To appear in EUROGRAPHICS 2014

Adaptive Texture Space Shading for Stochastic Rendering

M. Andersson,[†1,2] J. Hasselgren,[1] R. Toth,[1] and T. Akenine-Möller[1,2]

[1]Intel Corporation
[2]Lund University

Abstract

When rendering effects such as motion blur and defocus blur, shading can become very expensive if done in a naïve way, i.e. shading each visibility sample. To improve performance, previous work often decouple shading from visibility sampling using shader caching algorithms. We present a novel technique for reusing shading in a stochastic rasterizer. Shading is computed hierarchically and sparsely in an object-space texture, and by selecting an appropriate mipmap level for each triangle, we ensure that the shading rate is sufficiently high so that no noticeable blurring is introduced in the rendered image. Furthermore, with a two-pass algorithm, we separate shading from reuse and thus avoid GPU thread synchronization. Our method runs at real-time frame rates and is up to 3× faster than previous methods. This is an important step forward for stochastic rasterization in real time.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

1. Introduction

Stochastic rasterization [CCC87, AMMH07] continues to be an interesting path forward for realistic camera models for real-time rendering. Camera effects, such as motion blur and depth of field (also called defocus blur), are favorably expressed with this technique. For good quality, the number of samples per pixel often needs to be high, and due to motion or lens effects, standard multi-sampled anti-aliasing (MSAA) will often deteriorate to super-sampled anti-aliasing [MCH*11]. This, in turn, means that shading becomes prohibitively expensive. Hence, better methods are needed for shading in a stochastic rasterizer.

To this end, new hardware mechanisms for reusing shaded values for many different rasterized fragments in a stochastic rasterization setting have been proposed [RKLC*11, BFM10, CTM13]. Common to all of these approaches is that they assume that shading is constant for a certain point on a surface, regardless of motion and the lens. Note that this assumption is heavily used even in high-quality production renderers [CCC87].

In contrast to new hardware mechanisms, Liktor and Dachsbacher [LD12] present a method that works on current graphics hardware. They use a compact geometry buffer, which stores shading information independent of visibility. When computing and storing new entries in this buffer they rely on per-fragment synchronization and atomic counters. While our method has the same goals, i.e., reusing shaded values as efficiently as possible, it has been designed without high frequency synchronization. The contributions of our work are summarized below:

- A lock-free (no atomics) method for reusing shading values in a stochastic rasterizer.
- Sparse and adaptive evaluation of shading (triangles are shaded directly into a hierarchical data structure).
- An algorithm well suited for current graphics processors.
- Results show that our method is substantially faster than previous methods.

We believe that our method brings us one step closer to high-quality real-time rendering of stochastic effects on current graphics hardware.

2. Previous Work

The idea of decoupling shading rate from visibility sampling [CCC87] is old and has frequently been used in offline or photo-realistic rendering systems to improve performance. In the following, we will focus mainly on the more recent approaches targeting real-time performance and GPU

[†] magnus.andersson@cs.lth.se

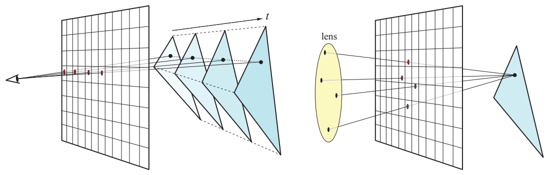

Figure 1: *Caching shaded values works for both motion blur (left) and defocus blur (right). The assumption, used by Cook et al. and many others after them, is that the shading at a certain point on a triangle is independent of time and lens position. Hence, shaded values can be cached and later reused.*

implementations. The idea of a shading cache is illustrated in Figure 1.

Many of the shading caches adopted for GPU implementation target temporal reprojection of shaded colors [NSL*07, SaLY*08]. While there is some overlap with our work, we only wish to reuse shaded values over different stochastic samples, and may therefore work with parametric texture space shading. In contrast, temporal reprojection algorithms typically work with screen space reprojection between frames and those techniques are thus not well suited for exploiting coherency between samples in a single frame.

As our algorithm works in object- or texture-space, it bears some resemblance to light mapping [AMHH08], which has been a popular technique (primarily) for baking static global illumination for computer games. Illumination maps were used by Arvo [Arv86] for storing the result of tracing rays from the light sources, and Heckbert [Hec90] stored radiosity in textures, which adapted their size to the shaded content. Some similarities can be seen between our approach and GPU accelerated light map generation [LTH*13], although those algorithms are typically more complex as they are targeted at solving light transport through the scene. However, a big difference between light map generation and our shading approach is that light maps are typically view-independent and should be suitable for viewing from any camera position. In contrast, since we will recompute the shading every frame, we can use the camera parameters to choose an appropriate sampling rate and texture filtering footprint. Texture space shading has also previously been used by Borshukov and Lewis [BL03] for realistic skin rendering.

Recently, several decoupled shading approaches targeting real-time rendering and graphics hardware [RKLC*11, BFM10, VTS*12, CTM13], have been developed. However, these papers focus on new hardware implementations, and are not easy to implement in the context of current generations of GPUs and graphics APIs. Of particular interest to us is the work by Vaidyanathan et al. [VTS*12] in which they derive minimum shading rates for defocus and motion blur. Liktor and Dachsbacher [LD12] built on the work by Ragan-Kelley et al. [RKLC*11] and presented a deferred shading cache approach. They implemented their algorithm using shader programs and showed performance gains on existing GPUs with stochastic rasterization. However, the algorithm requires per-sample synchronization when the shading cache is updated. This has a significant performance impact, and consequently the algorithm starts being beneficial only when expensive shaders are used.

Gribel et al. [GBAM11] use an object-space shading cache for offline rendering with semi-analytical methods. Similar to our approach, they also use a hierarchy and populate it as needed. They allocate a large chunk of memory for their cache, where subsets of the hierarchy are allocated lazily. Atomics must be used to avoid race conditions. Adapting this algorithm to run on the GPU would probably result in an algorithm similar to that of Liktor and Dachsbacher [LD12].

Although this paper focuses on the shading aspect, we have also implemented a stochastic rasterizer for evaluation purposes. Our implementation is very similar to that of McGuire et al. [MESL10], but modified to perform shading texture lookups in the pixel shader rather than computing actual shading. We also use the five-dimensional sample-in-triangle test proposed by Laine and Karras [LK11], as well as the back-face culling test for motion and defocus blur by Munkberg and Akenine-Möller [MAM11].

