Multisample anti-aliasing in deferred rendering

A. Fridvalszky and B. Tóth

Budapest University of Technology and Economics

Abstract

We propose a novel method for multisample anti-aliasing in deferred shading. Our technique successfully reduces memory and bandwidth usage. The new model uses per-pixel linked lists to store the samples. We also introduce algorithms to construct the new G-Buffer in the geometry pass and to calculate the shading in the lighting pass. The algorithms are designed to enable further optimizations, similar to variable rate shading. We also propose methods to satisfy constraints of memory usage and processing time. We integrated the new method into a Vulkan based renderer.

CCS Concepts

• ***Computing methodologies*** → ***Rasterization;*** *Antialiasing;*

1. Introduction

Deferred shading based rendering algorithms are popular in real-time three-dimensional applications, because they allow orders of magnitude more light sources than with classical forward shading algorithms. The disadvantage is that we cannot use the built-in multisample anti-aliasing algorithms of the GPU (MSAA). There are multiple solutions for this problem, but the increased memory and bandwidth consumption of the renderer is a common drawback. For this reason, it is typical to use post processing based anti-aliasing methods (e.g., FXAA). These techniques try to find and then blur edges on the picture, instead of sampling it with higher frequency. These methods are much faster than MSAA but they cannot always produce correct results, the picture may become blurry or fast camera movements may result in visible artifacts.

1.1. Deferred shading

Deferred shading [ST90] is a rendering technique that aims to increase the usable number of light sources in a scene or reduce the computational cost of lighting in case of complex geometry. The main idea is to divide the problem of rendering into two parts, the geometry pass and the lighting pass.

During the geometry pass the scene geometry is rasterized, but no shading is performed. Only the necessary attributes are collected (e.g., albedo, normals, depth) and stored in the so called G-Buffer. It is generally implemented as several frame sized textures, where every texel stores the corresponding pixel's data.

During the lighting pass light sources are processed. The classical approach rasterizes point light sources as spheres, where the radius corresponds to the effective range of the light. Tiled deferred shading [LHA*09] divides the camera space into smaller parts and generates a list of affecting light sources for each. The goal of both techniques is to reduce the number of unnecessary shading calculations and make the processing time of light sources independent of the scene geometry. During shading the G-Buffer is accessed and results are accumulated.

1.2. Multisample anti-aliasing

Aliasing (Figure 1) is a common problem during rendering. When we rasterize the scene geometry, the sampling rate is too low and aliasing occurs. The visible results are the jagged edges in the final image. One method to solve this is supersampling which renders the scene in a higher resolution than the target, then downsamples it. This solution targets the root of the problem, the low sampling frequency, but is very costly in real time environments.

Multisample anti-aliasing (MSAA [PMH02]) is a hardware accelerated optimization of supersampling. It aims to reduce the number of shading calculations by only doing supersampling where it is necessary. These parts are the edges of the rasterized triangles, where large differences can occur in the final color. With MSAA we still store multiple samples per pixel (just like with supersampling), but shading is only performed for some of them. Where the rasterizer detects that a triangle covers some sample in a pixel, it invokes the fragment shader only once, but the result will be written to each covered sample. After that, samples are averaged to compute the final color of the pixel.

1.3. Deferred shading with MSAA

When multisampling is applied to a deferred renderer we can no longer use the basic hardware accelerated process. The whole G-

DOI: 10.2312/egs.20201008

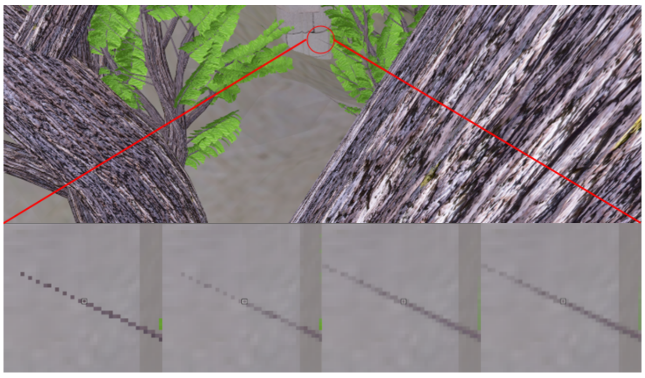

Figure 1: *Results of anti-aliasing. From left to right: NO AA, FXAA, 8× MSAA, Proposed Algorithm.*

Buffer must be created with higher sampling frequency (using MSAA). The information about pixel coverage must be stored in the G-Buffer. Increased size can already become a problem for mobile devices, but the memory bandwidth consumption makes it very taxing on desktop GPUs as well. A further problem is that during the lighting pass we do not want to calculate shading for duplicated sample data. A complex logic to select the unique samples for shading is unpractical for the massively parallel nature of the GPU, but if we settle with less accurate selections (e.g., simple, and supersampled pixel), then it will result in many unnecessary calculations.

