Eurographics Symposium on Parallel Graphics and Visualization (2015), pp. 1–9
C. Dachsbacher, P. Navrátil (Editors)

SIMD Parallel Ray Tracing of Homogeneous Polyhedral Grids

Brad Rathke[1,2] Ingo Wald[1] Kenneth Chiu[2] Carson Brownlee[3]

[1]Technical Computing Group, Intel Corp.

[2]Binghamton University, State University of New York

[3]Texas Advanced Computing Center, University of Texas

Abstract

Efficient visualization of unstructured data is vital for domain scientists, yet is often impeded by techniques which rely on intermediate representations that consume time and memory, require resampling data, or inefficient implementations of direct ray tracing methods. Previous work to accelerate rendering of unstructured grids have focused on the use of GPUs that are not available in many large-scale computing systems. In this paper, we present a new technique for directly visualizing unstructured grids using a software ray tracer built as a module for the OSPRay ray tracing framework from Intel. Our method is capable of implicit isosurface rendering and direct volume ray casting homogeneous grids of hexahedra, tetrahedra, and multi-level datasets at interactive frame rates on compute nodes without a GPU using an open-source, production-level ray tracing framework that scales with variable SIMD widths and thread counts.

1. Introduction

Rapid advancements in computing power and the sophistication of simulations have allowed scientists to simulate physical phenomena at increasingly large scales. Such simulations are challenging to effectively visualize at full resolution. Many techniques to optimize massive data rendering have focused on polygonal data or regular grids, however simulations often generate unstructured data in the form of tetrahedral, hexahedral, or other polyhedra which produce many challenges for interactive rendering. For isosurfacing, visualization tools such as *VisIt* [CBW*12] and *ParaView* [Hen04] use marching cubes [LC87] to extract a surface explicitly. This intermediate representation requires additional time and memory over the existing dataset and needs to be regenerated every time the isovalue is modified resulting in unnecessary computational overhead during data exploration.

Rasterizing the resulting triangles also presents several issues for performance and usability. Many of the resulting triangles may be occluded or of sub-pixel size in large datasets. Transparency presents another issue for traditional rasterization techniques due to the rendering order dependent nature of rasterizing transparent objects. Culling and depth peeling methods can alleviate problems with rasterization of transparent objects, but add complexity to the rendering algorithms. Directly ray tracing the unstructured grid, however, requires no intermediate representation, can accurately render transparency, and implicitly handles occluded or sub-pixel regions.

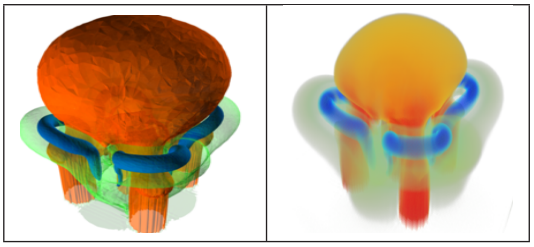


Figure 1: *Our method allows for interactively rendering unstructured data sets (in this example, the "Jets" of 12M tetrahedral cells), and supports both ray tracing of the implicit isosurfaces as well as direct volume ray casting. Left: Three semi-transparent (implicit) isosurfaces, with proper transparency handled by the ray tracer. Right: The same volume with direct volume ray casting, using a transfer function chosen to highlight the same features shown in the isosurface rendering.*

In the case of volume rendering, special-case solutions can be used that do not easily integrate with the rest of the visualization tool, use lower fidelity splatting, or resample

to a structured volume whose size must necessarily be restricted to fit the limited amount of GPU memory [SCCB05]. Resampling an unstructured grid into a structured grid will also cause some loss of data precision unless the new structured grid has a sufficiently large number of cells which can significantly increase the amount of memory used and may necessitate out-of-core rendering methods to achieve similar results.

In this paper, we describe our approach to integrating support for unstructured volumetric grids (currently tetrahedral and hexahedral meshes) into the OSPRay ray tracing framework, Section 3.2, with volumetric rendering and implicit isosurfacing. We demonstrate that this integration allows rendering isosurfaces with advanced shading effects like shadows, transparency, and ambient occlusion. In addition, by running completely on the CPU our method can take advantage of the full system memory, can be run on any non-GPU compute node, and can even operate on exactly the same data structures that middle-ware tools like *VTK* operate on.

We begin by exploring previous related works with regards to isosurface rendering and direct volume rendering of unstructured volumetric data in section 2. Section 3 discusses background information about preexisting systems used with our technique, including the Intel® SPMD Program Compiler (ISPC), the Embree ray tracing kernels, and the OSPRay ray tracing framework. A method overview is presented in section 4. Section 5 will describe the *min-max BVH* acceleration structure used in our technique. Sections 6 and 7 describe and discuss specific applications of our method to isosurface rendering and direct volume ray casting respectively. We describe our test execution environments and tabulate the results of our experiments in section 8. Finally, we summarize and conclude in Section 9.