3. Algorithm

Our shading approach is split into two passes, called the *shading pass* (Section 3.1) and the *stochastic rasterization pass* (Section 3.2). In the first pass, illustrated in Figure 2, we set up a hierarchical, i.e., mipmap-like, *shading texture* for each object. We do this by rasterizing the current object directly into the shading texture using the object's parametric shading space (or texture space) coordinates. During this pass, we also compute how each triangle will project on screen during the succeeding stochastic rasterization pass, and use that information to dynamically select an appropriate mipmap level to render into. In the second pass, where we typically wish to use stochastic rasterization effects, such as defocus and/or motion blur, we simply fetch the color value from the shading texture for each fragment. The division of our algorithm into two passes is crucial to performance because it enables lock-free lookups of shaded values. This should be compared to previous methods that require fine-grained thread synchronization [LD12].

By switching from the "shade on miss" model of the original cache systems, we lose the ability to shade only fragments that are visible from the camera and risk overshading if we choose a texture resolution that do not match the screen resolution. Therefore, it is important not only to perform back-face and frustum culling when setting up the shading texture, but also to select a mipmap level in the shading texture that closely matches the screen space signal frequency (including blur from depth of field).

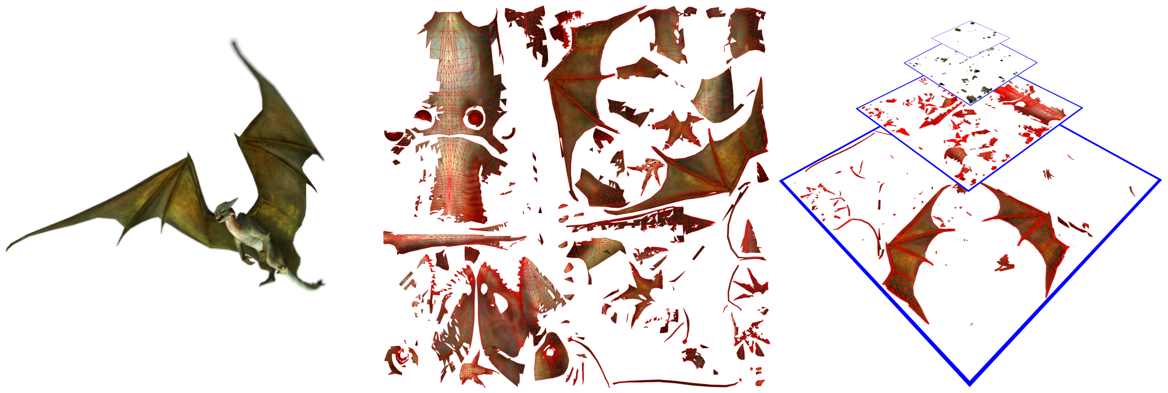

Figure 2: *An illustration of our object-space texture shading approach. Left: the Dragon object rendered with defocus blur using our object-space shading approach. Middle: the mesh rendered with shading to the object-space texture after view-frustum and back-face culling (for the camera used in the left image). Right: the same mesh rendered to object space, with an appropriate mipmap level selected per-triangle in the geometry shader. Note that the image in the middle is there for illustration purposes only—our method renders only to the structure, called a shading texture, shown to the right.*

We typically use one shading texture per object. If the scene requires more textures than is possible to allocate, we keep a pool of a few shading textures and alternate between pass one and pass two. It is beneficial to merge smaller objects and use the same shading texture whenever possible, since that minimizes the involved state changes when alternating between the passes.

Next, the two passes of our algorithm are described in more detail.

3.1. Shading Pass

The shading pass consists of two sub-passes, described in Section 3.1.3, in which we sparsely populate a hierarchical shading texture with shaded fragments. The stochastic rasterization pass can thereafter reuse these shaded values several times. In contrast to other research dedicated to lowering shading rate for stochastic rasterization, we propose to pay the shading cost up front, prior to visibility determination, which is similar to how Reyes handles shading [CCC87].

First, the mesh vertices need to be augmented with shading space coordinates, which must ensure that all surface points on the mesh can be uniquely mapped to a location in shading space. Texture coordinates are often already available and can double as the shading space so long as there are no overlaps. We would also like to point out that our algorithm can be used selectively, and alternative algorithms could be used for objects where non-overlapping atlases are an issue. Furthermore, like previous work [BFM10, RKLC*11, VTS*12, CTM13] all shading is calculated at the center of the lens and at a fixed time. The lack of view-dependent shading is widely regarded as an acceptable trade-off, given the complexity of the problem.

In Section 3.1.1 and 3.1.2, we describe the theory on how to conservatively determine the highest shading frequency required for any point on the primitive, in order to shade as sparsely as possible. Section 3.1.3 then describes how this is used in practice in order to generate the shading texture.

3.1.1. Shading Rate

Our algorithm for selecting shading rate is illustrated in Figure 3. Starting from a triangle in screen space, we use a method, called *anisotropic adaptive sampling* (AAS), proposed by Vaidyanathan et al. [VTS*12]. Given a lens with particular characteristics, AAS can be used to determine the screen-space shading rate needed in order to capture a certain desired percentage of the frequency content due to defocus blur.

We use the thin lens model, where the (non-signed) screen-space circle of confusion (CoC) radius can be modelled as:

$$r_c(w) = \left|\frac{c_0 + wc_1}{w}\right|, \tag{1}$$

where c_0 and c_1 are parameters derived from the camera's aperture and focal distance, and w is the vertex depth. Note that the focus plane is located at $w = -c_0/c_1$, since $r_c(-c_0/c_1) = 0$. Similar to the derivation of AAS, we find the smallest circle of confusion, $\min_w r_c(w)$, as shown in Figure 3B, and use it to bound the shading frequency. If the circle of confusion is less than a pixel, or if the triangle straddles the focus plane, the frequency is bound by the pixel grid instead. This gives us a shading grid spacing, or inverse shading rate, expressed in pixels:

$$d = \max(a(\min_w r_c(w)), 1), \tag{2}$$

where the function a depends on the aperture of the lens. We use a circular lens in our models and use $a(x) =$

