applied sciences

MDPI

Article

A Corner-Highlighting Method for Ambient Occlusion

Sukjun Park [1] and Nakhoon Baek [2,3,4,*]

[1] KOG Inc., Daegu 41937, Korea; pos02232@gmail.com
[2] School of Computer Science and Engineering, Kyungpook National University, Daegu 41566, Korea
[3] Department of Data Convergence Computing, Kyungpook National University, Daegu 41566, Korea
[4] Dassomey.com Inc., Daegu 41566, Korea
* Correspondence: oceancru@gmail.com; Tel.: +82-53-950-6379

Abstract: Graphical user experiences are now ubiquitous features, and therefore widespread. Specifically, the computer graphics field and the game industry have been continually favoring the ambient occlusion post-processing method for its superb indirect light approximation and its effectiveness. Nonetheless of its canonical performance, its operation on non-occluded surfaces is often seen redundant and unfavorable. In this paper, we propose a new perspective to handle such issues by highlighting the corners where ambient occlusion is likely to occur. Potential illumination occlusions are highlighted by checking the corners of the surfaces in the screen-space. Our algorithm showed feasibility for renderers to avoid unwanted computations by achieving performance improvements of 15% to 28% acceleration, in comparison to the previous works.

Keywords: ambient occlusion; screen space; corner highlighting

Citation: Park, S.; Baek, N. A Corner-Highlighting Method for Ambient Occlusion. *Appl. Sci.* **2021**, *11*, 3276. https://doi.org/10.3390/app11073276

Academic Editor: Federico Divina

Received: 19 February 2021
Accepted: 3 April 2021
Published: 6 April 2021

1. Introduction

With the development of computer hardware and software, users meet various computer graphics user interfaces. These graphical user experiences are ubiquitous features these days [1]. In the field of computer graphics and computer games, the realistic three-dimensional graphics scenes are widespread [2].

Large portions of the realistic three-dimensional rendering come from illuminations. Deeper and richer lightings give better illusions to viewers. Due to its heavy computation, for decades the game industry has been investing heavily on efficient and scalable real-time realistic rendering. *Ambient Occlusion* (AO) [3] is one of its kind; it decently illustrates the intensity of the illumination for places or pixels, by how much they are exposed to indirect lights.

This paper introduces a simple way to enhance the performance of the *Ambient Occlusion* algorithms by discarding areas which are not needed. Occlusion mostly occurs in concentrated places where objects are close to each other. Our algorithm, *Outline-dependent Screen-Space Ambient Occlusion* (OdSSAO), detects them through rendering model object outlines derived from its vertex normal vectors.

Our algorithm is not specific to certain ambient occlusion techniques; it may go alongside any other *screen-space* ambient occlusion methods such as *Image-space Horizon-Based Ambient Occlusion* (HBAO) [4] for instance. We also tested the stand-alone version of our method which is occluding all the highlighted regions without actual ambient occlusion computation, which will be discussed in later sections.

2. Related Works

The quality of real-time realistic rendering is often denoted highly dependent on approximating the indirect lighting in the scene [5,6]. Due to its excessive amount of computation, the game industry has been heavily investing in efficient and scalable illumination methods for decades. *Ambient Occlusion* [3] is one of them: It approximates the

intensity of the illumination for surfaces by how much they are blocked from lights and their radiance. In spite of its dated implementation, recent researches about its practical usage such as *VAO++* [7], *TSSAO* [8], and *Gaze-Dependent SSAO* [9] are present due to its infamous expensiveness.

Many of the current implementations are known to be originated from the *Screen-Space Ambient Occlusion* (SSAO) technique [10]. It computes illumination intensity depending on its neighbored pixels in the screen-space. While this screen-wide computation is recommended for scenes with complex surfaces, operations on smooth and non-occluded surfaces can be considered unnecessary and redundant.

Multi-resolution rendering techniques [11–13] are also applied to these applications. Specifically, the *Multi-resolution Screen-Space Ambient Occlusion* (MSSAO) [11] computes ambient occlusion by combining occlusion values using multiple *mipmap* levels of the *G*-buffer. It generates high quality results, mainly based on the real-time *mipmap* processing. The relatively high processing cost can be one of its drawbacks.

