Eurographics Symposium on Parallel Graphics and Visualization (2021)
M. Hadwiger, M. Larsen, F. Sadlo (Editors)

Faster RTX-Accelerated Empty Space Skipping using Triangulated Active Region Boundary Geometry

Ingo Wald[1] Stefan Zellmann[2] Nate Morrical[1,3]
[1]NVIDIA [2]University of Cologne [3]SCI Institute, University of Utah

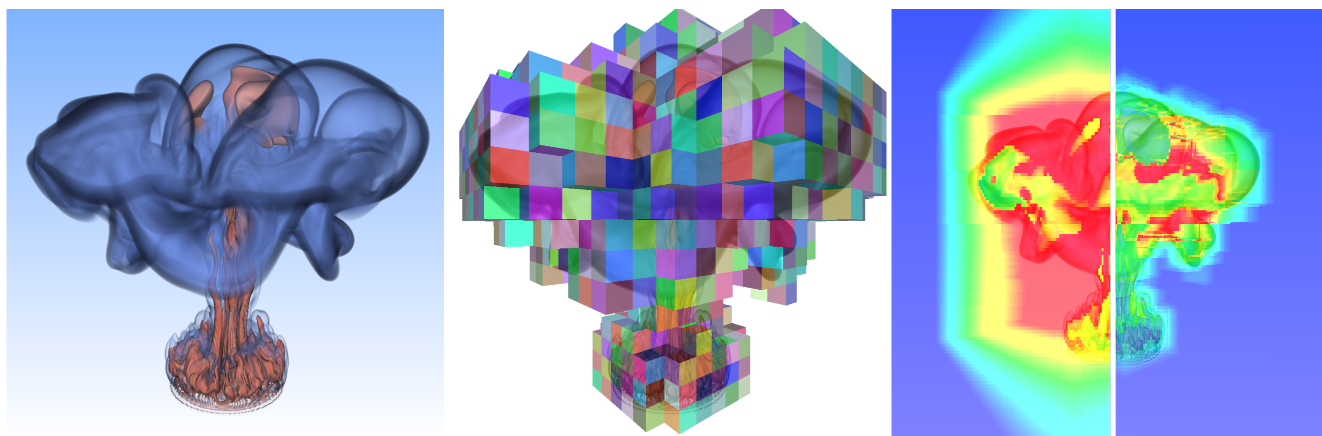

Figure 1: *An illustration of our technique on the* HEPTANE *volume data set. Left: a volume rendering of the* HEPTANE *data set using our chosen transfer function. Center: the same* HEPTANE *data set, with our low-resolution and GPU-generated boundary geometry. We use this boundary geometry to quickly and cheaply find where rays enter and respectively leave regions of non-zero opacity, thus focussing all volume integration to only those non-empty regions. Right: heat map renderings of the* HEPTANE *data set, once with our technique disabled for the left half of the image, and once with out technique enabled for the right half (blue is cheap, red is costly).*

Abstract

We describe a technique for GPU and RTX accelerated space skipping of structured volumes that improves on prior work by replacing clustered proxy boxes with a GPU-extracted triangle mesh that bounds the active regions. Unlike prior methods, our technique avoids costly clustering operations, significantly reduces data structure construction cost, and incurs less overhead when traversing active regions.

CCS Concepts

• Human-centered computing $\rightarrow$ ***Scientific visualization; Visualization techniques; • Computing methodologies*** $\rightarrow$ ***Ray tracing; Graphics processors;***

1. Introduction

Direct volume rendering is an important tool for visualizing 3D scalar data. To shade a given ray, that ray is marched through the volume by sampling the volume at discrete positions along this ray. These sampled scalar values are fed into a so-called *transfer function* that maps the sampled scalar to a color and opacity. Successive samples along the ray are then composited using the "over" operator. This ray marching process keeps on stepping, sampling, and compositing until either a maximum opacity has been reached, or the ray exits the volume, at which point the ray can be terminated.

There are many variations of this general ray marching scheme: for example by using more than one transfer function and/or scalar field, by using gradient shading in the color computation, etc. However, at its core, volume rendering is a process of using a transfer function to make regions of the volume partly or fully transparent, which in turn makes this rendering process useful for scientific visualization. By changing the transfer function interactively, users can choose to either emphasize or hide different parts of the volume, and can thereby explore the 3D structure of the underlying data.

The downside of direct volume rendering is that, if the chosen transfer function ends up making large parts of the volume fully or mostly transparent, rays passing through these regions end up taking lots of samples—and thus are costly, without useful impact on the image. Avoiding sampling such fully-transparent regions is called *space skipping*, and in and of itself refers to an entire family

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

DOI: 10.2312/pgv.20211042

of techniques. Of particular relevance to this paper is the technique introduced by Ganter and Manzke [GM19], which leveraged the then newly introduced RTX hardware ray tracing capabilities of NVIDIA's Turing architecture to realize fast, hardware accelerated space skipping. In their approach, Ganter and Manzke build a set of proxy boxes that represent active regions of the volume, then trace each ray into a (hardware accelerated) BVH over these boxes using OptiX, and only sample the volume within these proxy boxes.