2. Previous Work

For rendering isosurfaces, most visualization packages such as *ParaView* [Hen04] or *VisIt* [CBW*12] extract polygonal isosurfaces (typically using some variant of the *marching cubes* approach [LC87]), and render the resulting polygons using OpenGL. Isosurface extraction is well supported by tools like *VTK* [Sch06] but is still costly in terms of time and memory, and can produce intractably large numbers of triangles.

Rather than performing explicit isosurface extraction, a number of authors have proposed rendering isosurfaces via ray tracing to compute the ray/isosurface intersections on the fly. For structured grids, Parker et al. [PSL*98] presented a distributed shared memory approach to volume visualization using ray tracing. Wald et al. [WFM*05] presented a method for rendering implicit isosurfaces by building a min-max kd-tree across all voxels in a structured grid, such that the splitting planes of the tree align with the voxels of the original grid. Knoll et al. [KTW*11] made use of *min-max BVH* structures to facilitate coherent ray traversal of structured volumes and implemented explicit Intel® Streaming SIMD Extensions (SSE) vectorization of their volume integration function.

For unstructured grids, Marmitt et al. [MS06] explored using common real-time ray tracing techniques to find an initial voxel intersection, and then traversed subsequent cells using Plücker coordinate tests. Wald et al. [WFKH07] describe an approach that applies trees (in the form of BVHs) also to time-varying, tetrahedral data sets. They show the potential of the approach, but are limited to rendering isosurfaces, and rely on a special frustum traversal that would not easily fit a more general framework such as OSPRay.

For Direct Volume Rendering (DVR) of unstructured meshes, Childs et al. [CDM06] used a sample-based parallel distributed memory algorithm which relied on first resampling an unstructured grid into a structured grid. Muigg et al. [MHDG11] accomplished volume visualization of general polyhedral meshes using a GPU through a two-sided face sequence list data structure which allowed for easily walking a polyhedral mesh. Shirley et al. [ST90] describe a method of approximating direct volume rendering by projecting tetrahedra to a frame buffer as transparent triangles and rasterizing them.

3. Background

3.1. Software Defined Visualization

While most modern visualization packages rely on GPU based techniques *Software Defined Visualization (SDVis)* is the concept of using a purely software-based rendering stack to take advantage of the capabilities of CPU-based execution.

In software defined visualization all steps of the rendering process are accomplished entirely on the CPU, which allows for avoiding the constraints of working with a GPU, such as limited memory and bus transfer speed and eliminates the extra level of complexity that host-GPU communication and scheduling can introduce. CPU-only rendering also simplifies development of *in-situ* visualization algorithms by potentially using the same data structures in the same memory space as the running simulation.

3.2. OSPRay and Embree

OSPRay is an open-source ray tracing based rendering library for visualization. OSPRay is layered on top of Embree [WWB*14], a highly optimized set of CPU ray tracing kernels. In addition, OSPRay also uses ISPC (see Section 3.3) in order to achieve easy vectorization of the program, and thus improving performance. Building on top of the OSPRay framework allows us to take advantage of its preexisting implementations for CPU based ray tracing.

The volume rendering implementation in the current OSPRay release assumes that all volumes are structured, regular or rectilinear grids. Structured curvilinear and unstructured grids are not supported by the current version of OSPRay. However OSPRay is extensible through a system of modules and we will leverage this to add support for tetrahedral and hexahedral grids.

3.3. The Intel® SPMD Program Compiler

To achieve high utilization of CPU vector units we make use of the ISPC [PM12] language and compiler. The ISPC language is a subset of the C99 programming language and has been extended to facilitate as Single Program Multiple Data (SPMD) programming model. This allows for the program to be written as if it were standard C99 with a few extra storage class specifiers that affect how the CPU vector units will be used. The extensions present in ISPC allow for efficient use of CPU vector units without resorting to compiler intrinsics which are often not portable, and difficult to work with. Through our use of the ISPC compiler we are able to write code that will fully utilize processor vector units with a single code base for both traditional CPUs and Intel® Xeon Phi™coprocessors.

The storage type specifiers added by ISPC are *uniform* and *varying*. The ISPC compiler executes several program instances at once, one per vector lane (four processes for SSE, eight for AVX). A *uniform* value is shared between program instances, while a *varying* value will be different across program instances.

4. Method Overview

Algorithmically, our method is motivated by Wald et al.'s approach to rendering isosurfaces of tetrahedral meshes using *min-max BVH*s [WFKH07]; but we extend this method to also handle direct volume rendering and other unstructured grid representations, and we use a different set of choices for the actual implementation. As in Wald et al.'s approach, we build a *min-max BVH* over the base primitives of the unstructured grid (hexahedra or tetrahedra, in our case), and store the minimum and maximum possible attribute value of the respective subtree (Section 5). This data structure is then used both for ray-isosurface intersection (Section 6) as well as for efficiently locating volume samples in direct volume rendering (Section 7).

