Fast Approximate Visibility on the GPU using pre-computed 4D Visibility Fields

Athanasios Gaitatzes
University of Cyprus
75 Kallipoleos St.
P.O.Box.20537
Cyprus (CY-1678),
Nicosia
gaitat@yahoo.com

Anthousis Andreadis
Athens University of
Economics & Business
76 Patission St.
Greece (10434),
Athens
anthousis@gmail.com

Georgios Papaioannou
Athens University of
Economics & Business
76 Patission St.
Greece (10434),
Athens
gepap@aueb.gr

Yiorgos Chrysanthou
University of Cyprus
75 Kallipoleos St.
P.O.Box.20537
Cyprus (CY-1678),
Nicosia
yiorgos@cs.ucy.ac.cy

ABSTRACT

We present a novel GPU-based method for accelerating the visibility function computation of the lighting equation in dynamic scenes composed of rigid objects. The method pre-computes, for each object in the scene, the visibility and normal information, as seen from the environment, onto the bounding sphere surrounding the object and encodes it into maps. The visibility function is encoded by a four-dimensional *visibility field* that describes the distance of the object in each direction for all positional samples on a sphere around the object. In addition, the normal vectors of each object are computed and stored in corresponding *fields* for the same positional samples for use in the computation of reflection in ray-tracing. Thus we are able to speed up the calculation of most algorithms that trace rays to real-time frame rates. The pre-computation time of our method is relatively small. The space requirements amount to 1 byte per ray direction for the computation of ambient occlusion and soft shadows and 4 bytes per ray direction for the computation of reflection in ray-tracing. We present the acceleration results of our method and show its application to two different intersection intensive domains, ambient occlusion computation and stochastic ray tracing on the GPU.

Keywords

indirect lighting, pre-computed visibility, uniform distribution, hemisphere, tracing rays.

1. INTRODUCTION

The acceleration of the computation of the lighting equation in real-time on the GPU and especially the visibility term, one of the most intensive parts of the computation, is still a very active field of research. Ambient occlusion computation and real-time ray tracing are just two of the fields where the fast computation of the visibility queries is very important.

Ambient occlusion is defined as the attenuation of ambient light due to the occlusion of nearby geometry. It gives perceptual clues of depth, curvature, and spatial proximity and thus is important for realistic rendering. It is a technique that approximates the effect of indirect global illumination without trying to simulate the interplay of incident and reflected light.

Ray tracing is a general and versatile algorithm that performs image synthesis by shooting rays through each pixel, finding the closest intersection with the scene geometric entities. The generic backwards ray tracing algorithm is capable of capturing both local illumination and basic indirect specular effects such as mirror-like reflections and refraction.

In this paper we improve and expand the method proposed by Gaitatzes et al. [Gai08] by moving the implementation to the GPU, taking advantage of the shader units parallelism and demonstrating significant performance gains. While the core of the visibility queries mechanism remains the same, the paper shows how the method is adapted to both interoperate with a generic ray tracing system and accelerate the generation of high quality ambient occlusion. First, at pre-processing time, we construct the visibility field (Figure 1). It stores the intersection distances of a hemisphere of rays originating from sample points on the bounding sphere of an object

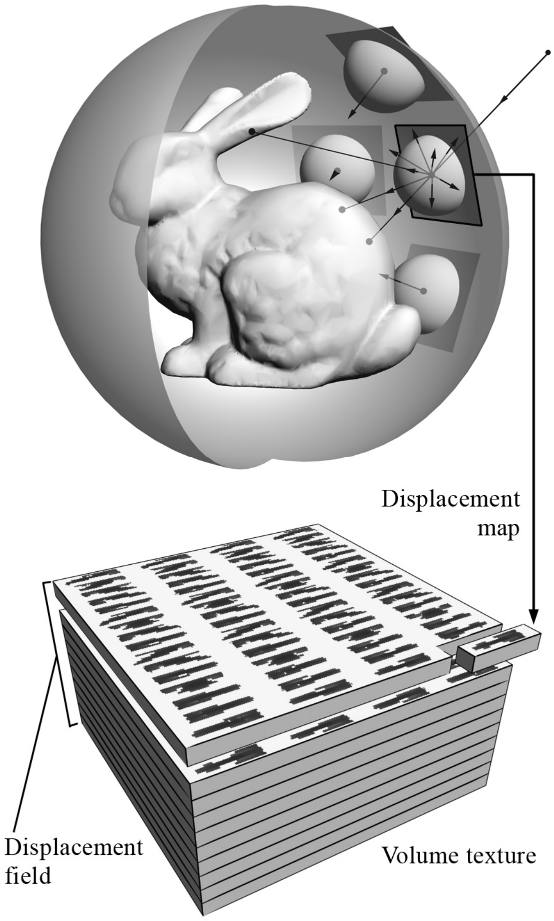


Figure 1: A hemisphere of rays emanating from the bounding sphere towards the object is pre-computed for a large number of sample points on the sphere. Bottom: Volume texture of the visibility field. Row by row each map is placed into a slice of the volume texture thus minimizing the volume space requirements. As a result a 512^3 volume will hold four 256^2 maps per slice.