In this work we target the previously mentioned problems with the combination of deferred shading and MSAA. We introduce a new method to mitigate these and implement it in a physically based renderer with Vulkan and C++.

2. Related work

To apply anti-aliasing Reshetov [Res09] proposed a post-processing based approach, which worked by searching various patterns in the final image and blending the colors in the neighborhood. It can be used efficiently in a deferred renderer. Lottes et. al [Lot09] and Jimenez et. al [JESG12] proposed similar techniques which applied various enhancements to the original approach. Chajdas et. al [CML11] used single-pixel shading with subpixel visibility to create anti-aliased images. Another branch of the anti-aliasing tehcniques uses previous frames to solve the problem. A recent variant, proposed by Marrs et. al [MSG*18] combines it with supersampling and raytracing. Liktor et. al [LD12] proposed an alternative structure for the G-Buffer which allows efficient storage of sample attributes. They used this structure both for stochastic rendering and for anti-aliasing. Crassin et. al [CMFL15] proposed a method for deferred renderers that reduces the stored and shaded sample count by merging samples that belong to the same surface. Schied et. al [SD15] replaced per-pixel samples with a triangle based representation to reduce the memory requirements of the G-Buffer. Our implementation of the G-Buffer uses a similar structure to the A-buffer, proposed by Carpenter et. al [Car84].

3. The proposed algorithm

The main problem of multisampling in case of deferred shading is the redundant storage of samples. The standard G-Buffers use textures to store per-pixel data. In case of multisampling we need larger textures to make space for more samples. By using 8× multisampling we effectively need eight times more memory. Most of this storage is unnecessary, because large part of the screen requires only 1-2 samples. This redundancy also causes further problems. MSAA is better than supersampling because it computes the shading on multiple samples, only where edges are found. This works for forward renderers, but during the lighting pass of a deferred renderer, this information is not available. It means that we must recover it manually or use supersampling, effectively losing all the benefits of MSAA.

Overview of the proposed technique is presented on Figure 2. The G-Buffer is divided into two parts. The first one contains one block of data for each pixel. It represents the standard G-Buffer and also contains the heads of per-pixel linked lists which store data for the rest of the samples, originating from the same pixel. These linked lists are in the second part of the G-Buffer.

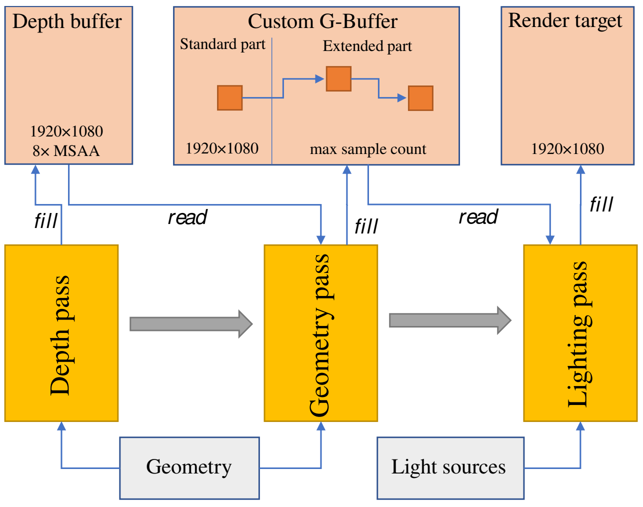

Figure 2: *Summary of the proposed algorithm.*

The idea is to construct the G-Buffer to prevent redundancy. Then during the lighting pass we know for certain that every block of data must be shaded and no unnecessary calculations will be done. The required size for the G-Buffer is reduced too.

To construct the G-Buffer, scene geometry is rasterized normally using the maximum desired multisampling frequency. The target framebuffer contains only a depth buffer with appropriate sampling rate. According to the behavior of standard multisampling, the fragment shader is invoked for every triangle-pixel intersection and each invocation represents one or more samples. The future contents of the G-Buffer are collected and calculated normally. The covered samples are checked if they contain a previously specified index. If that is the case, then the data is written into the first part of the G-Buffer. Otherwise a new block is allocated from the second part by using an atomic counter. The block is connected to the pixel's linked list with an atomic operation and the data is written into it. This way the G-Buffer becomes free of redundancy except for one case. That is when hidden objects are rasterized before the visible ones. We solve this by running a depth-only Z-prepass before the geometry pass.

During the lighting-pass the shading can be done by traversing the linked lists or the G-Buffer itself in an unordered manner, according to the implementation of the light sources. Our implementation followed the first approach. The light sources are stored in a buffer and for every sample we accumulate the shading for every light source. Then the results are averaged. This implementation of light sources is hardly optimal because every light source influences every part of the screen, even where its effect is unnoticeable. We chose this method because it is straightforward to implement. It is also easy to extend to tiled deferred shading, a popular variant of standard deferred shading.