Another multi-resolution method [12] divides the screen-space dynamically, and then, the illumination is rendered for each sub-image in an adequate resolution. This method tried to detect edge regions and focused on those regions for the ambient occlusion processing, which is similar to our method. In contrast, it performs ambient occlusion processing for multi-resolution images, and it also needs relatively-high processing power. The *Line-Sweep Ambient Obscurance* (LSAO) [13] method represents mathematical equations for the ambient occlusion processing.

This paper introduces a new way to enhance the performance of the ambient occlusion algorithms. Our algorithm, *Outline-dependent Screen-Space Ambient Occlusion* (OdSSAO), discards areas in advance where illumination is possibly not occluded. Ambient occlusion mostly occurs in concentrated places where orientations of nearby surfaces are disjoint. This also means places where surfaces are uniform, such as a huge wall, are possibly not occluded, standing as a redundant factor in total operation.

Our algorithm detects these areas by rendering the outlines of surfaces, derived by its vertex normal vectors over the original surfaces. If multiple outlines are drawn on a single pixel, our algorithm highlights the pixel as a possibly occluded point by storing it in the *stencil mask*. This mask layer is then utilized with the conventional *SSAO* algorithm [9] to process ambient occlusion only on those highlighted regions.

Our algorithm is originally aimed at highlighting the potential occlusion area, not computing the ambient occlusion itself. Thus, for comparison purposes, we implemented some of the well-known ambient occlusion algorithms on the top of ours: *Alchemy Ambient Obscurance* (AAO) [14] and the *Scalable Ambient Obscurance* (SAO) pixel sampling method [15]. In the following sections, we will represent the details of our method.

3. Our Algorithm

Our algorithm *OdSSAO* requires the basic deferred rendering pass to be performed. Algorithm 1 summarizes the rendering procedure in terms of *OpenGL* (Open Graphics Library) [16]. We first compute *world-space* vertex positions and their depths into the *G*-buffer. We next render pre-computed outline surfaces on the stencil layer. The rasterizer will draw back faces and front faces in sequence, with blending enabled. Depth testing with the previously rendered *depth buffer* should be enabled with writing disabled to avoid outline depth recording. The third step is to compute screen-space ambient occlusion with these generated buffers. We apply ambient occlusion only to the regions highlighted by the stencil layer. The final step is to perform a *bilateral blur* on the output image. Afterward comes the general deferred rendering pass such as lighting, effects and so on.

Algorithm 1: The brief summary of the rendering process in order.

step 1. render vertex positions and depth values to *G-buffer*.
enable depth test, depth mask writing, back face culling
disable blending
render to *G-buffer*

step 2. render pre-computed outline surfaces on the *stencil layer*.
disable depth mask writing
enable blending (reverse subtract mode), front face culling
step 2.1 render back face outlines to the *stencil layer*
enable blending (add mode), back face culling
step 2.2 render front face outlines to the *stencil layer*

step 3. compute screen-space ambient occlusion with the *G-buffer* and the *stencil layer*.
compute (screen-space) ambient occlusions
apply bilateral blurs
render the final scene with the ambient occlusion results

3.1. Outline-Dependent Masking

We assume ambient occlusion happens in places where surfaces, or groups of faces, are incoherently arranged. This means we also assume occlusion not to occur where surfaces are consistent and in opened surroundings. With this analogy, we aimed for an algorithm that can highlight surfaces where adjacent surfaces are close enough to occlude illumination. As a result, we brought up a simple approach to search this surface incoherency which is outlining.

We define the outline as a replica of the original surface where every vertex is extruded by its integrated normal vectors with an arbitrary extent. Figure 1 shows two-dimensional examples of the outline. Closed polygons are the model objects, or what we refer to a group of faces. The edges of those model objects are the original faces to be rendered, and the dotted lines around them are their outlines. Three-dimensional representation would be polyhedron with faces wrapping around. How to generate these outlines is explained further in Section 3.2.

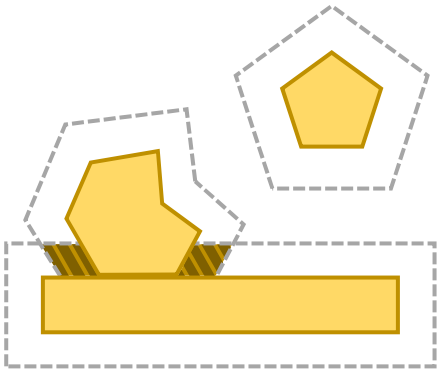


Figure 1. Two-dimensional representation of model objects, outlines (in dotted lines), and their intersections (in hatched lines).