Though this method is very effective in skipping empty regions, it also has several drawbacks: In particular, whenever the user changes the transfer function, proxy boxes over the voxels are computed on the CPU, followed by a BVH build over these boxes. This host-to-device proxy box transfer and BVH build hinder interactive transfer function edits used to explore the volume. In addition, during ray marching, only the BVH traversal part of the proposed space-skipping rays is hardware accelerated, and the actual ray-proxy box tests have to be done in a user intersection program, requiring costly back-and-forth's between hardware ray traversal and user defined intersection programs (in particular if there are several such boxes behind each ray).

In this paper, we introduce an extension of Ganter's method that follows the same core idea of using RTX accelerated rays for space skipping, and that even starts with the same basic active grid mask as their framework—but which significantly reduces these costs: Instead of a set of boxes that cover all active regions, we generate a low-resolution triangle mesh that describes only the *boundary* between active and inactive regions of the volume (See Figure 1). This boundary mesh is small, and can be generated in a CUDA kernel, entirely on the GPU. Since this boundary mesh is generated on the GPU, we can avoid any expensive host-to-GPU data transfers, improving RTX BVH construction performance during transfer function edits. Our generated space skipping mesh has a lower overall depth complexity compared to the BVH over boxes used by prior works (i.e., fewer ray-geometry intersections occur, and computed intervals span longer ranges), and since the mesh only uses native RTX data types (triangles), the entire space skipping process can now be done with hardware acceleration, thus leaving the CUDA cores free to do the actual volume sampling and shading.

2. Related Work

Volume rendering was first introduced by Drebin et al. [DCH88], and has since been used in many variations. An overview of recent work in this field can be found, for example, in a survey by Beyer al. [BHP15]. One of the key technologies to accelerate volume rendering is to employ some sort of space skipping to avoid sampling in regions that are known to only produce fully transparent samples; again we refer to Beyer al. [BHP15], and also Ganter and Manzke [GM19], for an overview of different approaches.

This paper primarily improves upon previous work by Ganter and Manzke [GM19], which used the recently introduced hardware ray tracing technology on Turing to perform RTX-accelerated space skipping. Before Ganter's work, the state of the art technique to perform space skipping was Hadwiger's "SparseLeap" [HAAB*17], which builds a data structure of proxy bricks that cover all regions of the volume that could produce non-transparent samples, and then uses hardware rasterization and programmable OpenGL shaders to efficiently find and skip regions of empty space. A key component of their technique is that their shader code will automatically merge the potentially numerous adjacent active bricks encountered along a pixel, thereby reducing the depth complexity and divergence that treating those bricks individually would have incurred.

Instead of using rasterization, Ganter and Manzke proposed to use the then newly introduced hardware ray tracing technology (in their case, through OptiX [PBD*10]) to find a similar set of proxy bricks, and reported both higher performance and more stable frame rates than for SparseLeap, while also pointing out that such hardware ray tracing based volume rendering (which unlike a raster-based approach always operates on arbitrary, individual rays) will also allow for easier integration into surface-based renderers. Instead of merging multiple successive bricks along a ray into a single segment they instead reduce depth complexity by greedily pre-merging neighboring bricks in a CPU-sided clustering step.

Our algorithm is based on computing a boundary mesh to determine non-empty sampling intervals. As such, it draws motivation from unstructured volume rendering where rays are intersected with the outside facing triangles of boundary tetrahedra to determine if ray segments are inside or outside the volume [BKS97].

Using hardware ray tracing for space skipping has also been proposed by Morrical et al. [MUWP19]. In their approach the authors compute similar proxy boxes over unstructured data, and also use rays to locate the active boxes along a ray. Unlike Ganter, their framework proposed to tessellate the boxes, which avoids the overhead of calling a user defined intersection program. Morrical et al. also proposed to use the leaves of a KD tree rather than proxy bricks derived from regular grids to account for the unstructured nature of their data.

Hardware accelerated ray tracing today is available through several different APIs and libraries. For our implementation we use OptiX [PBD*10] (more specifically, version 7.2), but the same algorithms also map to other APIs such as *DirectX Ray Tracing (DXR)* [Mic18] or *Vulkan Ray Tracing (VKR)* [Gro20].

3. Motivation

To understand the motivation behind our method, we will first summarize the prior techniques which use ray tracing hardware for space skipping, and discuss their imposed limitations.

The approach by Ganter and Manzke begins by interpreting the volume as a collection of smaller-resolution *bricks* of $K \times K \times K$ cells each. Based on the chosen transfer function, each of these bricks can be classified as being either *active* or *inactive*, resulting in what Ganter calls an *Active Brick Mask* for the underlying volume. Ganter then proposes to build a BVH over only the active bricks, and then use ray tracing to iterate through all bricks along a given ray, and sample the volume only within these boxes. The approach by Morrical et al. is similar, but their approach constructs a BVH over all bricks, active or inactive. During rendering, they check if a proxy brick intersected by a ray is marked as active before integrating.

