Viewport-Resolution Independent Anti-Aliased Ray Marching on Interior Faces in Cube-Map Space

Tianchen Xu
[1]State Key Lab of CS, Institute of Software, Chinese Academy of Sciences (CAS) & Univ. of CAS, China
[2]Advanced Micro Devices, Inc. (AMD)
xutc@ios.ac.cn

Wei Zeng
School of Computer Science, Sichuan University, China
veerzeng@163.com

Enhua Wu*
[1]State Key Lab of CS, Institute of Software, Chinese Academy of Sciences & University of CAS, China
[3]FST, University of Macau, China
weh@ios.ac.cn

Figure 1: Fast real-time (left) mesh-volume rendering using our ray marching on interior faces in cube-map space (mesh Dragon © Stanford University Computer Graphics Laboratory & environment map Tropical Beach © Blochi), and (right) multi-volumes rendering using our ray-traced hybrid method with inter-occlusions (meshes Bunny etc. © Stanford CG Lab & environment map Helipad Afternoon © Blochi)

ABSTRACT

This paper presents a novel approach to anti-aliased ray marching by indirect shading in cube-map space. Our volume renderer firstly performs ray marching on each visible interior pixel of a maximum-resolution-limited cube map, and then resamples (usually up-scales) the cube imposter in viewport space. By this viewport-resolution-independent strategy, developers can improve both ray-marching performance and its quality of anti-aliasing when allowing larger marching strides. Moreover, our solution also covers depth-occlusion anti-aliasing for mixed mesh-volume rendering, cube-map level-of-details (LOD) optimization for a further performance boost, and multiple-volume rendering by leveraging the GPU inline ray tracing. Besides, our implementation is developer-friendly and the performance-quality tradeoff determined by the parameter configuration is easily controllable.

SA '21 Technical Communications, December 14–17, 2021, Tokyo, Japan

ACM ISBN 978-1-4503-9073-6/21/12…$15.00
https://doi.org/10.1145/3478512.3488598

CCS CONCEPTS

• **Computing methodologies** → **Rendering**; *Visibility*.

KEYWORDS

Anti-aliasing, cube map, ray marching, ray tracing, real-time rendering, volume rendering

ACM Reference Format:
Tianchen Xu, Wei Zeng, and Enhua Wu. 2021. Viewport-Resolution Independent Anti-Aliased Ray Marching on Interior Faces in Cube-Map Space. In *SIGGRAPH Asia 2021 Technical Communications (SA '21 Technical Communications), December 14–17, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3478512.3488598

1 INTRODUCTION

Ray marching is a typical method for volume-data visualizations, which can convey the rendering effects of clouds, smoke fluid, and other thick translucent objects. In many use cases of real-time volume rendering, especially for fluid, the physics-based simulation is already a costly phase, and developers expect to avoid the rendering phase being another performance bottleneck. However, compared to trivial mesh-surface shading, ray-marched volume rendering is more compute-intensive, due to the multiple sampling accesses to a 3D volume texture for each pixel. In traditional screen-space ray marching, the sampling count has sensitive impacts on the rendering performance and quality. Too sparse sampling for a

high-resolution viewport would cause obvious aliasing artifacts, because the depth sampling frequency is much less than the breadth frequency. Hence, developers have to force coupling the lower bound of the sampling count to the viewport resolution, while it results in serious performance drops. Therefore, it is troublesome for developers to balance the performance-quality tradeoffs, and a viewport-resolution decoupled ray-marching method is urgently needed.

It is not easy to control the ray marching performance by the directly screen-space scheme, we bethought of ray marching onto a fixed-resolution imposter indirectly. However, it is a little unreliable to use a 2D billboard as the imposter, since the bounding rectangle of a projected volume is varying. Besides, simply upscaling a billboard in screen space may cause texel aliasing. Then, we come up with the idea of ray marching on cube-map faces, a special form of texture-space shading. These years, texture-space shading is popularly used in virtual reality for spatial-temporal coherence, and it can decouple the performance impact of viewport resolution [Hillesland and Yang 2016], but it will introduce duplicated workloads of processing the invisible texels. Therefore, visibility culling is often the necessary accompanying process for texture-space shading.

The major technical contribution of this paper is the proposal of our indirect cube-map ray marching scheme, as well as the related techniques on visibility culling, mesh-occlusion anti-aliasing, adaptive level of details (LOD) optimization, and multi-volumes rendering using hardware ray tracing.