We assume that the input to our method is one or more unstructured volumetric grids made up of hexahedral or tetrahedral cells and can also be easily expanded for other types of homogeneous polyhedral grids. Each volumetric grid is specified through a vertex array, one or more attribute arrays, and an array of vertex indices, with either 4 (tetrahedra) or 8 (hexahedra) vertex indices per cell. An overview of the datasets that we will be using for our experiments is given in Figure 4 and Figure 7.

On the systems level, our method is designed for and guided by the constraints of the OSPRay ray tracing framework; this paper describes how we make the algorithmic component work within this system. In particular, we describe how our framework makes use of ISPC for Instruction Set Architecture (ISA) independent vectorization, and what we have to do to work within the existing software architecture.

In our work we define a module to support two types of homogeneous (single polyhedra type) unstructured polyhedral grids, tetrahedral and hexahedral grids. We extend the OSPRay abstract types representing geometry and volumes to achieve both isosurface and DVR for unstructured grids. The new geometry type we define is used to wrap a *min-max BVH* which contains our tetrahedral or hexahedral mesh and is used for both isosurfacing and direct volume ray tracing. The new volume type is used to extend the OSPRay volume sampling functionality for unstructured datasets and is designed such that it is simply an interface for OSPRay to access the previously mentioned geometry type as a volume. These new types are designed such that OSPRay will be able to interact with them in exactly the same manner that it interacts with their base types.

5. The Min-Max BVH

At the core of our method is a *min-max BVH* nearly identical to the one described in [WSBW01] (albeit with different traversal kernels). We use a binary BVH in which each inner node points to two subtrees, while leaves point to a list of primitives. In addition, each BVH node contains the minimum and maximum attribute value that a sample in any cell in the respective subtree could possibly return: for intermediate nodes it is the minimum/maximum of these values for all primitives in that subtree, for leaf nodes it is the minimum/maximum of these values for all primitives contained in that leaf, and for each primitive it is the minimum/maximum attribute value at its vertices.

As in [WSBW01], our BVH is built over *all* primitives. This means that in the case of isosurface rendering the tree is built without regard to whether or not a given primitive can contain the isovalue of interest. Eventually such subtrees of uninteresting cells are culled during traversal by checking the *min-max BVH* ranges against the isovalue of interest.

Building the *min-max BVH* over all primitives makes adjusting the isovalue of interest at runtime trivial since the tree does not have to be rebuilt nor updated. The ability to adjust the isovalue of interest at runtime allows for rapid dataset exploration and analysis. Building the tree in such a way also allows the same *min-max BVH* to be used both for implicit isosurface rendering and direct volume ray casting.

In the case of data that is divided up into many subsections, such as simulation results from multiple nodes, we build a min-max BVH over all pieces.

```
struct MinMaxBVH2Node {
  float bounds_low[3];
  float scalar_range_low;
  float bounds_high[3];
  float scalar_range_high;
  int64 childCount : 3;
  int64 childRef   : 61;
};
```

Figure 2: *Data Layout of a* min-max BVH *Node.*

5.1. Relationship to Embree BVH Kernels

Though Embree (which OSPRay builds on) provides very efficient kernels for both building and traversing BVHs we do not use the Embree BVHs for the unstructured grid representation (though OSPRay still uses it for all other geometries), and instead build our own binary BVHs. This is because we require additional information in our BVH, namely the min/max of the value, and as of the writing of this paper it is not practical to extend Embree to do so. Furthermore, there is no ideal way to extend the existing Embree BVH build and traversal kernels to do the min/max culling of subtrees (Section 6) during traversal. It would of course be possible to integrate our method into Embree, since Embree is fully open source; and this would be expected to yield significant higher performance in both construction and traversal. Embree style parallel BVH build kernels could significantly reduce tree construction times allowing us to compute new trees at runtime for time varying data and improved intersection performance through hybrid traversal methods which have been shown to increase traversal performance by up to 50% compared to packeted traversal [WWB*14]. This, however, would require significant low-level, ISA-specific code that would make our system significantly harder to extend and maintain.

Since we are not using Embree's BVH, we could in theory also have used other data structures such as kd-trees as proposed in [WFM*05]. However, BVHs have other useful properties such as bounded and predictable memory use; and for highly *un*structured data such as tetrahedra and isovalue BVHs are significantly easier to build and traverse than kd-trees. Hence our entire framework builds on BVHs.

