deferred shading, and can be reduced by combining it with the linked list visibility buffer.

To evaluate the overhead of our method when tessellation shaders are used, we implemented a simple tessellation shader which computes the projected triangle size and sets the tessellation factor to obtain approximately equally sized triangles in image space. Figure 7 shows timings for the geometry pass as well as the total memory consumption for a flythrough animation. Note that we did not cull triangles from the tessellation shader to generate a high geometric workload. Our method is able to save space in both cases, however our method is not able meet the performance of deferred shading in case of the lower resolution. At the higher resolution our method is able to compete performance-wise.

Figure 6 shows the difference between attributes interpolated by the rasterizer and our method. The error increases for distant surfaces and surfaces observed from a shallow angle, which is to be expected since the magnitude of the derivatives increases in these regions.

Comparison to previous work Previous work [Kerzner and Salvi 2014; Crassin et al. 2015] reduces shading computations by merging surfaces and at the same time achieves a scene-independent reduction in memory consumption. Our method captures all visible surfaces per pixel and we therefore achieve the same image quality as standard MSAA methods, but nevertheless we could achieve better compression ratios in our experiments. Merging surfaces may lead to noticeable artifacts, however this merging could be integrated into our method as well to reduce shading costs. In contrast to our work, the aforementioned algorithms cannot support multi-rate shading.

The geometry pass in the method proposed by Burns et al. [2013] is cheaper compared to our method since it does neither require the z-prepass nor store visible triangles. They reference static mesh data which makes their method hard to use with tessellation, and more costly during shading since they interpolate attributes using a ray-triangle intersection that requires vertex transformations to be computed per pixel. Their visibility buffer stores the same amount of data as our method, but we additionally store the triangle buffer. This overhead, however, is minor, and by employing the linked list of visibility samples we can achieve even higher compression ratios than their method. In comparison to the method by Liktor et al. [2012] our method is more memory efficient because they need to store one G-Buffer sample per shading sample, whereas in our method the cost for storing triangles is amortized over several shading samples which is especially beneficial in case of multi-rate shading. Note that their method was designed for stochastic rasterization which cannot be used with our method.

6 Limitations

Employing a visibility buffer instead of a G-Buffer poses several challenges. Techniques that directly operate on the G-Buffer, for example deferred decals, cannot be used without adding additional buffers, thereby sacrificing storage benefits. Furthermore, since materials are evaluated during shading, either a more elaborate dispatch scheme must be employed for shading, or materials have to be evaluated using an uber-shader. For simplicity, we chose to employ an uber-shader approach for this work, but it may not be suitable for complex material shaders where potential branch divergence and increased register pressure might degrade the performance.

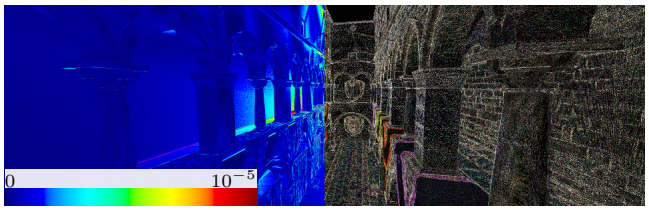


Figure 6: *Numerical errors due to our attribute interpolation. In the left part of the image the distance to the world-space position computed by the rasterizer is shown. The scene is scaled to fit into a* $[-1, 1]^3$ *cube. On the right the absolute difference between evaluated texture lookups is shown, amplified by* 10^3.

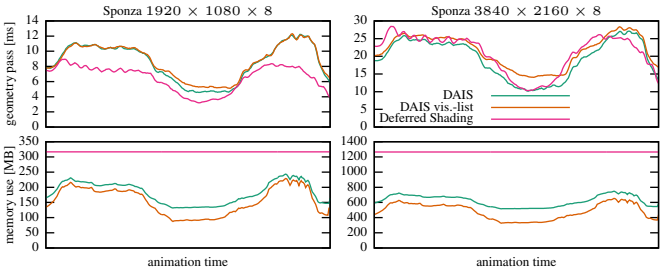


Figure 7: *Timings and total memory use for the geometry pass (including the z-prepass for DAIS) in the Sponza scene. A tessellation shader is used to create triangles with approximately equal screen-space size, generating up to* 13 *and* 26 *million triangles for a resolution of* $1920 \times 1080 \times 8$ *and* $3840 \times 2160 \times 8$ *respectively.*