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

As pointed out by Ganter, these approaches can result in many bricks that must be traversed by each ray (what Ganter calls *depth complexity*). To reduce this depth complexity, Ganter performs a clustering step that greedily merges neighboring active bricks.

For these methods, we observe the following costs:

clustering: The region clustering performed by Ganter reduces both BVH cost and depth complexity, but comes at a price. According to Ganter, at their finest brick size of $K = 8$, their clustering stage costs up to 112 ms for the FLOWER data set, and over a second for the SUPERNOVA. For the SUPERNOVA, even values of $K = 16$ and $K = 32$ still cost 107 and 13ms, respectively.

BVH build time: Once the clustering stage has produced a set of boxes, OptiX has to rebuild the BVH. We could not find actual measured performance numbers for this rebuild in Ganter's paper (likely because this cannot easily be independently measured in the version of OptiX Ganter used); however, it is worth noting that one major cost factor in building this BVH will be that boxes have to first be transferred from host to GPU.

remaining depth complexity: Though Ganter reports that clustering reduces the number of boxes by over $2\times$, even with this savings, there can still be many boxes along a ray. If neighboring rays intersect different sets of boxes with different numbers of samples per box, warp divergence will also quickly become an issue (this is why Sparseleap merges adjacent cell targets).

traversal cost: During rendering, rays will use hardware accelerated ray traversal units; however, in the case of Ganter's user-geometry, hardware acceleration only affects BVH traversal. For user primitives like boxes, the ray tracing pipeline must interrupt ray traversal and call a CUDA intersection program for every box encountered by the ray, leading to costly back-and-forth between ray tracing cores and shader cores.

Some of these costs (such as traversal cost and depth complexity) apply to every rendered frame, whereas others apply only when the transfer function changes.

4. Space Skipping using Active Region Boundary Geometry

The core idea of our approach is to holistically address all of these sources of overhead together, by replacing the hierarchy of 3D *bricks* with a set of triangles that cover only the boundary *surface* between active and inactive bricks. This boundary surface mesh has several beneficial properties over individual bricks:

- the resulting boundary surface will have relatively few triangles, and can be updated and rebuilt quickly
- as opposed to the previous costly brick clustering step, generating this boundary surface is trivially parallelizable, and can be done in a CUDA kernel on the GPU. This avoids any host-side bottlenecks and host-GPU data transfers
- since we only generate the *boundary* between active and inactive bricks, the problem of encountering multiple active bricks disappears, and depth complexity is reduced to how often a ray *transitions* between active and inactive regions. This reduces the number of space-skipping rays being cast, as well as the warp divergence between neighboring pixels
- since this surface is composed of purely hardware-accelerated primitives (BVH traversal and triangles), space-skipping rays no longer require costly CUDA intersection tests, reducing per-pixel cost and leaving the CUDA cores for volume integration

In the following section, we briefly describe our approach, which can be classified into two steps: Those that get executed exactly once at start-up, and those that get executed every time the transfer function gets modified.

4.1. Initial Set-Up

Our initial set-up consists of two operations: first, we create the grid of macrocells and corresponding active bricks mask, and after that we pre-allocate and pre-initialize the memory required for the triangle boundary mesh that we will generate later on.

4.1.1. Data Macrocell Grid, and Active Cells Mask

We start by building the same input data structure for active brick classification as Ganter and Manzke: Just like in his implementation, we support different brick sizes of $K \times K \times K$ input voxels per leaf. By default we use $K = 16$. Throughout the rest of this paper, we call this our *macrocell* grid, and call each cell therein a macrocell.

Assuming an input model of $V_x \times V_y \times V_z$ voxels, the size of this macrocell grid is $M_x \times M_y \times M_z$ macro cells, with $M_x = \left\lceil \frac{V_x}{K} \right\rceil$, etc. For each macrocell, we store the minimum and maximum scalar values of all the input voxels that map to this cell. We currently compute this macrocell grid on the host, then upload it to the GPU. To later hold the active brick masks, we also allocate GPU memory for a same-sized grid of booleans.

4.1.2. Preallocating Triangle Vertex and Index Arrays

Next, we allocate device-side memory for the vertex and index arrays that our GPU-side boundary mesh generation will later write into. We do not yet know how many triangles the surface will be made of, as this varies with the chosen transfer function; however, we can derive an upper bound on this triangle count, as we know the boundary geometry will always lie on faces of the macrocells.

For the boundary mesh vertices, we generate the list of *all* $N_{vtxMax} = (M_x + 1) \times (M_y + 1) \times (M_z + 1)$ possible macrocell grid vertices (using CUDA `float3`s), and have the later triangle generation produce indices that reference this vertex array. As a result, the final vertex array will contain some vertices not used by any triangles in the mesh, but this only comes at a cost to memory, and ultimately means we later do not have to track which vertices do vs do not get used.

For the triangles, we can compute the upper limit N_{triMax} on how many triangles could possibly be generated. We use this limit to pre-allocate an array of `int3` vertex indices. We also allocate a single device-side `numTriangles` counter that the triangle generation phase can later use as an atomic offset for where to write newly generated triangles to.