5.2. BVH Memory Layout

The data layout for our BVH nodes is depicted in Figure 2. All BVH nodes contains 6 floats for specifying that node's bounding box, two floats for the attribute range, and a 64-bit integer to encode the leaf or child pointers. An intermediate node in the tree will have a `childCount` of 0 while in a leaf node `childCount` will give the number of primitives as children. The 61-bit `childRef` value is dual purpose. If the node is a leaf this is an offset into a separate "item list" array of 64-bit integers that identify the primitives (though what exactly those 64-bit integers represent is up to the geometry using the BVH; the BVH does not care). If the node is an intermediate node, then `childRef` represents an offset from the beginning of tree storage to the location of the first of the two children. Sibling nodes are stored contiguously in memory so only a single offset is needed. This memory layout takes up only 40 bytes per *min-max BVH* node.

Using item lists allow for building the BVH without modifying the input vertex and index arrays, which is an important property when integrating into existing visualization packages.

Eliminating item lists would have required some degree of duplication of the primitives into the leaf nodes (due to the fact that nodes may overlap and a single primitive could be contained in more than one leaf), or rearranging the existing data. If we were to copy the primitives into BVH leaf memory we would be spending extra time during BVH build for the copies and would no longer be able to do easy linear traversal of the primitives. Rearranging the existing data is also not desirable as it would prevent integration with existing tools such as *ParaView*.

5.3. Min-Max BVH Construction

For the BVH construction code we intentionally chose a simple and straight-forward spatial median builder [MB90]. Though more advanced surface-area builders are often significantly faster to traverse[EGMM07], for our applications spatial median builds are sufficient. Since our BVH is built over *all* grid cells (not only those used for any specific isovalue) the distribution of our primitives is far more spatially uniform than typical surface data would be; so virtually none of the original assumptions of a surface area heuristic (SAH) [MB90] apply. We do, however, apply the SAH termination criterion when determining whether to split or create a leaf.

The build code is intentionally simple and is implemented in scalar C++. In each recursive partitioning step, we compute the bounding box of all primitives' centroids, and determine the plane that halves this box along its widest dimension. We then perform an in-place quick-sort partitioning step, and recursively build both left and right halves until the termination criterion tell us to make a leaf (due to encoding, we only allow leaves with less than 8 primitives).

5.4. Time Varying Data

We support time varying data by building a separate *min-max BVH* for individual time steps. This choice was made due to a prioritization of the runtime performance of switching between time steps over that of memory footprint. If our tree build times were similar to that of Embree[WWB*14] we would likely have chosen to build trees for each time step on the fly to save system memory. An example of time varying data is shown in Figure 3.

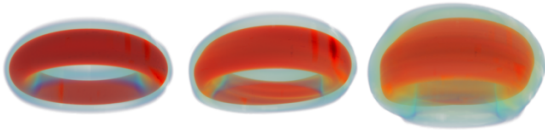


Figure 3: *Three time steps of the "Fusion" data set. Each time step is given its own OSPRay* `Geometry` *object and switching between them is accomplished by a simple pointer swap in the renderer, which then causes all rays traced to then reference the corresponding time step* `Geometry` *or* `Volume`*.*

Dataset	Size	SPMD	Packeted	Gain
Bucky	177k tets	68.9	103.2	50%
Jets	1M tets	79.9	92.0	15%
T-jet	12M tets	90.7	109.4	21%
SF1	14M tets	60.2	78.7	31%
Earthquake	47M hexes	14.1	46.3	228%

Table 1: *A comparison of both traversal algorithms on our* Xeon Node *(specifications in Section 8). Values are measured in millions of rays per second using the OSPRay obj renderer on implicit isosurfaces for the same views as Figure 4 and with shadowing disabled.*

5.5. ISPC Implementation

Traversal of the *min-max BVH* is implemented with the ISPC SPMD compiler to take advantage of all vector units available in the hardware. To accomplish this we maintain the structures used in the C++ implementation in the ISPC implementation and simply pass ownership of the pointers to ISPC. We evaluated two general methods of vectorized BVH traversal.

In the first method, which we call *SPMD traversal*, each program instance traverses the *min-max BVH* independently. The program instance keeps track of its own traversal stack. The SPMD algorithm is trivial to implement by simply allowing all traversal state variables to be *varying*, which allows each SIMD lane to perform its own, independent traversal.

In the second method, *packet traversal* [WSBW01], rays traverse the BVH in groups sized by the SIMD vector width. ISAs supporting AVX/AVX2 will create packets of eight rays while SSE capable machines will produce packets of only four rays. When traversing the BVH with a packet, each active ray must consider the same BVH node at the same time as all other rays in the packet. The packeted traversal algorithm is implemented very similarly to the SPMD algorithm, however all traversal state tracking variables are implemented using ISPC uniform types and must be treated as if they were shared memory between multiple threads of execution. If a ray has terminated earlier than its peers either by exiting the volume or successful intersection/sampling then that SIMD lane will be masked off for the remainder of the traversal.

