Vis Comput (2014) 30:697–706
DOI 10.1007/s00371-014-0968-8

ORIGINAL ARTICLE

Ray tracing via GPU rasterization

Wei Hu · Yangyu Huang · Fan Zhang · Guodong Yuan · Wei Li

Published online: 8 May 2014

Abstract Ray tracing is a dominant method for generating a wide variety of global illumination effects, such as reflections/refractions, shadows, etc. In this paper, we propose an efficient technique to perform nearly accurate ray tracing using the programmable graphics processor units (GPUs). With the aid of the linked-list A-buffer and the uniform voxel grid to represent scene geometry, the ray-scene intersection can be efficiently computed via the built-in rasterization on GPUs. Based on this novel ray-scene intersection technique, a new ray-tracing framework which supports various light transport algorithms is introduced, including Ray Casting, Whitted Ray tracing, Ambient Occlusion, Path Tracing, and so on. The experimental results demonstrate the accuracy and efficiency of our approach.

Keywords Ray tracing · Global illumination · Rasterization · GPUs

1 Introduction

Ray tracing is a classical method for rendering global illumination (GI) effects, especially reflections, refractions and shadows. The core problem of ray tracing is how to efficiently compute intersections of a ray with geometric scene primitives. For highly complex scenes, ray tracing usually becomes time-consuming due to excessive amount of intersection tests. Therefore, the primary approach to accelerate ray tracing focuses on improving the performance of ray-scene intersection tests. To avoid "brute-force" issue of intersection tests, many hierarchical data structures have been proposed and implemented on CPU, such as BSP tree, kd-tree, octree, uniform grid, etc, which have been comprehensively surveyed in [1].

Ray tracing on GPUs has been well studied in recent years [7–11]. The early methods [11] only employ GPUs as a high performance processor to accelerate ray tracing, with the similar data structures implemented on CPUs. Therefore, it is hard to integrate these techniques into the traditional hardware rasterization pipeline. Recently, some scene representation techniques, including voxelization and sampled textures on GPUs, have been proposed (see Sect. 2), and corresponding ray-tracing methods can achieve good performance since primary rays can be efficiently replaced by rasterization. However, these methods are hard to trace primary/secondary rays recursively in a unified framework, and generally are proposed for rendering limited global effects.

This paper proposes a ray-tracing method via hardware rasterization pipeline. Uniform grid voxels and per-fragment linked-list A-buffer [2,3] are adopted to represent the scene geometries, which are stored as texture buffers on GPUs. All rays are then tested against the scene to compute the intersections using these GPU-based scene representations. Our framework traces rays only via GPU rasterization, which leads to easy implementation and convenient integration into the rendering pipeline. In addition, compared with the existing rasterization-based techniques, our method achieves a better balance between accuracy and performance. Utilizing such a ray-tracing scheme, several ray-tracing style models, such as Ray Casting, Whitted Ray tracing [4], Ambient Occlusion [5], Path Tracing [6], etc, can be efficiently implemented, as shown in Fig. 1.

W. Hu (✉) · Y. Huang · F. Zhang · W. Li
College of Information Science and Technology,
Beijing University of Chemical Technology, Beijing 100029, China
e-mail: huwei@mail.buct.edu.cn

G. Yuan
Computer School, Beijing Information Science
and Technology University, Beijing 100010, China

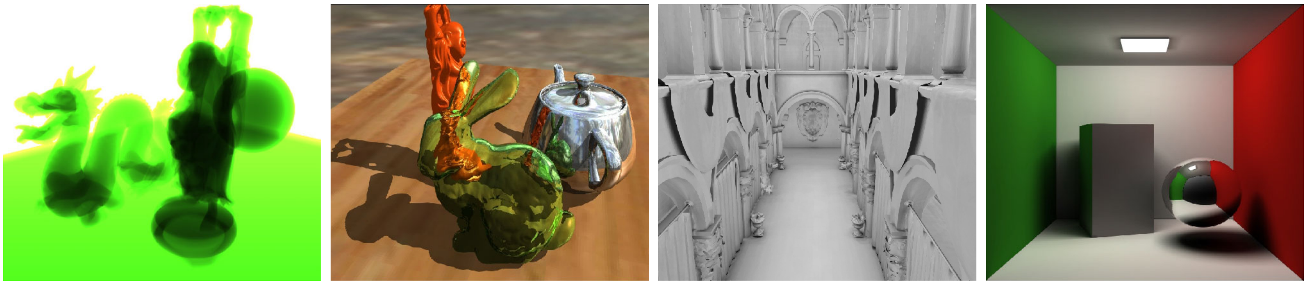

(a) 35 ms **(b)** 27 ms **(c)** 660 ms **(d)** 1030 ms

Fig. 1 From *left* to *right*, Ray Casting, Whitted Ray tracing, Ambient Occlusion, and Path Tracing results (size = 800 × 600) generated by our method with full dynamic scenes. All secondary effects (depth layers, shadows, reflections, and refractions) are produced by tracing rays via GPU rasterization. The depth of whitted ray tracing is 3, and we trace 128 rays per-fragment for rendering AO and 384 rays per-fragment for two-bounce path tracing (with importance sampling on lights) zooming in for details

This paper is organized as follows: First, the closely related works are surveyed in Sect. 2. Second, Sect. 3 describes the theory and implementation details of our ray-scene intersection algorithm. Then, how to build up a ray-tracing framework via the new intersection method, so as to implement various global light transport methods, is presented in Sect. 4. Section 5 shows the experimental results and discusses the comparisons with the state-of-the-art prior methods. Finally, the paper is concluded in Sect. 6 and proposes potential avenues for future research.

2 Related work

Over the last 20 years, various hierarchical structures to accelerate ray tracing on CPUs have been proposed as summarized in [1]. Recently, ray tracing on GPUs has attracted lots of research interests [7–10] since [11] in 2002. Nvidia also proposed Optix® ray tracing engine on GPUs [12]. However, the GPU is only regarded as a general purpose many-core processor in these methods, and hence it is nontrivial to integrate these methods into the existing OpenGL/Direct3D style rasterization pipelines.

