Screen-Space Ambient Occlusion Using A-buffer Techniques

Fabian Bauer
TU Darmstadt
Darmstadt, Germany

Martin Knuth
Fraunhofer IGD
Darmstadt, Germany

Arjan Kuijper
Fraunhofer IGD
Darmstadt, Germany

Jan Bender
TU Darmstadt
Darmstadt, Germany

***Abstract*—Computing ambient occlusion in screen-space (SSAO) is a common technique in real-time rendering applications which use rasterization to process 3D triangle data. However, one of the most critical problems emerging in screen-space is the lack of information regarding occluded geometry which does not pass the depth test and is therefore not resident in the G-buffer. These occluded fragments may have an impact on the proximity-based shadowing outcome of the ambient occlusion pass. This not only decreases image quality but also prevents the application of SSAO on multiple layers of transparent surfaces where the shadow contribution depends on opacity.**

We propose a novel approach to the SSAO concept by taking advantage of per-pixel fragment lists to store multiple geometric layers of the scene in the G-buffer, thus allowing order independent transparency (OIT) in combination with high quality, opacity-based ambient occlusion (OITAO). This A-buffer concept is also used to enhance overall ambient occlusion quality by providing stable results for low-frequency details in dynamic scenes. Furthermore, a flexible compression-based optimization strategy is introduced to improve performance while maintaining high quality results.

***Keywords*-rasterization; screen-space ambient occlusion; order independent transparency; real-time rendering**

I. Introduction

Traditional rasterization renderers make use of a depth buffer to discard occluded fragments when rendering into the main framebuffer. This means that screen-space effects can only be applied to the first visible layer of geometry limiting the application of deferred algorithms. However, the A-buffer introduced by Carpenter [1] allows to store and access each rasterized fragment in GPU memory. This enables to solve the order independent transparency (OIT) problem on a per-fragment level on the GPU.

Ambient occlusion (AO) in screen-space can also benefit from multiple fragment layers: Most of the time the ambient occlusion term is computed in screen-space and merely applied to the whole scene assuming opaque geometry only, disregarding the special considerations imposed by translucent materials. AMD's FirePro SDK [2] implements this approach. However, AO is only computed for the first layer leading to visible artifacts as shown in figure 3. Ambient occlusion for opaque geometry also benefits from a depth-complete scene representation compensating occlusion-based artifacts common in screen-space as illustrated in figure 2.

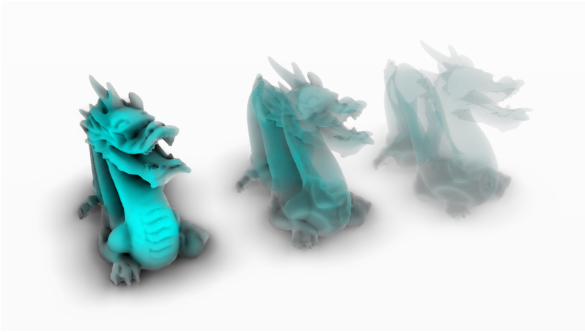

Figure 1. Multilayer ambient occlusion enabling transparent geometry based on material opacity

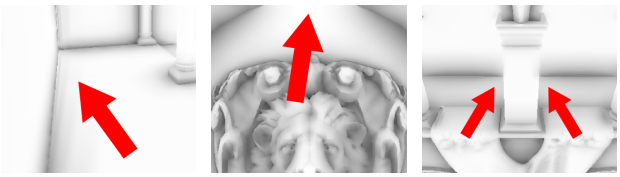

Figure 2. Errors resulting from single layer SSAO: Occluded fragments (left), triangles outside camera frustum (middle) and grazing angles (right)

In this paper we present a novel approach which includes multiple geometry layers to produce a coherent and more plausible ambient occlusion result including transparent and semi-transparent surfaces demonstrated in figure 1. Furthermore, different implementations of the A-buffer strategy are tested to give a detailed overview of the image quality and performance outcome.

II. Related Work

Order independent transparency for rasterization renderers is a classical problem in computer graphics. Modifying the blending function allows a fast handling of transparency [3] but introduces scene limitations. Therefore, numerous solutions have been developed to enable color correct alpha blending for arbitrary geometry on fragment level. Myers et al. [4] provide a fast but constricted OIT solution: A limited set of layers is stored within subpixels of multisample textures using stencil routing. A more flexible approach was presented by Bavoil et al. [5] called depth peeling. They

Published by the IEEE Computer Society

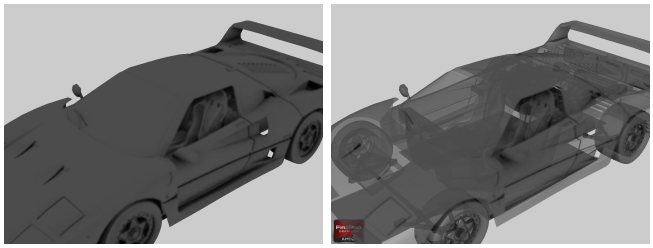

Figure 3. Wrongly accumulated colors for a transparent object (right) where the car interior is not shadowed, compared to full opacity (left) when handling a single layer of depth only.