Pre-allocating the upper bound of all possible vertices and triangles sounds excessive; however, we point out that at $K = 8$, $K = 16$ and $K = 32$, these upper bounds are three to five orders of magnitude lower than the number of voxels in the volume.

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

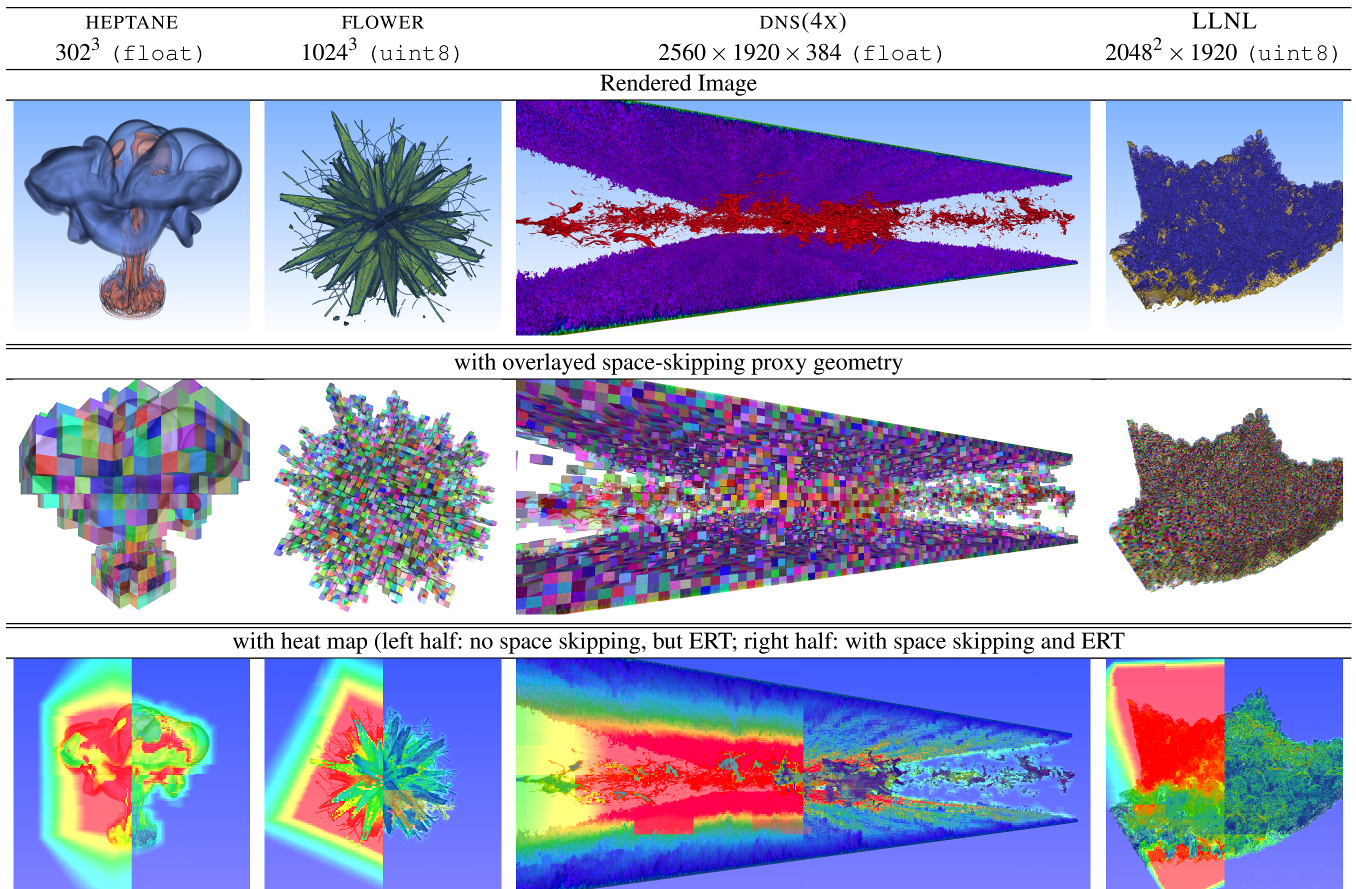

Figure 2: *The four data sets we use for our evaluation:* `heptane`*,* `flower`*,* `DNS-4` *(a 4× sub-sampled version of the full DNS data set), and the* `LLNL` *Richtmyer-Meshkov instability data set. Top row: direct volume rendering with the transfer function used during the evaluation. Center row: with our space-skipping geometry overlaid over the volume rendering. Bottom: the same (slightly zoomed out for context), with a heat map showing* `cudaTimer` *time per pixel with our space skipping method turned off (left half of each image) and with (right half) respectively turned on (right half).*

4.2. Space Skipping Data Structure (Re)build

The shape of this boundary mesh depends on the actual transfer function being used, and thus this mesh must be rebuilt every time the transfer function changes.

