Optimizing multisample anti-aliasing for deferred renderers

András Fridvalszky*
Supervised by: Balázs Tóth†

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest / Hungary

Abstract

Deferred renderers are popular in computer graphics because they allow using a larger number of light sources, but they have some drawbacks too. One of these is the inability to work together with traditional hardware-based multisample anti-aliasing. Multiple solutions exist to this problem, but their common drawback is the increased memory and bandwidth requirements. We propose a novel approach that eliminates unnecessary memory usage and improves performance while maintaining image quality. Our method is based on a new G-Buffer structure that uses per-pixel linked lists to store the samples. By limiting the number of pre-allocated blocks in the G-Buffer we can also satisfy strict requirements about memory usage and processing time. Similarly to variable rate shading, our method enables to selectively apply anti-aliasing either on preferred parts of the screen or on a per-object basis. We measured the new method using a Vulkan based renderer on scenes with different geometry complexity and characteristics while comparing performance and memory usage to the traditional techniques.

Keywords: Antialiasing, Deferred rendering, Multisampling

1 Introduction

Deferred shading based rendering algorithms are popular with real time three dimensional applications, because they make it possible to use 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, nowadays 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. The inherent consequences of these methods are that they are much faster than MSAA, but they cannot always produce correct results. The picture could become blurry or fast camera movement could result in visible artifacts.

1.1 Deferred shading

Deferred shading [11] 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 idea is to divide the rendering process into two parts, the geometry pass and the lighting pass (Figure 1).

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 usually implemented as several frame sized textures, where every texel stores the corresponding pixel's attributes for lighting.

During the lighting pass light sources are processed. There are multiple techniques to do this. The original one rasterizes point light sources as spheres, where the radius corresponds to the effective range of the light. Another technique, called tile-based deferred shading [5] divides the camera space into smaller parts and generates a list of affecting light sources for each. The goal is to reduce the number of unnecessary shading calculations and make the processing time of light sources independent from the scene geometry. During shading the G-Buffer is accessed and results are accumulated.

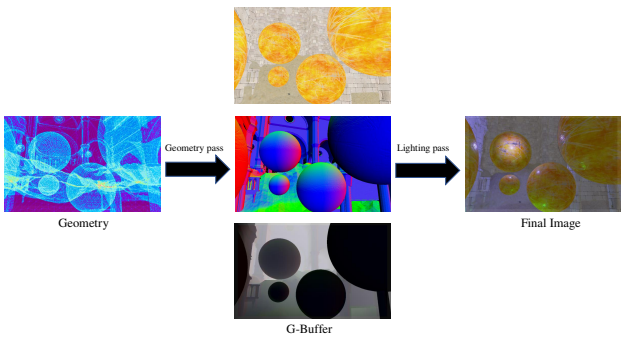

Figure 1: Visual representation of deferred shading.

*fandrasm@gmail.com
†tbalazs@iit.bme.hu

1.2 Multisample anti-aliasing

Aliasing is a common problem during rendering. For example when we rasterize the scene geometry, the sampling rate is too low and geometry aliasing occurs. The visible results are the jagged edges in the final picture. To solve this, we apply anti-aliasing methods in our rendering pipelines (Figure 2). One method is supersampling which renders the scene in a higher resolution than the target's, then downsamples it. This solution targets the root cause, the undersampling of frequencies, but it comes with high performance costs.

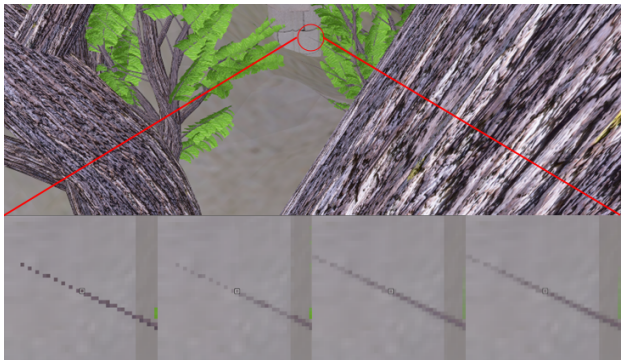

Figure 2: Results of anti-aliasing. From left to right: NO AA, FXAA, 8x MSAA, the Proposed Algorithm.