Some rasterization-based techniques have been proposed for achieving ray tracing to render various effects, such as reflection, refraction, caustics and shadows, on GPUs. The scene geometry is approximated by the corresponding sampled texture or the volumetric representation that can be stored as textures on GPUs, so it is feasible to compute the ray-scene intersections in the fragment shader via texture fetching. The environment map and the shadow map are just sampled forms for rendering approximate reflection, refraction and shadows. To improve accuracy of the intersection test, a straightforward solution is to store more geometry data in textures. Hence, single- or multiple-layered textures generated by rendering scenes from different sampling positions are used to approximate scenes [13–17]. The major limitation of these techniques is that the generated textures only represent part of the scene, or the representation and the corresponding ray-scene intersections depend on specific objects (reflectors/refractors) or sampling locations. Although rays can be traced fairly accurately (mirror reflection/refraction supported), these methods could not be used as general ray-tracing solutions. Layered depth images (LDIs) [18–20], which are constructed via depth peeling along orthogonal projections, are proposed to represent scene objects. Ray tracing on LDIs is firstly presented in [18]. However, due to the excessive memory cost of high-resolution LDIs, the rendering performance is low for accurate ray-scene intersections. Niebner et al. [19] accelerates the ray-scene testing by reducing sizes of LDIs, at the cost of high-frequency effects. Furthermore, rays cannot be traced within scene objects in these methods, so rendering refractions is not supported. Moreover, pre-processing is also required when generating many LDIs [19].

Voxel-based ray-tracing methods attract more and more attentions recently, because of geometry-independent scene description and the evolving of fast voxelization methods. Some scene voxelization and ray-voxel intersection techniques have been proposed for rendering global illumination interactively [21–26]. However, only diffuse indirect lighting (reflection) is rendered in [21–23], due to lack of precision caused by low-resolution uniform voxel grids. CUDA is used to construct voxel octrees in [24]. Rasterization-based efficient sparse voxel octree [25,26] can improve precision so that glossy reflections can be rendered interactively. Nevertheless, it is still difficult to compute accurate ray-scene intersections efficiently, and moreover, generating sparse voxel octree is relatively time-consuming so that it is not applicable for rendering fully dynamic scenes (see discussion in Sect. 5). Novak and Dachsbacher [27] proposed rasterized bounding volume hierarchy(RBVH) to represent a scene, and surface data are contained in BVHs as atlas for accurate ray tracing. However, RBVHs cannot be constructed

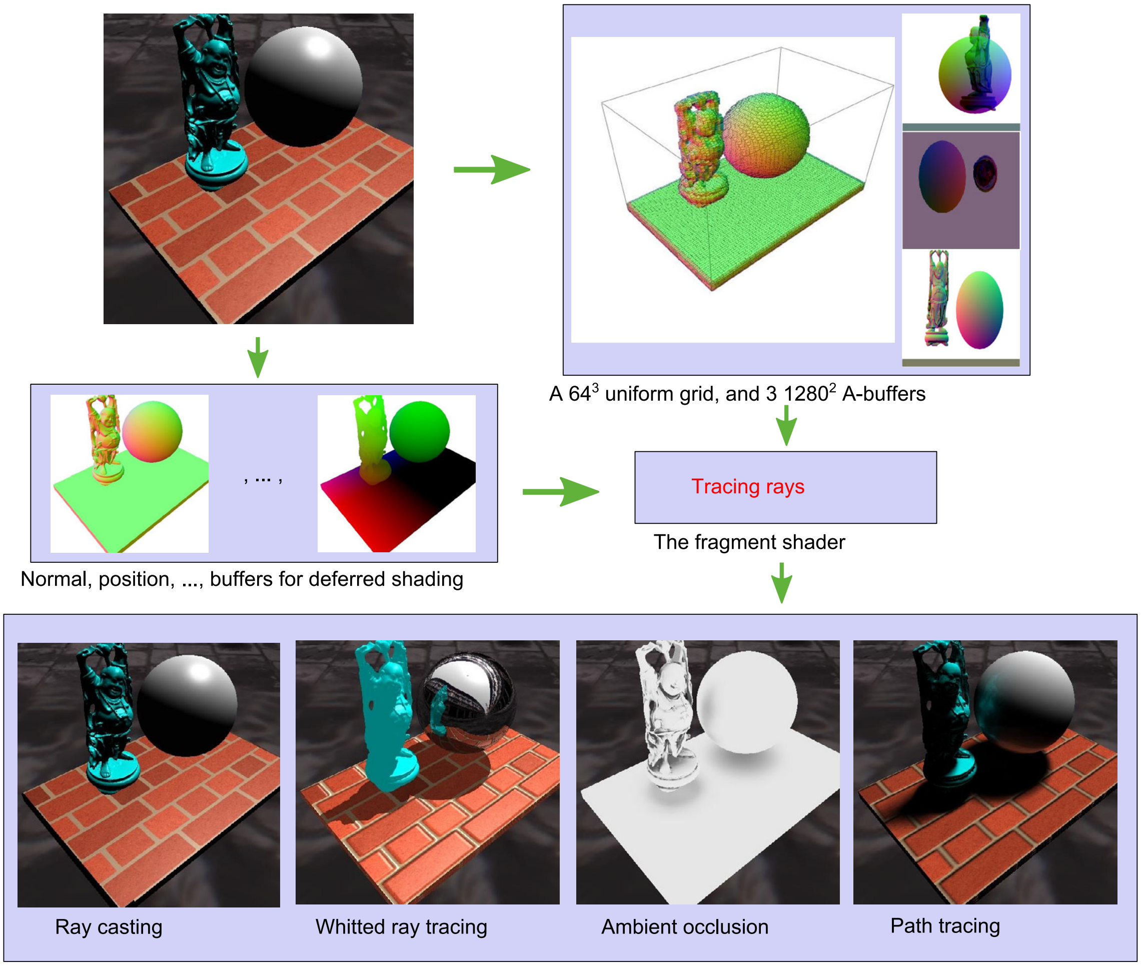

A 64^3 uniform grid, and 3 1280^2 A-buffers
, ... ,
Normal, position, ..., buffers for deferred shading
Tracing rays
The fragment shader
Ray casting
Whitted ray tracing
Ambient occlusion
Path tracing