and directed towards the model itself. We construct one map for each sample point (see Section 3.1). After the construction of the visibility field maps, we compactly fit them in one volume texture (see Section 4.1) for easy access on the GPU. In addition, all mesh information (i.e. coordinates, normals and materials) are stored in maps and passed on to the GPU. Then, at run time, when a ray from the environment towards an object (or vise versa) intersects its bounding sphere, we perform a simple ray-sphere intersection test and recover from the pre-computed maps the rest of the ray distance for the ray-object intersection test.

The advantage of the method described in Gaitatzes et al. [Gai08] is that the bulk of the computation is moved to a pre-processing stage. The results are stored in compact gray-scale textures; 1 byte per ray direction for the computation of ambient occlusion and soft shadows and 4 bytes per ray direction for the computation of reflection in ray-tracing, providing for each object a constant size of additional information, independent of the complexity of the original model. Then the real-time algorithm performs a simple intersection test with the bounding sphere of the object and a constant-time map lookup (see Section 3.2).

For dynamic scenes with rigidly moving objects, visibility fields accelerate the computation of the approximation of the indirect lighting term of the rendering equation to real-time frame rates as well as the computation of soft shadows and reflection in ray-tracing. The performance of this approach does not depend on the polygon count to a large extent; instead, it is directly related to the number of visible pixels shaded by the GPU. This is a significant advantage over existing approaches. In addition, our acceleration structure is flat by nature and thus more suited to the GPU architecture.

In Section 2 we give an overview of the previous work, followed by a description of our method in greater detail in Section 3. In Section 4 we discuss the GPU implementation and in Section 5 our results from the application of the visibility fields method in ray tracing and especially the benefit of shadow rays and secondary rays as well as secondary diffuse illumination (termed ambient occlusion).

2. BACKGROUND AND PREVIOUS WORK

We distinguish the previous work in two areas that both share the computation of the visibility function; the acceleration of the computation of ambient occlusion on the GPU and the acceleration of stochastic ray tracing algorithms on the GPU. Note that we apply our method only to a GPU-based ray tracing algorithm in order to compare timings with the fastest approach.

2.1 Ambient Occlusion on the GPU

In ambient occlusion the indirect component can be computed as:

$$A(\mathbf{x},\vec{\mathbf{n}}) = \frac{1}{\pi}\int_{\Omega} V(\mathbf{x},\vec{\omega}_o)\lfloor \vec{\omega}_o \cdot \vec{\mathbf{n}} \rfloor d\vec{\omega}_o$$

Where $V(\mathbf{x},\vec{\omega}_o)$ is an empirical function that maps distance from surface point x to the closest surface along direction $\vec{\omega}_o$ to visibility values between 0 (no occlusion) and 1 (full occlusion).

By tracing rays outward from a given surface point x over the hemisphere around the normal $\vec{\mathbf{n}}$, ambient occlusion measures the amount that a point is obscured from light. This average occlusion factor is used to simulate soft-shadowing.

Ambient occlusion (AO) computation on the GPU was first used by Bunnell [Bun05], who approximates the AO by modelling the receiver surface as disk-based occluders and evaluates the ambient occlusion caused by the disks using an analytic method. He uses a heuristic method to combine the shadows cast from multiple disks into a noise free image but requires high tessellation of scene geometry and a big pre-computation step.

Shanmugam et al. [Sha07] compute ambient occlusion as a post-processing pass based on a depth buffer from the eye's point of view. They split the AO computation into two phases, one for high frequency near detail, and another phase for low frequency detail with a wider search. The second phase allows large objects to inter-occlude as they pass next to each other. Their approach requires no scene-dependent pre-computations. On the downside, over occlusion artefacts might show up when multiple neighbouring spheres contribute occlusion to the same pixel.

Mittring [Mit07] does a full screen post-processing pass where z-buffer data is sampled around each pixel and an AO value is computed based on depth differences. Sampling occurs randomly in a sphere around each pixel, and AO is proportional to the number of sampled occluders. Like other screen space techniques, such as [Bav09], this view-dependent approach is fast, requires minimal or no pre-calculation, but cannot model AO correctly, because depth discontinuities, such as object edges and buffer boundaries, produce popping effects.