use a multipass rendering approach discarding the layers of previous passes via depth comparisons. The final image is created by blending the single layers together leading to quality OIT. Its drawbacks are the high (texture) memory consumption and the costly multipass rendering. Several optimizations exist which target these problems by reducing the number of passes required [6], by adding sorting to the method [7] or by constricting the computation to important parts of the scene [8]. AMD presented a novel method [9] of solving OIT by maintaining a Per-Pixel Linked List (PPLL) in texture memory using append/consume buffers. This way every rendered fragment is stored at a new memory location creating unsorted lists holding all information necessary for OIT in a single pass. As an extension to this method Intel demonstrated an approximation of OIT called Adaptive Transparency [10]. Not requiring fragment sorting, the approach achieves higher performance at the expense of image quality by using a transmittance function.

Screen-Space Ambient Occlusion covers methods and algorithms to solve the occlusion integral [11] in screen-space using texture sampling. First discussed by Crytek [12], it was used within their video game Crysis. Basically, AO is computed from the depth information contained inside the rendered frame, treating the screen information as a height-field.

Numerous derivatives, extensions and attributions have been made to improve on quality, performance and usability. Shanmugam and Arikan [13] extend the technique to incorporate distant occluders. Horizon-Based Ambient Occlusion (HBAO) by Bavoil et al. [14] improves the shadow quality by ray marching in image-space. Screen-Space Directional Occlusion (SSDO) [15] assumes a general shadowing direction also allowing for color bleeding and a single light bounce. To cover larger sampling kernels while still maintaining high performance, Huang et al. [16] presented a multiresolution algorithm (MSSAO) at the expense of texture memory. The screen depth of the current view can hide cavities important for AO. In [17] Vardis et al. tackle this problem by using the shadow buffer information of the present lights in a scene in conjunction with fusing the SSAO computations to a single image.

Another approach taking into account the whole scene is using a voxel representation to perform occlusion computations. Thiedemann et al. [18] use this capability to perform fast global illumination computations. A drawback is the high amount of memory necessary to store the voxel representation and the additional time needed for geometry voxelization. Another volumetric approach was presented by McGuire [19]: Influence of self occlusion is represented by polygonal volumes derived from the scene geometry. The technique gives good results but is scene complexity and fill-rate dependent. For volume rendering low-frequency lighting of volumetric data can be achieved by using a pre-computed radiance transfer (PRT) which has to be re-computed many times for dynamic scenes. A fast way of doing this re-computation utilizing the GPU is presented by Tobias Ritschel [20]. Fully shadowed regions can be avoided by relying on local ambient occlusion approximations as shown by Hernell et al. [21]. In an approach by Ropinski et al. [22] indirect illumination is achieved by capturing every possible light interaction of the volume structures in the scene in a preprocessing step.

Regarding multilayer ambient occlusion Bavoil et al. [23] presented an approach enabling HBAO calculations on opaque geometry checking multiple depth layers. The basic idea is to apply depth peeling to the scene until a sufficient amount of layers are available in texture memory. This multilayer concept alongside multiple camera angles was also used for SSDO to cover hidden regions. However, our approach allows to populate the A-buffer in a single geometry pass with sparse data structures improving on both performance as well as memory requirements. Going one step further we extend this principle to allow transparent surfaces to cast and receive AO shadows based on their respective material opacity.

III. Concept

Ambient occlusion for opaque geometry only considers the first geometry layer. However, when resolving OIT with AO, the shadowing factor must be computed for each layer before transparency blending operations are applied. That means that AO results must be available in the resolving pass where fragments are retrieved, sorted and then blended. Blending operations can not be dissolved and must be executed with sorted fragments since AO may darken the fragment color resulting in a different final blend color. Consequently AO is inseparably mixed into the color of transparent geometry preventing further processing like bilateral filtering. Moreover, since OIT has to be resolved in fullscreen resolution, the AO factors for such surfaces have to be determined with these dimensions as well, which might not always be acceptable regarding performance and the concept of the SSAO algorithm. Sacrificing color correctness for the sake of flexibility is possible by separating the AO calculation from the OIT resolving stage. The AO results are

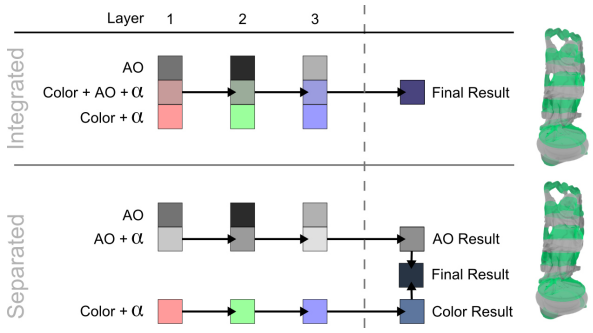

Figure 4. An example of the integrated and separated AO layer merging process. Using striped shading on the Stanford Happy Buddha mesh (right) reveals nearly no differences in the result.