Fig. 2 The uniform grid (with the average normal value for each cell) and A-buffers (with surface attribute values for each pixel) are used to represent the complex scene. Accurate ray tracing can be conducted in the fragment shader via deferred shading. Therefore, Ray Casting, Whitted Ray tracing (ambient lighting enabled), Ambient Occlusion, and Path Tracing can be implemented using our ray-tracing framework

using only rasterization-based pipeline (CUDA is required), and the construction time is still too slow to render complex dynamic scenes in real time.

There are also some rasterization-based global illumination methods, such as GPU-based bidirectional path tracing [28]. However, to the best of our knowledge, these methods are usually not general enough, and few of them focus on achieving fundamental high-precision ray-tracing frameworks.

Inspired by voxel- and LDI-based techniques, we present a novel method that integrates these two techniques to represent scene geometries, and a corresponding ray-scene intersection method to trace rays. Moreover, to demonstrate the capability of this new ray-tracing scheme, we implement various light transport methods, such as Ray Casting, Whitted-style Ray tracing, Ambient Occlusion, and Path Tracing, based on our ray-tracing framework. The framework of our algorithm is illustrated in Fig. 2.

3 Ray-scene intersection

As shown in Fig. 2, our approach first generates a two-level representation of the input scene, which includes a uniform voxel grid and three per-fragment linked-list A-buffers, to facilitate ray tracing. The uniform voxel grid is a coarse level of the actual geometric meshes, and the A-buffer represents scenes more precisely due to its high resolution. Note that a $1K \times 1K$ A-buffer corresponds to a resolution of a uniform grid of $1K^3$. This two-level representation allows to avoid using actual geometric meshes to compute ray-scene intersections, while still providing high accuracy and good efficiency. During the tracing process, a ray is traced against the uniform voxel grid first, and then if an initial intersection is detected, the ray is further tested against the A-buffers to compute the precise intersection point. Therefore, various GI effects can be efficiently rendered using our new ray-tracing technique.

3.1 Scene representation

We generate the uniform voxel grid and A-buffers for the input scenes via the rasterization-based rendering pipeline on GPUs. Thanks to the recently introduced image load, store and atomic operations in OpenGL, the two-level representation can be generated in a single pass.

First, we define the resolution of the voxel grid and the A-buffer. Then, a 3D texture to store the voxel grid and three texture buffers to store A-buffers are allocated.

Second, each triangle of the mesh is inputted into the GPU rendering pipeline. In the geometry shader, all triangle vertices are projected orthographically along the three main axes of the scene and outputted to the corresponding layers. The reason to use three projections is to avoid the degenerated triangle when its normal is perpendicular to the projection direction. Depth testing is disabled to prevent face culling, at least one fragment shader thread is generated for each triangle in the scene.

Lastly, for each fragment, the corresponding voxel grid can be computed easily, and we directly fill the corresponding grid with the fragment's normal for selecting the correct A-buffer (described in Sect. 3.2) using image store operations. Since multiple fragments can fall into the same grid, to average all the normals, the same technique proposed in [29] is used. Meanwhile, using the technique described in [3], we can generate three A-buffers containing all the attributes (the depth, position, normal, and BRDF, etc) that we want to store.

Furthermore, another rendering pass is processed to sort linked-list of each fragment, and then write sorting results back to A-buffers. This sorting process can further accelerate ray-scene intersection computation as discussed in Sect. 3.2.

3.2 Ray-scene intersection

The ray-scene intersection test is computed in the fragment shader after rasterization, and the scene (One uniform grid and three A-buffers) is accessed as textures and buffers.

To compute ray-scene intersections, we first traverse the ray on the coarse-level uniform grid. As shown in Fig. 3, the uniform voxel grid simplifies the computation of determining which voxel would be visited along a given ray. We use the classic 3D-DDA grid traversal algorithm [30] to step the ray along successive voxels (green-edge cells in the figure). If the current voxel contains scene geometry (red-filled cells in the figure), the precise intersection point is further determined using A-buffers. The 3D-DDA algorithm requires only a small number of steps to trace the ray, and the cost of traversal is linear to the number of the visited voxels. Since the resolution of the voxel grid is relatively low, the average traversal cost for this initial intersection test is small.

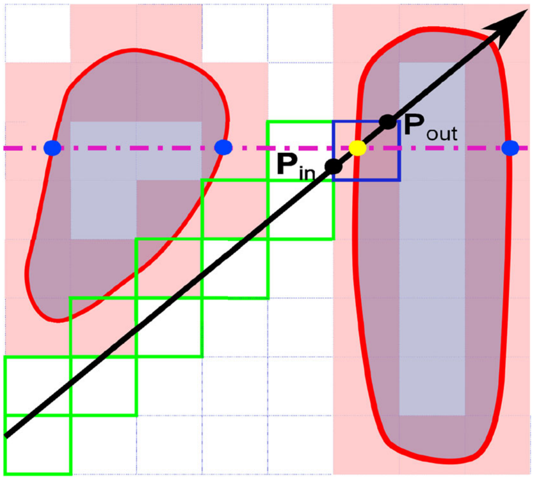


Fig. 3 Tracing the ray on the uniform grid first, and the precise intersection point is then computed by comparing depth values in the A-buffer, which is selected from 3 A-buffers using the average normal value in the current cell