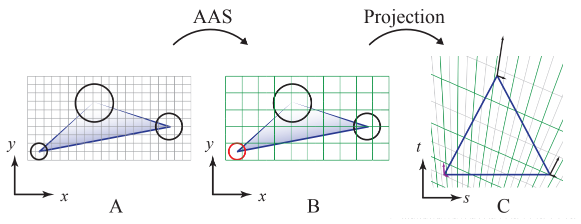

Figure 3: *Shading rate algorithm outline. A: triangle to be rendered. B: we use the AAS algorithm to derive a screen space shading grid, or inverse shading rate, which is dictated by the minimum circle of confusion shown in red. C: the shading grid is projected into shading space (note that it is not aligned with the Cartesian grid).*

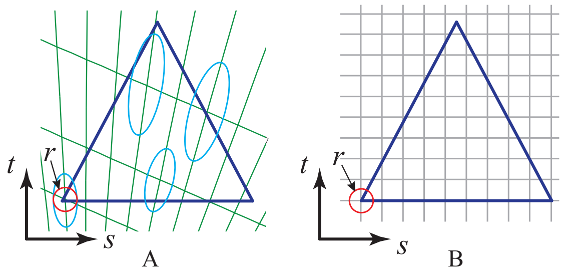

Figure 4: *Finding the highest frequency in texture space. A: transforming the AAS grid into texture space gives us a projective grid that does not align with the Cartesian grid we use for our shading space. B: the resolution of our shading space corresponds to the smallest radius (red circle to the left) of the filter kernels used.*

$x/2$ [VTS*12]. For mipmap level selection, we only consider defocus blur, while the full AAS algorithm handles the combination of both motion and defocus blur.

Once we have established a suitable shading grid spacing in screen space (Equation 2), we wish to transform it into our shading space. This is a projective transform, and we will therefore end up with a grid similar to that shown in Figure 3C. It should be noted that this grid is not aligned with the standard Cartesian grid, which we use for our shading space. However, if we assume that shading is filtered using EWA [Hec86] (or using hardware-accelerated anisotropic filtering), we obtain the Gaussian filter kernels illustrated in Figure 4A.

The filter kernels can be modeled as ellipses [Hec86], and are usually computed from the screen-space derivatives, $\frac{\partial \mathbf{T}}{\partial x}$ and $\frac{\partial \mathbf{T}}{\partial y}$, for a shading space texture coordinate, $\mathbf{T} = (s,t)$. We assume that the mapping $(x,y) \rightarrow \mathbf{T}$ is locally linear, and therefore the shading grid spacing computed using AAS (d from Equation 2) can be directly used to scale the screen-space derivatives: $d \cdot \frac{\partial \mathbf{T}}{\partial x}$ and $d \cdot \frac{\partial \mathbf{T}}{\partial y}$.

Given these derivatives, one may compute the minor axis

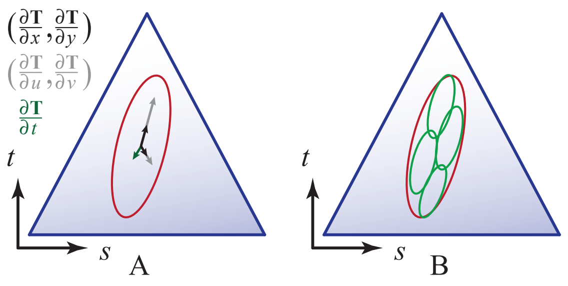
$\left(\frac{\partial \mathbf{T}}{\partial x}, \frac{\partial \mathbf{T}}{\partial y}\right)$
$\left(\frac{\partial \mathbf{T}}{\partial u}, \frac{\partial \mathbf{T}}{\partial v}\right)$
$\frac{\partial \mathbf{T}}{\partial t}$
t
s
A
t
s
B

Figure 5: *A: a triangle and its corresponding shading space along with texture derivatives with respect to x, y, u, v, and t. Given these five derivatives, we generalize the method by Loviscach [Lov05] to compute a single elliptical footprint, which takes all partial derivatives into account. B: depending on whether we access the shading texture once per rasterized pixel (MSAA) or once per sample (SSAA), we should pre-filter the shading texture with an appropriate footprint. The figure shows a pixel footprint used for MSAA as the red ellipse. If SSAA is used instead, we will get multiple lookups in the shading texture, illustrated as the green ellipses, and the footprints need to be scaled to reflect this.*

of the ellipse as described in Appendix A. As shown in Figure 4A, the smallest minor axis:

$$r = \min_{x,y \in tri} R_{minor}(x,y), \qquad (3)$$

of any ellipse over the whole triangle indicates the highest visible content frequency. It should be noted that this is a non-trivial function that is difficult to bound conservatively. We therefore conjecture that the minimum minor axis radius across a triangle occurs in one of the triangle vertices. Currently, we have no firm proof for that, but it matches previous knowledge that the mipmap level can be perspectively interpolated across a triangle in high quality [EWWL98]. Given the minimum radius, r, shown in Figure 4, we select the mipmap level as $\lfloor \log_2 r \rfloor$, and we end up with the grid shown in Figure 4B. This choice ensures that we shade densely enough to resolve details, while the actual filter footprint takes the non-discretized r into account.

In practice, the maximum shading frequency is limited by the finest mipmap level resolution. Having a low resolution shading texture and moving sufficiently close to an object may cause some blockiness, and therefore it is important to select a suitable maximum shading texture resolution for each asset. Selecting a resolution that is too high only affects memory consumption, as our shading system will pick an appropriate mipmap level for actual shading.

3.1.2. Filtering

After selecting an appropriate mipmap level for the triangle in the geometry shader, the triangle will be rasterized to the shading texture. The pixel shader is executed for

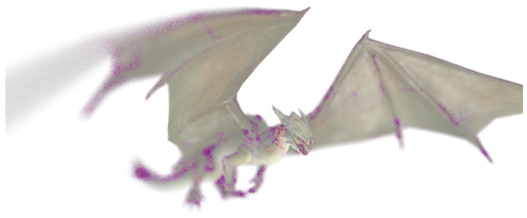

Figure 6: *Without conservative rasterization, the regions around triangle edges in the shading texture may have incomplete information, which propagates to the final image as shown here (emphasized in purple color).*

every pixel in the shading texture that overlaps the triangle, in the selected mipmap level. In the pixel shader, we compute pre-filtered shading values, filtering over the footprint of each pixel with respect to (x,y,u,v,t), where (x,y) are the screenspace coordinates, (u,v) are the lens coordinates and t is the shutter time. In the pixel shader, we must maintain enough triangle data so that we can compute $\frac{\partial \mathbf{T}}{\partial x}, \frac{\partial \mathbf{T}}{\partial y}, \frac{\partial \mathbf{T}}{\partial u}, \frac{\partial \mathbf{T}}{\partial v}, \frac{\partial \mathbf{T}}{\partial t}$, in order to determine the filter footprint. We use a generalized form of the filter formulated by Loviscach [Lov05], who only covered the case of screen space and motion blur filtering. The full derivation of how the generalization is done is found in Appendix B. The resulting combined filter has an elliptical kernel, which is used for computing the shading of the pixel.