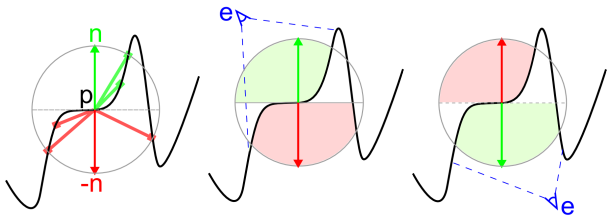

Figure 5. Opacity-based AO sampling for a point $\mathbf{p}$ and its normal $\mathbf{n}$ with camera-dependent weighting of exterior (green) and interior (red) AO

computed and then blended into a 2D texture using material opacity similarly to transparency color blending. An example for the *separated* as well as the color-correct *integrated* layer merging scheme is given in figure 4. As can be seen, the color difference between both approaches is very low and can be neglected.

Another aspect of AO with respect to OIT is the integration of the alpha-based opacity value into the AO formula. Geometry with high transparency should not shadow nearby geometry with the same intensity as fully opaque objects. This allows semi-transparent meshes to contribute to the AO term of the scene depending on material attributes.

Correctly merging OIT and AO offers multiple ways how transparent geometry can be interpreted. Each triangle the mesh is composed of can be considered as frontfacing or backfacing depending on the order of vertices in the buffer. Backfaces of opaque objects are never visible allowing the use of backface culling techniques to improve performance. Hence, the used triangles describe a solid volume of the object. A thin and possibly transparent hull can only be described by adding polygons for the interior geometry without relying on backface data. On the other hand, for high performance applications like video games it is common to interpret the triangle data as a very thin layer describing a hull of the object. These different interpretations of triangle geometry for transparent surfaces require a distinction regarding transparent color accumulation as well as AO gathering.

For solid volumes the AO formula can be applied introducing the sampled alpha values as a weight:

$$AO(\mathbf{p}) = \frac{\alpha_p}{\pi} \int_{\omega \in \Omega} \alpha_\omega V(\mathbf{p}, \omega) |\omega \cdot \mathbf{n}| d\omega, \qquad (1)$$

where $\mathbf{p}$ is the current position with the corresponding normal $\mathbf{n}$ and Ω is the sampling hemisphere. The visibility function V returns a value between 0 and 1 and describes the occlusion of $\mathbf{p}$ in direction ω. α_p and α_ω are the opacity values for $\mathbf{p}$ and for the sampled point in direction ω respectively. If geometry is interpreted as a thin object, additional factors have to be considered. AO is usually computed by sampling into a hemisphere oriented around the normal $\mathbf{n}$ in the current point $\mathbf{p}$. In this case transparent triangles actually consist of two sides making up the hull. This means the AO integral has to be resolved two times: Once for the usually applied hemisphere around the normal $\mathbf{n}$ in $\mathbf{p}$ and once more for the hemisphere oriented around $-\mathbf{n}$ as shown in figure 5.

As a matter of fact sampling now occurs in a complete sphere taking interior occlusion AO_i and exterior occlusion AO_e of points into account. These two occlusion factors have to be applied to the same point according to the alpha value of the surface:

$$AO(\mathbf{p}) = \alpha_p AO_e(\mathbf{p}) + (1 - \alpha_p) AO_i(\mathbf{p}). \qquad (2)$$

Another factor is the camera position: The weighting of AO_i and AO_e depends on the angle between the camera view vector and the normal. If the camera views a backface, the weights for AO_i and AO_e must be switched to allow correct blending of both AO values for the current image. Introduced with a special weighting function this assures correct color mixing for both AO terms depending on the translucency of the layer:

$$W(\mathbf{p}) = \begin{cases} \alpha_p & \text{if } \mathbf{p} \text{ is on frontface} \\ 1 - \alpha_p & \text{if } \mathbf{p} \text{ is on backface} \end{cases} \qquad (3)$$

$$AO(\mathbf{p}) = W(\mathbf{p}) AO_e(\mathbf{p}) + (1 - W(\mathbf{p})) AO_i(\mathbf{p}), \qquad (4)$$

where α_p describes the opacity value of the fragment color in $\mathbf{p}$.

