

# Sparse Volume Rendering using Hardware Ray Tracing and Block Walking

Mehmet Oguz Derin  
Morgenrot, Inc.  
Turkey  
oguz@morgenrot.net

Takahiro Harada  
Morgenrot, Inc.,  
Advanced Micro Devices,  
Inc.  
USA  
takahiro.harada@amd.com

Yusuke Takeda  
Hokkaido Univ.  
Japan  
ytakeda@sci.hokudai.ac.jp

Yasuhiro Iba  
Hokkaido Univ.  
Japan  
iba@sci.hokudai.ac.jp



Figure 1: Rgb volume datas captured by a tomography scanner rendered using the method on an AMD Radeon™Pro W6800.

## ABSTRACT

We propose a method to render sparse volumetric data using ray-tracing hardware efficiently. To realize this, we introduce a novel data structure, traversal algorithm, and density encoding that allows for an annotated BVH representation. In order to avoid API calls to ray tracing hardware which reduces the efficiency in the rendering, we propose the block walking for which we store information about adjacent nodes in each BVH node's corresponding field, taking advantage of the knowledge of the content layout. Doing so enables us to traverse the tree more efficiently without repeatedly accessing the spatial acceleration structure maintained by the driver. We demonstrate that our method achieves higher performance and scalability with little memory overhead, enabling interactive rendering of volumetric data.

## CCS CONCEPTS

• Computing methodologies → Rendering; Ray tracing.

## KEYWORDS

ray tracing, volume rendering, ray tracing hardware

## ACM Reference Format:

Mehmet Oguz Derin, Takahiro Harada, Yusuke Takeda, and Yasuhiro Iba. 2021. Sparse Volume Rendering using Hardware Ray Tracing and Block Walking. 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/10.1145/3478512.3488608>

## 1 INTRODUCTION

As the resolution and fidelity of volumetric capture devices increase, visualization becomes an even more critical component to explore the content [Takeda et al. 2021]. However, as the data distribution is unpredictable ahead of time, we need an algorithm that can help swift through such irregular data fast enough to trace rays for effects that can simulate real-life interaction of light with these mediums. Some modern GPU hardware recently started to have ray-tracing cores that natively support tracing non-uniform, axis-aligned bounding boxes for triangles [Advanced Micro Devices, Inc. 2020] [NVIDIA 2018].

Although the ray tracing hardware in modern GPUs are designed to accelerate ray-triangle intersection acceleration, there already are some works using the hardware to render structured volume data and unstructured volume data [Ganter and Manzke 2019] [Wald et al. 2021]. The focus of our paper is the visualization of the structured volume data, and thus, [Ganter and Manzke 2019] is relevant for our work. To reduce the number of ray casting operations invoked to visualize a volume, they group voxels to make a larger cell whose bounding volume is passed to the GPU. Our paper approaches the same problem using a different method that relies on additional information stored in each block to reduce ray casting operations.

Our approach's core idea is to facilitate access to the top-level blocks in our tree structure through hardware ray-tracing while providing a shortcut for walking into adjacent blocks for visualizing volumetric data, in our case consisting of RGBA voxels. Thus, our data does not require a transfer function as the input is not a scalar

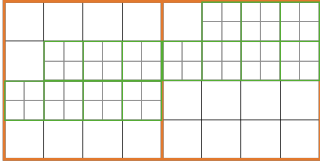
*SA '21 Technical Communications, December 14–17, 2021, Tokyo, Japan*

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in

*SIGGRAPH Asia 2021 Technical Communications (SA '21 Technical Communications)*, December 14–17, 2021, Tokyo, Japan.

<https://doi.org/10.1145/3478512.3488608>.



**Figure 2: Blocks and subblocks.** We allocate subblocks, which are uniform three-dimensional grids, each containing  $\#^3$  voxels, around the voxels where the data exists. <sup>3</sup> Subblocks are (at most) grouped to make a block, although it only stores indices of active subblocks for memory and traversal efficiency.

that needs such. First, we build blocks, which are collections of subblocks that consist of dense voxels. After, we request an acceleration structure compatible with ray tracing access from API. Then, we expose each block’s build-time generated adjacency information to compute kernels to act as shortcuts that improve performance as the data size grows over exhaustively querying the acceleration structure through hardware. Finally, we accelerate our algorithm by utilizing the mask field available in most ray-tracing APIs to store binned density representation.

## 2 SPARSE VOLUME RENDERING

We first describe the application of hardware ray tracing to volume rendering, which is the basis of our method. Then we extend it by adding block walking to make it efficient alongside hardware-accelerated space skipping. Finally, we propose block culling in which density range information is encoded in bits used to mask the traversal in the ray-tracing API.