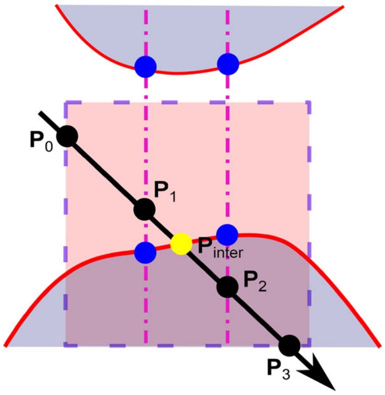


Fig. 4 $\mathbf{P}_{\text{inter}}$ is first identified between $\mathbf{P}_1$ and $\mathbf{P}_2$ through a linear searching, and then accurately computed by another binary searching [31]

If the current voxel contains geometry, we should further trace the ray on the A-buffers of the scene. In Fig. 3, the corresponding entry point $\mathbf{P}_{\text{in}}$ and exit point $\mathbf{P}_{\text{out}}$ could be easily computed with the 3D-DDA algorithm, and then these two points would be projected to the corresponding A-buffer, which is selected using the average normal value of the current voxel.

Given a point, we could access sorted link-listed values of the corresponding pixel in the A-buffer, and determine whether the point is inside scene objects by comparing depth values. Linear searching from $\mathbf{P}_{\text{in}}$ to $\mathbf{P}_{\text{out}}$ using 2D-DDA on the A-buffer could be employed to find the first intersection point $\mathbf{P}_{\text{inter}}$. Although the resolution of the A-buffer is high, the number of searching steps would be low with the restricted range between $\mathbf{P}_{\text{in}}$ and $\mathbf{P}_{\text{out}}$. To further improve the performance of ray tracing, we introduce a search algorithm inspired by [20], as illustrated in Fig. 4. First, we find the first

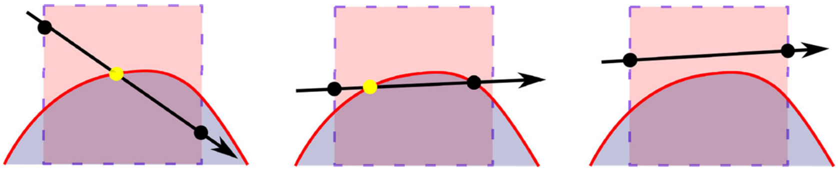
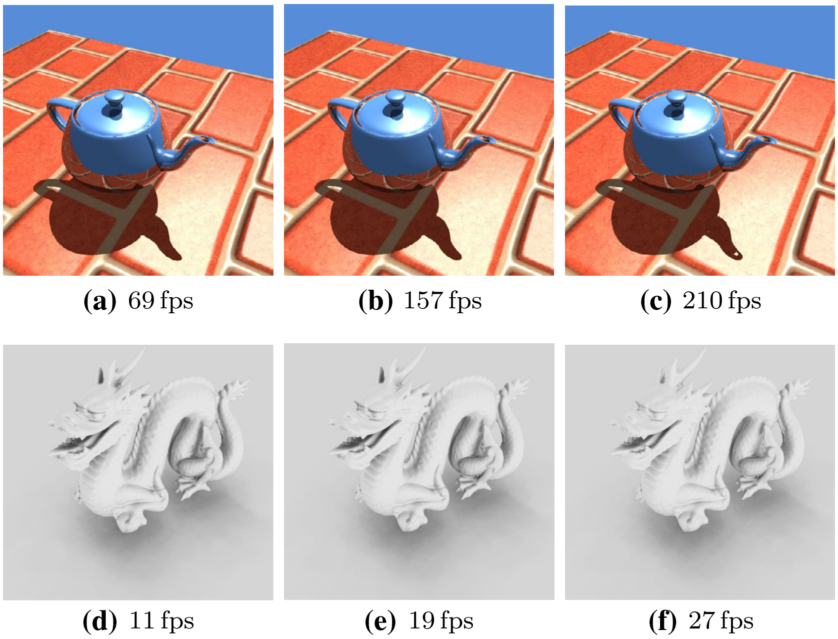


Fig. 5 We can assume that only one smooth surface is contained in a voxel cell. Therefore, we can simply judge the intersection by a binary searching between $\mathbf{P}_{\text{in}}$ and $\mathbf{P}_{\text{out}}$ (one-step linear search), especially for generating low-frequency effects



(a) 69 fps **(b)** 157 fps **(c)** 210 fps

(d) 11 fps **(e)** 19 fps **(f)** 27 fps

Fig. 6 Whitted Ray tracing (*top*) and Ambient Occlusion with 128 rays per-fragment (*bottom*) using 2D-DDA linear search (*left*), three-step linear search (*middle*), and one-step linear search (*right*). Notice some differences (see teapot spout and dragon horn) caused by sharp surfaces. The sizes of the uniform grid and A-buffers are 64^3 and 1024^2, respectively. The sub-caption indicates rendering frames per-second

point $\mathbf{P}_k$ inside (or outside for refraction) the object with an approximate linear search; Second, a binary search similar to [31] is conducted to find the intersection point $\mathbf{P}_{\text{inter}}$ between $\mathbf{P}_k$ and $\mathbf{P}_{k-1}$. An appropriate step length for the linear search might greatly reduce the comparison times to compute $\mathbf{P}_{\text{inter}}$. Moreover, in some cases (for example, tracing shadow rays), the accurate position of the intersection point is unnecessary, so the binary search can be ignored to improve the performance.