Multisample anti-aliasing (MSAA [9]) 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 colour. 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 colour 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-Buffer must be created with higher sampling frequency (using MSAA). The information about pixel coverage must be stored in the G-Buffer too. Increased size can already become a problem for mobile devices, but the memory bandwidth consumption makes it very taxing on desktop GPUs too. 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 analyse 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 modern renderer with Vulkan and C++.

2 Related work

To apply anti-aliasing Reshetov [10] proposed a post-processing based approach, which worked by searching various patterns in the final image and blending the colours in the neighbourhood. It can be used efficiently in a deferred renderer. Lottes et. al [7] proposed a different implementation based on the same idea with alternative edge detection mechanism. Jimenez et. al [4] adapted the original technique to work together with traditional MSAA and temporal supersampling to recover subpixel features.

Chajdas et. al [2] proposed a method that used single-pixel shading with sub-pixel visibility to create anti-aliased images.

Another branch of the anti-aliasing techniques uses previous frames to solve the problem. A recent variant, proposed by Marrs et. al [8] combines it with supersampling and raytracing.

Liktor et. al [6] proposed an alternative structure for the G-Buffer which allows efficient storage of sample attributes. They used this structure for stochastic rendering and also for anti-aliasing.

Salvi et. al [12] proposed a method for deferred renderers that reduces the stored and shaded sample count by merging samples that belong to the same surface. Crassin et. al [3] proposed a similar method that uses more complex criteria to group samples together and calculate aggregate values. Both methods are using a pre-pass before filling the G-Buffer to generate supporting information for subsequent passes.

Our implementation of the G-Buffer uses a structure similar to the A-buffer, proposed by Carpenter et. al [1]. The A-Buffer can be used for order independent transparency by collecting and sorting every rasterized fragment for each pixel in linked lists. Instead, we collect only the visible samples for every pixel and skip the sorting step.

3 The proposed algorithm

The main problem of multisampling in case of deferred shading is the redundant storage of samples. The standard G-Buffers are using textures to store per-pixel data. In case of multisampling we need to use larger textures to store more samples. By using 8x multisampling we effectively use eight times more memory. Most of it is unnecessary because a large part of the screen requires only 1-2 samples. This redundancy also causes further problems. MSAA is faster than supersampling, because only one sample is shaded for each fragment that is covered by

just one triangle. It is a great optimization for forward renderers, but during the lighting pass of a deferred renderer this information is not available. It means we must recover it manually or use supersampling, effectively losing all the benefits of MSAA.

The proposed technique consists of an alternative data structure for the G-Buffer and algorithms to build and use it. The G-Buffer is divided into two parts. The first one contains one block of data for each pixel. It represents the basic G-Buffer used in standard deferred shading. It also contains the heads of per-pixel linked lists that store data for the rest of the samples, originating from the same pixel. These linked lists are stored in the second part of the G-Buffer. The whole structure can be represented on the GPU as a Shader storage buffer object (SSBO).

The idea is that we construct the G-Buffer in a way 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 the appropriate sampling rate. According to the behaviour of standard multisampling the fragment shader is invoked for every triangle-pixel intersection and each invocation represents one or more samples. The attributes for lighting calculation are collected. The covered samples are checked if they contain a previously specified index. If that is the case then the collected 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. In our implementation we did the former. The light sources were stored in a buffer and for every sample we accumulated the shading for every light source. Then the results were averaged, weighted by the number of covered samples. 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 and we can reason better about the performance characteristics of different number of light sources. It is also easy to extend to tile-based deferred shading, a popular variant of standard deferred shading.

4 Implementation

After an overview of the proposed technique, we highlight the important details of our implementations.

4.1 The G-Buffer

In our shading model we needed the following attributes: albedo, normal, roughness, metallic, ambient occlusion factor (ao). We also needed a pointer to construct the linked list.

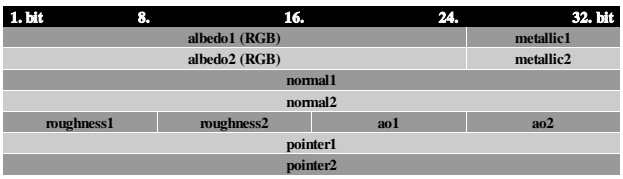

Figure 3: Structure of two interleaved block in the G-Buffer referring to two samples. Every sample uses 112 bits.

We interleaved every two block of data to prevent any unnecessary padding (Figure 3). It is another added flexibility of our data structure. The traditional texture based G-Buffer does not allow this. (It would require 16 bits padding for every sample.)