2.2 Real-time Ray Tracing on the GPU

Most GPU ray-tracing methods accelerate already established mechanisms for limiting the number of intersection tests. On the other hand, our approach provides an alternative and fast ray-surface intersection test, while it can certainly take advantage of the mentioned methods, to further improve final performance.

Carr et al. [Car02], Purcell et al. [Pur02], [Pur04], Karlsson et al. [Kar04] and Christen et al. [Chr05] implemented a streaming ray-triangle kernel on the GPU, fed by buckets of coherent rays and proximate geometry organized by a CPU process. However, there was a frequent communication of results from the GPU to the CPU over a narrow bus, negating much of the performance gained from the GPU kernel. Streaming geometry to the GPU became quickly the bottleneck.

To improve the performance of the GPU ray tracing, different acceleration structures have been widely adopted, such as the incorporation of kd-trees by Havran [Hav00] and Ernst et al. [Ern04]. However, these approaches had limited performance; by far not reaching the frame rates of the CPU based ray tracers. The main problem was the limited GPU architecture. Only small kernels without branching were supported. In addition a stack was usually required, which was poorly supported on GPUs. Foley et al. [Fol05] presented two implementations of a stack-less kd-tree traversal algorithm for the GPU, namely kd-restart by Kaplan [Kap85] and kd-backtrack. Foley showed, that on graphics hardware, there are scenes for which a kd-tree yields far better performance than a uniform grid. Although better suited for the GPU, the high number of redundant traversal steps led to relative low performance.

Besides grids and kd-trees there are also several other approaches that use a BVH as an acceleration structure on the GPU. Carr et al. [Car06] implemented a limited ray tracer on the GPU that was based on geometry images but it required careful parameterization of the geometry. It could only support a single triangle mesh without sharp edges. The acceleration structure was a predefined bounding volume hierarchy which could not adapt to the topology of the object. To alleviate the need for a stack Thrane et al. [Thr05] presented stack-less traversal algorithms for a BVH. They conclude that on the GPU, the bounding volume hierarchy traversal method is up to 9 times faster than that of a uniform grid and a kd-tree. Also, the technique proves the simplest to implement and the most memory efficient.

Horn et al. [Hor07] reduced the number of redundant traversal steps of kd-restart by adding a short stack. With their implementation on modern GPU hardware they achieved a high performance of 15–18M rays/s for moderately complex scenes. At the same time, Popov et al. [Pop07] presented a parallel, stack-less kd-tree traversal algorithm without the redundant traversal steps of kd-restart but with a poor GPU utilization of below 33%. With over 16M rays/s, their GPU ray tracer achieved similar performance as CPU based ray tracers. However, both GPU ray tracing implementations demonstrated only medium-sized, static scenes. Günther et al. [Gün07] presented a BVH based GPU ray tracing method for large models achieving close to real time rates using hard shadows.

3. APPROXIMATE VISIBILITY COMPUTATION

The computation of exact visibility is a time consuming task even for the new GPU architectures. We briefly describe here the visibility field acceleration method that follows that of Gaitatzes et al. [Gai08] but emphasizing the GPU architecture.

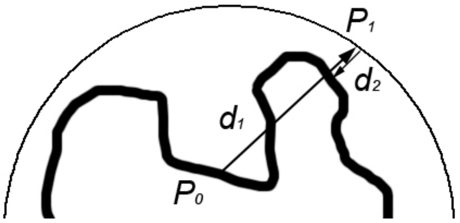


Figure 2: Visibility computation for intra-object occlusion.

3.1 Visibility Field Computation

The main idea of encoding visibility fields into maps is as follows. Consider a rigid object possibly moving through a scene. At a pre-processing step, from a discrete set of sample points on the objects bounding sphere, described as spherical coordinates (u, v), a hemisphere of rays is cast around the inward normal direction (Figure 1). For each ray (u, v, θ, φ), the closest distance between the bounding volume and the model surface is found and recorded as a compact integer value after being normalized by twice the sphere radius. Thus, for each sample point (u, v) a visibility gray-scale map is obtained that represents the distance travelled along the ray in the direction (θ, φ) before hitting the model surface. We define the *visibility field* of the object to be the collection of all visibility maps generated from all sample points on the bounding sphere of the object.