In practice, we let the hardware for anisotropic texture filtering effectively perform the filtering for us, which is in accordance with Loviscach's approach. An example of the combined filter for a sample position on a triangle with defocus and motion blur can be seen in Figure 5A.

We must make sure that the correct derivatives are used when pre-filtering, depending on how the shading texture is sampled in the subsequent stochastic rasterization pass, as illustrated in Figure 5B. If supersampling is used, several samples will be drawn from within the pixel, lens, and time footprints in the shading texture. The pixel, lens, and time derivatives must therefore be scaled prior to shading to ensure that the color value is correctly reconstructed. We assume that we have N evenly distributed sample points, and we therefore divide the (x,y) and (u,v) derivatives by $\sqrt{N}$ as this will distribute the pixel and lens area evenly among the samples. Similarly, the time derivative is divided by N since it is a one-dimensional attribute. For multisampling, we only wish to retrieve one shaded value per pixel, and thus $N = 1$. Regardless of what strategy we use for sampling the shading texture, we use the grid resolution of the selected shading texture mipmap level as a lower bound of the extent of the filter.

3.1.3. Populating the Shading Texture

As described in the previous subsections, we now have methods to compute an appropriate shading rate, i.e., mipmap level, per triangle. The actual population of the shading texture is done in two sub-passes. In the geometry shader of the first sub-pass, the triangles are conservatively view-frustum culled and are back-face culled with respect to the lens and time, using the method described by Munkberg and Akenine-Möller [MAM11]. The surviving triangles are then rasterized to the shading texture using the shading space coordinates transformed to the selected mipmap level as the position attribute. In the pixel shading stage, we recompute the texture footprint and perform the surface shading.

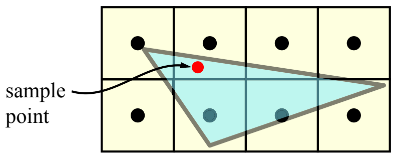

Figure 7: *With standard rasterization rules, only the two middle pixels in the bottom row would be considered to be inside this triangle since those pixels' center locations are the only ones that are inside the triangle. However, our sample locations when looking up shading in the shading texture can be arbitrary inside the triangle. The red sample point, for example, is inside the triangle, but the enclosing pixel would not be rendered to with standard rasterization. We overcome this with conservative rasterization, which processes all pixels that are touched by the triangle (which in this case includes all* 4×2 *pixels).*

Current graphics APIs do not allow dynamically selecting the output mipmap level, as mentioned in Section 3.1, and we therefore manually maintain our own mipmap hierarchy. Using this layout, we select a mipmap level for each triangle by scaling and offsetting the texture coordinates. This ensures that we can use a standard hardware accelerated bilinear texture lookup, instead of doing custom texture filtering in the pixel shader, when performing the stochastic rasterization pass (described in Section 3.2).

Using standard rasterization results in artifacts in the final image, which can be seen in Figure 6. The reason for this is that our sample locations are arbitrary within a triangle, while the rasterization hardware samples visibility at the pixel center. This is explained further in Figure 7. As described next, we use conservative rasterization [AMA05, HAMO05] in a second sub-pass to solve this problem.

In the second sub-pass, the same set of triangles are rendered with conservative rasterization. First, the culling and shading rate computations from the first pass are repeated. Next, we build the conservative outer hull [HAMO05] of the input triangle for the selected mipmap level. It is important that the shaded values that are already computed in the first sub-pass are not overwritten. To accomplish this, we

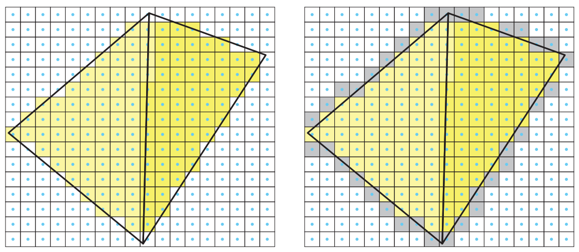

Figure 8: *The inner regions are yellow and the visible parts of the outer regions are gray. Left: the inner regions of two triangles are rendered in the first sub-pass. The depth is set to z_i. Right: the outer regions of the triangles are rendered using conservative rasterization in the second sub-pass, with depth z_o, where $z_o > z_i$. Since the depth of the inner region is closer than the depth of the outer region, the gray pixels are only located around the triangles.*

use a depth buffer in both the first and the second sub-pass with the depth test set to *less than*. The first sub-pass writes a smaller depth value the second sub-pass. This guarantees that the formerly shaded values have priority over the latter. An example of this process can be viewed in Figure 8.

3.2. Stochastic Rasterization Pass

In the second pass, we use a stochastic rasterizer, similar to that of McGuire et. al [MESL10], to render the final image. However, we want to point out that our shading algorithm is orthogonal to the rasterization algorithm. For example, we use optimized inside tests [LK11].

Since the shading texture was completely set up during the first pass, we do not need to handle cache misses and therefore the second pass becomes straightforward. Instead of traditional stochastic shading approaches [MESL10, MCH*11], we simply perform a texture lookup into the shading texture for each sample being rendered. We must use the exact same model as used in the first pass when computing the mipmap level, since only one level per triangle is populated with shaded data. The texture coordinates for the shading texture can be interpolated using the perspective-correct barycentric coordinates for a sample. The barycentric coordinates are computed as a byproduct of our sample-in-triangle inside test, which we also need to execute for each sample as part of the stochastic rasterization [AMMH07, MESL10].

