Computers & Graphics 37 (2013) 955–962

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Technical Section

Solving aliasing from shading with selective shader supersampling ☆

Christian Siegl [a,*], Quirin Meyer [a], Gerd Sußner [b], Marc Stamminger [a]

[a] Computer Graphics Group, University Erlangen-Nuremberg, Germany
[b] RTT AG, Munich, Germany

ARTICLE INFO

Article history:
Received 31 March 2013
Received in revised form
8 August 2013
Accepted 10 August 2013
Available online 4 September 2013

Keywords:
Antialiasing
Per-pixel linked lists
Sampling
GPU

ABSTRACT

Existing GPU antialiasing techniques, such as MSAA or MLAA, focus on reducing aliasing artifacts along silhouette boundaries or edges in image space. However, they neglect aliasing from shading in case of high-frequency geometric detail. This may lead to a shading aliasing artifact that resembles Bailey's Bead Phenomenon—the degradation of continuous specular highlights to a *string of pearls*. These types of artifacts are particularly striking for high-quality surfaces. So far, the only way of removing aliasing from shading is by globally supersampling the entire image with a large number of samples. However, globally supersampling the image is slow and significantly increases bandwidth consumption. We propose three adaptive approaches that locally supersample triangles only where necessary on the GPU. Thereby, we efficiently remove artifacts from shading while aliasing along silhouettes is reduced by efficient hardware MSAA.

© 2013 Elsevier Ltd. All rights reserved.

1. Introduction

Nobody will ever solve the antialiasing problem. – Jim Blinn [1]

Aliasing artifacts appear as jagged lines in still images and as flickering in animations. Given even the latest advances in computer graphics, there is still no comprehensive solution to the problem without significantly impacting performance, as observed by [2,3,1]. While it is perfectly understood that these artifacts occur due to violation of the Nyquist–Shannon sampling theorem [4], there still is no straightforward solution without disregarding performance.

The type of aliasing artifact we discuss in this work is illustrated in Fig. 1: The rendering shows a sideways view on the body of a BMW 3 series with specular highlights along sharp features. While this specular highlight should appear as one long continuous bright area, it degrades to a series of bright spots. This artifact resembles Bailey's Bead Phenomenon, which is observable during solar-eclipses: sun-rays that graze the moon occur as a bead chain. The white pixels in Fig. 1 can be considered the pearls, and the grey pixels the string of the bead.[1] Since we were not able to find an appropriate term for this mainly ignored kind of aliasing, we will refer to it as "bead chains" in this paper. When animating such a scene, the artifacts are amplified by an apparent flickering, as the bright spots begin to move and jump. What makes things even worse is that such fine highlights from folds are usually an important design element, and their proper display is particularly important.

Current antialiasing techniques are geared toward reducing aliasing along silhouette boundaries and visible edges, but they fail to antialias bead chains. The fundamental problem is that the highlight is both very bright and very thin, and any anti-aliasing method based on sampling *in image space* is likely to miss this feature, even with high supersampling. However, the appearance of the effect can be predicted well *in object space*, for instance by analyzing surface-normal vector variation. The idea is to single out those few triangles that are likely to generate artifacts and to enforce very high sampling rates only for these.

In this paper, we describe and examine three different variants to achieve this. We show that all these approaches remove aliasing from bead chains, and generate high-quality renderings in real-time. Our techniques focus on the bead chain artifacts, but the basic idea applies to all types of aliasing that are predictable in object space.

2. Previous work

Aliasing is well understood from a theoretical point of view. The Nyquist–Shannon sampling theorem [4] is violated and therefore artifacts occur. To get rid of these artifacts, many different methods have been developed, and there is a vast body of research on antialiasing techniques. For more details, see [5] and references therein.

☆This article was recommended for publication by M. Wimmer.

* Corresponding author. Tel.: +49 91318567274.
E-mail address: Christian.Siegl@cs.fau.de (C. Siegl).

[1] Note that the physical effect generating Bailey's Bead Phenomenon is different, it is only the visual similarity that is striking.

0097-8493/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cag.2013.08.002

Super-Sample Antialiasing (SSAA) is the most straightforward way of performing antialiasing. Spatial SSAA raises the image resolution and subsequently downsamples the intermediate image to the target image size. Temporal SSAA jitters the camera positions across a series of frames and accumulates the results in the target image [6]. These approaches, however, are costly with respect to both memory usage and performance. Especially fill-rate suffers, due to the high number of fragment shader evaluations. From an image quality point of view, however, SSAA with an arbitrarily high number of samples can be seen as the ground truth regarding antialiasing. It would be possible to increase the sampling rate to a point where the Nyquist–Shannon sampling theorem is met at every point of the scene.

Hardware supported Multi-Sample Antialiasing (MSAA [7]) also increases the sampling frequency, however, only one fragment shader per primitive covering a pixel is evaluated at the pixel center. MSAA currently is the most widespread technique for antialiasing and works very well in a wide range of applications, especially for geometrical edges. For better image quality, centroid sampling can be enabled: Using regular MSAA, the position of the fragment shader might be outside of the sampled primitive which introduces artifacts. With centroid sampling, one of the covered multi-samples is selected as position of the fragment shader evaluation. More recent hardware techniques such as CSAA [8] and EQAA [9] follow the same principle. To allow a finer assessment of the pixel area a primitive covers, coverage samples are introduced. These samples only determine pixel coverage and do not issue fragment shader evaluations.