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

As depicted in the top of Figure 4, the pointer consists of three parts. The first 26 bit stores the index of the next block in the linked list. Then 3 bits are needed to store the number of samples covered by the next block, and another 3 bits to store the index of one of these samples. The latter is needed to read the correct value from the multisampled depth buffer.

The final structure of the G-Buffer is shown at the bottom of Figure 4. The number of pre-allocated blocks must match the number of pixels, because even without anti-aliasing, one sample is needed. (This is the previously mentioned first part of the G-Buffer.) The number of dynamic blocks depends on the available memory, performance constraints and required quality. Additional samples after the first one are stored here in linked lists.

4.2 Light sources

We only used point light sources in our implementation. As a simplification we also implemented it as a uniform buffer and every fragment shader invocation iterated over

the entire list. This does not impact the performance comparisons, because a smarter light source implementation would affect each measured algorithm in the same way.

4.3 Z-prepass

A simple Z-prepass will not solve the problem of multiple fragment shader invocations for the same sample in the geometry pass, which can happen when triangles are intersecting with each other. At the intersection, fragments from different triangles can have the same depth value. This means that both fragments will allocate memory for the same sample location and it is possible that the resulting linked list will contain more fragments then the number of samples. This conflicts with the average calculation and causes visual errors. A solution for this is to use one bit of the stencil buffer to flag the sample after one invocation and discard subsequent ones.

4.4 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 is working well with deferred renderers. The most popular solutions are shadow mapping and its variations. Our approach does not interfere with these methods and they can be applied to the resolved image without changes.

5 G-Buffer Optimizations

Our method stores samples in the G-Buffer precisely, when we decide to shade them. This characteristic allows us to further reduce memory usage by storing fewer samples for certain pixels in the geometry pass. To apply arbitrary maximum sample count for a pixel, we would require another 32 bit atomic counter for every one of them. It is also impossible to ensure deterministic behaviour, so small flickering can ruin the results. Instead, we opted to use a maximum of 1 sample. This needs no further memory storage or complexity because we just need to discard every sample except the one, which goes to the first part of the G-Buffer. The remaining question is how to decide if a pixel only needs one sample. We provide four different methods, which can be used either exclusively or simultaneously.

5.1 Filtering based on location

We can select certain areas on the screen in advance to disable anti-aliasing by limiting the maximum number of samples to one. It goes well with certain kind of programs, where a large part of the screen would get blurred (e.g., car racing) or with VR where edges of the screen need less detail. The only limitation is the allowed complexity for selecting parts of the screen.

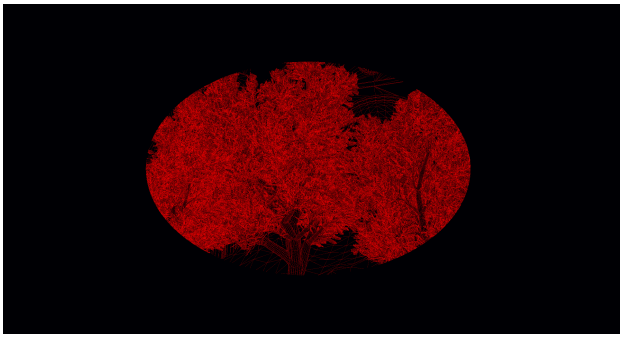

Figure 5: Visualization of sample usage after filtering by an ellipse (black - 1 sample, red - 8 samples).

We implemented two methods based on an ellipse (Figure 5) and a rectangle. The latter is given with top, bottom, left and right values while the former with its centre, width and height. Outside of the selected region, only one sample is stored and no anti-aliasing is performed, while in the inside everything remains the same as before (dynamic sample count).

The advantage is that we know beforehand how many pixels are going to be filtered. We can reason about the memory requirements better and we can reduce it preemptively. The disadvantage is that no dynamic adjustments are done and edges of the selection are clearly visible if a complex, unblurred object intersects it.

5.2 Filtering based on depth

We can also use the depth buffer to specify a threshold for anti-aliasing (Figure 6). It is great for certain scenes, where background objects are always getting blurred or barely visible (e.g., fog). The problem is, that we must also select these objects precisely and reliably based only on depth. It helps that we can apply per-frame thresholds based on previous frames or by using some other heuristics.

5.3 Filtering based on objects

Sometimes there are objects with very simple geometry (or some other special properties) that need no anti-aliasing (e.g., spheres).

These can be flagged beforehand and excluded from using multiple samples (Figure 6). It is a very specific solution which must be used with care but can boost performance immensely in certain environments while maintaining image quality.