2 RELATED WORK

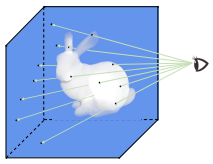
Since ray marching [Tuy and Tuy 1984] was proposed to a 3D object to 2D display, many research works have drawn attention to the volume rendering [Drebin et al. 1988], which is performed by simulating the absorption of light along the ray path to the viewer. To gain performance, several optimizations have been used, including a volume bricking scheme and a shallow data hierarchy [Parker et al. 2005]. Later, the solution by [Crassin et al. 2009] is based on an adaptive data representation depending on the current view and occlusion information coupled to an efficient ray-marching rendering algorithm. Furthermore, [Morrical et al. 2019] introduced a method for rendering unstructured meshes using a combination of a coarse spatial acceleration structure and hardware-accelerated ray marching.

In addition to the performance problems, ray marching can also cause aliasing. [Muñoz 2014] applied higher order adaptive solvers into faster convergence, which can be used to reduce the bias in some cases, but the errors may still not be acceptable for certain applications. In recent years, new statistically unbiased Monte Carlo methods were proposed [Novák et al. 2018][Kettunen et al. 2021], which randomly query the volume densities along a ray and compute a visibility estimated from these values.

Above all, we can find that most prior achievements have focused on global illumination and transmittance estimation models, while their sampling scheme of ray marching is still vanilla, which initiates rays in screen space. Therefore, we still have potential for improvement to generate anti-aliased volumes in an efficient way for real-time rendering, but it would be challenging.

3 TECHNIQUES

The basic idea of our method is to deploy ray marching for each pixel on the interior faces of a cube-map imposter, and then upscale the marching result on the viewport by drawing the cube interior faces with cube mapping (texture sampling by direction vectors). We propose a 3-pass rendering pipeline:



Light map generation pass From each voxel of a 3D light map, we cast a ray to the light source, and perform ray marching in the light-map space for the transmittance.

Cube ray marching pass As the core pass, to each pixel on the cube-map faces, we cast a ray from the viewer position, and perform ray marching in the local volume space (domain $[-1, 1]$) with the inverse world transformation.

Cube resampling pass We draw the interior faces of a cube on the viewport and resample the ray-marched texels with simple cube mapping.

Then, we will discuss some detailed techniques that can tackle the new problems introduced by the pipeline above.

3.1 Interior versus Exterior Faces

Shading on the surfaces (exterior faces) is a straightforward thought when we mention cube-map space ray marching. In this scheme, there are at most 3 visible faces only. However, it would cause an obvious texel-aliasing artifact when the viewer is close to a cube corner (Figure 2 (a)), due to the under sampling in the near-viewer zone with a perspective projection. Using the interior faces instead, the artifact can be eliminated (Figure. 2 (b)), since the texel shading rate on the far faces can be over the pixel sampling rate on the screen, while there are maximally 5 visible faces to hold more texel data. Moreover, the interior-face scheme naturally includes the case of the viewer inside the volume, and thus no extra computations need to be introduced to handle this case in particular.

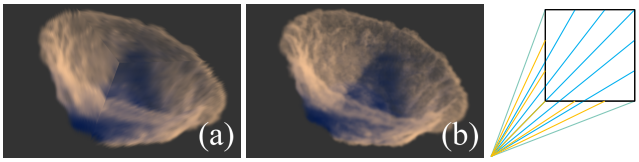


Figure 2: (a) Exterior-face scheme causes texel-aliasing, and (b) interior-face scheme can hold more rays of data to eliminate the artifact.

For the interior-face scheme, the invisible faces can be culled per face in advance. We generate an 8-bit visible-face mask, and each visible bit $b_{\text{visibility}}(i)$ for cube face $i = 0, \ldots, 5$ can be fast calculated by the following conditional equation:

$$b_{\text{visibility}}(i) = \begin{cases} 1, & (\boldsymbol{E}(\lfloor i/2 \rfloor) > -1) \wedge (i \& 1) \\ 1, & (\boldsymbol{E}(\lfloor i/2 \rfloor) < 1) \wedge \neg(i \& 1) \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where $\boldsymbol{E}$ is the viewer position transformed into the local cube map space, and its index (function argument) refers to components

$x = 0$, $y = 1$, and $z = 2$ respectively; the bitwise-AND operator is denoted by '&'.