The increasing popularity of deferred shading rendering techniques raises the necessity for screen-space antialiasing methods, such as morphological antialiasing (MLAA) [10]. The basic idea behind MLAA is to take the final image, detect edges, re-vectorize these edges, and re-sample the image. Therefore, MLAA is a post-processing step, and it easily integrates nearly in any rendering pipeline. Techniques, such as FXAA [11], SRAA [12], and SMAA [13] further improve MLAA. All these techniques exhibit good image quality, while adding only little overhead. However, they only perform antialiasing for visible edges and cannot improve on other aliasing artifacts like bead chains.

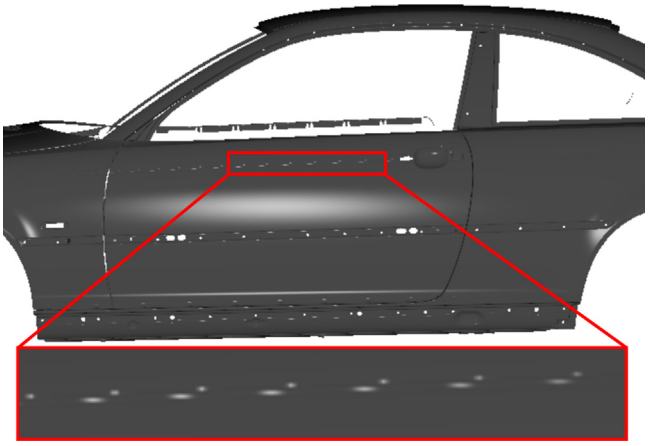

Fig. 1. Sideways view on the body of a BMW (500 k triangles) with specular highlights along sharp features without any antialiasing. While this specular highlight should appear in the form of one long continuous bright area, it degrades to a series of bright spots.

A survey paper that discusses surface shading was published by [14]. In this paper approaches toward pre-filtering the shading information (normals, horizon maps,…) that is covered by one pixel are discussed. The idea is to pre-filter for example the normals that occur in the area of one pixel. Since averaging normals linearly is not correct they rather pre-filter normal distribution functions.

So far we have focused on real-time rendering techniques for antialiasing. There are two nonreal-time approaches our work is loosely based on that have to be mentioned. First the well known Reyes rendering architecture [15]. The idea of subdividing an object into micro-polygons until their size is approximately the size of one pixel, comes very close to our approach of resampling in object space. Second, in ray tracing adaptive supersampling is very widespread. An early implementation was described by [16]. The idea is to adaptively refine the sampling at image positions based on a criterion that includes information such as edges between objects and contrast of the area. This comes close to the adaptive supersampling algorithms we will describe later on.

In his work “Pyramidal Parametrics”, [17] is the first one to describe the “bead chain” artifact with its annoying characteristics while moving. His solution to the problem is a mip-mapped (pyramidal) parametric illumination map that is able to antialias the aliasing from shading. He also suggests a solution where the local curvature of a model is limited. While both these solutions sound interesting, the first one is geared toward environment maps and the second one is prohibited in our case, as we do not want to change the model.

Nankervis shows an interesting approach toward antialiasing [18]. His idea is to separate shading from geometric sampling. To achieve this he renders the scene at a lower resolution and adds the geometry to per-pixel linked lists. In a second pass, the linked lists are traversed and coverage is evaluated at a higher resolution. We will use a similar approach later on, but while Nankervis supersamples the coverage, we supersample the shading.

3. Problem analysis

In the section, we analyze the sample configurations depicted in Fig. 2 and find that *triangle size* and *normal variation* are the two causes for bead chain artifacts. This helps us understand, why known antialiasing methods are incapable of removing bead chain artifacts.

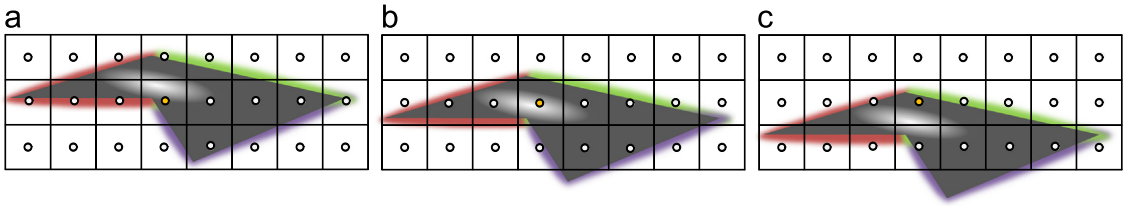

Fig. 2. Three sample configurations showing the two problems leading to bead chains. Looking at the orange sampling location, (a) shows that the green triangle, containing the specular highlight, is not sampled at all. Moving to (b) the specular highlight is sampled. In (c) the green triangle is sampled but the specular highlight is missed. For the orange sample location this results in (a) dark, (b) bright, and (c) dark. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this article.)