Our methodology for comparing those two methods was to collect performance metrics in terms of rays per second for each of our test datasets and compare their average value as well as how well each algorithm scales as cell count and therefore tree complexity increases. Each dataset tested is shown in Figure 4.

From Table 1, we can see that the packeted traversal algorithm is more performant with all our test datasets and scales significantly better as well. In particular, packeted traversal achieves teh highest performance increase with our hexahedral (Earthquake) dataset; the significant increase in percentage speedup from packet traversal of the hexehedral dataset in comparison to the tetrahedral datasets is due to the scalar overhead incurred during intersection being dwarfed by the amount of traversal work necessary for such a large number of cells. Based on these results we choose to implement a packet traversal as our default algorithm and all future results will reflect an assumption of using packet traversal.

6. Application to Isosurface Ray Tracing

We render an implicit isosurface that is calculated at traversal time with no previous isosurface extraction. This is accomplished by interpolation of the values at the vertices that make up each primitive. By building a *min-max BVH* optimized for ray traversal over all of the primitives that make up a volume we are able to achieve interactive frame rates.

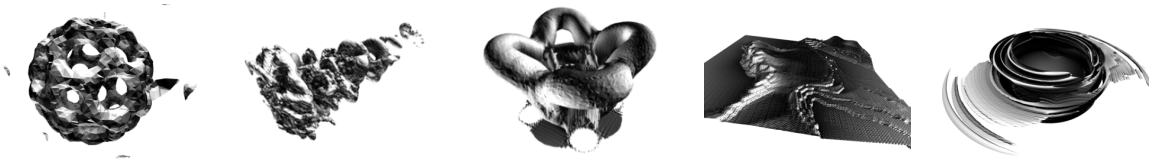
6.1. Multiple Isosurfaces

There are two cases which can create multiple isosurfaces in a single scene. The first case is that a scene may have multiple volumes each represented as an isosurface. The second case is that of a single volume represented by multiple isosurfaces – which in turn implies two possible implementations.

For the case in which there are multiple volumes, implementation is trivial. Since each volume is represented by its own geometry object all interactions are already provided by the existing OSPRay systems.

Multiple isosurfaces generated from a single geometry can be implemented naively by simply re-traversing the *min-max BVH* for each isosurface individually. However, such a naive implementation would perform poorly. Instead we execute traversal for all isosurfaces at once, and test for all isovalues at each step. If any isovalue intersection for the group of isovalues is successful we return a hit. This allows us to avoid excessive re-traversal of the *min-max BVH*.

For all cases, after the intersection is returned it is left to the renderer to decide whether or not to continue tracing from the intersection location in order to find intersections with other surfaces in the scene.



Bucky Ball (Bucky) (177k tetrahedra) | Turbulent Jet (T-jet) (1M tetrahedra) | Jets (12M tetrahedra) | San Francisco Bay (SF1) (14M tetrahedra) | Japan Earthquake (47M hexahedra)

Figure 4: *The five datasets used in our experiments (in increasing geometric complexity), in this figure rendered as isosurfaces using an ambient occlusion renderer.*

6.2. Implicit Min-Max BVH Traversal

When rendering implicit isosurfaces the *min-max BVH* is traversed in search of cells containing any isovalues of interest. Subtrees in which our isovalue of interest may not exist are skipped over entirely.

6.3. ISPC Implementation

On the ISPC side, we render isosurfaces by wrapping the *min-max BVH* in an extended OSPRay `Geometry` type named `IsosurfaceGeometry`. `IsosurfaceGeometry` is treated by OSPRay just like any other `Geometry` type and can be used by any OSPRay surface renderer. The `IsosurfaceGeometry` type contains a *min-max BVH*, and, in order to work as an OSPRay Geometry has to implement two functions inherited from the base geometry class. The `intersect` function which is the interface between the ray caster and the geometry itself and is used to generate ray-surface intersection information. The `postIntersect` function also needs to be implemented; it uses the intersection information generated by the `intersect` function to interface between the geometry and the renderer for the purpose of shading.

6.4. Ray-Cell Isosurface Intersection

Once primitives potentially containing the isovalue are found via traversing the *min-max BVH*, we then do a ray-isosurface intersection test with those primitives, using the Neubauer method [NMHW02] (also see Figure 5). The surface normal on the hit point (if found) is generated through central differences.