By default, we use a per-sample frequency shader for the stochastic rasterization pass. However, as an optimization it is possible to run a per-pixel shader while still inside-testing each sample. In this case, we only need to perform a single texture lookup per pixel in an algorithm that mimics multisampling. However, due to API limitations, it is only possible to output a single depth value if the shader is executed on a per-pixel basis and this may lead to artifacts similar to the shading approach proposed by McGuire et al. [MESL10]. The artifacts may be quite severe in some cases, as illustrated in Figure 9. However, the multisampling approach performs significantly better than supersampling approaches on current GPUs, and it may be valuable if performance is crucial.

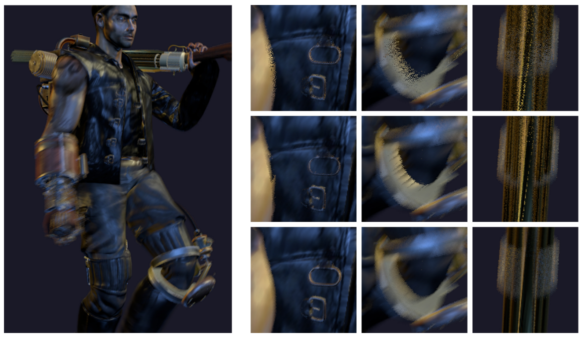

Figure 9: *Examples of artifacts caused by writing a single depth compared to correct per-sample depth. The top row uses the depth of the last covered sample, the middle row uses the depth of the nearest sample, and the bottom row uses the correct per-sample depth.*

After the stochastic rasterization pass has finished, the image is complete. Since the hardware is used to composite the framebuffer from all triangles in the second pass, blending works as in any forward renderer. Note that this is not possible with deferred shading methods, such as the one proposed by Liktor and Dachsbacher [LD12].

4. Results

We compare our implementation to the deferred shading approach by Liktor and Dachsbacher [LD12] and stochastic rasterization with supersampled shading. The original stochastic rasterizer proposed by McGuire et al. [MESL10] relied on multisampling, but it is easy to extend to supersampling and we used this as reference. When possible, we implemented both supersampled and multisampled versions of the algorithms, since multisampling is desired if performance is a primary concern. We use supersampling unless otherwise is indicated.

Our implementation of Liktor and Dachsbacher's algorithm differs somewhat from their description. In their implementation, a cache maintains *ssIDs*, which uniquely identify a shading point in shading space, coupled with a particular primitive. In the geometry shader, each primitive is assigned a range of ssIDs, which is an operation that requires an atomic counter to avoid collisions. However, they implemented their algorithm using OpenGL 4.2, while we based all our algorithms on Direct3D 11, which lacks unordered access binding for the geometry shader stage. We remedied this limitation by adding a 32-bit `primitiveID` field to each cache entry to uniquely identify each shading point and primitive coupling. The hash function was also modified to

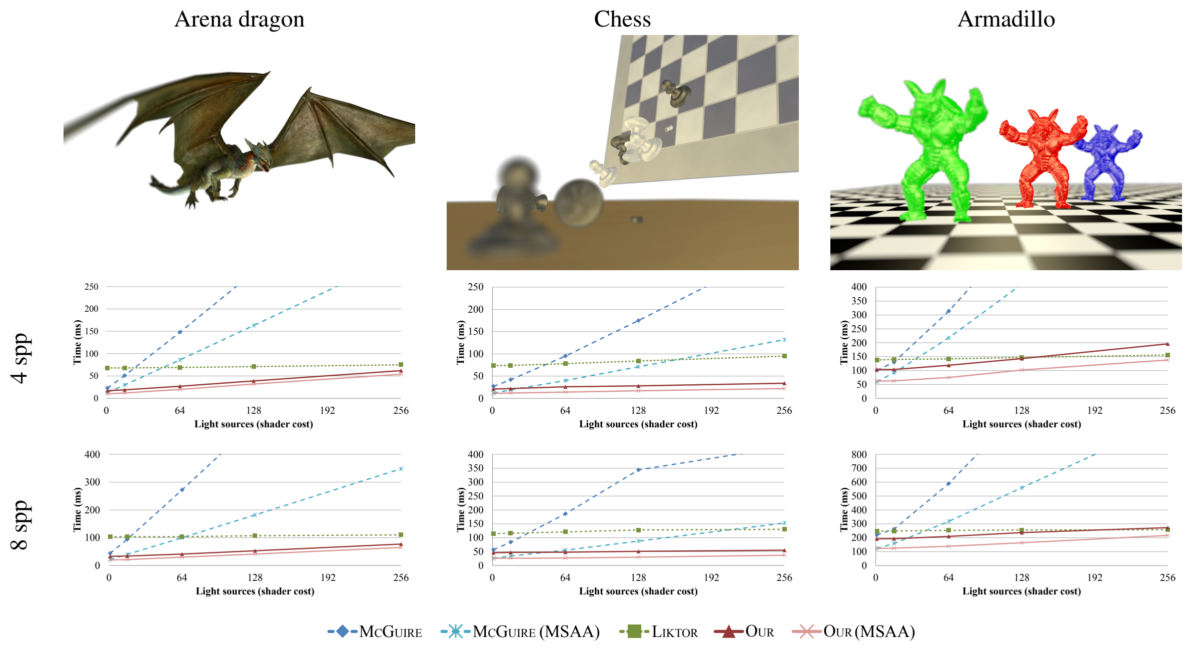

Figure 10: *Results for our test scenes rendered with a variety of algorithms. In the diagrams, we present render times in milliseconds (lower is better) for* 4 *and* 8 *samples per pixel (spp), respectively. The algorithms called* McGuire, Liktor*, and* Our *are directly comparable in terms of image quality. The algorithms with (MSAA) in their names rely on multisampling, and only do a single shader execution per pixel, with a single shading texture lookup. While this is beneficial for performance, we can only output a single depth value per pixel, and there is a risk of artifacts due to inaccurate depth test, as illustrated in Figure 9.*

offset the cache address based on the `primitiveID`. The cache size was set to 64k entries, and increasing this number did not result in significantly fewer shader executions. In our implementation, the shading rate of defocused triangles in Liktor and Dachsbacher's algorithm is also reduced using the AAS approach [VTS*12] and filtered with the generalized form of Loviscach's method [Lov05]. Our goal has been to put together an implementation that is as close as possible to Liktor and Dachsbacher's to make for a fair comparison.