Our method requires a geometry prepass which may impact the performance in case of high geometry workloads and expensive vertex shaders. We therefore advise to use effective means of culling with our method. Highly tessellated scenes also pose a challenge, as all visible triangles need to be stored, and the costs for the two geometry passes are amplified. The storage costs of our algorithm depend on the number of attributes since for each additional scalar attribute we need to store three scalars in the triangle buffer. As mentioned previously in Section 3.2, referencing triangles from the original mesh can be beneficial in case of high attribute counts.

Multi-rate shading samples are spawned in the geometry pass. Because they are not generated directly by the rasterizer and shaded in a separate compute pass, techniques based on the stencil buffer cannot be used. Violating the assumption of a low-frequency shading signal might lead to aliasing. Since we do not interpolate between shading samples our technique might create blocky artifacts. Vaidyanathan et al. [2014] evaluated the image quality for a similar multi-rate shading approach.

7 Conclusion and Future Work

In this paper, we presented a novel approach to deferred shading suitable for high visibility sampling frequencies. We are able to improve the performance compared to deferred shading, while at the same time dramatically reducing memory costs. In contrast to deferred shading, our approach enriches the shading phase with knowledge about the partial derivatives of all stored attributes. Using triangle references, we can identify duplicate visibility samples per pixel and thereby avoid redundant shading. We do not require random

access to all triangles since we identify visible triangles using two passes and store these triangles in a buffer, thereby enabling the use of tessellation. Our method is sensitive to the number of drawn triangles since it uses two geometry passes. We showed that the size of the visibility buffer can be reduced even further by storing a linked list of visibility samples per pixel, with moderate computational overhead.

The visibility buffer is of low entropy, as many visibility samples store the same reference. Our technique therefore works well with memory compression, which is already applied transparently by current GPUs [NVIDIA 2014], and can be beneficial for energy efficiency. Furthermore, our method is much more cache friendly than deferred shading since for many visibility samples, the same triangle data is requested, in contrast to deferred shading where each visibility sample corresponds to an individual block of memory.

Since geometric attributes can be evaluated during shading, our method could improve or actually enable new screen-space techniques. An interesting direction of future research is to eliminate the z-prepass by using a dynamic memory allocation scheme that is able to identify and free the memory for triangles that became completely occluded. Our method might benefit from small changes to the graphics pipeline. If the triangle setup performed by the rasterizer was exposed in the fragment shader, our method would not need to rely on the geometry shader to pass through the whole triangle, and furthermore the compute pass for the derivatives of the attributes would be superfluous.

Acknowledgements

The first author is supported by a LGF stipend of the State of Baden-Württemberg. We would like to thank Crytek for the *Sponza* scene and Guillermo M. Leal Llaguno for the *San Miguel* scene.

References

AKELEY, K. 1993. Reality engine graphics. In *Proc. of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, SIGGRAPH '93, 109–116.

BLINN, J. F., AND NEWELL, M. E. 1978. Clipping using homogeneous coordinates. In *Computer Graphics (Proc. SIGGRAPH)*, vol. 12, 245–251.

BURNS, C. A., AND HUNT, W. A. 2013. The visibility buffer: A cache-friendly approach to deferred shading. *Journal of Computer Graphics Techniques (JCGT) 2*, 2, 55–69.

CLARBERG, P., TOTH, R., AND MUNKBERG, J. 2013. A sort-based deferred shading architecture for decoupled sampling. *ACM Trans. on Graphics (Proc. SIGGRAPH) 32*, 4, 141:1–141:10.

CRASSIN, C., MCGUIRE, M., FATAHALIAN, K., AND LEFOHN, A. 2015. Aggregate g-buffer anti-aliasing. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 109–119.

DACHSBACHER, C., AND STAMMINGER, M. 2005. Reflective shadow maps. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '05, 203–231.

DAVIDOVIČ, T., ENGELHARDT, T., GEORGIEV, I., SLUSALLEK, P., AND DACHSBACHER, C. 2012. 3D rasterization: a bridge between rasterization and ray casting. In *Proc. of Graphics Interface*, 201–208.

FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. Data-parallel rasterization of micropolygons with defocus and motion blur. In *Proc. of ACM SIGGRAPH / Eurographics conference on High Performance Graphics*, 59–68.

JIMENEZ, J., ECHEVARRIA, J. I., SOUSA, T., AND GUTIERREZ, D. 2012. SMAA: Enhanced subpixel morphological antialiasing. *Computer Graphics Forum 31*, 355–364.

KERZNER, E., AND SALVI, M. 2014. Streaming g-buffer compression for multi-sample anti-aliasing. In *Proc. of ACM SIGGRAPH / Eurographics conference on High Performance Graphics*, 1–7.

KOPF, J., COHEN, M. F., LISCHINSKI, D., AND UYTTENDAELE, M. 2007. Joint bilateral upsampling. *ACM Trans. on Graphics 26*, 3.

LARSON, G. W. 1998. LogLuv encoding for full-gamut, high-dynamic range images. *Journal of Graphics Tools 3*, 1, 15–31.

LAURITZEN, A. 2010. Deferred rendering for current and future rendering pipelines. *SIGGRAPH Course: Beyond Programmable Shading*.

LIKTOR, G., AND DACHSBACHER, C. 2012. Decoupled deferred shading for hardware rasterization. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 143–150.

NVIDIA, 2014. White paper: NVIDIA GeForce GTX 980. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.

RAGAN-KELLEY, J., LEHTINEN, J., CHEN, J., DOGGETT, M., AND DURAND, F. 2011. Decoupled sampling for graphics pipelines. *ACM Trans. on Graphics 30*, 3, 17:1–17:17.

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

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

SCHUELER, C. 2007. Normal mapping without pre-computed tangents. In *ShaderX5: Advanced Rendering Techniques*, W. F. Engel, Ed. Charles River Media.

VAIDYANATHAN, K., SALVI, M., TOTH, R., FOLEY, T., AKENINE-MÖLLER, T., NILSSON, J., MUNKBERG, J., HASSELGREN, J., SUGIHARA, M., CLARBERG, P., ET AL. 2014. Coarse pixel shading. In *Proc. of ACM SIGGRAPH / Eurographics conference on High Performance Graphics*, 9–18.

A Computing Partial Derivatives for Perspective Correct Attribute Interpolation

The barycentric coordinates λ_i of a point (x, y) in relation to a triangle $p_i = (u_i, v_i)$ can be computed by

$$\begin{aligned}\lambda_1(x,y) &= \frac{(v_2 - v_3)(x - u_3) + (u_3 - u_2)(y - v_3)}{D},\\ \lambda_2(x,y) &= \frac{(v_3 - v_1)(x - u_3) + (u_1 - u_3)(y - v_3)}{D},\\ \lambda_3(x,y) &= 1 - \lambda_1(x,y) - \lambda_2(x,y),\end{aligned} \tag{3}$$

with $D = \det(p_3 - p_2, p_1 - p_2)$. From Eq. 3, the partial derivatives of the barycentric coordinates with respect to x and y are derived as

$$\begin{aligned}\tfrac{\partial \lambda_1}{\partial x} &= \frac{y_2 - y_3}{D}, \quad \frac{\partial \lambda_2}{\partial x} = \frac{y_3 - y_1}{D}, \quad \frac{\partial \lambda_3}{\partial x} = \frac{y_1 - y_2}{D},\\ \frac{\partial \lambda_1}{\partial y} &= \frac{x_3 - x_2}{D}, \quad \frac{\partial \lambda_2}{\partial y} = \frac{x_1 - x_3}{D}, \quad \frac{\partial \lambda_3}{\partial y} = \frac{x_2 - x_1}{D}.\end{aligned} \tag{4}$$

Given the partial derivatives of the barycentric coordinates, we can compute the partial derivatives of an attribute a/w using

$$\frac{\partial a/w}{\partial x} = \sum_i \frac{\partial \lambda_i}{\partial x} \cdot \frac{a_i}{w_i}, \quad \frac{\partial a/w}{\partial y} = \sum_i \frac{\partial \lambda_i}{\partial y} \cdot \frac{a_i}{w_i}. \tag{5}$$