3.2 Mesh Occlusions

When rendering a scene with mixed meshes and volumes, the draw-process of opaque meshes will generate a depth buffer (z-buffer) for occlusion culling called z-test. During ray marching, we obtain the mesh-occlusion position derived by the depth-unprojection to clamp the max ray length $T_{\max}$ for culling the occluded ray steps. If using direct screen-space ray marching, the z-buffer and the viewport are in the same space whose pixels are one-to-one coupled. Meanwhile, using our cube-map space ray marching, the ray-casted space is totally different from the z-buffer with the unmatched shading and sampling rates, thus causing the mesh-occlusion artifact, as shown in Figure 3 (a).

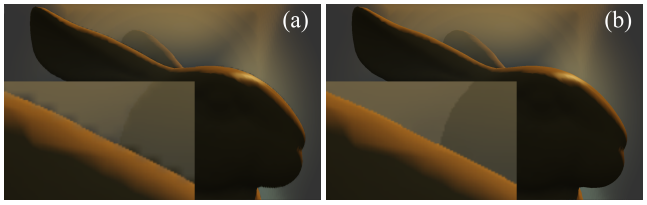


Figure 3: (a) Mesh-occlusion artifact due to different spaces of cube map and z-buffer, and (b) our bilateral filter applied instead of simple bilinear sampling

In order to eliminate the artifact (Figure 3 (b)), we make a 2×2 tiny bilateral filter to the cube-map samples during the resampling pass. Instead of hardware bilinear sampling, we gather the 2×2 raw samples that would be used for bilinear interpolation when sampling a texture, and then filter each samples by the following formula:

$$\boldsymbol{s}(i) = \frac{\boldsymbol{s}_{\text{raw}}(i) w_{\text{z}}(z_{\text{cube}}(i), z_{\text{buffer}}) w_{\text{bilinear}}(i)}{\sum_{j=1}^{4} w_{\text{z}}(z_{\text{cube}}(j), z_{\text{buffer}}) w_{\text{bilinear}}(j)}$$

$$w_{\text{z}}(z_{\text{cube}}, z_{\text{buffer}}) = \max\left(1 - k_{\text{z}} |z_{\text{buffer}} - z_{\text{cube}}|, 0\right)$$

where $\boldsymbol{s}_{\text{raw}}$ and w_{bilinear} denote the raw sample i and its bilinear weight correspondingly. w_{z} is the edge-stopping function of the bilateral filter, in which z_{buffer} and z_{cube} are linear depth values derived from the z-buffer and cube map respectively, and k_{z} is an adjustment coefficient (0.5 by default in our experiments). Hence, we need to introduce another cube depth map and blit (transfer with a nearest-filter sampler) the z-buffer data into it during the cube ray marching pass.

3.3 Adaptive Level of Details

When the volume goes away from the viewer, we can reduce the ray marching resolution to a coarser level for the further performance boost. Certainly, we can always calculate the ideal values of the cube map resolution and ray-marching sampling count according to the volume position. However, developers usually expect to limit the upper bounds of the cube map resolution (the size at MIP-level 0) and the sampling count for performance-quality control. Therefore, we design a process to compute the optimized cube-map MIP-level that satisfies the aforementioned requirements:

1) We firstly project the 8 corner vertices of the cube into viewport space and store them for vertex indexing later.
2) We couple the vertex pairs into edges by indexing the projected vertices, and evaluate the edge lengths in pixels.
3) We conservatively choose the maximum edge length of the cube and scale it as the ideal cube-map resolution S_{ideal}, and derive the ideal sampling amount N_{ideal} (float number) as well:

$$\begin{cases} S_{\text{ideal}} = \dfrac{\max_{i=1,\ldots,12}\{\|\boldsymbol{P}(e_{\text{end}}(i)) - \boldsymbol{P}(e_{\text{begin}}(i))\|\}}{k_{\text{u}}} \\ N_{\text{ideal}} = \dfrac{k_{\text{c}} S_{\text{ideal}}}{\sqrt{3}} \end{cases}$$

where $\boldsymbol{P}$ denotes the projected vertex position and its edge begin and end indices are denoted by e_{begin} and e_{end}; k_{u} and k_{c} are the extra user-specified screen-upscaling and sampling-count scaling coefficients respectively.
4) We clamp the ideal sampling count N_{ideal} with the user-specified maximum sampling count $N_{\max}$ to get the actual sampling count N, and inversely derive the cube-map resolution S.