To evaluate the performance, we measured the average rendering times for three different scenes with defocus blur. The *Arena dragon* scene contains one object with 74k triangles. The *Chess* scene is comprised of 12 objects with a total of 29k triangles. The *Armadillo* scene uses a high triangle count of 640k triangles. We can control shading complexity by computing shading using between one and 256 directional light sources. This allows us to study how the different algorithms scale with increasing shader cost. All images were rendered at 1920×1280 resolution. Throughout our experiments, the finest mipmap level resolution of the shading texture was set to 2048^2, which worked well for all of our test scenes.

All benchmarks were run on a PC with an Intel Core i7 965 CPU, 6 GB RAM and an NVIDIA 780 GTX GPU with 3 GB RAM (the benchmark application is completely GPU bound). Figure 10 shows the rendering time in milliseconds for the different scenes using $4\times$ and $8\times$ MSAA. We note that the algorithm by Liktor and Dachsbacher [LD12] is not affected as much by shader complexity, and at some point, their algorithm may be faster than ours. In the rightmost diagrams in Figure 10, Liktor and Dachsbacher's algorithm is faster at 256 light sources, and in the leftmost diagrams we estimate that it will be faster when using about 512 light sources. For reasonably complex shaders, our algorithm has performance characteristics that makes it very desirable for real-time rendering. The results are very encouraging since our algorithm outperforms the best competing algorithm by up to $3\times$ in some cases.

The rendered image quality of our algorithm closely matches that of the reference solution. In Figure 11, we compare the image quality of McGuire's algorithm using 128 samples per pixel against our algorithm with 8 and 128 samples per pixel.

4.1. Timings by Render Pass

We have timed the different passes in our algorithm. Not surprisingly, the stochastic rasterization pass (Pass 2 Rast.), which includes the lookups in the shading texture, makes

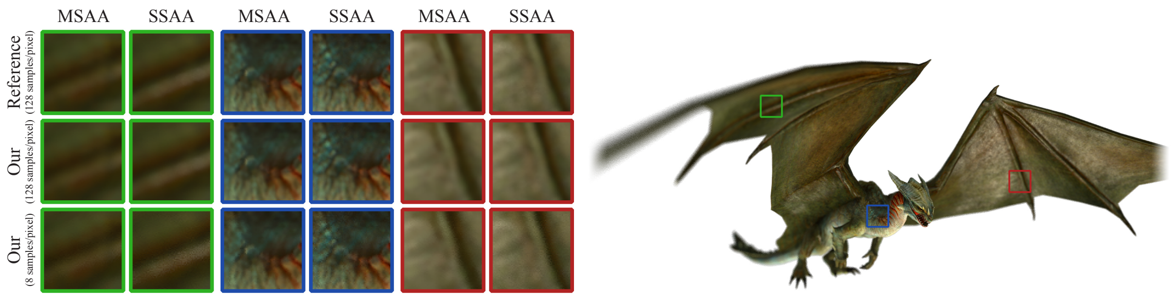

Figure 11: *Image quality comparison between our algorithm with 8 and 128 samples/pixel and McGuire's algorithm with 128 samples/pixel. We ran the algorithms both with MSAA and SSAA enabled. Notice that the highlight in the green cutouts is less prominent for MSAA compared to SSAA. The blue cutout shows that we are able to maintain high quality, even when the defocus effect is close to zero. In the red cutout, we note that some texture detail is lost for MSAA, but this occurs both for our algorithm and for the reference. Overall, we find that the major difference between the images stem from noise due to the lower number of samples drawn.*

up for the bulk of the time. Even though MSAA speeds up the stochastic rasterization pass (Pass 2 MSAA) considerably, it is still a major part of the total rendering time. Roughly $80-90\%$ of the time is spent on the inside tests within Pass 2, which means that the shading texture lookups are relatively inexpensive. When creating the shading texture, we note that the sub-pass performing conservative rasterization (Pass 1 Cons.) requires more time than the sub-pass doing normal rasterization (Pass 1 Shade) even though it writes to considerably fewer pixels. The reason for this is mainly that the geometry shader of the conservative rasterizer is quite expensive, and offsets whatever shading cost we gain from the many pixels failing the depth test. This is most apparent in the Armadillo scene as it has a very high triangle count, and we note that our algorithm would benefit greatly from faster algorithms for conservative rasterization [AMA05]. Finally, we note that 1–2 ms of the frame time is spent in miscellaneous setup tasks, such as animation, frame buffer clearing, and multi-sampling resolve, which are not related to our texture space shading algorithm. The table below shows a breakdown of rendering times (in milliseconds), of the Dragon and Armadillo scenes from Figure 10 with 16 light sources for shading.

[ms]	**Dragon**		**Armadillo**	
	4×	8×	4×	8×
Misc	2.3	2.3	0.8	0.8
Pass 1 Shade	2.1	2.1	3.8	3.8
Pass 1 Cons.	2.3	2.3	11.9	11.9
Pass 2 Rast.	13.8	29.3	87.5	178.8
Pass 2 (MSAA)	(7.1)	(16.6)	(46.1)	(108.3)
Total	20.5	36.0	104.0	195.3
Total (MSAA)	(13.6)	(21.0)	(62.1)	(124.8)

5. Conclusions and Future Work

Shading cost needs to be substantially reduced in order to make stochastic rasterization a viable rendering method. To this end, we have presented a novel technique for reusing shading for stochastic depth of field rasterization on current GPUs, and shown significant performance improvements of up to 3×. In contrast to deferred shading methods, we also support blending. There are several interesting aspects that we would like to research further in the future. Liktor and Dachsbacher's [LD12] algorithm shades very sparsely due to their approach doing deferred shading, which means that they get full benefit of occlusion and never have to shade any occluded samples. However, it has to explicitly sync threads, which proved expensive, and therefore the method performs better only when very expensive shaders are used. It would be useful to investigate how our algorithm would perform with occlusion queries and/or software occlusion culling on the CPU [Col11], which is often done in mature game engines.

To minimize state changes for very large scenes, we would like to develop a system for dynamically allocating properly-sized shading textures (or using a part of a shading texture) on a per-object basis. This would not require any artist interaction and at the same time it would reduce state changes significantly, which would also increase the performance of our algorithm. Finally, we would like to extend the frequency analysis to motion blur and to the combination of motion and defocus blur. Currently, our system does not compute optimized shading rates for motion blur—however we would like to point out that our system can already handle these effects too, even though we may over-shade.