4.2.1. Active Cell Classification

The first step in rebuilding the boundary mesh is to recompute the active bricks mask from the macrocell grid. We do this by calling a CUDA kernel where each thread maps to exactly one macrocell. The respective CUDA thread computes which part of the transfer function this macrocell's stored value range maps to, and checks if any of these transfer function values have a non-zero alpha. This boolean result is written into the active bricks mask. This process is similar to the one used by Ganter and Manzke, except that our brick mask is stored and updated entirely on the GPU.

4.2.2. Boundary Triangle Generation

Next, we clear the device-side `numTriangles` counter, and run three templated copies of a CUDA kernel which generates all triangles with X, Y, and Z orientation, respectively. In the first instance, we look at all faces of the macrocell grid that are perpendicular to the X axis (there are $(M_x+1) \times M_y \times M_z$ of those), and run a CUDA kernel where each thread maps to one of those faces.

For each such face, the corresponding thread computes the indices of the two macrocells sharing that face, and then uses those indices to determine if the respective macrocells are active or not. Indices that would map to outside the macrocell (which can occur for faces on the outside of the volume) get treated as if the macrocell was inactive.

The given face is a boundary face if and only if one of the adjacent cells is active, and the other is not. If this is not the case, the kernel returns without generating any triangles. Otherwise, the thread computes the indices of the four vertices that span the current face, and uses those four indices to generate two triangles that span that face. The winding order of these triangles are set to point away from the active towards the inactive cell. The thread then allocates storage in the triangle buffer by atomically increasing the `numTriangles` by two, and then writes the generated triangle indices into the proper location of the indices buffer.

We then call this face generating kernel two more times—once

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

for the Y, and once for the Z direction—and finally download (only) the value of the `numTriangles` counter to the host. Macrocells, active brick mask, and triangles never leave the GPU.

CUDA/C++ pseudo code for the template to identify boundary cells and insert triangles into the boundary mesh for one dimension is given in Listing 1.

```
1  template <int DIM>
2  generateBoundaryMeshKernel(int &numTriangles,
3                             int3 *triangles,
4                             bool activeCells[][][])
5  {
6    // Current and previous macro cell indices
7    int3 curr = getGlobalThreadIndex();
8    int3 prev;
9    if constexpr(DIM==0)
10     prev = curr - { 1, 0, 0 };
11   else if constexpr(DIM==1)
12     prev = curr - { 0, 1, 0 };
13   else if constexpr(DIM==2)
14     prev = curr - { 0, 0, 1 };
15
16   // Only proceed if at boundary
17   if (activeCells[prev]==activeCells[curr])
18     return;
19
20   // Get vertex list indices for the boundary
21   // quad connecting prev and curr macro cells
22   int vi[4] = getBoundaryQuad<DIM>(currIndex);
23
24   // Generate two more triangles and
25   // append to global list
26   int triID = atomicAdd(&numTriangles,2);
27
28   if (activeCells[prev]) {
29     // active boundary, facing from prev to curr
30     triangles[triID+0] = { vi[0], vi[1], vi[2] };
31     triangles[triID+1] = { vi[0], vi[2], vi[3] };
32   } else if (acticeCells[curr]) {
33     // active boundary, facing from curr to prev
34     triangles[triID+0] = { vi[1], vi[0], vi[2] };
35     triangles[triID+1] = { vi[2], vi[0], vi[3] };
36   }
37 }
```

Listing 1: *CUDA/C++ pseudo code identifying boundary cells and generating boundary surfaces. The templated kernel is called once per X, Y, and Z orientation.*

4.2.3. BVH Rebuild

Lastly, we update the triangle count and index array pointer of the OptiX triangle geometry to reference our generated mesh, and ask OptiX to rebuild the corresponding BVH. Next, we compact the BVH to conserve memory, and build with the "optimize for traversal" flag rather than the "optimize for build time" flag. Omitting compaction and asking for a build-time optimized BVH (as well as looking into refitting rather than rebuilding) could further improve our BVH construction performance, but we found these optimizations to be unnecessary.

4.3. Space Skipping during Volume Rendering

During rendering, volume rays can use this boundary mesh data structure to find where the ray enters and respectively exists any active regions. First, each ray is traced against this triangle mesh, where the corresponding closest hit program records both the hit distance t_{hit} to the boundary mesh as well as the OptiX "hit kind" flag, which specifies whether the intersected triangle was hit on its front or back side (both of these values are easily available in the closest-hit program).

If the ray does not hit any geometry at all, we know that ray will not traverse any active volume regions, and thus we terminate the ray. If the ray does hit a back-facing triangle, we know the ray must be inside the volume, and know the next segment to integrate is $[t_0, t1] = [0, t_{hit}]$. Otherwise, we know the ray enters the volume at $t_0 = t_{hit}$, and we trace the same ray again with `ray.tmin=t0` to determine the exit distance t_1.

Once the $[t_0, t_1]$ interval has been determined, we perform volume integration over this interval. In our case, we use both gradient shading and early ray termination, but we note that how exactly this sampling and shading is done is completely orthogonal to our technique.

After integrating the given interval, we check if the ray has terminated early, and if so, return. Otherwise, we iterate by tracing another ray from t_1 to find any other potential segment to integrate, and so on.