3.2 Visibility Field Indexing

During the real-time part of the execution an incident ray to the object intersects its bounding sphere and the distance between the ray origin and the intersection point is recorded. The intersection point **q** is transformed into the object coordinate system: $\mathbf{q}' = \mathbf{M}^{-1} \cdot \mathbf{q}$, where **M** is the transformation matrix with respect to the reference frame of the ray. We need to acquire the closest point (u, v) on the sphere for which we have a visibility map and therefore the index of the corresponding visibility map. In addition we need to transform the corresponding (θ, φ) of the incident ray into a visibility map cell coordinates. The indexing is performed following the methodology proposed in Gaitatzes et al. [Gai08]. We can now index into the *visibility field* for the given ray (u, v, θ, φ) and extract the distance information which is then added to the intersection distance above and this is our approximated distance value of the ray origin from the object's surface.

```
1: for all emanating rays do
2:    if ray intersects bounding sphere of occluder object
3:       discretize intersection point (u, v)
4:       discretize ray (φ, θ)
5:       access distance in visibility field volume
6:    end
7:    use distance for occlusion approximation
8: end
9: compute occlusion at pixel x
```

Algorithm 1: Pseudo code of shader algorithm for AO rendering, using visibility fields.

A special case arises when the rays originate from the object being queried for visibility. As we can see in Figure 2, when a ray originates on the object at point $\mathbf{p}_0$, the distance d_1 in direction $\overrightarrow{p_0p_1}$ is computed and compared to distance d_2 in direction $\overrightarrow{p_1p_0}$ which is extracted from the visibility map at point $\mathbf{p}_1$. If d_1 is greater than d_2 then point $\mathbf{p}_0$ is occluded.

4. Visibility Fields on the GPU

4.1 Ambient Occlusion

Directional ray samples on a reference hemisphere aligned with the z-axis are pre-computed and stored in a texture for passing to the GPU. In the fragment shader (Algorithm 1), the pre-computed ray directions are transformed according to the local normal vector and intersected with the bounding sphere of each occluder. We are able to handle both rays originating outside and inside the bounding sphere for inter-object and intra-object occlusion respectively. The only difference in the computation is the respective step to compute the final ray-object intersection distance at line 7 of Algorithm 1.

The indexing of the visibility fields is executed entirely on the GPU as is the Monte Carlo ray casting to evaluate the resulting ambient occlusion. The visibility maps are compacted and stored into a single 3D texture as slices, as shown in Figure 1. As the number of positional samples (i.e. visibility maps) can exceed the maximum volume texture dimension supported by the hardware, we compact as many visibility maps on each 2D slice of the volume as the texture hardware permits.

4.2 Ray tracing

For our proof-of-concept case study, we wanted to further improve ray-tracing timings of an already fast ray tracer. We used the method of Amit Ben-David et al. [Ami07] that implemented both a CPU and a fast GPU ray tracer by exploiting a BVH acceleration structure that has been proven to work better in some cases [Gün07] and is better suited for dynamic scenes. We did not replace the primary ray intersection tests because the regularity of the ray distribution emphasized the sampling pattern on the bounding sphere. Furthermore GPU rasterization provides better timings for the primary rays pass. In conjunction with the fact that for complex (and therefore time consuming) scenes with elaborate materials, most time is spend on secondary rays, we applied the *visibility fields* method only to secondary rays, including shadow rays. To capture the intricate reflection effects of non-perfect reflection surfaces and to highlight the advantage of our method when intersection tests increase significantly, we extended the implementation to stochastic ray-tracing.

As in the case of the ambient occlusion computation, the rays are stored in a 2D map but this time are re-computed for each running pass. For the ray-object intersection the visibility maps are used in a fragment shader on the GPU (similar to Section 4.1) along with the additional pre-computed maps of normals. The generated fragments correspond to intersection test results and the fragment shader returns the intersection point and distance to the actual surface as extracted from the visibility field. These results are used for shading or for spawning secondary rays for the next ray-tracing iteration.

Visibility field directional samples

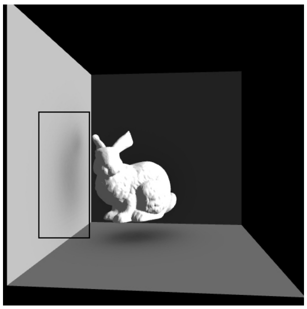
		32 x 32	64 x 64	128 x 128
Visibility field positional samples	1090			
		81.2 ms, RMS 0.59578	82.7 ms, RMS 0.58886	82.9 ms, RMS 0.58606
	4226			
		83.2 ms, RMS 0.45392	83.3 ms, RMS 0.42404	
	16642			
		84.2 ms, RMS 0.42054		

Figure 3: Inter-object AO of a bunny model using the visibility fields method with 256 rays per pixel implemented on the GPU. We report the draw time and the RMS error. On the bottom right the reference image rendered on the GPU using 256 rays per pixel in 7126 ms. The model itself is rendered using fixed-pipeline direct rendering.