### 2.1 Rendering using Hardware Ray Tracing

**2.1.1 Data Structure.** Hardware ray-tracing (RT) APIs such as DXR and Vulkan RT take a list of triangles or Axis aligned bounding boxes (AABBs) from the user, then it builds an acceleration structure in the driver, which we use to compute the closest intersection. Ray query API provides a way to use the ray-tracing hardware inside the compute stage to obtain the closest hit AABB (or triangle). We use this ray query API to accelerate the rendering of sparse volumes. We could pass every single voxel as an AABB which is too inefficient. Thus, we group voxels into uniform-sized blocks, whose AABB we pass to the API. We employed a two-level sparse data structure for a block. A block contains indices of subblocks. Subblock is a dense representation of voxels as shown in Fig. 4. This data structure not only allows us to save memory but improve the rendering efficiency.

**2.1.2 Rendering.** The rendering is implemented in a compute kernel where we use a hardware ray tracing query to find the first intersection of the ray against the blocks. After finding the block, we process the block immediately, although it could be deferred if preferred. Since our blocks and subblocks make up a two-level hierarchical grid, we use two-level DDA to traverse the voxel data on the ray [Bresenham 1965] [Museth 2014]. Once we finish traversing a single block, we launch another ray tracing query to find the next block, then traverse the voxels in the block. We repeat this until we

exit the volume or reach the user-selected termination condition, alpha becoming 1, or first hit depending on the rendering mode specified.

### 2.2 Block Walking

As data shape is unpredictable, there can be scenes with dense regions and sparse regions at different points in the grid. In addition, user-controlled runtime filtering and blending decrease the predictability of whether the first hit block will yield a hit voxel. Thus, we need to access AABBs along the ray repeatedly by executing multiple ray-tracing queries. This pattern can become very inefficient as we scale the data.

We propose block walking, which eliminates hardware ray tracing queries to realize an efficient volume rendering. The algorithm with block walking enabled is illustrated in Fig. 3 (a). Although the first ray query cannot be avoided, most of the subsequent ray queries can be avoided by storing adjacency information of each block. The purpose of this is to take a shortcut into adjacent blocks without launching another query, whose overhead is not negligible. As a result, this version, after HDDA, introduces the following condition: if parametric  $\ell$  after HDDA is inside of a neighboring block, we start an HDDA directly without going back to the ray-tracing hardware queries as illustrated in Fig. 3 (b). If there is no adjacent block, we launch a ray query to find the next closest block if parametric  $\ell$  is still less than the maximum  $\ell$ .

### 2.3 Block Culling

In contrast to the lack of acceleration structure access input for traversal strategy, ray-tracing APIs have a field for guiding the intersection function to perform runtime culling of top-level acceleration structure elements. This culling mask field is an integer that API uses as a bit field to allow implementations to flag specific instances, which are top-level primitives.

We utilize the density member of stored ranges to compute a culling mask for each block we feed into the RT API. To achieve this, we bin the density ranges to the maximum number of flags that API allows. This way, we can skip over blocks that might not meet the user-specified content thresholds.

Mask calculation happens as follows: we compute max and min of the voxels in a subblock and block, use them to compute  $\langle \delta = b = 5 : \rangle A(\langle \delta = /3), \langle 0G_b = 5 : \rangle A(\langle 0G/3)$ , then flip the bits between these, where 3 is the quantization divider. We calculate this quantization divider by dividing the maximum possible density value by the maximum number of bits the API allows for acceleration structure annotation. We use the ranges we store in blocks and subblocks to skip regions during traversal if these values are not in the range of the interest. This method is effective when the user removes a large part of the data, but the modification is more fine-grained than what API allows it to represent.

## 3 IMPLEMENTATION

We have implemented our algorithms using Vulkan 1.2 with buffer device address, acceleration structure, ray query extensions, and GLSL with corresponding extensions. Ray generation, traversal, and offscreen target resolve all happen in a compute kernel stage. Then, depending on launch configuration, this offscreen target