A. Ambient Occlusion Formula

Usually a fragment entry stores color with alpha used as opacity and depth. Since AO has to be computed for every visible fragment on the screen every entry in the list has to store the normal of the face as well. Applied on deferred texture buffers, SSAO algorithms usually check the pixel neighborhood of each screen coordinate to sample fragment data. To additionally account for transparent surfaces every sample position is checked for an entry in the OIT fragment buffer. If the screen coordinate is covered by transparent fragments, they are fetched from the buffer and decomposed into the necessary spatial components. Each of these layers

may contribute to the AO term of this sample but to maintain a uniform distribution of sampling directions only the maximum shadow value gathered from these fragments is used. This process of computing AO for a given position is described in algorithm 1.

Algorithm 1 Multilayer ambient occlusion computation for a given view-space position $\mathbf{p}$ using fragment lists

```
1:  procedure ComputeSSAO(p)
2:      ao_final ← 0
3:      c ← 0
4:      for all ω in Ω do
5:          s ← GetScreenCoordinate(p + ω/p.z)
6:          fraglist ← ABuffer.fetch(s)
7:          ao ← 0
8:          for all f in fraglist do
9:              ao ← max(ao, GetAO(p, f))
10:         ao_final ← ao_final + ao
11:         c ← c + 1
12:     return ao_final/c
```

As is common with SSAO techniques the normalized sampling direction ω is scaled according to the current view-space depth $\mathbf{p}.z$ of the current position p. The function *GetAO* calculates the AO term for the given position $\mathbf{p}$ with respect to the sampled layer fragment f. Its mode of operation therefore depends on the AO algorithm of choice. If the screen coordinate of position $\mathbf{p}$ is covered by transparent fragments, special care has to be taken: Similarly to resolving OIT the fragment list of the current position $\mathbf{p}$ is retrieved and sorted by depth. Then the respective occlusion results are calculated and blended according to the alpha value of the fragment. Finally, the blended result is stored in a 2D render target to allow further processing.

To increase AO quality for opaque geometry as well the fragment list concept can be applied in a similar fashion: In this case it is usually enough to store solely the linear depth value of the fragment in the A-buffer. The shadowing factor is only computed for the first layer of depth, while occlusion sampling is performed for every layer by traversing the fragment lists of each sample position (see algorithm 1).

B. List Compression

Fetching and processing multiple texels for each sample is time consuming. In order to speed up the ambient occlusion sampling stage, an optimization strategy similar to Intel's Adaptive Transparency technique [10] is applied. The basic idea is to compress the OIT fragment list by discarding fragments which are unlikely to contribute to the ambient shadowing term of the scene.

The compression is performed in the OIT resolving stage where the fragment list is retrieved, sorted and blended. Each fetched list element is inserted into a fixed size array a where each entry $a[i]$ at index i consists of the depth $a[i]_z$ and opacity $a[i]_\alpha$ of the fragment. The length m of the array defines the maximum allowed number of nodes and therefore the quality of the compression. The array is initialized with depth values of 1. Inserting a fragment f is done by looking up the index i of a where $a[i]_z < f_z < a[i{+}1]_z$. This assures a sorted order of the nodes. In addition to each OIT list entry, fragments of opaque geometry have to be considered as well and are inserted into the array with an opacity of 1.

As soon as the insertion function is called more than m times, the list is compressed by eliminating the node at index i which is retrieved in the following way:

$$i = \min_j(a[j]_\alpha(1.0 - a[j]_z)), \tag{5}$$

where $0 \leq j < m$. For one part this criterion makes sure that fragments with high opacity are favored while for the other part fragments with low depth are preferred opposed to those resident in greater distance. The search for this index can optionally be executed only for the latter half of the array to preserve fragment data closer to the camera. With this criteria fragments which are close to camera and/or have high opacity values are favored while highly transparent and/or distant fragments are dropped. The resulting array therefore only contains the most relevant occlusion information which can be stored in a 2D buffer. For example a texture of the format RGB32UI can store up to four nodes.

Using the same criteria as above, normals can be compressed and stored in a 2D texture as well. In conjunction with the depth and opacity texture this information can be used to reconstruct the most important list elements where AO sampling has to be conducted in the SSAO pass. Texel fetches demanded by the sampling process are done relying on the compressed depth and opacity information where each of the m list elements is checked for occlusion shadowing. In summary, the original fragment list buffers are only needed to blend the transparent color and are never queried for ambient occlusion calculations. By reducing texture buffer access to a single texel fetch per sample instead of repeated queries induced by a fragment list traversal, the memory access coherency and cache utilization for the ambient occlusion pass is significantly improved.

IV. Implementation

The approaches described above were implemented in OpenGL using three different fragment list memory architectures to test performance and memory requirements. Traversing the elements of a texture buffer can be expensive. Modern GPUs improve performance by using texture caches. These caches only work for neighboring texels in directions implied by the buffer data type in one, two or three dimensions. Thus, neighbor access is much faster then random access. It is therefore wise to implement efficient memory layouts regarding this fact.