5. Tests and Results

We implemented the real-time part of the above algorithm using the OpenGL® Shading Language [Kes06] on a 32bit Intel Core 2 Quad Q6600 at 2.4 GHz CPU and 4GB of main memory equipped with a GeForce 8800 GTS GPU with 512MB of texture memory. The window size was set to 512x512 for a total of 262144 pixels.

5.1 Ambient Occlusion

For most of the test runs the active pixels were about 200000 as only 75% of the window was rendered (the rest being black).

To acquire a reference image against which to compare our acceleration method in speed but mainly in image quality, we implemented ambient occlusion on the GPU using the uniform grid acceleration structure (see Figure 3 bottom-right).

We observe (in Figure 3) that the RMS error of the images compared to the reference image of the bunny, is very low and the achievable draw time, even for large models, is real-time. Based on the RMS error using 4226 64x64, visibility maps gives the same results as using maps of size 16642 32x32. We also infer from Figure 4 that the draw time is unaffected by the number of maps used thus the space required for the *visibility maps* depends only on the image quality that we would like to achieve.

In Figure 5 the visibility fields were used for the generation of intra-object occlusion but because the ray sphere intersection algorithm always succeeds at finding an intersection (worst case since we are inside the bounding sphere of the object) the rendering times are up to 4 times slower than the

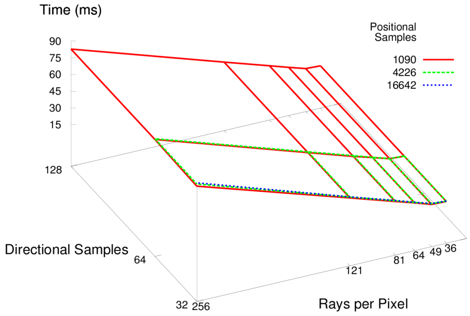
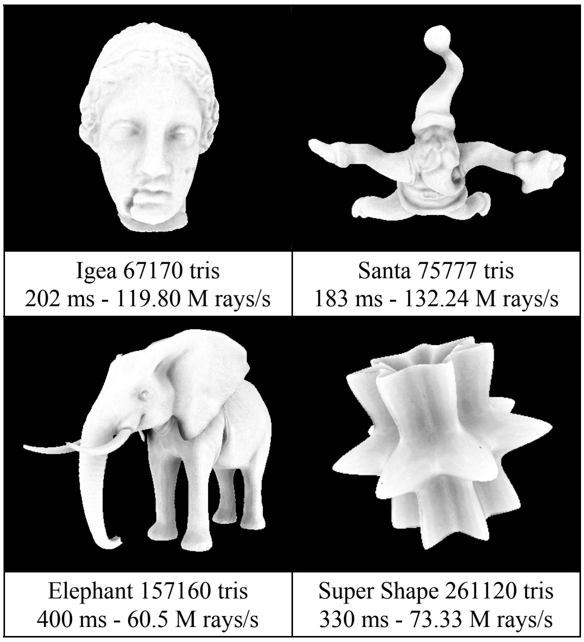


Figure 4: The draw time of the bunny model (39000 tris) plotted against different rays/pixel versus the size of the visibility maps.



Igea 67170 tris
202 ms - 119.80 M rays/s

Santa 75777 tris
183 ms - 132.24 M rays/s

Elephant 157160 tris
400 ms - 60.5 M rays/s

Super Shape 261120 tris
330 ms - 73.33 M rays/s

Figure 5: Intra object ambient occlusion rendered on the GPU using 16642 64x64 visibility maps requiring 65 MB of space and 121 rays per pixel.

inter-object occlusion case. Still the performance rate is above the one reported by Horn et al. [Hor07]. We also observe that more visibility maps are required in this case in order to render a believable image. We attribute this to the fact that multiple rays, with small angular differentiation, originating on close points on the object, hit the same sample point on the objects bounding sphere. Thus the same visibility map is used and the occlusion result looks grainy. When more maps are used the problem is alleviated.

In Figure 6 we show the Sponza Atrium rendered with several large polygon models inside it. The resulting draw time is contributed to the rendering method that uses one pass for each caster model. Just before each caster model is drawn, we enable subtractive blending (with OpenGL blend equation GL_FUNC_REVERSE_SUBTRACT), in effect, removing colour from the image. The poor draw time is also attributed to the fact that non-visible pixels (the Sponza Atrium has a lot of non-visible geometry) are not culled before the fragment shader is executed on the GPU.