4. Implementation

In the following sections we highlight the important details of our implementations.

4.1. The G-Buffer

Our implementation used the physically based Cook-Torrance shading model. According to this we need the following attributes: albedo, normal, roughness, metallic, ambient occlusion factor (ao). We also need a pointer to construct the linked list.

1. bit 8.	16.	24.	32. bit
albedo1 (RGB)			metallic1
albedo2 (RGB)			metallic2
normal1			
normal2			
roughness1	roughness2	ao1	ao2
pointer1			
pointer2			

Figure 3: *Structure of one block in the G-Buffer, where every sample uses 112 bits.*

We interleave every two blocks of data to prevent any unnecessary padding (Figure 3).

1. bit 26.	29.	32. bit
index	sample count	sample index

1. bit 32.	…	
next pointer	pre-allocated blocks	dynamic blocks

Figure 4: *Structure of the pointer and the G-Buffer.*

As we can see in Figure 4 (top), the pointer consists of three parts, where 3 bits are needed to store the number of covered samples and another 3 bits to store the index of one of these samples. The latter is needed to read the correct depth from the multisampled depth buffer. These refer to the pointed block. Remaining bits are used for the index of the next block.

The final structure of the G-Buffer is in Figure 4 (bottom). The number of pre-allocated blocks must match the number of pixels. The number of dynamic blocks depends on the available memory, performance constraints and required quality.

4.2. Light sources

We only used point light sources in our implementation. We also implemented it as a uniform buffer and every fragment shader invocation iterated over the entire list. It is inefficient, but allows better insight into the performance characteristics of our algorithm.

The performance directly depends on the number of shading calculations. The number of light sources (and their implementation) affects it, but only because it increases the shading cost. So by applying smarter light source implementation, like tiled deferred shading, we would not change the direct performance hit of our algorithm.

It is also easy to incorporate these optimizations within our method. For example, construction of the culled light source buffers can be directly added to our solution. The global light source list must be exchanged with the dynamically constructed buffers in the shader, but no other actions are required.

4.3. Shadows and transparent materials

In our implementation we chose to not implement shadows or transparency. Transparency is a common problem for deferred renderers and they are generally handled separately. Shadow calculation works well with deferred renderers, the most popular solutions are shadow mapping and its variations. Our approach does not interfere with these methods.

5. Evaluation

We benchmarked the performance characteristics of our implementation on multiple GPUs (Nvidia GTX970 and GTX1050). We compared the processing time and memory usage to an implementation without anti-aliasing, to a version of FXAA and to the traditional implementation of MSAA.

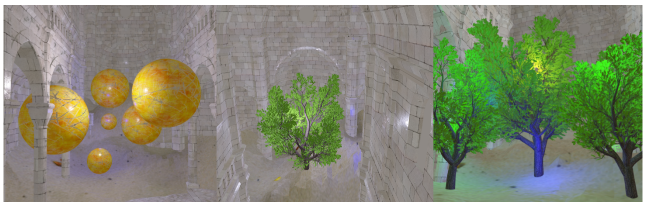

Figure 5: *Test scenes used during evaluation.*

We used three test scenes. These scenes contain 1.4, 8.7, and 23.8 million vertices, respectively (Figure 5).

First we measured the required memory for 1920×1080 resolution (Table 1). Our proposed method has flexible memory requirements so only minimum and maximum values are given. In case of minimum values no anti-aliasing will be performed. The actual memory requirements for full anti-aliasing depend on the scene geometry (Figure 6).

	No AA	MSAA		Proposed algorithm			
		4x	8x	4x		8x	
				Min	Max	Min	Max
G-Buffer	23.73	94.92	189.84	27.69	110.74	27.69	221.48
Z-Buffer	7.91	31.64	63.28	31.64	31.64	63.28	63.28
Total	31.64	126.56	253.12	59.33	142.38	90.97	284.76

Table 1: *Memory consumption of the anti-aliasing methods in Mbytes.*

	MSAA	Proposed algorithm	
	4x	4x	8x
1. scene – 5 light sources	5.2648	3.8092	4.3629
1. scene – 50 light sources	21.987	12.5951	14.0772
2. scene – 5 light sources	7.9021	8.1721	8.97702
3. scene – 5 light sources	14.069	19.1354	21.3446
3. scene – 50 light sources	33.293	31.7038	38.6841