4.4. Implementation Notes

The above algorithm can be implemented in a variety of ways and APIs. For our implementation, we chose to use CUDA for the active mask generation and boundary mesh generation kernels, and OptiX 7 as our ray tracing API.

We use OptiX through the OWL library [Wal20], and in particular use OWL's CUDA inter-op capabilities to interface with CUDA; e.g., the vertex and index arrays are created as OWL buffers, with the CUDA kernels writing into these buffers. For the acceleration structure, we use an `OWLTrianglesGeom` for the triangle mesh, and an `OWLTrianglesGroup` for the corresponding BVH. Since OWL does not support 3D textures, we create the 3D volume texture in CUDA, and pass that texture to an OptiX raygen program as an `OWL_USER_TYPE(cudaTextureObject_t)`.

All actual rendering in our code is performed in a `raygen` program that generates and traces the rays into the volume and bounding mesh geometry. The resulting space skipping ray segments march through the volume, sampling a 3D texture, computing local gradients, and transferring samples using transfer function texture look-ups. To reduce aliasing, we use interleaved sampling [KH01], and we use a fixed sampling rate of $dt = 0.5$ (i.e., an average of two samples per cell). For the ray-surface intersection, we use a closest hit program that uses per-ray data to pass hit distance and triangle orientation back to the raygen program. Since we know that no any-hit program is being used, we trace the ray with the `DISABLE_ANYHIT` flag.

For our evaluation, we also include two reference ray marchers. The first reference ray marcher disables the space-skipping rays,

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

where volume rays march over the entire volume bounds, sampling through potentially inactive regions. The second reference ray marcher follows the technique presented by Ganter and Manzke, where space skipping rays march over active proxy-boxes using `OWLUserGeom`, where BVH traversal is done in hardware and intersection testing is done in a CUDA intersection program. In all other aspects, all ray marchers are exactly the same, including the use of early ray termination, and avoiding gradient computations for fully-transparent pixels.

The raygen program also allows for super-sampling with an interactively controllable number of samples per pixel, performs progressive refinement using an accumulation buffer, and also supports various useful helper tasks such as overlaying boundary surface intersections, a performance heat map, etc. For GUI and transfer function editing, we use the freely available *cuteeOWL* library that comes with OWL [WZ20]. After acceptance, our implementation will also be made publicly available.

5. Results

For our evaluation, we used an NVIDIA RTX 8000 GPU with 4,608 CUDA cores (boost clock of 1.7GHz), hardware ray tracing support, and 28 GB of GDDR6 memory. For reference, the PC that this GPU is in has a 2.2 GHz, 4-core Intel CPU, 128 GBs of RAM, and runs Ubuntu Linux 18.04, CUDA 10.2, NVIDIA driver 440.44, and OWL version 1.0.5.

To evaluate our framework, we use four widely used volume data sets as shown in Figure 2. To facilitate an easy comparison to Ganter and Manzke, we include the FLOWER data set. We could not locate a copy of the larger SUPERNOVA data set, but have replaced this with two other data sets of similar to larger size (DNS(4X) and LLNL).

In our comparison we focus on comparing to Ganter and Manzke [GM19], and refer readers to his paper for a detailed comparison to SparseLeap [HAAB*17].

5.1. Data Structure Build/Update Time

The main bottleneck for Ganter and Manzke's framework was the time to update the data structure, dominated by the time to perform CPU-side clustering, and for uploading and rebuilding the BVH. In tables 1 and 2 we report the corresponding timings for our boundary mesh method and Ganter's, for different macrocell sizes (our default is 16).

Compared to Ganter's approach, our active brick classification can be done entirely on the GPU, and its cost (T_{tf}) becomes negligible. Clustering time T_C does not apply any more in our framework, and its equivalent—triangle extraction time T_{tris} is in the low milliseconds range even for the finest tested macrocell grid. BVH construction is currently the biggest cost in our framework, but this is in the low milliseconds even for our largest data sets and finest macro cell resolutions. Corresponding BVH build times from Ganter's approach are approximately equal to ours for smaller datasets, but for larger datasets we see significant performance improvements.

When adding all different update costs together, our total build time remains well within an interactive range. Even for our default macrocell size of $K = 16$, our total build time never exceeds 3.5 ms.