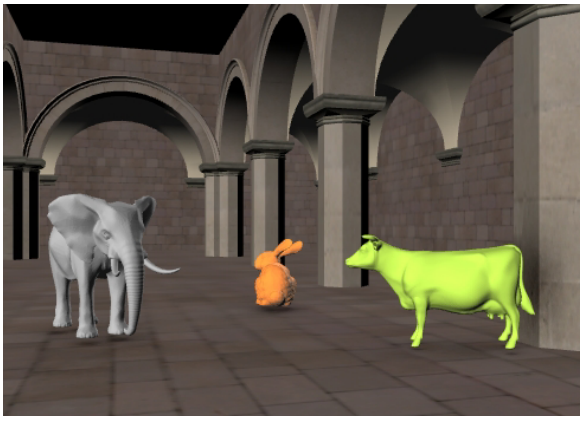


Figure 6: A scene of the Sponza Atrium with a bunny (38889 tris), a cow (92864 tris) and an elephant (157160 tris) rendered in three passes (one per object) with the visibility fields algorithm using 4226x64x64 maps and rendering in 2.5 frames per second.

Even though the *visibility fields* method is only an approximation, it does a very good job at preserving image quality given the low memory requirements and achieved draw time.

5.2 Ray tracing

In Figure 7 we show a close-up of the bunny ears of using the visibility-fields method. We show that very good results of soft shadows can be achieved while using 20 shadow ray samples along with 4226 64x64 visibility maps (i.e. 16.51MB of memory).

In Figure 8 we render a slightly more complex scene using 3 light sources of radius 2. As in the previous cases, the rendering time is almost completely affected by the primary rays which perform triangle intersection tests. Our method completes the rendering in 3268 ms, of which 70% is for the shadow rays. It produces a very good approximation of soft shadows using 20 shadow rays per pixel. For the total of 11,838,600 shadow rays, this corresponds to 1.9323 10^{-4} ms per shadow ray which is a very encouraging result. In the corresponding BVH GPU method to produce sharp shadows using just 1 shadow ray per pixel, the draw time is 48047 ms to compute the final image. Of that time 70% is used for the 591930 shadow rays yielding 5.682 10^{-2} ms per shadow ray.

In Figure 9 we use the visibility fields algorithm to render non-perfect-mirror reflections. The polished reference image is rendered with 4 rays for each reflective pixel leading to slower rendering times. However, we notice from the images and the RMS factor that the reflected sub region of our method is much closer to the result of the brushed metal reference image than the perfect mirror reference image. This strengthens our position that the proposed method is suitable for stochastic ray-tracing, as the quality of the rendered image is comparable to the reference image. Furthermore, the rendering time, even using 4 rays per reflective pixel, is very close to ray-casting without secondary rays.

6. Limitations of the Visibility Fields

The *visibility fields* method is not very well suited for elongated models. The occlusion produced, even when using 16642 maps is pretty grainy. In addition

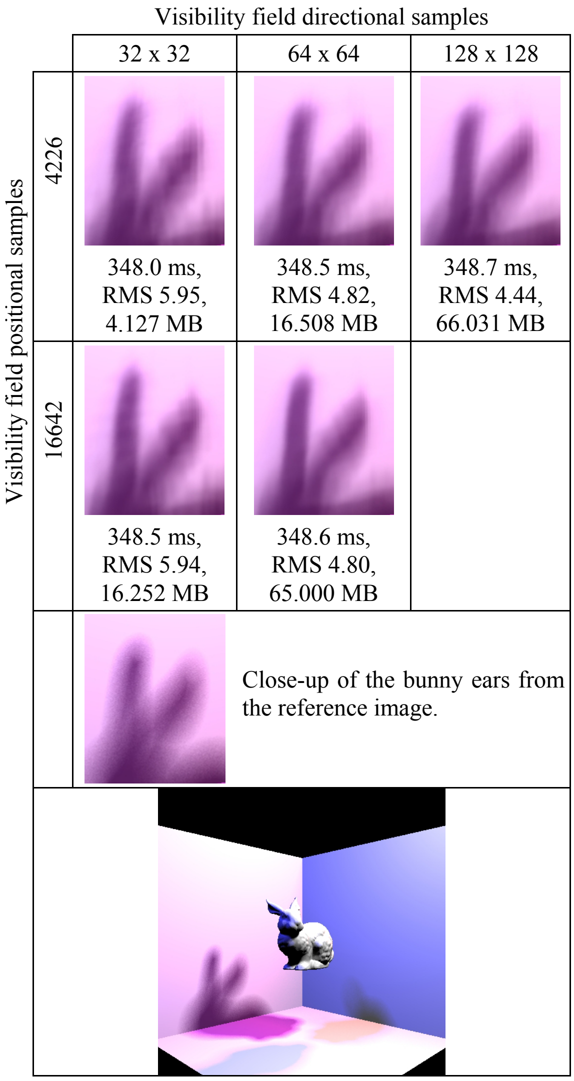

Visibility field directional samples
32 x 32
64 x 64
128 x 128
Visibility field positional samples
4226
348.0 ms, RMS 5.95, 4.127 MB
348.5 ms, RMS 4.82, 16.508 MB
348.7 ms, RMS 4.44, 66.031 MB
16642
348.5 ms, RMS 5.94, 16.252 MB
348.6 ms, RMS 4.80, 65.000 MB
Close-up of the bunny ears from the reference image.