We can easily spot the mismatching surfaces by rendering these prepared outlines. The mismatching surface, or surface incoherency, is the case where nearby surfaces have distant surface normal vector orientations. When a view ray enters more than two outlines without exiting, we consider the hit point of the ray as an outline intersection. This intersection check specifies mismatching surfaces in ease, which is what our algorithm highlights as possibly occluded areas. Figure 1 demonstrates these outline intersections. Dotted lines represent outlines and highlighted areas represent their intersections. View rays hitting the

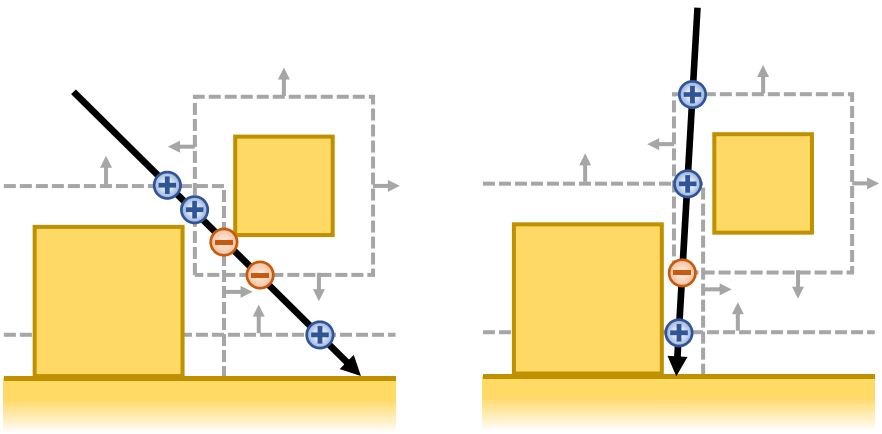
surface inside this highlighted region are likely to be occluded. As we can see in this figure, concave corners are what our algorithm is locating before computing ambient occlusion.

The actual implementation of the corner highlighting takes a few steps. We first render the base model object information to the screen-space *G*-buffer. Rendered buffers hold the world-space position, the normal vector, and the depth value of the pixel. Generating this *G*-buffer is a necessary step in deferred rendering; it provides essential data for the illumination in later processing.

Rendering outlines come next to check the intersection aforementioned. We detect the view ray entering and exiting outlines by rendering them. We consider rendering outline faces equivalent to the view ray crossing the outlines. View ray stops when it confronts the model object surface, thus crossing the outline faces behind the model objects are ignored. Accordingly, depth testing with previously obtained depth buffer is mandatory. In this step, writing to the depth buffer is disabled to avoid depth testing with outline faces.

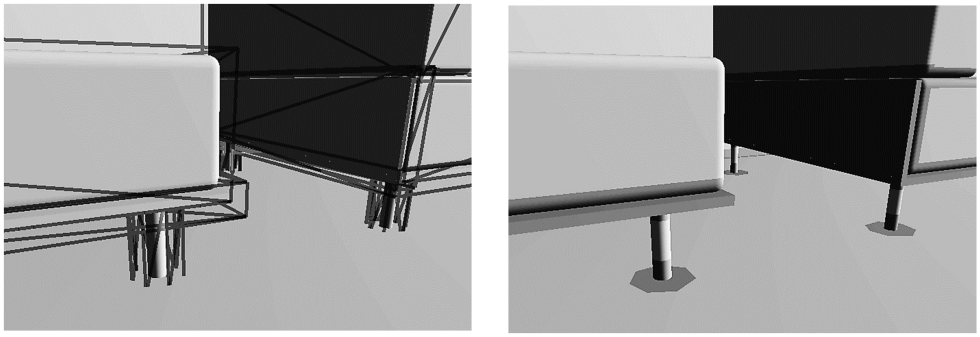
Outline intersection can be measured by counting the outline faces rendered on a pixel. Add 1 when the view ray enters the front face of the outline. Subtract 1 when enters back face vice versa. When this summation ends above 2, it implies that the pixel hit by the view ray is in the intersection region, and thus corresponding pixel can be marked as a possibly occluded area.