$$\begin{cases} N = \min(\lceil N_{\text{ideal}} \rceil, N_{\max}) \\ S = \dfrac{\sqrt{3}\min(N_{\text{ideal}}, N)}{k_{\text{c}}} \end{cases}$$

5) The cube-map MIP-level (LOD) can be finally evaluated by $\ell = \max(\lfloor \log_2 S/S_0 \rfloor)$ with the finest texture size S_0 at MIP-level 0.

3.4 Ray-traced Multi-volumes Rendering

For real-time multiple volumes rendering, we make use of indirect compute to archive volume-granularity visibility culling, and hardware ray tracing to guarantee order-independent transparency.

Firstly, we add a preemptive volume-culling pass by dispatching compute shaders. Like GPU amplification and mesh shader scheme, 8 threads per volume are invoked to process viewport-visibility culling of the entire volumes, vertex projections (step 1 in section 3.3), edge length estimations with 2 edges per thread (3.3 step 2), cube-map LOD estimations (3.3 steps 3–5), volume classifications, and cube-face visibility mask generations (3.1 Eqn. (1)), in sequence. Here, a new volume classification step is added after the LOD determined. It first estimates the total visible pixels of the cube map and the pixel count of the directly projected volume, and then compare them to determine the optimized scheme between the indirect cube-map and direct screen-space ray marching. The final output of this pass can be encoded into a 16-byte descriptor including volume instance ID, volume source ID, ray sampling count, LOD, rendering scheme, and cube-face visibility mask.

Subsequently, we finish the cube ray marching pass for all the visible volumes marked with cube-map scheme. Finally, we draw the cube interior faces for all visible volumes as opaque objects. For each visible volume in the nearest layer, if it has the ray marched result in cube-map space, we directly fetch the result, otherwise we deploy the screen-space ray marching. Then, for each rasterized pixel, we launch inline ray query to trace the transparency-object occluded but visible volume pixels in the next layer, and shade them in the similar way as the nearest layer. We repeat the ray tracing processes using inline ray query for the remaining layers and finish the shading of all visible volumes.

4 RESULTS

In our experiments, we have designed several groups of comparative test cases to verify the quality and performance of our method. All test cases ran in a machine equipped with an AMD Radeon™ RX 6800 GPU of boost clock 2105MHZ.

For the quality verification, as shown in Figure 4, the rendering results of our method are overall same to the references, while our method can provide more smoothly anti-aliased shading in some detailed pixel regions. As the upper bound of the input volume resolution is constrained by developers for performance control, a high screen-space resolution would not provide a sensitive quality improvement, which is just over-sampling. Contrastively, our method executes the ray marching shading at an appropriate resolution.

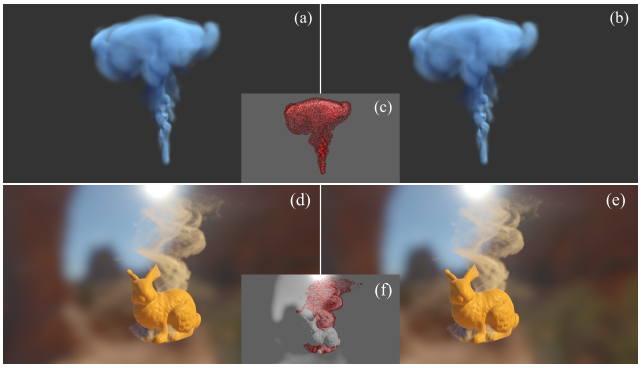


Figure 4: Rendering results of (a–c) fluid smoke and (d–f) devil and bunny using (a)(d) our cube-map space ray marching and (b)(e) direct screen-space ray marching respectively; (c)(f) difference visualizations (mesh Bunny © Stanford CG Lab & environment map Arches PineTree © Blochi)

For the performance, we list the comparative performance results in Table 1. We can find that the performance of direct screen-space ray marching (DSSRM) drops rapidly with increasing viewport resolution, while our cube-map-space ray marching (CMSRM) can keep nearly constant performance in high-resolution cases, and our adaptively hybrid method for multi-volumes can even win in all resolution cases.