Figure 7: Close-up of the bunny ears rendered using the visibility fields for the generation of soft shadows using 3 lights and 20 shadow ray samples on the GPU. We report the required time, the RMS error and the total space requirements. Bottom: Reference image rendered using the BVH method with 3 lights and 256 rays per pixel taking 913,210 ms on the GPU.

models that are highly concave would fail to produce accurate visibility maps as it would not be possible to record all of the tight concavities of the model.

7. Conclusions

We have presented the *visibility fields*, a discretization of the visibility around an object, implemented on the GPU. We have shown how it can be used for an interactive inter-object ambient occlusion approximation computation. For the intra-object occlusion case the number of required maps is large and the draw time needs improvement when the model covers a lot of pixels on the screen. But in a game environment where several models exist on the screen and their coverage is not very big, the intra-object occlusion method can be used even for high triangle count models.

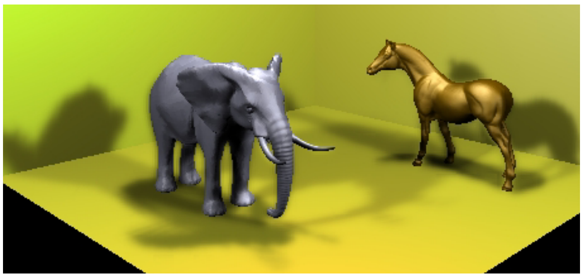
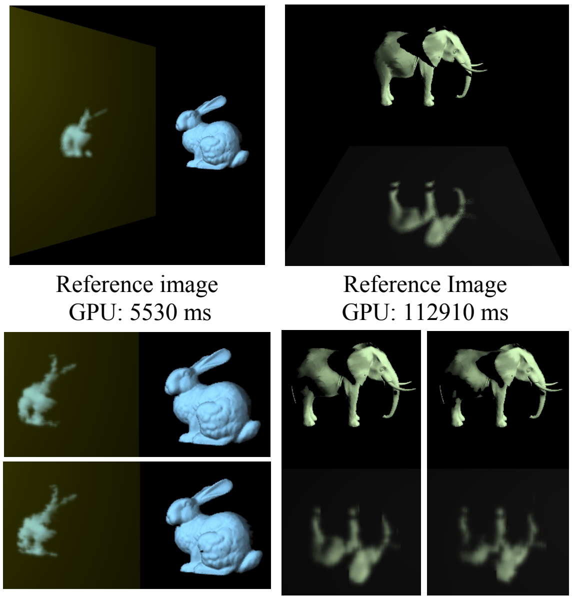


Figure 8: Close-up of a more complex scene using 3 point lights and 20 shadow ray samples rendered in 3268 ms using the visibility fields method. The BVH GPU method for sharp shadows takes 48047 ms.

The method especially favours large model data sets, where we maintain a constant computation time, independent of the model complexity. Our method is robust, has a relatively small memory footprint



Reference image
GPU: 5530 ms

Reference Image
GPU: 112910 ms

(top) 4226 64x64 maps
441 ms, 4.480 RMS error,
66.031 MB used

(left) 4226 64x64 maps
1900 ms, 8.137 RMS error,
66.031 MB used

(bottom) 4226 32x32 maps
440 ms, 4.555 RMS error,
16.508 MB used

(right) 4226 32x32 maps
1897 ms, 9.392 RMS error,
16.508 MB used

Figure 9: Polished reflection of the elephant (157160 tris) and the bunny (39000 tris) using 4 rays per reflective pixel. First row: Reference images using the BVH method (GPU draw times). Second row: Close-up view of our visibility fields GPU method where we report the draw time, the RMS error and the space requirements.

against comparable existing methods and the time required to generate the visibility maps depends only on the complexity of the occluder geometry. In addition, the number and resolution of the maps used in the *visibility fields* can be adjusted depending on the required accuracy and the available memory. The same maps can be used for both inter and intra-object ambient occlusion computation.

Furthermore, our algorithm can be applied to ray tracing calculations where exact ray hits are not critical, for example for shadow and secondary ray intersection tests, such as soft shadow rays and Monte Carlo ray tracing.