3.1. Triangle size

The screen-space size of a triangle is one cause for aliasing artifacts due to shading. Between the configurations in Fig. 2 (a) and (b), the sample scene is moved only slightly, resulting in a vastly different outcome for the pixel containing the orange sampling location. While the green triangle in Fig. 2(a) is not sampled at all, it is sampled three times in Fig. 2(b). Only a slight movement of the triangles, the orange sample point can either be a dark pixel (Fig. 2(a)), or a bright pixel (Fig. 2(b)). This leads to flickering artifacts during animations and bead chains for still images.

3.2. Normal variation

The second cause of aliasing artifacts due to shading is the triangle's per-vertex normal variation. In Fig. 2(b) and (c) the sampling location, represented by an orange circle, covers the green triangle. Even though the triangle is sampled in both cases, we obtain bead chains, as the pixel is bright in Fig. 2(b) and dark in Fig. 2(c). This is because the shading highlight is sampled in Fig. 2 (b) but it is missed in Fig. 2(c). Therefore, the bead chains are due to the size of the specular highlight (and not due the size of the triangle). The size of specular highlights computed with per-pixel lighting can be predicted by analyzing how much the three per-vertex normals of a triangle vary amongst each other: the higher the variation the smaller the highlight. In the example Fig. 2, the variation is high.

These two properties, which can be summed up as follows:

```
if(screenspace triangle    < pixel size and
   specular highlight size < pixel size)
then
    triangle is critical
```

will lead to the effects we described as bead chains.

3.3. Using known methods

Existing methods (see Section 2) are not able to effectively antialias the described shading artifacts with acceptable performance. Hardware techniques, such as MSAA, CSAA, or EQAA, are geared toward antialiasing silhouette boundaries and the number of multi-samples is limited on current GPUs. Thus, triangles smaller than the sample spacing are not necessarily rasterized, as they might not be covered by any sample at all. MSAA improves upon bead chain type artifacts, as a pixel may contain information from multiple triangles. Due to the small size of triangles, it is important to enable centroid sampling to prevent fragment shader evaluation outside of the triangle. However, only *one* fragment shader evaluation per-pixel and primitive is carried out. Therefore, MSAA-type methods fail to account for aliasing inside a primitive.

SSAA yields better results from a quality point of view, as every supersample is associated with one fragment shader evaluation. With SSAA, we could increase the sampling frequency to a point at which no more aliasing is visible. However, SSAA globally supersamples the image, also in locations, where fewer samples would suffice. In order to deal with aliasing from shading, high supersampling rates are necessary. For real-time scenarios, this is prohibitive regarding performance and memory consumption. Yet, we use SSAA as a "ground truth" in terms of image quality.

MLAA based techniques fail to antialias bead chains, as they only work for visible edges in image space. Adapting MLAA to handle aliasing from shading requires detecting and resolving the artifacts in image space. Detecting the given type of artifacts is difficult, as they occur in a variety of different patterns. Correctly removing them is even more difficult, as too much information is already lost in image space.

4. Adaptive supersampling

Based on the analysis of Section 3, we propose approaches tailored to deal with the special properties of bead-chain type artifacts. To handle other types of aliasing, MSAA proved to be a very good choice. Therefore, we render the entire scene using MSAA with centroid sampling into an off-screen render target, first. Taking this initial image and depth buffer, we make use of the fact that we know exactly which parts of the geometry lead to aliasing from shading. By placing additional samples on these critical triangles and blending them with the initial image, we can antialias bead chains.

The implementation of sorting out critical parts of geometry is also shared among all our implementations and performed in a preprocessing step. We categorize triangles based on their per-vertex normal variation, which is independent of the viewing transformation. Implemented using geometry shaders, triangles can be re-categorized quickly upon changes in geometry. The categorization works as follows: If one of the three mutual angles between the per-vertex normals of a triangle is above a user-specified threshold, we mark the triangle as critical. Depending on the lighting model and its parameters, we found that normals varying more than 2–12 degrees are prone to show bead chain artifacts. Note that we used Phong lighting and this rather simple criterion is sufficient. The decision if a triangle is critical may become more complex for more sophisticated lighting models.

In the following three sections, we will present algorithms for generating and blending additional samples. Following the basic idea of alleviating aliasing from shading by selectively forcing additional shader evaluations, we call our approach SSS.

4.1. Geometry shader SSS

The first implementation, that follows the previously described idea, mainly uses the geometry shader stage of the graphics pipeline. Inside the geometry shader, the screen-space size and the per-vertex normal variation of a triangle is computed. Based on these two factors, the triangle is either discarded because of not being critical in the current view, or marked to trigger additional fragment shader evaluations.

We can further reduce the number of triangles processed with any of our three algorithms using an *early-exit heuristic*: In the geometry shader, we evaluate the per-vertex normals of the triangle and compute how far it is away from a specular highlight (considering Phong lighting in our case). If this value is below a threshold, we assume that the triangle can be considered as no longer critical regarding aliasing from shading. Therefore, this triangle is not further processed. Fig. 3 and Table 1 show that our heuristic significantly reduces the amount of triangles processed and therefore improves rendering times by a large amount.