5.4 Filtering based on edges

A common problem with MSAA is that it applies to every triangle edge, even when they come from the same object

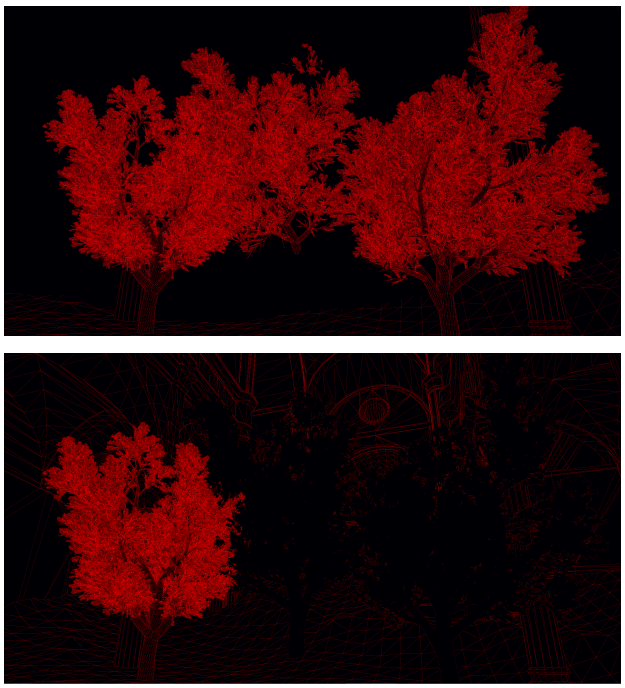

Figure 6: Top: Visualization of sample usage after filtering by a depth threshold. The cathedral and the rearmost tree is partially excluded from anti-aliasing. Bottom: Visualization of sample usage after filtering out a pair of trees.

and no anti-aliasing would be needed. To reduce these false-positive cases we can use edge detection on the depth buffer after the Z-prepass to flag true edges on the screen (Figure 7). These can be used to further constrain the anti-aliasing.

The problem with this approach is that by only using subpasses we can access the depth only for the currently processed pixel. This makes it difficult to use robust edge detection algorithms that would make use of the neighbouring pixels and directional information. We decided to calculate the minimum and maximum depth values for each pixel (from the samples) and threshold the difference. Results were not satisfactory, because many true edges were excluded.

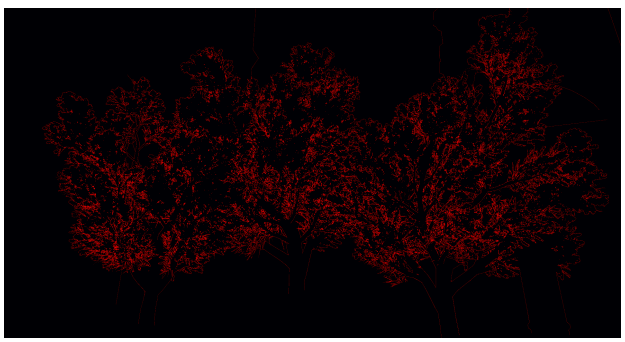

Figure 7: Visualization of sample usage after edge detection on the depth buffer.

With multiple renderpasses and with a more robust edge detection algorithm, better results could be achieved, but it could also hinder the performance on mobile devices.

6 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.

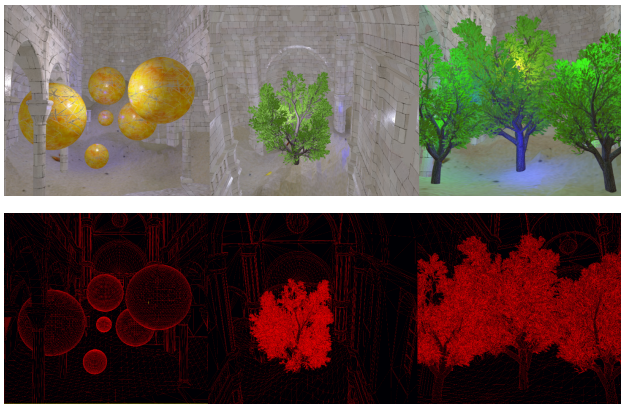

Figure 8: Test scenes used during evaluation, and visualized sample usage.

We used three test scenes with various geometry complexity for our measurements. These scenes contain 1.424.145, 8.699.022 and 23.820.168 vertices respectively (top row of Figure 8). The number of samples used for each scene is also shown on the bottom row of Figure 8.