List elements in the buffer typically consist of three essential pieces of information encoded into three 32-bit unsigned integers:

- The four-component color of the fragment where the alpha channel describes the opacity of this layer.
- A single floating point depth value used as a sorting criterion when resolving the fragment list.
- The fragment normal in view-space for the AO computation. For thin geometry the normal is negated according to the camera position, following figure 5.

Since sorting and blending is needed in the OIT and the AO pass, it can be more efficient to store the sorted fragment lists again after transparency is resolved. Then the AO pass is able to blend the shadow results of fetched list entries without further arrangements.

A. Per-Pixel Linked List

The PPLL implementation is based on the work of Yang et al. [9] where a single geometry pass stores rasterized fragments in texture memory. In the following fullscreen pass the fragment pool is queried for each list entry of the current pixel position to enable sorting and blending operations. This imposed memory architecture implies heavy cache thrashing since list elements are scattered throughout texture memory.

B. Linear List

This approach is based on the OpenGL Insights chapter by Knowles et al. [24] and stores elements of the same list at nearby memory slots to improve texel cache usage. For OIT the authors did not observe a performance increase with linear list buffer management compared to the basic linked list approach. However, since OITAO relies on even more buffer fetches, the benefit of texture caching is more promising. The downsides of this implementation are the expensive prefix sum computation and the fact that transparent objects have to be rendered twice to count and store the fragments.

C. Texture3D

The third implementation of OITAO relies on a three-dimensional texture representing the fragment lists for each pixel. A single geometry pass stores the fragments at the texture position corresponding to the pixel position, while a subsequent fullscreen pass is used to sort and blend the lists. Texel caching for fragments now applies to all three dimensions at the cost of maximized memory requirements.

V. Results

To test the image quality and performance for opacity-based AO two different SSAO algorithms were implemented and extended: A simple single pass approach utilizing a Poisson disk sample pattern to check pixel neighborhoods and a multiresolution approach based on the implementation by Hoang et al. [16].

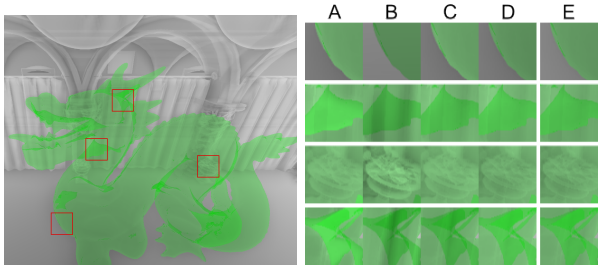

Figure 6. Image quality comparison between different compression criteria and the brute force method.

OIT and AO calculations rely on a heavy amount of texture and image operations. The difference between *image* and *sampler* data access is the method used to retrieve the texels on the GPU. Accessing *sampler* types is done using the so called *FastPath*, while *image* types are queried using the *CompletePath* adopting AMD's APP SDK terminology [25]. Enabling the use of load and store operations as well as atomic methods, the *CompletePath* is not able to take advantage of the texel cache as is the case with basic *sampler* reading calls. Since the presented approaches heavily rely on OpenGL's *image* type functionality, this loss in performance has to be considered.

The following results are obtained on an AMD Radeon 7850 GPU at default clock speeds and 1GB video memory. Ambient occlusion is always computed for the full viewport resolution of 1280×720 with 16 depth layers.

A. Opacity-Based Ambient Occlusion

For OITAO the test scene consists of the Stanford Dragon mesh with 100,000 triangles exhausting multiple depth layers and a dense triangle structure. Since the color distance to the integrated approach is negligibly low, the following tests only consider OITAO approaches implementing separated blending to allow flexible AO computations.

Applying compression to each fragment list while resolving transparency reduces quality by taking only the most important fragments into account. For medium workload scenes four nodes are usually enough to reproduce the brute-force result on a semi-transparent mesh as demonstrated in figure 8. This is due to the fact that highly transparent layers are dropped while nearly opaque layers obscure visibility of remaining layers.

However, on heavy workload many layers with potentially high shadowing factors are discarded resulting in a more visible difference as shown in figure 7. Additionally dynamic scenes may suffer from popping artifacts: Moving geometry or changing material opacity may require important layers which were present before to be discarded from the list. Therefore, the precision of the AO calculations is reduced resulting in visible inconsistencies. In such cases more coherent results are achieved by preserving the first nodes in the compressed list. The compression criterion presented

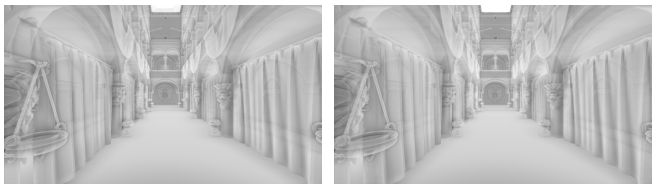

Figure 7. Results of brute force (left) and compression-based (right) OITAO on heavy workload. Most pixels are covered by transparent geometry (up to 32 layers) requiring 1084.72 ms and 107.42 ms per frame respectively.