To trigger additional fragment shader evaluations inside the current triangle, point primitives are generated and passed down the rendering pipeline. Points are generated pseudo randomly on the inside of the triangle with additional samples along the border. The depth buffer is locked during this rendering pass and the points are drawn with a depth offset to ensure rasterization. The depth offset is required, because we use the depth buffer of the offscreen rendered complete scene at this point to prevent points from hidden geometry to become visible. For correct blending of the points the color value resulting from lighting computations inside the fragment shader stage is weighted with

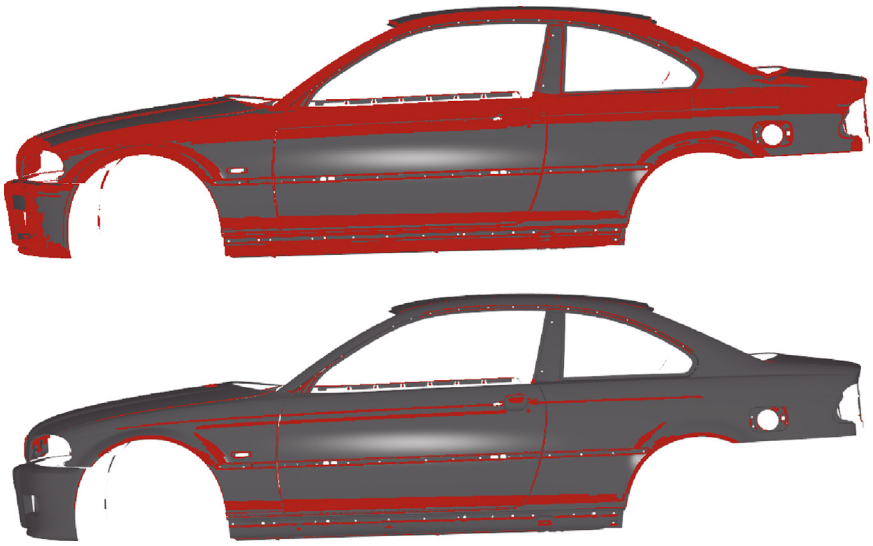

Fig. 3. Effect of using a heuristic based on the vicinity to a specular highlight for discarding critical geometry. Notice the massive drop in the number of triangles processed.

Table 1
Rendering times in ms of our three approaches without (w/o) and with (w) the early-exit heuristic. For an explanation of "Close up" and "Distant". Refer to Section 5.

Method	Close up		Distant	
	w/o	w	w/o	w
Geometry shader	46.0	17.2	8.4	4.6
Compute shader	20.4	8.3	16.3	4.9
PPLL	69.0	10.9	29.1	5.7

the size of the point. The size of the point is determined by the size of the source triangle and the number of generated points.

To get the final image, the additional samples are written to an offscreen render target and are blended with the previously rendered MSAA antialiased image. This is achieved in an additional rendering pass.

The hardware tessellation in DirectX 11 is unsuitable for our purposes because the tessellator stage cannot output point primitives. We could configure the tessellator to generate a large amount of triangles to simulate point rendering, however, this is very costly from a performance point of view.

4.2. Compute shader SSS

Starting from the point based implementation using the geometry shader, we implemented a second version that uses compute shaders. Using compute shaders, we aimed at achieving a better occupancy of the graphics card by having more control over how the processing takes place.

Each compute shader processes one triangle. The vertices have to be transformed into screen space and samples are generated like described in Section 4.1. Taking these sample positions the lighting computations are performed right inside the compute shader and therefore the rest of the rendering pipeline is cut out. This improves performance. The resulting color values have to be written into an off-screen render-target using a scattered write.

DirectX 11 currently does not offer the required synchronized add operation for floating point values. There is a workaround using *InterlockedCompareExchange* that includes a busy wait which introduces a massive performance hit. Considering that we only have to deal with values between 0 and 1, we decided on converting the floating point values to scaled integer values. For integer values, a synchronized add operation is available.

Similar to Geometry Shader SSS, the final image again has to be composited in an additional rendering pass.

4.3. Per-Pixel linked list SSS

The last and maybe most advanced approach we took, is utilizing per-pixel linked lists.

With the introduction of atomic memory access operations, atomic counters, and random memory access in DirectX 11 and OpenGL 4.2, it is possible to maintain one linked lists per pixel on the GPU in real-time [19].

To achieve this, we require a *node-buffer* and a *head-buffer* stored in GPU memory. The *node-buffer* contains the data of the list elements and an offset to the previous entry of the list. For every pixel of the render target, the *head-buffer* stores an offset to the last list element in the node-buffer. The head-buffer therefore has the same size as the render target.

When adding an element to a per-pixel linked list, a global counter that points to the first free element of the node-buffer is incremented. The payload is written to the node-buffer. Using atomic operations to avoid race conditions, the pixel's head-buffer entry is exchanged with the value of the global counter. The previous entry of the head-buffer serves as an offset to the previous list element.

The core of the algorithm consists of two passes: In the first pass, we create *per-pixel linked lists*, containing all the information required for antialiasing (Section 4.3.1). A second pass *resolves* the per-pixel linked lists, and creates antialiased pixels (Section 4.3.2) that are blended with the previously rendered complete scene.