First we measured the required memory for 1920x1080 resolution (Table 1). Our proposed method has flexible memory requirements so only minimum and maximum values are given. In the case of minimum values no anti-aliasing will be performed. The actual memory requirements for full anti-aliasing depend on the scene geometry (Figure 9).

	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.

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 a large number of shading calculations must be performed.

Figure 9: Complex scene for memory requirements. It needs 115.31 MB memory for 8x and 75.01 MB for 4x anti-aliasing. In the 8x case it is even smaller than the memory requirements of the traditional 4x MSAA implementation.

	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
2. scene – 50 light sources	21.8875	16.5104	18.9172
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 without any G-Buffer optimization (ms).

On small scenes with many light sources our technique is able to deliver better anti-aliasing results and even better performance than previous solutions. It reacts well to larger anti-aliasing settings too (left of Figure 10), because it effectively reduces unnecessary shading operations. Complex geometry can present a problem, because of the Z-prepass, but only in extreme cases and even then, with equally large number of light sources it still outperforms the traditional method (right of Figure 10).

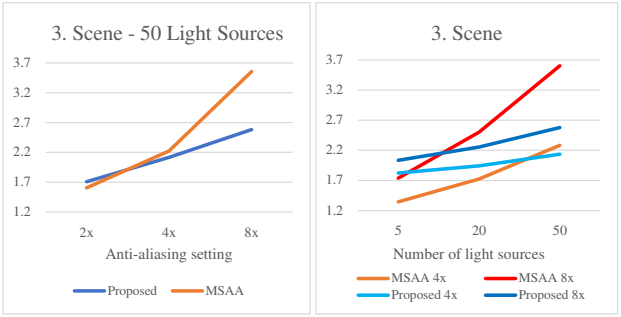

Figure 10: Relative processing times of the algorithms with different anti-aliasing settings (left) and number of light sources (right). The measured values represent the relative performance of the techniques, compared to the implemantation without anti-aliasing.

7 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 performance is better for small scenes while it also reacts well to more light sources and higher sample count than the traditional method. The flexible data structure of the G-Buffer prevents any redundancy and enables to further reduce the allocated storage by selectively applying anti-aliasing. These characteristics permit us to aggressively limit the G-Buffers size and consequently satisfy strict requirements about performance and memory consumption.

8 Acknowledgements

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

References

[1] Loren Carpenter. The a-buffer, an antialiased hidden surface method. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 103–108, 1984.

[2] Matthäus G Chajdas, Morgan McGuire, and David P Luebke. Subpixel reconstruction antialiasing for deferred shading. In *SI3D*, pages 15–22. Citeseer, 2011.

[3] Cyril Crassin, Morgan McGuire, Kayvon Fatahalian, and Aaron Lefohn. Aggregate g-buffer anti-aliasing. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, pages 109–119, 2015.

[4] Jorge Jimenez, Jose I Echevarria, Tiago Sousa, and Diego Gutierrez. Smaa: enhanced subpixel morphological antialiasing. In *Computer Graphics Forum*, volume 31, pages 355–364. Wiley Online Library, 2012.

[5] Aaron Lefohn, Mike Houston, Johan Andersson, Ulf Assarsson, Cass Everitt, Kayvon Fatahalian, Tim Foley, Justin Hensley, Paul Lalonde, and David Luebke. Beyond programmable shading (parts i and ii). In *ACM SIGGRAPH 2009 Courses*, page 7. ACM, 2009.

[6] Gábor Liktor and Carsten Dachsbacher. Decoupled deferred shading for hardware rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 143–150. ACM, 2012.

[7] Timothy Lottes. Fxaa. *White paper, Nvidia, Febuary*, 2009.

[8] Adam Marrs, Josef Spjut, Holger Gruen, Rahul Sathe, and Morgan McGuire. Adaptive temporal antialiasing. In *Proceedings of the Conference on High-Performance Graphics*, page 1. ACM, 2018.

[9] James Peterson, Robert Mullis, and Gregory Hunter. Multi-sample method and system for rendering antialiased images, October 3 2002. US Patent App. 09/823,935.

[10] Alexander Reshetov. Morphological antialiasing. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 109–116. ACM, 2009.

[11] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 197–206. ACM, 1990.

[12] Marco Salvi and Kiril Vidimče. Surface based antialiasing. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 159–164, 2012.