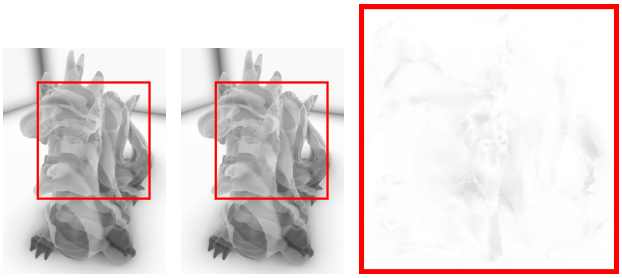

Figure 8. Image distance comparison (right) of brute force (left) and approximated (middle) OITAO on semi-transparent geometry

in section III-B is then only applied to fragments in the back of the array where AO inaccuracies are obscured by the alpha blending operations executed in the front. Figure 6 illustrates the quality improvements of applying list element preservation (column C) compared to naive compression schemes and the brute force method (column E). As this comparison shows, preserving the first half of the list can improve quality in areas covered by highly transparent surfaces close to the camera. Disregarding opacity and simply keeping the closest m layers of the list results in missing shadowing information for geometry in the back (column A). Applying the opacity- and depth-based compression criteria (column B) to the whole array results in missing occlusion information for the dragon mesh which therefore neither casts nor receives shadows.

Alternatively and at the expense of performance the uncompressed fragment list can be used to calculate and blend AO while sampling still relies on the compressed list (column D). For complex scenes this further improves quality and produces an AO map close to the brute force result while the AO passes require about 30% more time.

Generally, the performance of an SSAO algorithm heavily relies on the number of passes, the amount of samples taken and the corresponding sample kernel distance in image-space. The single pass Poisson OITAO approach uses 16 uniformly distributed sample directions with a depth-depending length. Performance of the PPLL, Texture3D, Linear List (LL) and compression-based implementations are compared to the Dual Depth Peeling (DDP) approach of Bavoil et al. [23] in table I.

As predicted the Texture3D approach is able to make best

Stage	DDP	PPLL	LL	Tex3D	Compr.
Clear	0.87	0.20		0.01	0.20
Count			1.06		
Prefix Sum			3.58		
Draw	3.65	1.68	0.71	0.60	1.67
Resolve OIT	0.46	0.81	0.51	0.50	2.14
AO	11.47	12.17	9.81	9.75	6.70
Total	16.45	14.86	15.67	10.86	10.71

Table I
POISSON DISK OITAO SINGLE FRAME PERFORMANCE IN MS

use of the texture cache. The downside is a considerably high memory consumption compared to the sparse list approaches: For example an eight layer 3D texture always requires 84.37 MB of video memory compared to the 4.63 MB dynamically allocated by the sparse list approaches for the test setup. However, the expensive fragment counting and prefix sum computations needed to maintain the Linear List structure lead to a significant performance slowdown, which is not amortized by the improved list traversal.

Since the test scene is sufficiently covered with 8 layers the DDP approach achieved competitive results. On the other hand, timing results of the *Draw* stage increase with more complex scenes which demand a lot of re-rendering, whereas the other approaches rely on a single geometry pass only.

Compressing the PPLL into a 2D texture decreases performance in the compression stage but improves AO performance since the fragment pool does not have to be queried anymore. This changes for scenes with heavy OIT workload as shown in figure 7, where the Compressed approach outperforms the brute force solutions by a minimum factor of 10, taking four nodes into consideration. To store the compressed list additional memory requirements of 24 bytes per-pixel are needed for four nodes and 16 bit depth precision.

As the results in table II show, multiresolution AO is the most taxing OITAO approach since the expensive fragment list traversal is required multiple times. The costly Linear List pre-computations are now fully amortized due to the multiple passes required to generate the AO term.

B. Opaque Ambient Occlusion

To test multilayer AO on opaque geometry the following tests are conducted on the Sibenik Cathedral mesh by Marko Dabrovic. A comparison between single layer HBAO and its A-buffer counterpart as well as the ray tracing result using Blender is given in figure 9. The direct quality improvements resulting from the additional shadows are demonstrated on the right side of the figure, where AO is computed for the first layer of depth only with respect to occluded geometry resident in the 16 layer A-buffer. This improvement is especially apparent for AO algorithms catching low-frequency occlusion detail.