Since the uniform voxel grid size is relatively small, in most scenarios, we can assume that only one smooth surface is contained in a voxel cell. Therefore, if the entry point $\mathbf{P}_{\text{in}}$ is outside the object, there are two circumstances: the exit point $\mathbf{P}_{\text{out}}$ is inside or outside the object. If $\mathbf{P}_{\text{out}}$ is inside the object (see the left one in Fig. 5), a binary searching can rapidly identify $\mathbf{P}_{\text{inter}}$. If $\mathbf{P}_{\text{out}}$ is outside the object, our algorithm might miss the accurate $\mathbf{P}_{\text{inter}}$ if the linear-search step length is large (see the middle one in Fig. 5). Since the possibility that the ray intersects the object is low under this circumstance, usually a three-step linear search is enough for generating results with high-precision, and one-step linear search is also appropriate for rendering low-frequency effects. Figure 6 gives some comparisons of searching techniques. We can see that three-step linear search is visually close to ground truth 2D-DDA search, and even a one-step linear search is sufficient to well approximate the low-frequency effects. We notice that there are some differences caused by sharp surfaces. Since the assumption of one-step linear search, one smooth surface in a voxel cell, is not obeyed here.

Since the times of depth comparison in the A-buffer are limited by three-step/one-step linear search, the resolution of A-buffers does not have great impact on the performance of the step search. Therefore, high-resolution A-buffers are preferred and adopted in our implementation.

4 Illumination model implementation

Based on the new ray-scene intersection technique, various global light transport methods can be fully implemented on GPUs (see Fig. 2). Deferred shading is employed to avoid costly ray tracing for occluded fragments.

In our implementation of Ray Casting, primary rays are generated and traced against scene objects, and final colors are accumulated in a single fragment shader. Since multiple intersections are naturally supported, we can render multi-layer effects(see Fig. 1a) by recursively tracing rays until rays are out of the bounding box.

In the Whitted Ray tracing, reflection rays and refraction rays are traced recursively for each intersection in the fragment shader, and shadow rays are generated at all intersection points. Since recursion is not allowed in GLSL, we implement the recursive ray tracing with loops and stacks. In our implementation, we set maximum ray tracing depth $= 3$ to achieve satisfying results.

Tens or hundreds of sampling rays are required for each fragment in Ambient Occlusion and Path Tracing. In our implementation, tracing more than 64 rays per-fragment in one single fragment shader is currently not supported due to the constrained number of shader instructions and OpenGL window time-out limitations. Fortunately, Ambient Occlusion and Path Tracing both can be computed using sum and average operations. To trace more rays efficiently, a multiple-pass rendering pipeline is essential. Two framebuffers ($\mathbf{FB}_{\mathrm{curr}}$ and $\mathbf{FB}_{\mathrm{comp}}$) and corresponding multiple-pass rendering methods are introduced to composite final results, as shown in Algorithm 1.

There are two proposed multiple-pass rendering methods, the progressive rendering method and the general rendering method in our solution. The progressive method iteratively refines the final result after each rendering passes, and can be applied for interactively previewing the complex lighting effects. The procedure $PRender()$ is executed for each frame, and the procedure $InitCompBuffer()$ (and the representation construction) is executed only when the display content is changing (moving camera, updating scene objects, etc). In our general rendering method, the procedure $InitCompBuffer()$ is also executed for each frame, so the final result is totally re-generated. λ is the sampling number in a single rendering pass, and $\lambda \times \eta$ is the total number of sampling rays of current fragment. According to our experiments, a short shader is preferred for better performance, and we set $\lambda = 8$ for AO and $\lambda = 3$ for path tracing (due to multi-bounce path length) in our implementation. Figure 7 illustrates the multi-pass composition results with different rendering passes.

Algorithm 1 Composition for multi-sampling methods

```
▷ Init composition buffers
procedure INITCOMPBUFFER()
    Enable the framebuffer $\mathbf{FB}_{comp}$;
    Clear color and depth buffers;
    $\eta = 1$;
end procedure

▷ The Progressive Rendering method
procedure PRENDER()
    ▷ Rendering objects
    Enable the framebuffer $\mathbf{FB}_{curr}$;
    Clear color and depth buffers;
    Draw objects with $\lambda$ sampling rays;
    Disable the framebuffer $\mathbf{FB}_{curr}$;
    ▷ Accumulate $\mathbf{FB}_{curr}$
    Enable the framebuffer $\mathbf{FB}_{comp}$;
    Enable alpha blending;
    Set glBlendFunc(GL_ONE, GL_ONE);
    Draw a quad with the texture $\mathbf{FB}_{curr}$;
    Disable the framebuffer $\mathbf{FB}_{comp}$;
    ▷ Composite results
    Clear output color and depth buffers;
    Display $\mathbf{FB}_{comp}$ according to $\eta$ and $\lambda$;
    $\eta$++;
end procedure

▷ The General Rendering method
procedure GRENDER(int $renderPasses$)
    ▷ Rendering objects
    INITCOMPBUFFER();
    for $\eta = 1$ to $renderPasses$ do
        Enable the framebuffer $\mathbf{FB}_{curr}$;
        Clear color and depth buffers;
        Draw objects with $\lambda$ sampling rays;
        Disable the framebuffer $\mathbf{FB}_{curr}$;
        ▷ Accumulate $\mathbf{FB}_{curr}$
        Enable the framebuffer $\mathbf{FB}_{comp}$;
        Enable alpha blending;
        Set glBlendFunc(GL_ONE, GL_ONE);
        Draw a quad with the texture $\mathbf{FB}_{curr}$;
        Disable the framebuffer $\mathbf{FB}_{comp}$;
    end for
    ▷ Composite results
    Clear output color and depth buffers;
    Display $\mathbf{FB}_{comp}$ according to $renderPass$ and $\lambda$;
end procedure
```

5 Results and comparisons

Our technique was implemented in OpenGL/GLSL and all the results were rendered on a PC equipped with an Intel Xeon® CPU E5-1620 with 3.7 GHz and an Nvidia GeForce® 780 GTX graphics card. The sizes of uniform grids and A-buffers should be well investigated to keep well balance between quality and performance. According to our experience, a $(\sqrt{3n})^3$ uniform grid is a optimal choice for n^2 A-buffers. Except of the marked places, all the rendering results in the paper were generated using 64^3 uniform grids and 1280^2 A-buffers, and the size of the screen buffer is 800×600. Note that all performance data are recorded for full dynamic scenes except with the progressive rendering method, that is to say, uniform voxel grids and A-buffers are re-constructed for each frame.