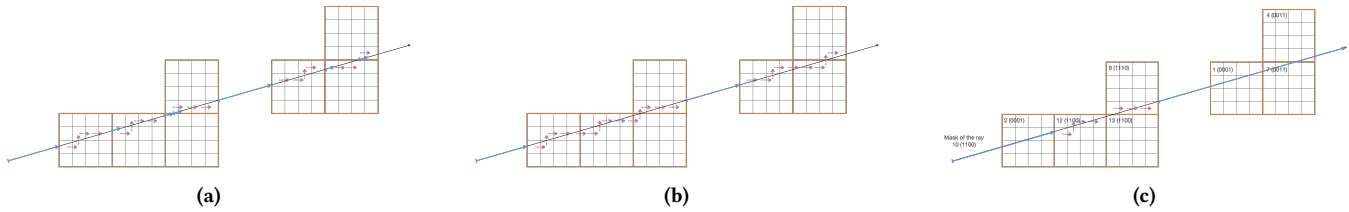


Figure 3: (a) Hardware Ray tracing algorithm. Single ray tracing is visualized. The blue and pink arrows show the hardware ray tracing queries and DDA steps. (b) Hardware Ray-tracing algorithm with block walking. Ray traversal with acceleration structure access, block walking, and re-access of acceleration structure is visualized. We start by accessing the acceleration structure through hardware to get the closest block to the ray start after the minimum  $\ell$ . Once we reach the boundary voxels found in boundary subblocks, we then walk into the adjacent block directly using the embedded spatial information if it is present. If not, we inquire about the next closest block through hardware. This process repeats until our traversal reaches satisfying conditions. (c) Hardware Ray-tracing algorithm with block culling. Ray traversal with a culling mask is visualized. In addition to our steps for hardware ray-tracing, we provide a culling mask computed from the density range specified by the user. When building the acceleration structure, the blocks are annotated with cull masks, so the hardware can skip over these blocks if the  $\#$  operation between traversal mask and block mask does not return a value larger than zero.

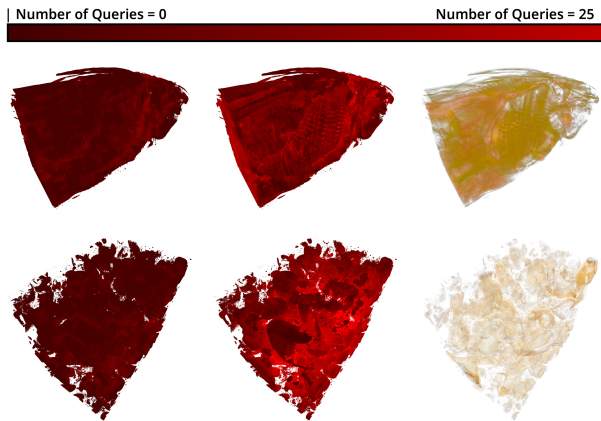


Figure 4: The number of queries it takes to reach content value for walking on (left) and walking off (middle), with desired output (left) for fish and rock data.

gets presented to the screen or exported to an image file. Blocks, subblocks, and data are all stored in Vulkan buffers. We calculate AABBs for the Vulkan acceleration structure from the data, which is accessed through ray queries. We pass the runtime configuration of the user through a push-constant structure. This structure includes the culling mask for the ray query besides settings that do not impact runtime, like clear color.

The addresses of subblocks are stored as unsigned 8-bit inside blocks. Therefore, the maximum number of subblocks that blocks contain is  $6^3$  ( $6^3 \leq 256$ ). We chose  $8^3$  as the size of individual subblocks with a parametric implementation that makes it possible to tweak this size. From these 8-bit integers, we access subblocks by multiplying the stored index with the size of subblock structure, which we embed voxel data to, appending it to buffer device address obtained from API and supplied to compute kernel, and access fields of subblock through this address.

Vulkan ray-tracing API requires cull masks to be 8-bit only. As a result, we divide our density domain max of 256 (as our data is a

four-component vector of unsigned 8-bit integers where the last component represents the density) by 32 to represent the density range of blocks as a bitfield. Thus, blocks correspond to top-level instances.

Our test data are RGBA volume data captured by a tomography device. Thus, we do not apply any transfer function to the volume data during preprocessing but adjust the density cut off for different visualization and determine member fields like minimum and maximum values of regions.

#### 4 RESULTS

We used an AMD Radeon PRO™W6800 GPU on 21.Q2.1 driver for our evaluation. The system has a AMD Ryzen Threadripper PRO 3955WX, 512 GBs of RAM, and runs Windows 10, Vulkan 1.2.

We benchmarked our framework with four different scenes that contain RGBA voxels with 8-bits per channel to evaluate the performance of algorithms with a resolution of 2048x2048 pixels. We use two modes per scene configuration. First, block traversal measures the direct impact of block-walking and block-culling algorithms through traversal of all blocks a ray intersects with, starting from closest to the camera and writing to a scalar at each block visit. Secondly, blend mode traverses voxels and sums their value until value accumulation results in enough content, which either means being opaque for early termination or containing greater than zero content for blending with the user-specified background.

We measured the performance of the kernel using Vulkan’s timestamp queries. We write a timestamp before dispatch and another one after barrier post-dispatch and extract runtime from these according to physical device-reported information. In the tables, we report the average of 128 such measurement samples.

We observe a 5-10x reduction in the number of queries (acceleration structure accesses) when walking is enabled. This reduction translates to 1.5x performance improvement on average with performance degradation on sparse cases in block traversal mode. However, we see the performance improvements shrink in blend mode due to costly access to individual voxel values. It improved all the cases except for two cases where sparsity becomes high. When we enable block culling, we see that it can bring in up to