Table 2: *Computation times of the anti-aliasing methods (ms).*

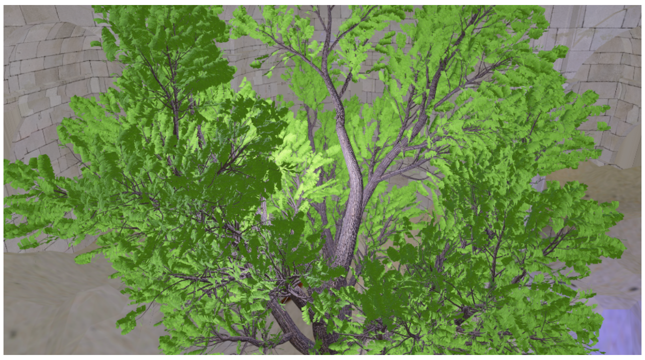

Figure 6: *Complex scene for memory requirements. It needs 115.31 MB memory for 8× and 75.01 MB for 4× anti-aliasing. In the 8× case it is even smaller than the memory requirements of the traditional 4× MSAA implementation.*

We also measured the processing times of the anti-aliasing algorithms. As we can see in Table 2, our algorithm performs well in environments where large number of shading calculations must be performed. On small scenes with many light sources it can even apply better anti-aliasing while remaining faster than the previous solution. It reacts well to larger anti-aliasing settings too (left of Figure 7), because it never does any unnecessary calculations. Large scene geometry can present a problem, because of the Z-prepass, but only in extreme cases and it still outperforms the traditional method (right of Figure 7).

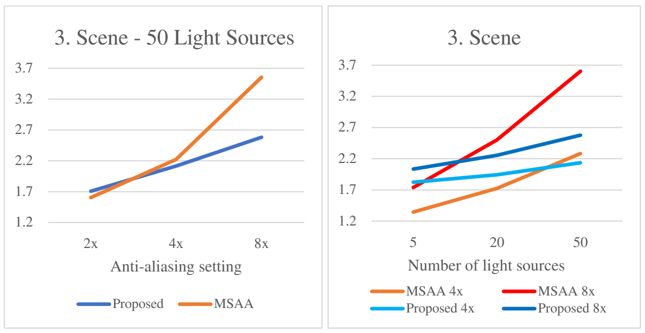

Figure 7: *Relative processing times of the algorithms with different anti-aliasing settings (left) and number of light sources (right). The measured values represent the relative performance of the techniques, compared to the implemantation without anti-aliasing.*

6. Conclusion

The proposed algorithm can apply multisample anti-aliasing in a deferred renderer without unnecessary memory allocations and complex shading logic of traditional methods. The flexible data structure of the G-Buffer prevents any redundancy and makes possible to specify strict requirements about performance and memory consumption.

7. Acknowledgements

This work has been supported by OTKA K-124124, and by the EFOP-3.6.2-16-2017-00013.

References

[Car84] CARPENTER L.: The a-buffer, an antialiased hidden surface method. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (1984), pp. 103–108. 2

[CMFL15] CRASSIN C., MCGUIRE M., FATAHALIAN K., LEFOHN A.: Aggregate g-buffer anti-aliasing. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games* (2015), pp. 109–119. 2

[CML11] CHAJDAS M. G., MCGUIRE M., LUEBKE D. P.: Subpixel reconstruction antialiasing for deferred shading. In *SI3D* (2011), Citeseer, pp. 15–22. 2

[JESG12] JIMENEZ J., ECHEVARRIA J. I., SOUSA T., GUTIERREZ D.: Smaa: enhanced subpixel morphological antialiasing. In *Computer Graphics Forum* (2012), vol. 31, Wiley Online Library, pp. 355–364. 2

[LD12] LIKTOR G., DACHSBACHER C.: Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games* (2012), ACM, pp. 143–150. 2

[LHA*09] LEFOHN A., HOUSTON M., ANDERSSON J., ASSARSSON U., EVERITT C., FATAHALIAN K., FOLEY T., HENSLEY J., LALONDE P., LUEBKE D.: Beyond programmable shading (parts i and ii). In *ACM SIGGRAPH 2009 Courses* (2009), ACM, p. 7. 1

[Lot09] LOTTES T.: Fxaa. *White paper, Nvidia, Febuary* (2009). 2

[MSG*18] MARRS A., SPJUT J., GRUEN H., SATHE R., MCGUIRE M.: Adaptive temporal antialiasing. In *Proceedings of the Conference on High-Performance Graphics* (2018), ACM, p. 1. 2

[PMH02] PETERSON J., MULLIS R., HUNTER G.: Multi-sample method and system for rendering antialiased images, Oct. 3 2002. US Patent App. 09/823,935. 1

[Res09] RESHETOV A.: Morphological antialiasing. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), ACM, pp. 109–116. 2

[SD15] SCHIED C., DACHSBACHER C.: Deferred attribute interpolation for memory-efficient deferred shading. In *Proceedings of the 7th Conference on High-Performance Graphics* (2015), pp. 43–49. 2

[ST90] SAITO T., TAKAHASHI T.: Comprehensible rendering of 3-d shapes. In *ACM SIGGRAPH Computer Graphics* (1990), vol. 24, ACM, pp. 197–206. 1