Stage	DDP	PPLL	LL	Tex3D	Compr.
Clear	0.88	0.18		0.01	0.19
Count			1.21		
Prefix Sum			3.63		
Draw	3.64	1.82	0.70	0.60	1.99
Resolve OIT	0.51	0.91	0.58	0.58	2.22
Downsample	0.41	0.41	0.43	0.40	0.53
AO Pass 1	2.95	2.90	2.50	2.96	0.91
AO Pass 2	7.33	7.11	5.76	6.86	2.54
AO Pass 3	18.61	21.31	15.33	18.66	8.17
AO Pass 4	7.90	8.95	7.86	7.13	5.82
Total	42.23	43.59	38.00	37.20	22.37

Table II
MULTIRESOLUTION OITAO SINGLE FRAME PERFORMANCE IN MS

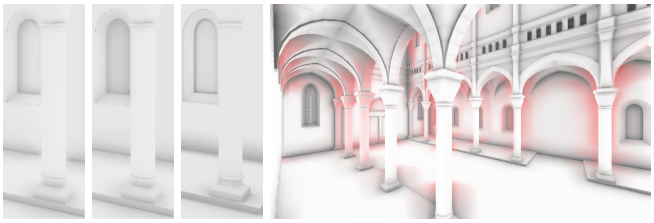

Figure 9. From left to right: Comparison between traditional HBAO, multilayer HBAO and Blender AO. Right side shows multilayer ambient occlusion improving on traditional results with additional shadowing (red).

Performance benchmarks are executed in a compute shader implementation of HBAO [26]. List compression is applied in the same compute pass where nodes are stored in shared memory instead of texture memory. The top four layers of the rendered scene are favored in the compression criterion. Since linear depth is the only necessary information for AO sampling, the fragment pool entries are more sparing compared to an OIT A-buffer.

For the HBAO approach 8 samples in X and Y direction are fetched from shared memory to determine geometric proximity. Benchmark results for Depth Peeling (8 layers), brute force PPLL traversal, list compression and default HBAO implementation which operates on the first layer of depth only, are shown in table III. Even though rendering a single frame with multilayer AO takes about ten times as long as the default approach, the improved coherency and temporal stability increase image quality significantly adding depth to dynamic scenes with more plausible shadows.

VI. DISCUSSION

A problem of Nvidia's depth peeling based approach [23] is the expensive re-rendering of geometry for every single layer which is not needed for the algorithm provided above. With the proposed approach computation is not only faster but also memory efficient, easily supporting multiple layers due to the fragment list structure which can be used to dynamically adjust texture buffer sizes on-the-fly. Therefore, the approach improves on Nvidia's multilayer algorithm in

Stage	Depth Peeling	Brute force	Compr.	Default
Clear	0.53	0.18	0.18	
Draw	4.14	1.61	1.60	0.26
AO	13.98	14.41	10.14	1.11
Total	18.65	16.20	11.92	1.37

Table III
HBAO SINGLE FRAME PERFORMANCE IN MS

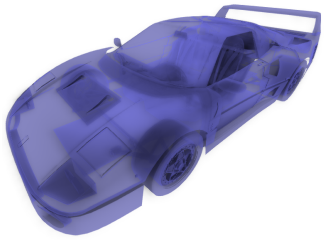

Figure 10. Multiresolution OITAO applied to a semi-transparent mesh

every way and can be considered a viable alternative for improving AO quality in screen-space.

Storing the fragment entries in a 3D texture proved to be the fastest brute force method taking advantage of texel caches at the expense of texture memory. However, for heavy workload the list compression scheme performs best while still maintaining a plausible AO effect.

Regarding transparency the presented approaches are able to produce high quality results generating and applying AO for every geometry layer stored in the fragment list structure. This of course depends on the SSAO algorithm and the number of texel fetches needed as well as the maximum number of geometry layers stored in memory. For instance MSSAO requires multiple AO passes which need a large amount of texel fetches due to the repeated fragment list traversal. Relying on list compression the tested approaches are able to improve in performance at an expense in image quality depending on the complexity of transparent geometry.

SSAO for opaque geometry also benefits from the additional depth layers. Compared to depth peeling a downside is the order of fragments in the list pool: List entries are randomly inserted into their respective lists, which theoretically requires a complete traversal to account for the most important layers. Therefore, the maximum amount of layers plays an important role and must be chosen with respect to the scene geometry.

VII. CONCLUSION AND FUTURE WORK

We presented a novel, multilayer SSAO approach to effectively compute AO on transparent surfaces while improving temporal stability and image quality for opaque geometry. To reduce computational complexity with a minor sacrifice in image quality a flexible list compression scheme is introduced. Different methods were discussed to allow

the computation of AO terms using fragment list entries as shadow receivers and emitters opposed to traditional 2D deferred buffers. Additionally opaque geometry receives occlusion from transparent objects taking the corresponding opacity into account. In contrast to the implementation of the AMD FirePro SDK (see figure 3), the car interior is now shadowed using every geometric layer of the scene enabling plausible AO as shown in figure 10. By applying the A-buffer fragment list principle to opaque geometry, the occlusion-induced problem of screen-space based algorithms emerging from a single 2D depth buffer was solved as well. This proved to be a worthwhile addition to SSAO computations, which can easily be applied to most approaches solving the AO integral in screen-space.