Figure 2 represents how corner checking is performed. We have the same scene setup with two different view rays, left and right. Filled rectangles represent original model objects and the dotted edges wrapping around are their outlines, and small arrows showing their normal vector directions. The $+$ and $-$ symbols on the figure show where view ray enters and exits the outline the model objects. Left view ray sums up to 1, showing its contact point is not inside the corner. Right view ray sums up to 2 on the other hand, meaning its point is inside the corner and is likely to be occluded. With this simple approach, we generate the stencil layers for ambient occlusion. Figure 3 shows the actual implementation results of our algorithm.



(**a**) The contact point of the ray is in the corner. (**b**) Not in the corner.

Figure 2. An example of how the corner checking is performed. The $+/-$ symbols show that the view ray is entering/exiting the outline through front/back faces.



(**a**) Rendered in the wireframe mode. (**b**) Outline intersections are highlighted.

Figure 3. Drawing the intersections. Those intersections are candidates for the ambient occlusion processing.

3.2. Generating Outlines

Our algorithm is inspired by the *model object outlining technique* [17] which is frequently found in many video games as a highlighter for certain 3D objects. This method introduces generating outlines by creating a scaled replica of the 3D objects in run time. Figure 4 visualizes the approaches we have made. Filled polygons represent model objects and thin edges represent their generated outlines with each method.

The first method is a simple scaling. It generates an outline by scaling the model object itself. While it is a simple approach, it mostly ends in generating incorrect outlines as seen in the figure on the left. The next approach is the *Normal Vector Extrusion*. For the illumination purpose, many rendering pipelines require model object files to include per-vertex normal vectors, and the Normal Vector Extrusion method benefits from this.

We extrude each vertex with its normal vector to a certain extent. This method guarantees constant outline thickness, but corners are mostly discontinuous, leading to a miscalculation on view ray enter and exit counting. This flaw can be seen on the center in the Figure 4.

(**a**) simple scaling (**b**) normal vector extrusion (**c**) integrated normal vector extrusion

Figure 4. Different outline generation methods. The integrated normal vector extrusion shows the best results.

To avoid this issue, we opted for our last approach: *Integrated Normal Vector Extrusion*. Each vertex in the model object is shared among multiple faces, and faces have different normal vector orientations. As shown in Algorithm 2, for each vertex, we generate a new normal vector by integrating (or averaging) all the unique normal vectors of the faces that the vertex takes place. Note that similar face normal vectors are treated as a duplicate, and one among them should be valid for integrating. Since this operation is much to handle in real-time, we pre-compute the model object data and save them as a pre-processed data. The last image in Figure 4 depicts the implementation result of Algorithm 2. Unlike previous methods, you can observe an accurate outline for the model object with consistent thickness and continuous corners.

Algorithm 2: Pseudo-code to generate integrated normal vectors for each vertex.

```
step 1. reset vertex normal vectors.
    for each vertex;
        reset that vertex's candidate set to be empty.

step 2. build up candidate normal vectors.
    for each face;
        calculate the face normal vector n⃗.
        foreach vertex in that face;
            if n⃗ is not included in the vertex's candidate set,
                add n⃗ to the vertex's candidate set.

step 3. calculate the integrated normal vectors.
    for each vertex;
        calculate the average vector of the vertex's candidate set.
        set that average vector as the integrated normal vector.
```

In spite of the preparation, our outline method does not always guarantee a perfect result. There are two major causes to be treated with caution. The first is when the model object is *not* in the form of a closed polyhedron. Closed polyhedron refers to the model object with no opened faces or corners on the surface. In other words, the back face of the model object should not be visible in the camera view from any possible directions. This can be avoided by either filling the gaps (closing the face) or generating custom outlines solely for the troubling ones.

The latter case is about scaling the concave corners. Our outlining mechanism cannot spot concave corners by default. While corners in between two different model objects can be distinguished, the local corner information of a single model object is lost due to integrating vertex normal vectors. The best solution to address this issue is to convexify the outline model object and bake the local ambient occlusion of the model object into its texture, so there is no need for real-time checking the local static concave corners.