6.5. Shading

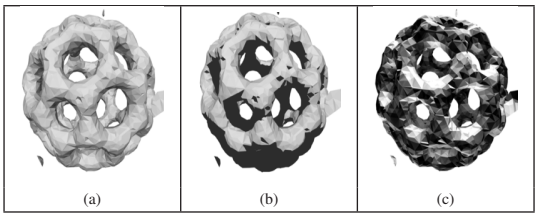
Shading the implicitly generated isosurfaces is handled by the already implemented renderers in the OSPRay framework. The central difference is calculated and used as the surface normal at intersection time and the rest is handled by the existing renderer with no modifications necessary.

7. Application to Direct Volume Ray Tracing

OSPRay volume rendering is sample based. The renderer will generate samples which are tested against the volume.

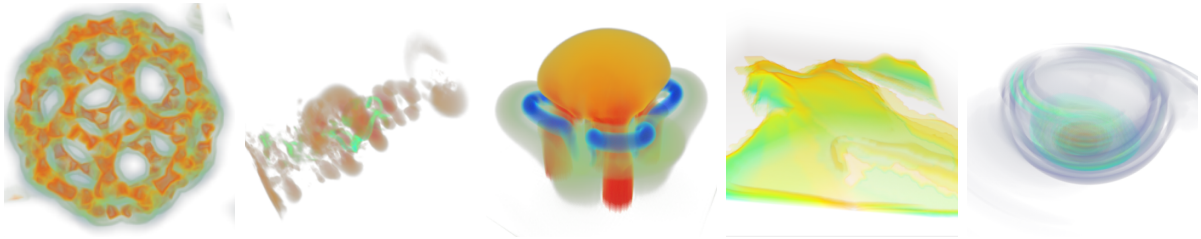
```
t_in = inf; t_out = -inf;
for each cell plane do
    t_phit = intersect(ray, plane)
    t_in = min(t_phit, t_in) t_out = max(t_phit, t_out)
end
if t_in > t_out then return NO_HIT
t_0 = t_in; t_1 = t_out; v_0 = interpolate(ray, cell, t_0); v_1 =
interpolate(ray, cell, t_1);
for i=1..N do
    t = t_0 + (t_1 − t_0) (iso−v_0)/(v_1−v_0)
    if sign(interpolate(ray, cell, t)) == sign(v_0 - iso)
    then
        t_0 = t; v_0 = interpolate(ray, cell, t)
    else
        t_1 = t; v_1 = interpolate(ray, cell, t)
    end
end
t_hit = t_0 + (t_1 − t_0) (iso−v_0)/(v_1−v_0)
return t_hit
```

Figure 5: *Our cell intersection algorithm (used for all cell types). We first test that the ray intersects a given cell, generating t_{in} and t_{out} in the process. We then apply Neubauer's method with N=2 to generate t_{hit} through successively subdividing the line segment intersecting the cell and interpolating along it.*



(a) (b) (c)

Figure 6: *The bucky ball volume dataset rendered with (a) no shadows, (b) shadows, (c) ambient occlusion We can trivially choose different preexisting renderers or even extend OSPRay with a new renderer with which to visualize the dataset when rendered as an implicit isosurface.*



Bucky Ball (Bucky) (177k tetrahedra) | Turbulent Jet (T-jet) (1M tetrahedra) | Jets (12M tetrahedra) | San Francisco Bay (SF1) (14M tetrahedra) | Japan Earthquake (47M hexahedra)

Figure 7: *The same datasets as in Figure 4, this time rendered with direct volume ray casting using appropriate transfer functions.*

We extend OSPRay's existing structured `Volume` type with a new subclass supporting unstructured volumes. The new `Volume` type for unstructured data contains a full `Geometry` object as is used for the implicit isosurface rendering.

To interface properly with the OSPRay volume renderer new `Volume` type must implement a `computeSample` function which returns the interpolated value of the scalar grid within the `Volume` at a given sample position. The `computeSample` function is required to traverse the *min-max BVH* with regards to the sample position. It is important to note that since a given sample position can exist within both subtrees of the *min-max BVH* we must keep a stack of subtrees that have not yet been visited so that we can backtrack if necessary and continue traversal. Once we find a leaf that potentially contains a primitive for which our sample position is valid we test each primitive in turn to check that the sample position is contained within the primitive. Much of this final primitive intersection code is highly scalar in nature which does impact performance.

7.1. ISPC Implementation

Just like for our implicit isosurface rendering the rays used to generate sample positions are handled as a packet. For each ray in a packet we sample along it until we either sample outside of the scene, or it is not possible for further samples to contribute to the final image.

Sampling is handled during traversal of the *min-max BVH* structure. Once it is determined that the sample position exists within a cell we simply calculate the scalar value at the sample position by interpolation from the cell vertices. For illustrative purposes interpolation from a tetrahedral cell is detailed in Figure 8.