4.3.1. Create per-pixel linked lists

To resolve the problem of triangles not covering sampling locations, we decided on performing conservative rasterization [20]. Using this technique, a triangle is enlarged such that for every pixel the triangle intersects, a fragment shader evaluation is ensured. This is depicted in Fig. 4. The screen-space bounding box for the green triangle is computed and enlarged by half a pixel in every direction. The enlarged bounding box covers all pixel centers of pixels the triangle intersects. These computations are carried out in a geometry shader, and the enlarged bounding box is passed down the rendering pipeline alongside all information

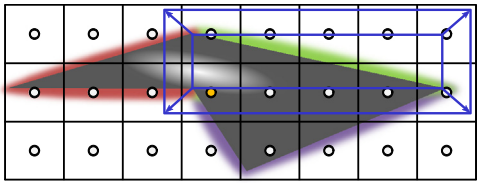

Fig. 4. Conservative rasterization. The screen-space bounding box is computed and enlarged by half a pixel in every direction. Therefore, the pixel center of every pixel the triangle intersects is covered and a fragment shader is evaluated.

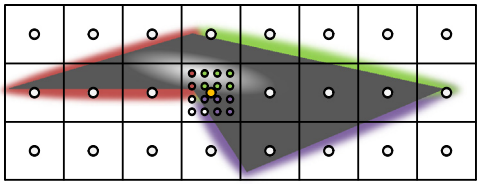

Fig. 5. Supersampling inside the fragment shader. The sampling frequency is adapted to the per-vertex normal variation to register all specular highlights. The resulting color is written into the per-pixel linked lists alongside the coverage and depth value. Per-pixel linked lists are blended based on coverage and depth values in a separate rendering pass.

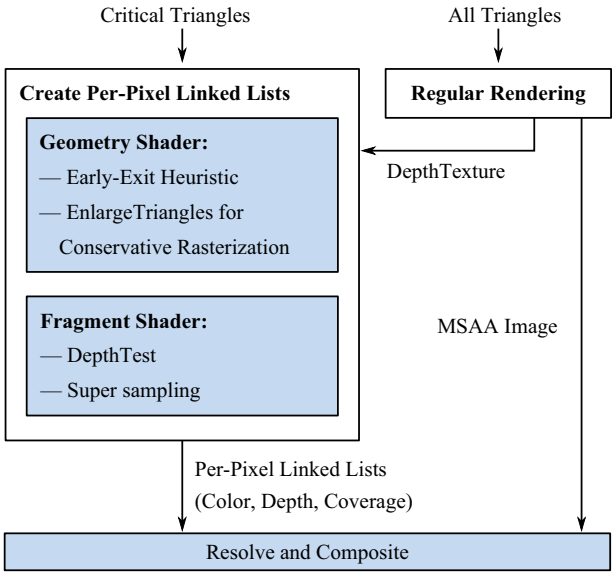

Fig. 6. Overview over the complete rendering pipeline of the per-pixel linked list approach.

about the original triangle for correct sampling in the fragment shader. To improve performance, an early exit like described in Section 4.1 is added to discard triangles that are not critical in the current view.

The second problem that has to be addressed is the small size of specular highlights (high per-vertex normal variation). To correctly determine the final pixel color, supersampling the pixel area is required, as shown in Fig. 5. Each fragment shader thread supersamples its pixel area. By adapting the supersampling rate to the per-vertex normal variation, the specular highlight is no longer missed.

In Fig. 5, for the pixel showing the colored supersampling locations, three fragment shaders are evaluated. One for every triangle that intersects the pixel. Every fragment shader samples all supersampling locations and computes a color value. The number of samples that cover the sampled triangle (six in case of the green triangle) divided by the total number of samples (16 in this case) embody the coverage of the triangle on the pixel. The color value, the coverage value, and the depth value are appended to the per-pixel linked list.

Hardware *depth test* has to be disabled while creating per-pixel linked lists to ensure that every possible fragment shader is evaluated. However, some fragments are too far behind the visible surface and therefore do not contribute to the final image. To further accelerate per-pixel linked list SSS, we manually discard these fragments by testing every fragment's depth against the depth buffer from the previously rendered full scene.

4.3.2. Resolve

In a second pass, we resolve the per-pixel linked lists to determine the final color of the pixel. A screen-aligned quadrilateral is rendered to issue one fragment shader evaluation per pixel of the final image. We process the per-pixel linked list entries front to back. The colors are multiplied with their coverage value and added up. The unaltered coverage values are also added up. We can stop processing the list when the added coverage values reach 1.0, resulting in full coverage of the pixel. If the coverage value does not reach 1.0 after processing the entire list, the color value has to be blended with the previously rendered image of the scene (see Section 4). Because we perform blending with the previously rendered offscreen render target at this point, no additional pass for compositing is required.

The described two rendering passes are embedded into the rendering pipeline depicted in Fig. 6.

4.4. Visibility

One last aspect that we want to elaborate on is visibility. The presented algorithms perform antialiasing of bead chains with an important presumption: We assume to deal with a closed surface of a solid material (no transparency) and only consider the front-most parts. This assumption is critical for the presented kind of algorithms, as we depend on the possibility to independently process triangles. Considering that bead chains mainly occur on solid shiny surfaces like car paint, the presumption is very close to the circumstances we are confronted with. Merely under some extreme viewing angles, it becomes possible to see artifacts that originate from this presumption. For the general case these artifact do not occur, the depth testing performed against the depth buffer of the MSAA-rendered off-screen render-target.