Another solution is to avoid concave vertex normal vectors. Vertices taking the concave part of the local corner can be excluded from computing the integrated normal vector. However, this can easily go wrong if not properly treated; even if all the integrated normal vectors are correctly assigned, gaps will start to appear when the extrusion is longer than the nearby faces. Such case will lessen the accuracy of the corner highlighting, resulting in a non-optimal ambient occlusion evaluation.

4. Results

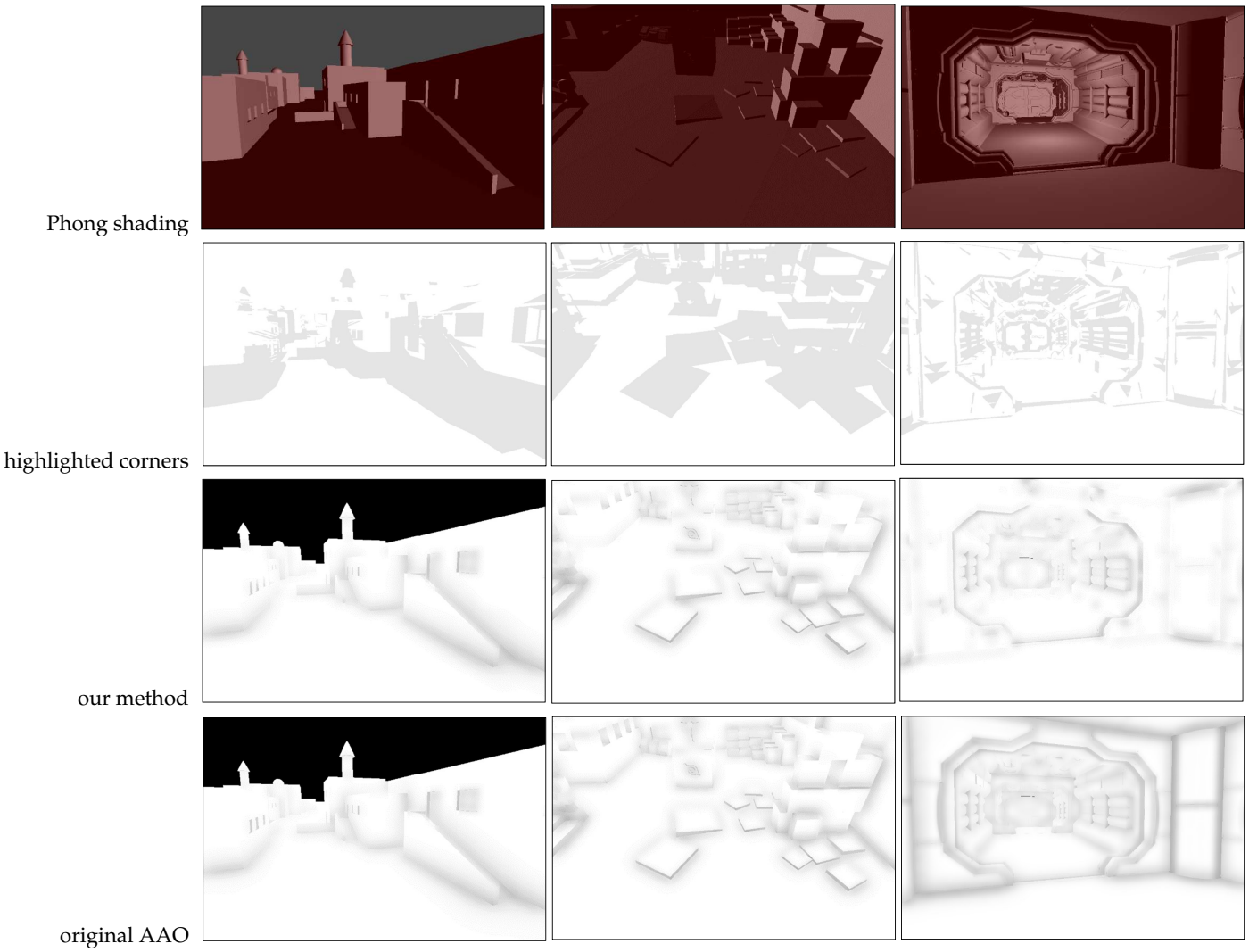
To test our algorithm, we rendered several scenes with the ambient occlusion post-processing pass on a custom C++ OpenGL engine. Any type of screen-space derived ambient occlusion variant is compatible with our algorithm. Among many promising ambient occlusion methods, we chose *AAO* (Alchemy Ambient Occlusion), a widely used screen-space ambient occlusion method in many video game rendering engines. The *screen-space ambient occlusions* with and without our highlighting method are both implemented to be compared. Our target was to achieve identical rendering results with the original pipeline but with improved rendering performance. Tests are done in Windows 10 PC with NVIDIA GeForce GTX 1070 graphics card and Intel Core i7-6700K 4.00 GHz CPU.