7.2. Amortizing Tree Traversal Overhead

To help amortize the overhead of traversing the *min-max BVH* for each sample we generate a group of samples along each ray all at once. We are then traversing the tree with a packet of groups of samples, rather than a packet of samples. We speculatively descend the *min-max BVH* if any sample

```
inline varying float interpolate (uniform Tet &tet, const
    varying vec3f P)
{
  const varying float u =
      dot(P, tet.plane[1].N)+tet.plane[1].d;
  const varying float v =
      dot(P, tet.plane[2].N)+tet.plane[2].d;
  const varying float w =
      dot(P, tet.plane[3].N)+tet.plane[3].d;
  return
  (1.f-u-v-w)*tet.vertex[0].val+
  u * tet.vertex[1].val+
  v * tet.vertex[2].val+
  w * tet.vertex[3].val;
}
```

Figure 8: *Scalar value interpolation from a tetrahedral cell via trilinear interpolation.*

	Group Size					
Dataset	1	4	8	16	24	32
Bucky	54.6	**56.5**	56.0	54.9	52.4	49.6
Jets	5.2	5.2	**5.4**	5.3	5.2	5.1
T-jet	4.0	4.4	**4.5**	**4.5**	4.4	4.3
SF1	**9.9**	9.6	9.5	9.0	8.7	8.4
Earthquake	12	13.7	15.9	17.7	**17.8**	17.3
Average difference from baseline across datasets						
	0	+.74	+1.12	**+1.14**	+.56	-.2

Table 2: *A comparison of different sample group sizes for each dataset as measured on our* Xeon Node. *Values are in millions of rays per second. The values in bold represent the highest achieved for the dataset.*

for any ray is found to exist within a given subtree. This will result in some unnecessary traversals, but is offset by the number of traversals that no longer have to start from scratch at the root of the tree. To a lesser extent, we also have a performance gain from a reduction in the number of function calls made since we also no longer need to call `traverse` for individual sample positions and, in particular, have fewer memory accesses.

In Table 2 we evaluate traversal performance for various sample group sizes and observe a trend of larger sample group sizes correlating with higher performance on relatively larger datasets with an outlier of the SF1 dataset. The SF1 dataset performs poorly with large sample groups due to

Step size	Group Size 1	4	8	16	24	32
2.5	2.4	2.9	3.1	3.2	**3.3**	**3.3**
5	3.5	3.9	**4.1**	**4.1**	**4.1**	4.0
7.5	4.5	4.6	**4.7**	**4.7**	**4.7**	4.5
10	5.2	5.3	**5.4**	5.3	5.2	4.9
12.5	5.9	**6.0**	**6.0**	5.8	5.7	5.5
15	**6.5**	**6.5**	**6.5**	6.3	6.1	6.0

Table 3: *Performance scaling of sample group size against ray marching distance on the* Xeon Node*. All measurements are in millions of rays per second, and all measurements were taken using the Jets dataset. The values in bold are the highest throughput achieved for the step size noted on the left of the row. For reference, the ray marching distance used for the Jets dataset in Table 2 was 10.*

how thin the volume is relative to the view angle, as such our extra samples would have been culled as they are beyond the volume bounds but are forced to traverse the BVH anyway.

In Table 3 we explore the relationship between ray marching distance and sample group width. We found that group size and step size have an inverse relationship with regards to performance; that is to say larger group sizes give a larger performance increase only when step size is relatively small. This relationship is important because a smaller step size implies a greater visual fidelity, and as such it is notable that it is possible to regain some lost performance when enhancing visual fidelity by increasing sample group size.

Given our observations in Table 2 and Table 3 we choose a sample group size of 16 for future measurements as this size on average gave the best performance for the level of visual fidelity shown for each scene in Figure 7.

8. Results and Discussion

We gathered performance data for two workstation nodes.

- A node equipped with two Intel® Xeon® E5-2687W v3 processors running at 3.1GHz and with 128GB of system RAM (referred to as the *Xeon Node*)
- A node equipped with two Intel® Xeon® E5-2680 processors running at 2.7GHz 46GB os system RAM and three Intel® Xeon Phi™ 7120A coprocessors (referred to as the *Phi Node*)

Our method is performs well on a single general compute node equipped with a Xeon CPU (Table 4) with the ray throughput equivalent to 22.3 frames per second for implicit isosurface ray casting and 8.5 frames a second for DVR of our largest dataset (Earthquake) assuming a 1080p resolution. On the *Phi Node*, we measured ray throughput equivalent to 48.9 frames per second when performing implicit isosurface ray casting and 8 frames per second for DVR with the same dataset and resolution (Table 4).

Interestingly DVR rendering does not scale as well as implicit isosurface rendering on our Xeon Phi node and for the Earthquake dataset actually performs more slowly than on a standard Xeon processor. We currently believe that this is because in several cases our method devolves into near-scalar execution, a task for which the Xeon Phi is not well suited. Degeneration into scalar execution is due to our current lack of hybrid traversal algorithms; the addition of which would allow packets that have diverged into single-ray traversal to still take advantage of SIMD execution. With hybrid traversal algorithms we expect that performance on the *Phi Node* would increase significantly.