Acknowledgements

We thank the reviewers for their insightful comments, and thank David Blythe, Marco Salvi, Charles Lingle, Tom Piazza, and the Advanced Rendering Technology team at Intel for their support. We would also like to thank Jacob Munkberg, Petrik Clarberg, Jim Nilsson and Björn Johnsson for all the fruitful discussions, and Gabor Liktor for answering our questions about his research. Tomas is a Royal

Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation.

References

[AMA05] Akenine-Möller T., Aila T.: Conservative and Tiled Rasterization Using a Modified Triangle Set-Up. *Journal of Graphics Tools, 10*, 3 (2005), 1–8. 5, 8

[AMHH08] Akenine-Möller T., Haines E., Hoffman N.: Real-time rendering. 417–419. 2

[AMMH07] Akenine-Möller T., Munkberg J., Hasselgren J.: Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware* (2007), pp. 7–16. 1, 6

[Arv86] Arvo J.: Backward Ray Tracing. SIGGRAPH '86 Course Notes - Developments in Ray Tracing, 1986. 2

[BFM10] Burns C. A., Fatahalian K., Mark W. R.: A Lazy Object-Space Shading Architecture With Decoupled Sampling. In *High Performance Graphics* (2010), pp. 19–28. 1, 2, 3

[BL03] Borshukov G., Lewis J. P.: Realistic Human Face Rendering for "The Matrix Reloaded". In *ACM SIGGRAPH Sketches & Applications* (2003), pp. 1–1. 2

[CCC87] Cook R. L., Carpenter L., Catmull E.: The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)* (1987), vol. 21, pp. 96–102. 1, 3

[Col11] Collin D.: Culling the Battle Field. Game Developer's Conference, 2011. 8

[CTM13] Clarberg P., Toth R., Munkberg J.: A Sort-based Deferred Shading Architecture for Decoupled Sampling. *ACM Transactions on Graphics, 32*, 4 (2013), 141:1–141:10. 1, 2, 3

[EWWL98] Ewins J., Waller M., White M., Lister P.: MIP-map Level Selection for Texture Mapping. *IEEE Transactions on Visualization and Computer Graphics, 4*, 4 (1998), 317–329. 4, 10

[GBAM11] Gribel C. J., Barringer R., Akenine-Möller T.: High-Quality Spatio-Temporal Rendering using Semi-Analytical Visibility. *ACM Transactions on Graphics, 30*, 4 (August 2011), 54:1–54:11. 2

[HAMO05] Hasselgren J., Akenine-Möller T., Ohlsson L.: Conservative Rasterization. *GPU Gems 2* (2005), 677–690. 5

[Hec86] Heckbert P. S.: Survey of Texture Mapping. *IEEE Computer Graphics & Applications, 6*, 11 (Nov. 1986), 56–67. 4, 9

[Hec90] Heckbert P. S.: Adaptive Radiosity Textures for Bidirectional Ray Tracing. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)* (1990), pp. 145–154. 2

[LD12] Liktor G., Dachsbacher C.: Decoupled Deferred Shading for Hardware Rasterization. In *Symposium on Interactive 3D Graphics and Games* (2012), pp. 143–150. 1, 2, 6, 7, 8

[LK11] Laine S., Karras T.: *Efficient Triangle Coverage Tests for Stochastic Rasterization*. NVIDIA Technical Report NVR-2011-003, Sept. 2011. 2, 6

[Lov05] Loviscach J.: Motion Blur for Textures by Means of Anisotropic Filtering. In *Eurographics Symposium on Rendering* (2005), pp. 7–14. 4, 5, 7, 10

[LTH*13] Luksch C., Tobler R. F., Habel R., Schwärzler M., Wimmer M.: Fast Light-Map Computation with Virtual Polygon Lights. In *Symposium on Interactive 3D Graphics and Games* (Mar. 2013), pp. 87–94. 2

[MAM11] Munkberg J., Akenine-Möller T.: Backface Culling for Motion Blur and Depth of Field. *Journal of Graphics, GPU, and Game Tools, 15*, 12 (2011), 123–139. 2, 5

[MCH*11] Munkberg J., Clarberg P., Hasselgren J., Toth R., Sugihara M., Akenine-Möller T.: Hierarchical Stochastic Motion Blur Rasterization. In *High Performance Graphics* (2011), pp. 107–118. 1, 6

[MESL10] McGuire M., Enderton E., Shirley P., Luebke D.: Hardware-Accelerated Stochastic Rasterization on Conventional GPU Architectures. In *High Performance Graphics* (2010), pp. 173–182. 2, 6

[NSL*07] Nehab D., Sander P. V., Lawrence J., Tatarchuk N., Isidoro J. R.: Accelerating Real-Time Shading with Reverse Reprojection Caching. In *Graphics Hardware* (2007), pp. 25–35. 2

[RKLC*11] Ragan-Kelley J., Lehtinen J., Chen J., Doggett M., Durand F.: Decoupled Sampling for Graphics Pipelines. *ACM Transactions on Graphics, 3*, 30 (May 2011), 17:1–17:17. 1, 2, 3

[SaLY*08] Sitthi-amorn P., Lawrence J., Yang L., Sander P. V., Nehab D., Xi J.: Automated Reprojection-based Pixel Shader Optimization. In *ACM Transactions on Graphics* (2008), vol. 27, pp. 127:1–127:11. 2

[VTS*12] Vaidyanathan K., Toth R., Salvi M., Boulos S., Lefohn A.: Adaptive Image Space Shading for Motion and Defocus Blur. In *High-Performance Graphics* (2012), pp. 13–21. 2, 3, 4, 7

Appendix A: Elliptical Texture Filter

The task at hand is to find the minimal radius of the elliptical footprint on the triangle. The projection of a pixel with circular footprint in the plane of the triangle is an ellipse with arbitrary orientation. We let a screen space coordinate be defined as (x,y) and texture space coordinates (s,t).