We report the results of rendering three different scenes:

- *Small Desert City*, outdoor alleyways with moderate occluded corners.
- *Cartoon Temple Building*, inside of a temple filled with rectangular blocks.
- *Corridor*, science-fiction themed corridor where concave vertices are not properly handled.

Figure 5 shows the rendering results in the top-to-bottom order. For comparison, we excluded shared operations such as *bilateral blur* and *lighting*. We sampled 4 nearby pixels per pixel for computing ambient occlusion without down-sampling. Each scene is

rendered in static for 10 s with the screen resolution of 1080 × 720. Table 1 summarizes our performance results of rendering scenes shown in Figure 5.



(**a**) Small Desert City (**b**) Cartoon Temple (**c**) Corridor

Figure 5. Comparison of ambient occlusion results in three distinct scenes.

Table 1. Performance comparison of the results.

		Prev. AAO	Our Method
Small Desert City	total triangles	536,326 tri	1,072,672 tri
	avg. rendering time	1282 µs	782 µs
	worst proc. time	1309 µs	1086 µs
Cartoon Temple	total triangles	17,678 tri	35,356 tri
	avg. rendering time	1098 µs	926 µs
	worst proc. time	1274 µs	1237 µs
Corridor	total triangles	291,517 tri	583,034 tri
	avg. rendering time	1308 µs	631 µs
	worst proc. time	1322 µs	923 µs

The first row depicts the basic Phong shading rendering results to show the composition of the scene. The second row depicts the *highlighted corners* generated by our algorithm for each scene. Areas highlighted in grey imply that they have a high potential

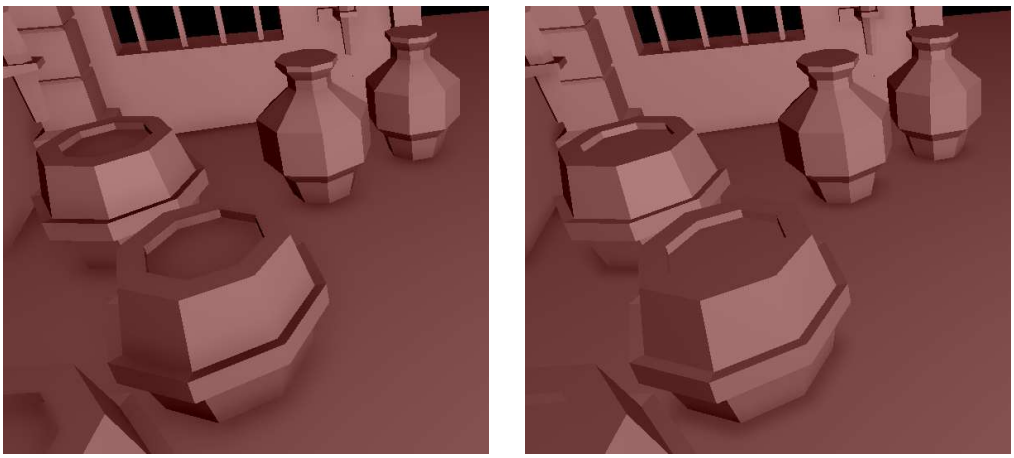
to be occluded. The third row shows the results of our method, which operated only on the highlighted areas from the previous row. The last row shows the results of the *original AO* implementation. Bilateral blur was active for generating these images but was omitted when measuring the performance.

For the scenes of *Small Desert City* and *Cartoon Temple*, we can observe that corners are properly highlighted. These two scenes are composed of multiple flat walls, leaving only a portion of the place to be occluded. While the result ambient occlusion image generated with and without our algorithm is nearly identical, the one with our algorithm was 15% faster.