5. Results and discussion

Fig. 7 shows the results of our implemented algorithms compared to other previously mentioned antialiasing techniques on an Intel Core i7-2600 at 2.8 GHz, 4 GB of RAM and an Nvidia GeForce GTX 470. The model shows a BMW E46 consisting of 500 k triangles rendered at a resolution of 1280×720. We decided on using MSAA with centroid sampling as the most widely used antialiasing technique. For quality comparison, we use SSAA.

Going from no AA to MSAA (Count: 8, Quality: 32) shows little improvement in quality. The results of MSAA vary depending on the current view configuration and the aliasing along the specular highlight is still very obvious, especially while moving. Review the accompanying video for a better understanding of the aliasing on a moving object.

Going to $16 \times$ SSAA, the gaps between the bright spots begin to close and quality improves noticeably. The step to $64 \times$ SSAA again shows an apparent improvement. Looking at the selected specular highlight, bead chains are no longer visible.

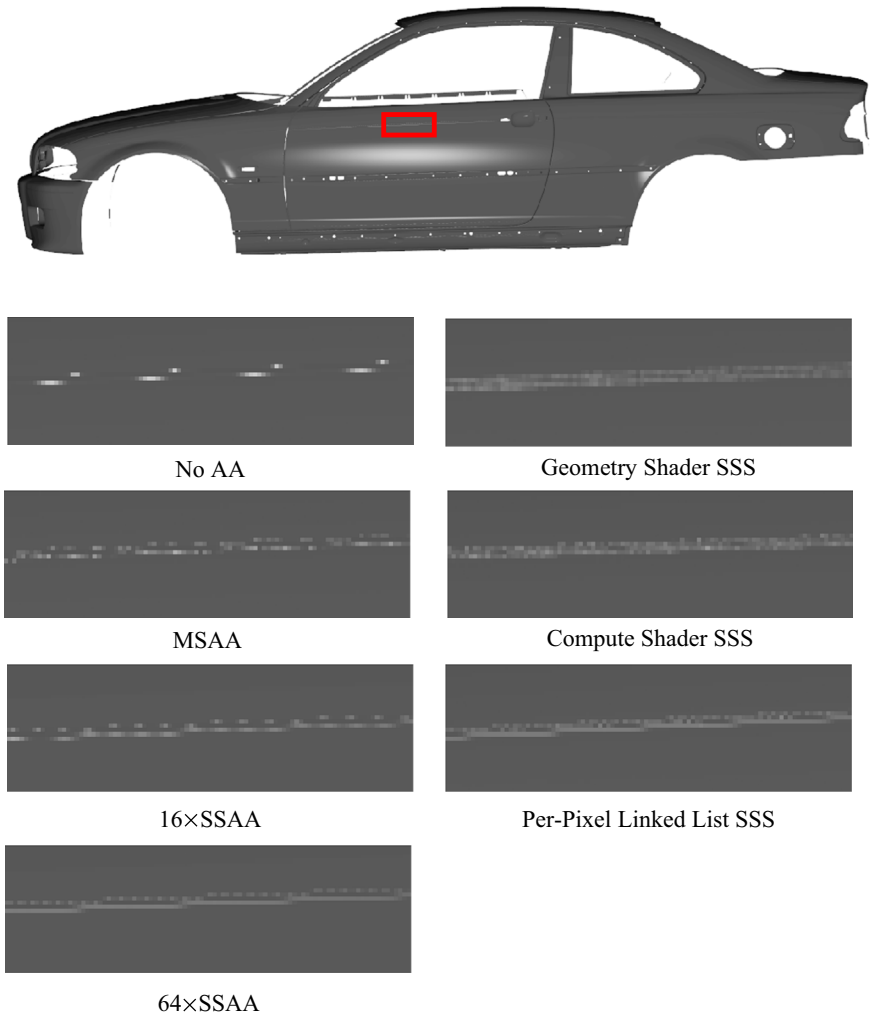

Fig. 7. Results of geometry shader SSS, compute shader SSS and per-pixel linked list SSS compared to SSAA, MSAA, and no AA. Review the accompanying video to get a better impression.

All our algorithms show a perceived quality that is well above no antialiasing, MSAA and $16 \times$ SSAA. Quality is on par or even above $64 \times$ SSAA. The differences in the appearance between our algorithms are mainly due to differences in the sampling process and the number of samples required to achieve the presented image quality. We aimed at an image quality similar to $64 \times$ SSAA.

Note that the geometry and compute shader based approaches are even able to antialias the staircase artifacts that still show in $64 \times$ SSAA and are also visible in per-pixel linked list SSS.

Fig. 8 shows the radiator grill of a truck. On this high-frequency detail, aliasing from shading appears on every cylindrical surface along the grill. Again, MSAA shows only very little improvement. Going to $64 \times$ SSAA or our per-pixel linked list SSS shows very good quality. The chain like structure of the specular highlights is no longer visible.