5 CONCLUSIONS

In this paper, we propose a novel approach to anti-aliased ray marching by indirect shading in cube-map space. Our method leverages texture-space shading to prevent volume rendering from rapid performance drops in high-resolution viewport cases without any quality tradeoff. During the implementation, we also propose some detailed techniques to overcome the new problems introduced by the cube ray marching scheme, such as interior-face shading and mesh occlusions, and to further boost the performance, such as adaptive LODs and ray-traced hybrid multi-volumes.

As we have only considered the mesh-volume occlusion cases, but lack handling the volume-volume overlap cases, which is the main potential limitation of our current proposal. In terms of the future work, we will explore the per-texel visibility culling and LOD for our scheme by taking advantage of sampler feedback feature of the latest GPU architectures, and integrate more advanced transmittance estimation algorithms into our ray marching.

Table 1: Time-cost statistics (in ms) of our tested applications (grid-size 256^3 for single-volume cases and 128^3 for multi-volumes); CMSRM extra memory cost over DSSRM

Test case	View res.	CMSRM	DSSRM	Ex-mem.
Fluid smoke	1080p	7.63	7.87	5.98MB
	2K	7.69	8.13	
	4K	7.69	8.93	
Devil bunny	1080p	1.61	1.99	5.98MB
	2K	1.68	2.32	
	4K	1.84	3.34	
Cloud dragon	1080p	1.67	2.12	5.98MB
	2K	1.81	2.51	
	4K	1.96	4.06	
Multi-volumes	1080p	2.43	3.15	95.625MB
	2K	3.46	5.05	
	4K	6.02	11.49	

ACKNOWLEDGMENTS

The authors would like to thank the associate editors and all anonymous reviewers for their valuable comments. This research is supported by NSFC (62072449, 61632003), Macao FDCT (0018/2019/AKP), and AMD.

REFERENCES

Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. 2009. GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (Boston, Massachusetts) *(I3D '09)*. Association for Computing Machinery, New York, NY, USA, 15–22. https://doi.org/10.1145/1507149.1507152

Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. 1988. Volume Rendering. *SIGGRAPH Comput. Graph.* 22, 4 (June 1988), 65–74. https://doi.org/10.1145/378456.378484

K. E. Hillesland and J. C. Yang. 2016. Texel Shading. In *Proceedings of the 37th Annual Conference of the European Association for Computer Graphics: Short Papers* (Lisbon, Portugal) *(EG '16)*. Eurographics Association, Goslar, DEU, 73–76.

Markus Kettunen, Eugene D'Eon, Jacopo Pantaleoni, and Jan Novák. 2021. An Unbiased Ray-Marching Transmittance Estimator. *ACM Trans. Graph.* 40, 4, Article 137 (July 2021), 20 pages. https://doi.org/10.1145/3450626.3459937

Nate Morrical, Will Usher, Ingo Wald, and Valerio Pascucci. 2019. Efficient Space Skipping and Adaptive Sampling of Unstructured Volumes Using Hardware Accelerated Ray Tracing. In *2019 IEEE Visualization Conference (VIS)*. IEEE, Vancouver, BC, Canada, 256–260. https://doi.org/10.1109/VISUAL.2019.8933539

Adolfo Muñoz. 2014. Higher Order Ray Marching. *Computer Graphics Forum* 33, 8 (2014), 167–176. https://doi.org/10.1111/cgf.12424

Jan Novák, Iliyan Georgiev, Johannes Hanika, and Wojciech Jarosz. 2018. Monte Carlo methods for volumetric light transport simulation. *Computer Graphics Forum* 37, 2 (2018), 551–576. https://doi.org/10.1111/cgf.13383

Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles Hansen, and Peter Shirley. 2005. Interactive Ray Tracing for Volume Visualization. In *ACM SIGGRAPH 2005 Courses* (Los Angeles, California) *(SIGGRAPH '05)*. Association for Computing Machinery, New York, NY, USA, 15–es. https://doi.org/10.1145/1198555.1198754

Heang K. Tuy and Lee Tan Tuy. 1984. Direct 2-D display of 3-D objects. *IEEE Computer Graphics and Applications* 4, 10 (1984), 29–34. https://doi.org/10.1109/MCG.1984.6429333