Although of these positive results, there are drawbacks to our method as mentioned previously. The accuracy of our algorithm greatly lessens when concave vertices in the scene are not properly prepared in advance. Room model objects, for instance, are generally faced inwards, which eventually leads to generating several concave corners. Such case should be handled differently when integrating its normal vectors. The result of the last scene Corridor pinpoints this problem. Unlike the previous two, this scene was not prepared in advance and uniformly computed integrated normal vectors for all model objects, including the room. The result shows significant accuracy losses; corners are not properly tracked and mostly washed out as shown in the highlighted stencil layer figure.

5. Stand-Alone Implementation

Besides using our algorithm as a medium step for rendering ambient occlusion, we also tested out the stand-alone version of our algorithm. Our approach this time is using the highlighted corner stencil buffer as an ambient occlusion algorithm as-is. This lets us generate good enough ambient occlusion effects without implementing actual ambient occlusion operations. Figure 6 shows the attempt we made.



(**a**) The previous AAO method. (**b**) Our new implementation.

Figure 6. Comparison between AO and stand-alone implementation of our algorithm.

The left image is rendered with AAO, and the right image is rendered with the stand-alone implementation of our algorithm. The original method rendered the scene, computed ambient occlusion on a full screen-space wide with 4 sample rate, and then executed the bilateral blur. Our method rendered the scene, rendered corner highlighting stencil layer (rendering outline model objects), and then executed the bilateral blur.

Although it does not show identical results, it gives suitable ambient occlusion visuals that are not far off from the original. With this semi-decent precision, we were able to top the performance over the original by 28%: Default AAO implementation showed the average rendering speed of 1785 µs, while our stand-alone implementation showed 1388 µs. On the whole, it showed a decent trade-off between occlusion accuracy and rendering performance.

6. Discussion

Our algorithm stands strong when there are not many corners to be occluded in the scene. The fundamental of the corner highlighting method is to omit unnecessary ambient occlusion computation on not-occluded surfaces. The more spacious and simple the scene is, the better the performance will result. Two test scenes *Small Desert City* and *Cartoon Temple* highly meet this requirement. Their walls and floors are sufficiently flat and objects are sparsely placed, allowing a minimal chance for ambient occlusion to happen.

On the other hand, our algorithm stands weak on rendering complex scenes where the majority of the place needs occlusion check. Our method intends to check possibly occluded areas. If all areas are affected by the occlusion, our algorithm will highlight the entire screen, which leads to no more than a rendering waste. If the goal of the rendering pipeline is, as an example, to support high-quality first-person shooting games, this is not the algorithm to have an interest in; such scene setups are recommended to run ambient occlusion *without* our algorithm.

Another downside of our algorithm is that it requires a considerable amount of time and work to generate model object outlines *prior to the real-time operation*. Not only preparing *integrated normal vectors* for the outlines, but the developer also has to be aware of its local concave corners and its complexity. For instance, generating an outline for a room model object is often treated apart from the regular ones since its faces are towards inside.

Improper outline generation may cause false region highlighting, *over-extruded* outline faces, excessive outline mesh rendering, and so on. Our algorithm is based on a trade-off between ambient occlusion computing and meshes rasterizing; we focus on less ambient occlusion computation in the cost of more mesh rasterization. Outline model objects are also recommended to be simplified.

7. Conclusions

As the results imply, our *Outline-dependent Screen-Space Ambient Occlusion* method shows the performance boost on its kind. By highlighting the potential corners, we were able to discard unwanted computations. We also have shown the potential of our algorithm by the stand-alone implementation.

On top of an intermediate step to the full ambient occlusion algorithm, we also tested out using the corner highlighting method itself as an ambient occlusion method. As a result, we found an efficient way to render ambient occlusion in a reasonable quality without its actual implementation; we were able to greatly enhance the rendering performance with the small cost of accuracy.

Conclusively, we present *Outline-dependent Screen-Space Ambient Occlusion* (OdSSAO), a new approach to handling the indirect lighting in real-time. We expect our method can help to reduce the performance overhead of some occlusion algorithms and their kind. We also anticipate that our algorithm can make an adequate exploit on the real-time rendering fields, especially for performance-weighted low-end devices. More optimizations will be explored in the near future.