5.1 Accuracy of ray-scene intersection

To verify the accuracy of our ray-scene intersection method, we compare Whitted ray-tracing results rendered using our framework to the references generated with Optix ray-tracing engine [12]. Figure 8 illustrates the results of this comparison. It can be seen from the figure, our results closely match the references.

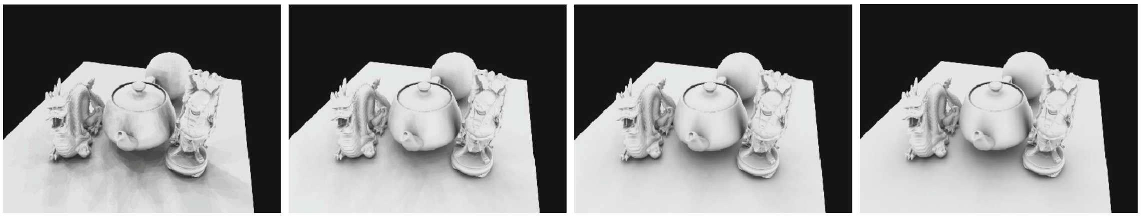


Fig. 7 AO effects with different rendering passes (the sampling number $\lambda = 8$ for one single pass). About 0.75 ms is required for tracing one sampling ray. Note that the timing of constructing scene representation is not included

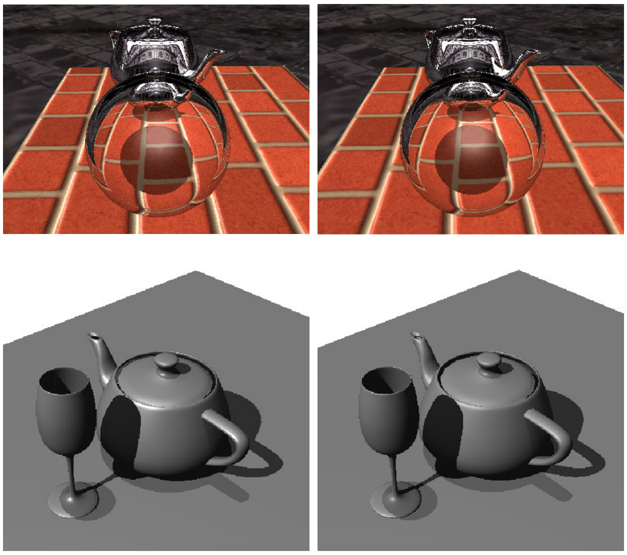


Fig. 8 Comparison of ray-tracing accuracy as implemented in our method (*left*) and Optix (*right*). The ray-tracing depth values are 3 in the top one. Two point lights are used in the bottom one

5.2 Comparison with related techniques

Regular voxel [22,23] and sparse voxel octrees (SVOs) [26] were proposed to represent scene objects for rendering global illumination on GPUs. Restricted to memory limitation and traversal performance, the low-resolution regular voxel is preferred in [22,23]. Although some low-frequency effects can be rendered via regular voxels only, to render accurate results as ours, the sampling number or the voxel resolution should be higher, which would cause performance and memory problems (see Fig. 9). It is clear that the same results could be generated with 1280^3 and 64 sampling rays using regular voxel grids in Fig. 9, but the GPU memory capability limits the size of voxel grids.

Compared with regular voxels, SVOs can describe scenes with higher precision. Crassin et al. [26] implemented global illumination computation using Octrees and cone-tracing, and achieved good performance. However, the SVO technique also has limitations on dynamic construction performance. Moreover, the SVO structure is well suited to cone-tracing, but accurate tracing rays on leaf level of SVOs is time-consuming in the fragment shader. We compared the method with our implemented SVO as shown in Table 1, our method spends less time on constructing scene representations and tracing rays achieving the same precision (the precision that a $1K \times 1K$ A-buffer can achieve is the same as a uniform grid of $1K^3$, and a 10-level octree). Therefore, our algorithm is more applicable for dynamic scenes and complex illumination models.

Admittedly, some low-frequency effects are better supported in SVO cone-tracing, since a few number of cones can approximate Ambient Occlusion and indirect illumination well. Therefore, SVOs can render Ambient Occlusion of static scenes more rapidly. However, Cone-tracing is naturally not suitable for illumination methods that require high-precision ray-object intersection (hard shadow, mirror reflections, soft shadows under area lights, etc). On the contrary, our method can trace rays more precisely with high per-

Fig. 9 Comparison of rendering AO with our technique (*left*) and only voxel grids [23] (*right two*). The general rendering method is used here. The sub-caption indicates the size of voxel grid, the sampling rays per-fragment, and the rendering frames per-second (fps)

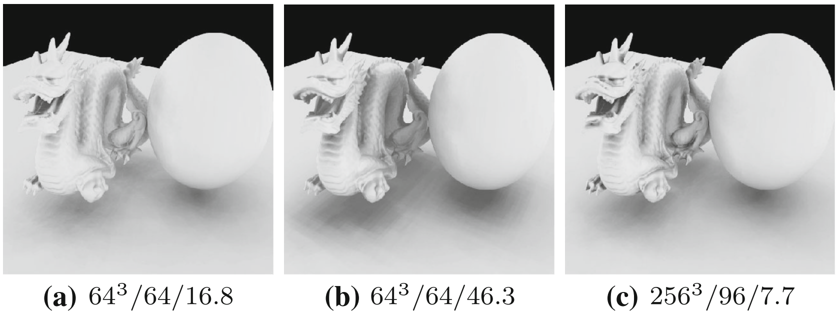
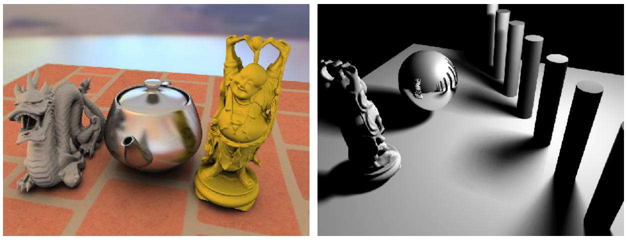