We have shown that in the above mentioned cases the production of the desired image is accelerated while the results remain close to the reference images. The hybrid method we propose favours large model data sets as in ambient occlusion. This result is expected as all triangle intersection tests for shadow and secondary rays are replaced with constant time operations. In this way rendering time is affected mostly by the primary rays that give us the visibility of the scene.

8. REFERENCES

[Ami07] Amit B., Elber G.: GPU Ray Tracing. Master's Thesis, Technion Israel Institute of Technology, 2007.

[Bav09] Bavoil, L., Sainz, M.: Image-space horizon-based ambient occlusion. In ShaderX7 - Advanced Rendering Techniques, Delmar, 2009.

[Bun05] Bunnell M.: Dynamic ambient occlusion and indirect lighting. In GPU Gems 2, pages 223–234. Addison Wesley, 2005.

[Car02] Carr A. N., Hall D. J., Hart C. J.: The Ray Engine. In Proc.Graphics Hardware 2002, pg. 37–46, Sep. 2002.

[Car06] Carr A. N., Hoberock J., Crane K. Hart C. J.: Fast GPU Ray Tracing of Dynamic Meshes using Geometry Images. In Proceedings of Graphics Interface 2006, Quebec, Canada, June 07-09, 2006.

[Chr05] Christen M., Engel W., Hudritsch M.: Ray Tracing on GPU. Diploma Thesis Univ. of Applied Sciences Basel (FHBB), 2005.

[Ern04] Ernst M., Vogelgsang C., Greiner G.: Stack Implementation on Programmable Graphics Hardware. In Proceedings of the Vision, Modelling, and Visualization Conference 2004 (VMV 2004), pp. 255–262.

[Fol05] Foley T., Sugerman J.: Kd-tree acceleration structures for a GPU ray tracer. In Proc. Graphics Hardware, pages 15–22, 2005.

[Gai08] Gaitatzes A., Chrysanthou Y., Papaioannou G.: Presampled Visibility for Ambient Occlusion. In Proc. of the 16-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG '2008), Czech Republic, February 2008.

[Gün07] Günther J., Popov S., Seidel H.-P., Slusallek P.: Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In Proc of the IEEE / Eurographics Symposium on Interactive Ray Tracing 2007, pp. 113–118.

[Hav00] Havran V.: Heuristic Ray Shooting Algorithms. Ph.D. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[Hor07] Horn R. D., Sugerman J., Houston M., Hanrahan P.: Interactive k-D Tree GPU Raytracing. In Proceedings of the Symposium on Interactive 3D Graphics and Games, ACM Press, pp. 167–174, 2007.

[Kap85] Kaplan R. M.: Space-Tracing: A Constant Time Ray-Tracer. In Proc. Computer Graphics 19, 3 (July 1985), pg. 149–158. (Proceedings of SIGGRAPH 85 Tutorial on Ray Tracing).

[Kar04] Karlsson F., Ljungstedt C. J.: Ray tracing fully implemented on programmable graphics hardware. Master's Thesis, Chalmers Univ. of Technology, 2004.

[Kes06] Kessenich J., Baldwin D., Rost R.: The OpenGL Shading Language. Version 1.2.8. 3Dlabs, Inc. Ltd. 2006.

[Mal88] Malley T. J. V.: A shading method for computer generated images. In Master's Thesis, Computer Science Department, University of Utah, June 1988.

[Mit07] Mittring M.: Finding next gen: CryEngine 2. In ACM SIGGRAPH 2007 Courses, San Diego, California, August 05-09, 2007.

[Pur02] Purcell J. T., Buck I., Mark R. W., Hanrahan P.: Ray tracing on programmable graphics hardware. In Proc. SIGGRAPH, 2002.

[Pur04] Purcell J. T.: Ray Tracing on a Stream Processor. Ph. D. Dissertation, Stanford University, 2004.

[Pop07] Popov S., Günther J., Seidel H.-P., Slusallek P.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. In Proc. of Computer Graphics Forum 26(3), pp. 415–424, 2007, (Proceedings of Eurographics)

[Sha07] Shanmugam P., Arikan O.: Hardware accelerated ambient occlusion techniques on GPUs. In Proceedings of the 2007 Symposium on interactive 3D Graphics and Games, Seattle, Washington, April 30 - May 02, 2007.

[Shi97] Shirley P., Chiu K.: A low distortion map between disk and square. In Journal of Graphics Tools 2, 3 (1997).

[Sla02] Slater M.: Constant time queries on uniformly distributed points on a hemisphere. In Journal of Graphic Tools 7, 1 (2002), pp. 33–44.

[Thr05] Thrane N., Simonsen L.O.: A comparison of acceleration structures for GPU assisted ray tracing. Master's Thesis, University of Aarhus, Denmark, 2005.