Author Contributions: Software, writing—original draft presentation, visualization, S.P.; Conceptualization, methodology, investigation, writing—original draft presentation, writing—review and editing, supervision, funding acquisition, N.B. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (Grand No. NRF-2019R1I1A3A01061310).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Baek, N.; Kim, K. Design and implementation of OpenGL SC 2.0 rendering pipeline. *Clust. Comput.* **2019**, *22*, 931–936. [CrossRef]
2. Baek, N. An emulation scheme for OpenGL SC 2.0 over OpenGL. *J. Supercomput.* **2020**, *76*, 7951–7960. [CrossRef]
3. Bunnell, M. Dynamic Ambient Occlusion and Indirect Lighting. IIn *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*; Pharr, M., Fernando, R., Eds.; Addison-Wesley Professional: Boston, MA, USA, 2005.
4. Bavoil, L.; Sainz, M.; Dimitrov, R. Image-Space Horizon-Based Ambient Occlusion. In Proceedings of the ACM SIGGRAPH 2008 Talks, SIGGRAPH'08, Los Angeles, CA, USA, 28 July–1 August 2008; Association for Computing Machinery: New York, NY, USA, 2008; p. 22. [CrossRef]
5. Baek, N.; Yoo, K. Emulating OpenGL ES 2.0 over the desktop OpenGL. *Clust. Comput.* **2015**, *18*, 165–175. [CrossRef]
6. Baek, N.; Kim, K.J. An artifact detection scheme with CUDA-based image operations. *Clust. Comput.* **2017**, *20*, 749–755. [CrossRef]
7. Bokšanský, J.; Pospíšil, A.; Bittner, J. VAO++: Practical Volumetric Ambient Occlusion for Games. In *Eurographics Symposium on Rendering: Experimental Ideas & Implementations*; EGSR'17; Eurographics Association: Goslar, Germany, 2017; pp. 31–39. [CrossRef]
8. Mattausch, O.; Scherzer, D.; Wimmer, M. Temporal Screen-Space Ambient Occlusion. In *GPU Pro 2*, 1st ed.; Engel, W., Ed.; A. K. Peters, Ltd.: Natick, MA, USA, 2011.
9. Mantiuk, R. Gaze-Dependent Screen Space Ambient Occlusion. In *Computer Vision and Graphics*; Chmielewski, L.J., Kozera, R., Orłowski, A., Wojciechowski, K., Bruckstein, A.M., Petkov, N., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 16–27.
10. Bavoil, L.; Sainz, M. Screen Space Ambient Occlusion. In *NVIDIA Developer Download*; NVIDA: Santa Clara, CA, USA, 2008.
11. Hoang, T.D.; Low, K.L. Multi-Resolution Screen-Space Ambient Occlusion. In Proceedings of the 17th ACM Symposium on Virtual Reality Software and Technology, VRST'10, Hong Kong, China, 22–24 November 2010; Association for Computing Machinery: New York, NY, USA, 2010; pp. 101–102. [CrossRef]
12. Besenthal, S.; Maisch, S.; Ropinski, T. Multi-Resolution Rendering for Computationally Expensive Lighting Effects. *J. WSCG* **2019**, *27*. [CrossRef]
13. Timonen, V. Line-Sweep Ambient Obscurance. *Comput. Graph. Forum* **2013**, *32*. [CrossRef]
14. McGuire, M.; Osman, B.; Bukowski, M.; Hennessy, P. The Alchemy Screen-Space Ambient Obscurance Algorithm. In Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, HPG'11, Vancouver, BC, Canada, 28 July 2011; Association for Computing Machinery: New York, NY, USA, 2011; pp. 25–32. [CrossRef]
15. McGuire, M.; Mara, M.; Luebke, D. Scalable Ambient Obscurance. In Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics, EGGH-HPG'12, Paris, France, 25–27 June 2012; Eurographics Association: Goslar, Germany, 2012; pp. 97–103.
16. Segal, M.; Akeley, K. *The OpenGL Graphics System: A Specification*; Khronos Group: Beaverton, WA, USA, 2019.
17. Valenta, D. Pixel-Perfect Outline Shaders for Unity. 2018. Available online: www.videopoetics.com/tutorials/pixel-perfect-outline-shaders-unity/ (accessed on 10 March 2021).