In Table 2, we provide performance numbers for two viewing configurations. The first view ("close up") shows the BMW filling the full screen, in the second configuration ("distant") the BMW only covers about 10% of the screen. Aliasing from shading becomes especially problematic for distant views as more critical triangles will fall onto the area of one pixel. However on a smaller object details are harder to see. The resulting performance numbers vary between these two configurations as the number of fragment shader evaluations changes. For the geometry- and compute-shader-based algorithms, the two different configurations result in a change in the size of the affected triangles and therefore a change in the number of generated point samples. For Per Pixel Linked List SSS more pixels are covered resulting in more fragment shader evaluations (similar to SSAA)

Considering performance, we first have to compare our algorithms to MSAA which is the most common technique for antialiasing. Point-based rendering using geometry shaders comes closest to the time of MSAA for the distant view configuration but can not keep up for the close up view. Overall, our fastest implementation is the compute shader based approach. Given that all our implementations are based on MSAA and perform additional sampling on top, this is not unexpected. Considering that MSAA can not provide acceptable results in terms of quality, we also have to compare our algorithms to SSAA. Only $64 \times$ SSAA shows the quality we want to achieve. All our algorithms are able to outperform $64 \times$ SSAA under all viewing conditions. Thereby, our compute-shader-based implementation is three times faster than $64 \times$ SSAA.

Table 3 shows a decomposition of our three algorithms into their parts. A render pass with MSAA is the first step in all three cases. The next step in all cases is the generation of additional samples which takes up most of the time. Note, that this part is also the one that dominates the scaling behavior of the overall algorithms. Finally the resolve pass combining the gathered information takes a similar amount of time than the initial rendering pass.

Note, that the performance of our algorithms highly depends on the number of processed triangles. Only visible critical triangles

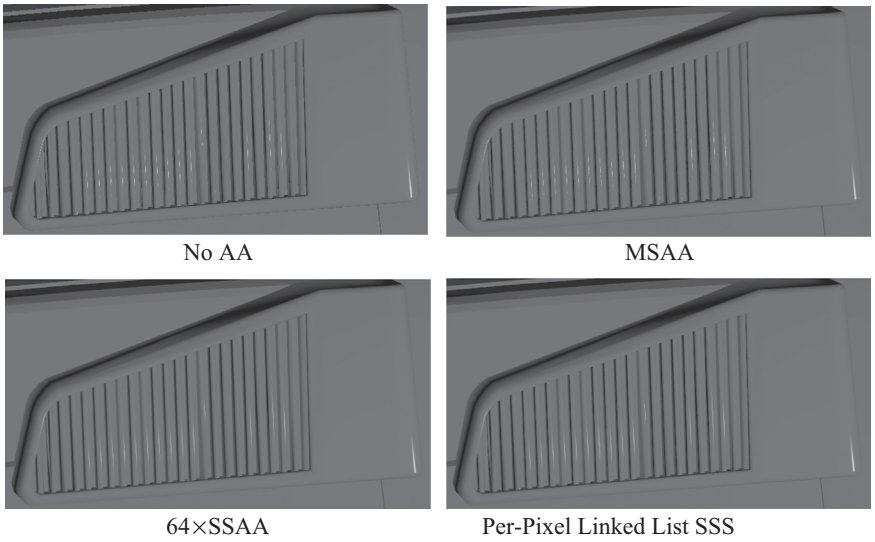

Fig. 8. Results of per-pixel linked list SSS compared to the standard antialiasing algorithms on the high-frequency detail of a radiator grill.

Table 2
Rendering times and frame rates of geometry shader SSS, compute shader SSS and per-pixel linked list SSS compared to SSAA, MSAA, and no AA.

Method	Close up		Distant	
	Time (ms)	Rate (fps)	Time (ms)	Rate (fps)
No AA	1.6	520	1.6	520
MSAA	1.9	340	1.9	340
16 × SSAA	4.8	175	3.6	240
64 × SSAA	25.1	38.7	13.1	73
Geometry shader	17.2	56	4.6	175
Compute shader	8.3	113	4.9	189
PPLL	10.9	89	5.7	152

Table 3
The three antialiasing techniques decomposed into their parts. In all cases the generation and evaluation of additional samples takes up the biggest amount of time.

Renderpass	Geometry shader		Compute shader		PPLL	
	Close up (ms)	Distance (ms)	Close up (ms)	Distance (ms)	Close up (ms)	Distance (ms)
Render	1.4	1.2	1.4	1.2	1.4	1.2
Sample	14.0	2.3	5.5	2.3	8.4	3.9
Resolve	1.8	1.1	1.4	1.4	1.1	0.7

that are close to a specular highlight are treated with our algorithm. In configurations where such triangles are not present, the overhead for performing one of our pipelines is minimal. By classifying less triangles as critical and adjusting the early-exit heuristic, the number of processed triangles can easily be influenced to trade-off rendering time versus image quality. Also the number of generated samples resulting from a triangle's screen-space size and per-vertex normal variation can be adapted to either favor performance or quality.

Regarding performance, there is another factor that has to be taken into account. While SSAA supersamples the entire scene, our approaches are highly adaptive. Consider an entire scene with more objects. Using SSAA the entire scene has to be supersampled. Using one of our algorithms the overhead is strictly confined to the objects that can show bead chains and even only to the triangles within these objects that are critical.