The elliptical footprint in texture space, centered around $(0,0)$, is:

$$E(s,t) = As^2 + Bst + Ct^2, \tag{4}$$

where (s,t) is inside the ellipse if $E(s,t) < F$, and from Heckbert [Hec86], we have the following:

$$\begin{aligned} A(x,y) &= (\partial t/\partial x)^2 + (\partial t/\partial y)^2, \\ B(x,y) &= -2\left(\frac{\partial s}{\partial x}\frac{\partial t}{\partial x} + \frac{\partial s}{\partial y}\frac{\partial t}{\partial y}\right), \\ C(x,y) &= (\partial s/\partial x)^2 + (\partial s/\partial y)^2, \\ F(x,y) &= \left(\frac{\partial s}{\partial x}\frac{\partial t}{\partial y} - \frac{\partial s}{\partial y}\frac{\partial t}{\partial x}\right)^2. \end{aligned} \tag{5}$$

Furthermore, if we introduce $r = \sqrt{(A-C)^2 + B^2}$, the minimum radius of the ellipse is given by:

$$R_{\text{minor}}(x,y) = \sqrt{2F/(A+C+r)}. \tag{6}$$

To determine the highest resolution mipmap level needed somewhere over the triangle, our task is to find the minimal value of R_{minor} over the surface of the triangle. More formally we search for:

$$\min_{(x,y)\in\text{Tri}}\left[R^2_{\text{minor}}(x,y)\right] = \min_{(x,y)\in\text{Tri}}\left[\frac{2F}{A+C+r}\right]. \quad (7)$$

The derivative of a perspective correctly interpolated attribute can be expressed as [EWWL98]:

$$\begin{aligned}\frac{\partial s}{\partial x}(x,y) &= \frac{c_3x+c_4}{Q^2},\\ \frac{\partial s}{\partial y}(x,y) &= \frac{c_5y+c_6}{Q^2},\\ &\dots\end{aligned} \quad (8)$$

where:

$$Q = c_0x+c_1y+c_2. \quad (9)$$

The expression we try to minimize becomes a high order irrational function. While it is possible to simplify this function somewhat, we have yet to find any elegant solution to finding the minimum, or proving that the minimum lies in any of the triangle's vertices.

Appendix B: Filter Derivation

Loviscach [Lov05] showed how to efficiently filter textures in time and space by extending elliptical weighted average (EWA) filtering to handle motion. We generalize this approach to handle Gaussian filter kernels in n dimensions. Like Loviscach, we make some local approximations about the various dimensions that we wish to integrate over. First, we assume that the texture coordinates are locally linear with regard to the augmented variables, which means that we locally ignore effects such as perspective distortion. EWA already uses this approximation in (x,y), using concentric ellipses in (s,t). Second, Loviscach also approximates (s,t) linearity in time, and we will do the same for all additional variables. Finally, we locally assume that all variables are independent of each other, again in line with Loviscach's work.

Sums of independent normal distributions are normal distributions. The Gaussian distribution in 2D is:

$$X = \begin{pmatrix}s_0\\t_0\end{pmatrix} + \alpha\begin{pmatrix}\partial_x s\\ \partial_x t\end{pmatrix} + \beta\begin{pmatrix}\partial_y s\\ \partial_y t\end{pmatrix}, \quad (10)$$

where α and β are normal distributions with zero mean. Without loss of generality, we assume α and β are $N(0,1)$ distributions, as the variance can be chosen arbitrarily by pre-scaling the partial derivatives. If we augment with a number of additional independent distributions, we get:

$$Y = \begin{pmatrix}s_0\\t_0\end{pmatrix} + \sum_i^n \gamma_i\begin{pmatrix}\partial_{X_i} s\\ \partial_{X_i} t\end{pmatrix}, \quad (11)$$

where $\gamma_1 = \alpha$, $\gamma_2 = \beta$, $X_1 = x$, $X_2 = y$, and all γ_i are again $N(0,1)$ distributions.

Next, we focus on the distribution around the point (s_0,t_0). Summing distributions is most easily accomplished using characteristic functions. The characteristic function of a sum of distributions is the product of the characteristic functions of the distributions. The characteristic function for the sum in Equation 11 is:

$$\begin{aligned}\Phi_Y(p,q) &= \mathbf{E}\left(e^{i\begin{pmatrix}p\\q\end{pmatrix}\cdot\left[\sum_i^n\gamma_i\begin{pmatrix}\partial_{X_i}s\\ \partial_{X_i}t\end{pmatrix}\right]}\right)\\ &= e^{-\frac{p^2}{2}\sum_i^n(\partial_{X_i}s)^2}\cdot e^{-pq\sum_i^n\partial_{X_i}s\partial_{X_i}t}\cdot e^{-\frac{q^2}{2}\sum_i^n(\partial_{X_i}t)^2}.\end{aligned} \quad (12)$$

The distribution described by Equation 11 can be expressed as a sum of two independent distributions:

$$Z = \begin{pmatrix}s_0\\t_0\end{pmatrix} + \zeta\begin{pmatrix}e\\f\end{pmatrix} + \eta\begin{pmatrix}g\\h\end{pmatrix}, \quad (13)$$

where ζ and η are $N(0,1)$ distributions. The characteristic function for Z is:

$$\begin{aligned}\Phi_Z(p,q) &= \mathbf{E}\left(e^{i\begin{pmatrix}p\\q\end{pmatrix}\cdot\left[\zeta\begin{pmatrix}e\\f\end{pmatrix}+\eta\begin{pmatrix}g\\h\end{pmatrix}\right]}\right)\\ &= e^{-\frac{p^2}{2}(e^2+g^2)}\cdot e^{-pq(ef+gh)}\cdot e^{-\frac{q^2}{2}(f^2+h^2)}.\end{aligned} \quad (14)$$

Comparing Equation 14 with Equation 12 reveals that these are the same distribution as long as the following equalities are true:

$$\begin{aligned}e^2+g^2 = A &:= \sum_i^n(\partial_{X_i}s)^2,\\ ef+gh = B &:= \sum_i^n\partial_{X_i}s\partial_{X_i}t,\\ f^2+h^2 = C &:= \sum_i^n(\partial_{X_i}t)^2.\end{aligned} \quad (15)$$

The system of equations (15) is underdetermined with three equations for four variables. Loviscach solved this equation in three dimensions by aligning one distribution to either the s- or t-axis, depending on which is more numerically robust [Lov05]. If $A > C$, the following expressions are used:

$$e = \sqrt{A}, \quad f = B/e, \quad g = 0, \quad h = \sqrt{C-f^2}, \quad (16)$$

and otherwise:

$$h = \sqrt{C}, \quad g = B/h, \quad f = 0, \quad e = \sqrt{A-g^2}. \quad (17)$$