A possible extension of the list compression scheme could be provided by integrating additional information like the number, depth and opacity of fragments dropped by the criterion in the AO sampling process to make stochastic assumptions about proximity based shadowing. The AMD OpenGL extension *AMD_sparse_texture* providing partially resident texture allocation could be used to implement memory efficient fragment list structures. This would improve the Texture3D approach by combining good quality and performance with reduced memory requirements.

ACKNOWLEDGMENT

The work of Jan Bender was supported by the Excellence Initiative of the German Federal and State Governments and the Graduate School CE at TU Darmstadt. The research leading to these results has received funding from the European Commission's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 285026.

REFERENCES

[1] L. Carpenter, "The A-buffer, an antialiased hidden surface method," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 103–108, Jan. 1984.

[2] O. Zegdoun, "Amd firepro technology sdk," http://developer.amd.com/tools-and-sdks/graphics-development/firepro-sdk/, July 2013.

[3] H. Meshkin, "Sort-independent alpha blending," in *GDC Session*. Perpetual Entertainment, 2007.

[4] K. Myers and L. Bavoil, "Stencil routed A-Buffer," in *ACM SIGGRAPH 2007 sketches*, 2007.

[5] C. Everitt, "Interactive order-independent transparency," 2001.

[6] L. Bavoil and K. Myers, "Order independent transparency with dual depth peeling," Nvidia, Tech. Rep., 2008.

[7] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu, "Efficient depth peeling via bucket sort," in *Proc. High Performance Graphics*. ACM, 2009, pp. 51–57.

[8] M. Maule, J. Comba, R. Torchelsen, and R. Bastos, "Hybrid transparency," in *Proc. I3D*. ACM, 2013, pp. 103–118.

[9] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz, "Real-time concurrent linked list construction on the GPU," in *Proc. Eurographics conference on Rendering*, 2010, pp. 1297–1304.

[10] M. Salvi, J. Montgomery, and A. Lefohn, "Adaptive transparency," in *Proc. High Performance Graphics*. ACM, 2011, pp. 119–126.

[11] G. S. Zhukov, A. Iones, "An ambient light illumination model," in *Proc. Rendering Techniques*, 1998.

[12] M. Mittring, "Finding next gen: Cryengine 2," in *ACM SIGGRAPH 2007 courses*. ACM, 2007, pp. 97–121.

[13] P. Shanmugam and O. Arikan, "Hardware accelerated ambient occlusion techniques on GPUs," in *Proc. Interactive 3D graphics and games*. ACM, 2007, pp. 73–80.

[14] L. Bavoil, M. Sainz, and R. Dimitrov, "Image-space horizon-based ambient occlusion," in *SIGGRAPH talks*, 2008.

[15] T. Ritschel, T. Grosch, and H.-P. Seidel, "Approximating Dynamic Global Illumination in Screen Space," in *Proc. Interactive 3D Graphics and Games*, 2009.

[16] T.-D. Hoang and K.-L. Low, "Multi-resolution screen-space ambient occlusion," in *Proc. VRST*, 2010, pp. 101–102.

[17] K. Vardis, G. Papaioannou, and A. Gaitatzes, "Multi-view ambient occlusion with importance sampling," in *Proc. Interactive 3D Graphics and Games*. ACM, 2013, pp. 111–118.

[18] S. Thiedemann, N. Henrich, T. Grosch, and S. Müller, "Voxel-based global illumination," in *Proc. Interactive 3D Graphics and Games*. ACM, 2011, pp. 103–110.

[19] M. McGuire, "Ambient occlusion volumes," in *Proc. High Performance Graphics*, 2010, pp. 47–56.

[20] T. Ritschel, "Fast gpu-based visibility computation for natural illumination of volume data sets," *Eurographics (Short Papers)*, pp. 57–60, 2007.

[21] F. Hernell, P. Ljung, and A. Ynnerman, "Local ambient occlusion in direct volume rendering," *IEEE TVCG*, vol. 16, no. 4, pp. 548–559, 2010.

[22] T. Ropinski, J. Meyer-Spradow, S. Diepenbrock, J. Mensmann, and K. Hinrichs, "Interactive volume rendering with dynamic ambient occlusion and color bleeding," in *Computer Graphics Forum*, vol. 27, no. 2, 2008, pp. 567–576.

[23] L. Bavoil and M. Sainz, "Multi-layer dual-resolution screen-space ambient occlusion," in *SIGGRAPH talks*, 2009.

[24] P. Knowles, G. Leach, and F. Zambetta, "Efficient layered fragment buffer techniques," in *OpenGL Insights*. CRC Press, 2012, pp. 279–292.

[25] *AMD Accelerated Parallel Processing Programming Guide*, AMD, December 2012.

[26] L. Bavoil, "Horizon-based ambient occlusion using compute shaders," *Nvidia DirectX 11 SDK*, 2011.