The last advantage of the presented algorithms is their memory consumption. Using $64 \times$ SSAA on an image of size 1280×720 takes up 225 MB of memory to store the supersampled image. With our algorithms, this amount of memory can be reduced significantly. Regarding the point-based approaches, no additional memory is required as the samples are generated on-the-fly. Regarding per-pixel linked list SSS, we need some memory to store the per-pixel linked lists. In the presented example, we accounted for an average number of per-pixel linked list entries of four. This results in 27 MB of memory consumption. By introducing a mechanism for recognizing buffer overflows, this amount of memory could be reduced even further.

Given all these advantages, there is one possible drawback we want to comment on: In our examples, we use the rather simple Phong lighting model. In case of compute shader SSS and per-pixel linked list SSS, the lighting has to be evaluated inside a loop in the shader. This means that for very complex shaders, the performance might suffer.

6. Conclusion

Aliasing from shading is a subject that is mostly ignored by current techniques for antialiasing. The most prevalent solution regarding this topic often is to adapt shading or the model itself. While this is possible in certain applications, it is not feasible for others. This is especially true for the highly detailed car models we work with.

In this work, we have presented three novel methods for performing antialiasing of bead chains. It became obvious that the artifacts occur due to small triangle size and high per-vertex normal variation resulting in small specular highlights. Taking these insights, we presented three algorithms that are capable of resolving the problem while performing better than SSAA and for certain configurations even close to MSAA. At the same time all implementations offer the flexibility to be adjusted on the fly to meet runtime limits while preserving optimal image quality.

Appendix A. Supplementary material

Supplementary data associated with this article can be found in the online version of http://dx.doi.org/10.1016/j.cag.2013.08.002.

References

[1] Blinn JF. Jim Blinn's corner: ten more unsolved problems in computer graphics. IEEE Computer Graphics & Applications 1998;18:86–9.
[2] Newell ME, Blinn JF. The progression of realism in computer generated images. In: Proceedings of the 1977 annual conference. p. 444–8.
[3] Heckbert P. Ten unsolved problems in rendering. In: Workshop on rendering and algorithms, graphics interface '87. p. 1–4.
[4] Shannon CE. Communication in the presence of noise. Proceedings of the IRE 1949;37:10–21.
[5] Jimenez J, Gutierrez D, Yang J, Reshetov A, Demoreuille P, Berghoff T, et al. Filtering approaches for real-time anti-aliasing. In: ACM SIGGRAPH 2011 courses, SIGGRAPH '11. p. 6:1–329.
[6] Haeberli PE, Akeley K. The accumulation buffer: hardware support for high-quality rendering. In: Computer graphics (proceedings of SIGGRAPH '90), vol. 24. p. 309–18.
[7] Akeley K. Reality engine graphics. In: SIGGRAPH '93, ACM, 1993. p. 109–16.
[8] Young P. CSAA (Coverage Sampling Antialiasing). ⟨http://developer.nvidia.com/csaa-coverage-sampling-antialiasing⟩, 2007.
[9] Advanced Micro Devices, EQAA Modes for AMD 6900 Series Graphics Cards. ⟨http://developer.amd.com/sdks/radeon/assets/EQAA%20Modes%20for%20AMD%20HD%206900%20Series%20Cards.pdf⟩, 2011.
[10] Reshetov A. Morphological antialiasing. In: Proceedings of the conference on high performance graphics 2009. In: Proceedings of the 2009 ACM symposium on high performance graphics, ACM, 2009. p. 109–16.
[11] Lottes T. FXAA (Whitepaper), ⟨http://www.ngohq.com/images/articles/fxaa/FXAA_WhitePaper.pdf⟩, 2009.
[12] Chajdas MG, McGuire M, Luebke D. Subpixel reconstruction antialiasing for deferred shading. In: Symposium on interactive 3D graphics and games '11. p. 15–22.
[13] Jimenez J, Echevarria JI, Sousa T, Gutierrez D. SMAA: enhanced subpixel morphological antialiasing. Computer Graphics Forum 2012;31:355–64.
[14] Bruneton E, Neyret F. A survey of nonlinear prefiltering methods for efficient and accurate surface shading. IEEE Transactions on Visualization and Computer Graphics 2012;18:242–60.
[15] Cook RL, Carpenter L, Catmull E. The Reyes image rendering architecture. In: Computer graphics (proceedings of SIGGRAPH '87), vol. 21. p. 95–102.
[16] Painter J, Sloan K. Antialiased ray tracing by adaptive progressive refinement. In: Computer graphics (proceedings of SIGGRAPH '89), vol. 23. p. 281–8.
[17] Williams L. Pyramidal parametrics. In: Computer graphics (proceedings of SIGGRAPH '83), vol. 17. p. 1–11.
[18] Nankervis A. Linked list antialiasing and edge resampling, ⟨http://naixela.com/alex/⟩, 2012.
[19] Yang JC, Hensley J, Gruen H, Thibieroz N. Real-time concurrent linked list construction on the GPU. Computer Graphics Forum 29:2010.
[20] Jon Hasselgren LO, Akenine-Möller Tomas. Conservative rasterization. In: GPU Gems 2. Addison-Wesley Professional, 2005. p. 677–90.