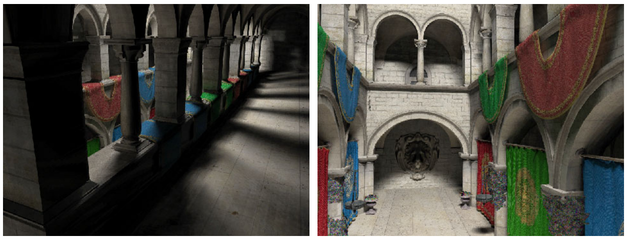
Table 1 Performance comparison between our method and SVO. To achieve the same precision, we trace rays on leaf nodes of SVO

Scene	Method	Construction (ms)	Tracing rays (ms)	Others (ms)	Total (ms)
Figure 1b	Ours	13.2	11.6	2.7	27.5
Figure 1b	SVO	42.8	84.6	3.8	131.2
Figure 8top	Ours	10.8	8.7	2.6	22.1
Figure 8top	SVO	35.6	79.6	2.9	109.1



(a) 1389 ms **(b)** 820 ms

Fig. 10 Lighting from the environment map (*left*) and the rectangle area light (*right*) with 1,024 sampling rays per-fragment. The Ward BRDF is used to render the teapot, and the sphere is rendered with mirror reflections. The progressive rendering method is used here (128 passes with $\lambda = 8$). The sub-caption indicates the total rendering timing (including representation constructing timing)



(a) 5110 ms **(b)** 4030 ms

Fig. 11 The Sponza model under area lights with path tracing. The progressive rendering method is used here. A 256^3 size of the uniform grid is used here, due to complex scene objects. Low performance is caused by the high depth-level of A-buffers and the large size of the uniform grid. The sub-caption indicates the total rendering timing (including representation constructing timing)

formance, which results in the fact that almost all the ray-tracing-based illumination methods, even Monte-Carlo path tracing, can be implemented using our ray-tracing framework.

5.3 Other results

In Fig. 10, we present two examples with reflections and soft shadows by sampling the environment map (left) and the area light (right). The BRDF importance sampling technique [32] is employed for generating sampling rays on the teapot in the left figure.

Similar to other voxel-based methods, our solution is suitable for rendering small-scale and near-field illumination, due to the limitations of GPU capabilities. Yet, this does not mean that our solution can only be applied for rendering simple scenes. Figure 11 gives our path tracing results of the Sponza model under rectangle area lights.

6 Conclusion and future work

We presented a 100 % rasterization-based ray-tracing framework in this paper. Uniform voxel grids and linked-list A-buffers were employed to represent complex scenes accurately, and a new ray-intersection method based on such scene representations was developed for tracing rays. Most global light transport methods could be implemented in the fragment shader. Our method has better tradeoff between accuracy and performance, compared with existing solutions based on GPU rasterization. Note that all secondary effects are generated by tracing rays in our method. Actually, our method can benefit the traditional OpenGL/Direct3D applications. For example, our solution can be applied to render only mirror reflections/refractions, and shadows can still be generated using shadow mapping. Therefore, we believe that our method can be directly applied in computer games or utilized as a fast preview method for sophisticated lighting simulations in off-line rendering.

As a sampling-based method, our method certainly has some limitations, and there exist some future space to improve the quality and the performance. Water-tight objects are required for tracing rays accurately, and some gracile objects might be missed in A-buffers due to limited resolutions. As for some large and complex scenes, the sizes of uniform voxel grid and A-buffers, as well as the layer depth of A-buffers, would be large to keep high-precision ray tracing, which would result in memory and performance problems or even beyond the capability of current GPUs. It might be solved using multiple-layer or dynamic-loaded buffers. Furthermore, utilizing multiple-type buffers (static and dynamic objects, near and far objects, coarse and precise representation, etc) could also accelerate representation construction and ray-scene intersections too. Moreover, replacing our uniform voxel grids with SVOs could reduce memory costs for large static scenes too. Finally, based on our framework, in future we will further study the implementation of more complicated global illumination methods, such as photon mapping, bidirectional path tracing, and so on.

Acknowledgments We would like to thank the anonymous reviewers for their constructive comments. We thank Zhao Dong for his valuable suggestions. This work was supported jointly by Nation Nature Science Foundation of China (No. 61003132) and 973 Program of China (No. 2011CB706900).