model	#cells	#active	#tris	T_{tf}	T_{tris}	T_{BVH}	T_{total}
	macro-cell size $K=8$						
HEPTANE	54.9K	12.3K	18.1K	$42\mu s^\star$	$54\mu s^\star$	1.6ms	2.1ms
FLOWER	2.1M	180K	318K	$134\mu s^\star$	$115\mu s^\star$	4.8ms	5.4ms
DNS(4X)	3.7M	660K	1.5M	$206\mu s^\star$	$225\mu s^\star$	12.3ms	13.1ms
LLNL	15.7M	2.9M	4.2M	$998\mu s^\star$	$904\mu s^\star$	28.2ms	30.4ms
	macro-cell size $K=16 (default)$						
HEPTANE	6.9K	2K	3.4K	$47\mu s^\star$	$66\mu s^\star$	$918\mu s^\star$	1.4ms
FLOWER	275K	33K	78K	$59\mu s^\star$	$57\mu s^\star$	3.0ms	3.6ms
DNS(4X)	461K	122K	366K	$67\mu s^\star$	$69\mu s^\star$	5.3ms	5.8ms
LLNL	1.97M	475K	422K	$153\mu s^\star$	$119\mu s^\star$	5.4ms	6.0ms
	macro-cell size $K=32$						
HEPTANE	1K	358	856	$37\mu s^\star$	$44\mu s^\star$	$589\mu s^\star$	1.0ms
FLOWER	33K	6.8K	19K	$37\mu s^\star$	$45\mu s^\star$	1.8ms	2.2ms
DNS(4X)	58K	26K	98K	$40\mu s^\star$	$48\mu s^\star$	2.8ms	3.2ms
LLNL	246K	69K	54K	$52\mu s^\star$	$59\mu s^\star$	2.3ms	2.8ms

Table 1: *Statistical data and timings for the different stages of building our data structure: T_{tf} is time for active cell classification (based on transfer function); T_{tris} time for boundary triangle mesh generation; T_{BVH} time to rebuild the OptiX BVH; and T_{total} is total data structure rebuild time. (*: Any timings $< 1ms$ should be considered with caution, and might be better interpreted as simply "less than one millisecond")*

model	T_{tf}	T_{tris}	T_{BVH}	T_{total}
	macro-cell size $K=8$			
HEPTANE	$39\mu s^\star$	3.3ms	2.1ms	5.9ms
FLOWER	$142\mu s^\star$	53.2ms	5.0ms	58.7ms
DNS(4X)	$256\mu s^\star$	136.7ms	11.6ms	149.1ms
LLNL	$991\mu s^\star$	641.5ms	44.2ms	687.2ms
	macro-cell size $K=16 (default)$			
HEPTANE	$42\mu s^\star$	$528\mu s^\star$	1.3ms	2.4ms
FLOWER	$55\mu s^\star$	9.9ms	2.7ms	13.1ms
DNS(4X)	$64\mu s^\star$	25.6ms	4.4ms	30.4ms
LLNL	$154\mu s^\star$	95.1ms	8.9ms	104.5ms
	macro-cell size $K=32$			
HEPTANE	$37\mu s^\star$	$101\mu s^\star$	1.0ms	1.6ms
FLOWER	$38\mu s^\star$	1.8ms	1.4ms	3.6ms
DNS(4X)	$46\mu s^\star$	5.8ms	2.5ms	8.8ms
LLNL	$52\mu s^\star$	14.1ms	3.1ms	17.7ms

Table 2: *Statistical data and timings for the different stages of building the data structure by Ganter and Manzke: T_{tf} is time for active cell classification (based on transfer function); T_{tris} time for boundary triangle mesh generation; T_{BVH} time to rebuild the OptiX BVH; and T_{total} is total data structure rebuild time. (*: Any timings $< 1ms$ should be considered with caution, and might be better interpreted as simply "less than one millisecond")*

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

Method mc-size	Base NA	Ganter K=8	 K=16	 K=32	Ours K=8	 K=16	 K=32
	1 ray per pixel						
HEPTANE	34	79	81	67	100	93	71
FLOWER	14	95	88	54	154	111	64
DNS(4X)	13	24	25	19	40	34	23
LLNL	8	23	27	19	34	40	43
	4 rays per pixel						
HEPTANE	9	17	18	15	29	22	18
FLOWER	3	26	18	11	36	23	13
DNS(4X)	3	5	4	3	9	7	5
LLNL	2	4	4	5	7	8	9

Table 3: *Frames-per-second comparison for the four models shown in Figure 2, using a resolution of* 3840×2160 *and at different macrocell sizes.*

space-skip force rebuild	Base off N/A	Ganter on off	 on on	Ours on off	 on on
HEPTANE	34	81	68	93	84
FLOWER	14	88	41	111	83
DNS(4X)	13	25	14	34	28
LLNL	8	27	7	41	32

Table 4: *Impact of space skipping and rebuild time on render performance, for the four models shown in Figure 2, using a resolution of* 3840×2160*.*

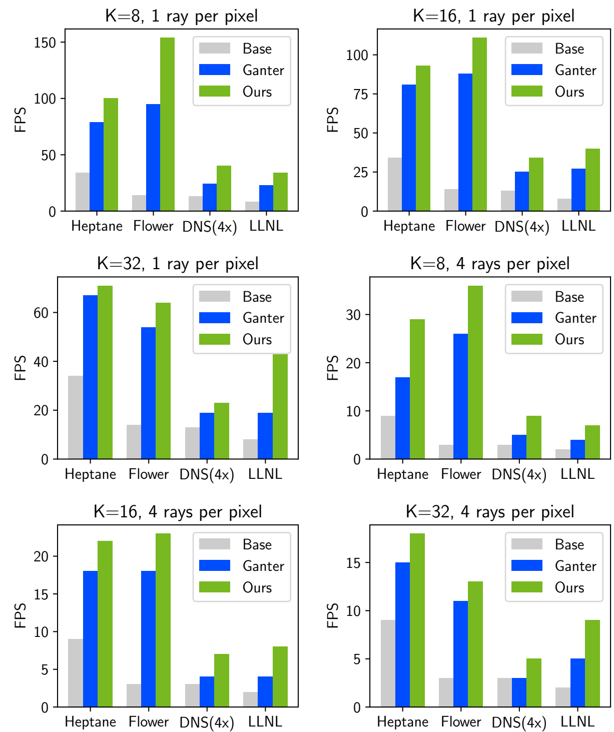

Figure 3: *Frames-per-second comparison, visualization of the results presented in Table 3.*