References

1. Wald, I., Ize, T., Kensler, A., Knoll, A., Parker, S.: Ray tracing animated scenes using coherent grid traversal. ACM Trans. Graph. **25**(3), 485–493 (1999)
2. Carpenter, L.: The A-buffer, an antialiased hidden surface method. In: Proceedings of the 11th annual conference on computer graphics and interactive techniques (SIGGRAPH'84), pp. 103–108 (1984)
3. Yang, C., Hensley, J., Grun, H., Thibieroz, N.: Real-time concurrent linked list construction on the gpu. In: Proceedings of the 21st Eurographics conference on Rendering(EGSR'10), pp. 1297–1304 (2010)
4. Whitted, T.: An improved illumination model for shaded display. Commun. ACM **23**(6), 343–349 (1980)
5. Zhukov, S., Iones, A., Kronin, G.: Proceedings of the Eurographics, Workshop, pp. 45–55 (1998)
6. Kajiya, J.: The rendering equation. In: Proceedings of the 13th annual conference on Computer graphics and interactive techniques (SIGGRAPH'86), pp. 143–150 (1986)
7. Wang, R., Wang, R., Zhou, K., Pan, M., Bao, H.: An efficient GPU-based approach for interactive global illumination. ACM Trans. Graph. **28**(3), 1–8 (2009) (artical no. 91)
8. Aila, T., Karras, T.: Architecture considerations for tracing incoherent rays. In: Proceedings of the Conference on High Performance Graphics (HPG'10), pp:112–122 (2010)
9. Aila, T., Laine, S.: Understanding the efficiency of ray traversal on GPUs. In: Proceedings of the Conference on High Performance Graphics (HPG'09), pp:113–122 (2009)
10. Zhou, K., Hou, Q., Wang, R. Guo. B.: Real-time KD-tree construction on graphics hardware. ACM Trans. Graph. **27**(5), 1–11 (2008) (artical no. 126)
11. Purcell, T., Buck, I., Mark, W., Hanrahan, P.: Ray tracing on programmable graphics hardware. ACM Trans. Graph. **21**(3), 703–712 (2002)
12. Parker, S., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., Mcallister, D., Mcguire, M., Morley, K., Robison, A., Stich, M.: Optix: a general purpose ray tracing engine. ACM Trans. Graph. **29**(4), 1–13 (2010) (artical no. 66)
13. Umenhoffer, T., Patow, G., Kalos, L.: Robust multiple specular reflections and refractions. GPU Gems **3**, 387–407 (2010)
14. Rosen, P., Popescu, V., Hayward, K., Wyman, C.: Non-pinhole approximations for interactive rendering. IEEE Comput. Graph. Appl. **31**(6), 68–83 (2011)
15. Wyman, C.: Interactive image-space refraction of nearby geometry. In: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia (GRAPHITE'05), pp. 205–211 (2005)
16. Kalos, L., Umenhoffer, T.: Specular effects on the GPU: state of the art. Comput. Graph. Forum **28**(6), 1586–1617 (2009)
17. Yao, C., Wang, B., Chan, B., Yong, J., Paul, J.: Multi-image based photon tracing for interactive global illumination of dynamic scenes. Comput. Graph. Forum **29**(4), 1315–1324 (2010)
18. Burger, K., Hertel, S., Kruger, J., Westermann, R.: GPU rendering of secondary effects. In: Proceedings of International Workshop on Vision, Modeling and Visualization (VMV'07), pp. 51–61 (2007)
19. Niebner, M., Schafer, H., Stamminger, M.: Fast indirect illumination using layered depth images. Vis. Comput. **26**(6), 679–686 (2010)
20. Zhang, C., Hsieh, H, Shen, H.: Real-time reflections on curved objects using layered depth textures. In Proceedings of IADIS International Conference on Computer Graphis and Visualization (2008)
21. Kaplanyan, A., Dachsbacher, C.: Cascaded light propagation volumes for real-time indirect illumination. In: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games (I3D'10), pp. 99–107 (2010)
22. Thiedemann, S., Henrich, N., Muller, S.: Voxel-based global illumination. In: Proceedings of the 2011 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games (I3D'10), pp. 103–110 (2011)
23. Papaioannou, G., Menexi, M., Papadopoulos, C.: Real-time volume-based ambient occlusion. IEEE Trans. Vis. Comput. Graph. **16**(5), 752–762 (2010)
24. Laine, S., Karras, T.: Efficient sparse voxel octrees. In: Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D'10), pp. 55–63 (2010)
25. Crassin, C., Neyret, F., Lefebvre, S., Eisemann, E.: Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In: Proceedings of the 2009 symposium on Interactive 3D graphics and games, pp. 15–22 (2009)
26. Crassin, C., Neyret, F., Sainz, M., Green, S., Eisemann, E.: Interactive indirect illumination using voxel cone tracing. Comput. Graph. Forum **30**(7), 1921–1930 (2011)
27. Novak, J., Dachsbacher, C.: Rasterized bounding volume hierarchies. Comput. Graph. Forum **31**(2), 403–412 (2012)
28. Tokuyoshi, Y., Ogaki, S.: Real-time bidirectional path tracing via rasterization. In: Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games(I3D'12), 183–190 (2012)
29. Crassin, C., Green, S.: Octree-based sparse voxelization using the GPU hardware rasterizer. In: OpenGL Insights, pp. 303–318 (2012)
30. Amanatides, J., Woo, A.: A fast voxel traversal algorithm for ray tracing. In: Proceedings of EuroGraphics'87, pp. 3–10 (1987)
31. Policarpo, F., Oliveira, M., Comba, J.: Real-time relief mapping on arbitrary polygonal surfaces. In: Proceedings of the 2005 symposium on Interactive 3D graphics and games(I3D'05), pp. 155–162 (2005)
32. Colbert, M., Krivanek, J.: Real-time shading with filtered importance sampling. Comput. Graph. Forum **27**(4), 1147–1154 (2008)

Wei Hu is currently an associate professor of computer science and technology, at Beijing University of Chemical Technology, China. He received his Ph.D. degree from Tsinghua Univ. in 2006. His research interests include computer graphics, computational photography, etc.

Yangyu Huang is currently a M.S. candidate in computer science in Beijing University of Chemical Technology, China. His research interests include computer graphics and computer vision.

Fan Zhang is currently an associate professor of electronic and information engineering at the Beijing University of Chemical Technology. He received the Ph.D. degree in signal and information processing from Institute of Electronics, Chinese Academy of Science, Beijing, China, in 2008. His research interests are synthetic aperture radar signal processing, high performance computing and scientific visualization.

Guodong Yuan is currently an assistant professor in computer science at Beijing Information Science and Technology University. He received his Ph.D. degree from Tsinghua University in 2008. His research interests include computer graphics and visualization.

Wei Li is currently with the College of Information Science and Technology at Beijing University of Chemical Technology, Beijing, China. He received his Ph.D. degree in electrical and computer engineering from Mississippi State University, Starkville, MS, USA, in 2012. Subsequently, he spent 1 year as a Postdoctoral Researcher at the University of California, Davis, CA, USA. His research interests include statistical pattern recognition, hyperspectral image analysis, and data compression.