5.2. Render Time

To evaluate our method's impact on render performance during interactive volume exploration, we also measured our sample volume renderer's frame rate using the high-quality settings and the configurations shown in Figure 2. We took measurements once with space skipping completely disabled (i.e., geometry is neither extracted, nor are any space skipping rays being traced), once with Ganter and Manzke's method, and once with our method enabled. For reference we also report performance if data structure update is performed every frame.

As can be seen from these experiments (see Figure 3, Tables 3 and 4) our method always provides significant speedups over the reference method, even if the cost for data structure updates is fully factored in. In particular, even for our largest models rebuilding the data structure, where Ganter's method results in hundreds of milliseconds of build time, we only see a small impact on total frame rate. All models we tested remain fully interactive even if data structure rebuilding is forced every frame. A visual illustration of the impact of space skipping can also be seen in the heat-maps (with and without space skipping) provided in the bottom row of Figure 2.

6. Summary and Conclusion

We have presented a method that builds on the same basic ideas as prior works—namely, to use RTX accelerated ray tracing to realize space skipping in volume rendering—but modify these techniques to significantly reduce several sources of overhead that prior techniques suffer from. In particular, we build on the same basic macro-cell/active brick mask as Ganter and Manzke, but replace their BVH over active brick regions with a BVH over only the boundary between active and inactive regions. This results in a much lower depth complexity and fewer (and cheaper) rays traced for space skipping.

In addition, we avoid clustering completely, and can easily extract both active brick mask and boundary geometry natively on the GPU, at significantly lower cost. During rendering, our method will integrate over exactly the same regions as prior methods, but at a lower cost to find these active regions, and with significantly lower cost of rebuilding the data structure, maintaining highly interactive rates even when the user modifies the transfer function.

Our biggest cost factor during data structure update currently is the BVH build time. Reducing this—e.g., by employing refitting—is an obvious avenue for future work. Far more interesting however is to evaluate if the same method can also be applied to adaptive sampling, or for non-structured data such as tetrahedral meshes or adaptive mesh refinement data.

References

[BHP15] BEYER J., HADWIGER M., PFISTER H.: State-of-the-Art in GPU-Based Large-Scale Volume Visualization. *Computer Graphics Forum* (2015). 2

[BKS97] BUNYK P., KAUFMAN A., SILVA C. T.: Simple, fast, and robust ray casting of irregular grids. In *Scientific Visualization Conference (dagstuhl '97)* (1997), pp. 30–30. 2

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

[DCH88] DREBIN R. A., CARPENTER L., HANRAHAN P.: Volume Rendering. *ACM SIGGRAPH Computer Graphics (Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '88)* (1988). 2

[GM19] GANTER D., MANZKE M.: An Analysis of Region Clustered BVH Volume Rendering on GPU. In *Proceedings of High Performance Graphics (HPG)* (2019). 2, 6

[Gro20] GROUP K.: Ray Tracing in Vulkan. Available online: https://www.khronos.org/blog/ray-tracing-in-vulkan, 2020. 2

[HAAB*17] HADWIGER M., AL-AWAMI A. K., BEYER J., AGUS M., PFISTER H.: SparseLeap: Efficient empty space skipping for large-scale volume rendering. *IEEE Transactions on Visualization and Computer Graphics* (2017). 2, 6

[KH01] KELLER A., HEIDRICH W.: Interleaved Sampling. In *Rendering Techniques 2001 (Proceedings of the Eurographics Workshop on Rendering)*. 2001. 5

[Mic18] MICROSOFT: Direct3D 12 Ray Tracing. Available online: https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-raytracing, 2018. 2

[MUWP19] MORRICAL N., USHER W., WALD I., PASCUCCI V.: Efficient Space Skipping and Adaptive Sampling of Unstructured Volumes Using Hardware Accelerated Ray Tracing. In *Proceedings of IEEE Visualization (Short Papers Track)* (2019). 2

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A.: OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)* (2010). 2

[Wal20] WALD I.: Introducing OWL: A Node Graph Abstraction Layer on top of OptiX 7. Available online: https://bit.ly/3pGPIWy, 2020. 5

[WZ20] WALD I., ZELLMANN S.: CuteeOWL: QT5 based 3D Viewer and Transfer Function Editor Widgets. Open Source: https://github.com/owl-project/cuteeOWL/, 2020. 6

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.