Dataset	Size	Xeon Node ISO	Xeon Node DVR	Phi Node ISO	Phi Node DVR
Bucky	177k tets	103.2	54.9	365.0	25.3
Jets	1M tets	92.0	5.3	282.0	9.7
T-jet	12M tets	109.4	4.5	418.9	9.8
SF1	14M tets	78.7	9.0	190.8	12.1
Earthquake	47M hexes	46.3	17.7	101.6	16.7

Table 4: *Performance in millions of rays per second (higher is better) for both the Xeon and Phi nodes. Isosurface (ISO) renderings are done with the OSPRay 'obj' renderer and shadows are disabled and for the Phi node we generated 100 samples per pixel per frame rather than the standard single sample per pixel per frame. DVR renderings were done with no differences in settings between the workstation and Phi nodes.*

8.1. Comparison to Other CPU Methods

Unlike some other CPU based isosurface visualization algorithms our method does not require any pre-computation of the volume data to extract an isosurface. Isosurfaces are generated implicitly and never exist explicitly in memory, as such our memory overhead during isosurface rendering consists only of that which is needed to store the *min-max BVH*.

Our method also allows for usage of the same BVH tree structure to do both isosurface rendering and direct volume ray tracing. As such it is feasible to rapidly and dynamically toggle between isosurface and volume rendering during run time.

8.2. Comparison to GPU Methods

Unlike the popular projected tetrahedra technique ours is not an approximation. Rather than approximating cell projection by splatting transparent triangles onto the frame buffer we are directly sampling the data for each pixel.

Although similar methods have been developed for GPUs they still suffer from the constraints of available GPU memory which can be orders of magnitude lower than the available system memory. For instance, the Earthquake dataset consumes approximately 10GB of memory at runtime for the grid, BVH, and other in memory data structures and many datasets are orders of magnitude larger. Our CPU method can be compiled and run on any general compute node and requires no specialized hardware.

9. Conclusions

We have presented a technique for interactive visualization of homogeneous unstructured volumetric grids usable for both implicit isosurface visualization and direct volume ray tracing. We used packeted *min-max BVH* traversal in a ray tracer implemented with an SPMD compiler to achieve high performance across both CPUs and Intel Xeon Phi coprocessors while avoiding the need to write specialized code for each platform. We find our method to perform adequately and with minimal memory overhead on general compute nodes without the use of a GPU.

The method as presented performs well for isosurface rendering and can also be used for direct volume ray tracing. Although performance with regards to direct volume ray casting is lower than desired, through progressive refinement methods our algorithm is capable of tolerating high step sizes when sampling along a ray which helps to solve our performance issues while maintaining high image quality.

The main barrier to using our method for in-situ visualization of simulation data is currently the build time required for our *min-max BVH*, and our rendering performance is currently limited by our intersection and interpolation kernels. We leave as future work a parallelized and non-scalar *min-max BVH* build and update kernels such that quickly rebuilding or updating an existing BVH for time varying data during simulation is feasible as well as finding methods to either reduce the number of intersections/interpolations necessary to produce quality images or reduce the number of attempts to intersect/interpolate irrelevant nodes. Future work may also include auto-tuning some of the fixed parameters we are currently using, such as sample group size. It would also likely be a large performance gain to supplement our technique with some low-level vectorized kernels for tasks for which we currently default back to scalar execution.

References

[CBW*12] CHILDS H., BRUGGER E., WHITLOCK B., MEREDITH J., AHERN S., PUGMIRE D., BIAGAS K., MILLER M., HARRISON C., WEBER G. H., KRISHNAN H., FOGAL T., SANDERSON A., GARTH C., BETHEL E. W., CAMP D., RÜBEL O., DURANT M., FAVRE J. M., NAVRÁTIL P.: VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*. Oct 2012, pp. 357–372. 1, 2

[CDM06] CHILDS H., DUCHAINEAU M., MA K.-L.: A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization* (2006), EGPGV '06, pp. 153–161. 2

[EGMM07] EISEMANN M., GROSCH T., MAGNOR M., MÃIJLLER S.: Automatic Creation of Object Hierarchies for Ray Tracing Dynamic Scenes. In *IN WSCG SHORT PAPERS PROCEEDINGS* (2007). 4

[Hen04] HENDERSON A.: *The ParaView Guide: A Parallel Visualization Application*. Kitware, Nov. 2004. 1, 2

[KTW*11] KNOLL A., THELEN S., WALD I., HANSEN C. D., HAGEN H., PAPKA M. E.: Full-resolution Interactive CPU Volume Rendering with Coherent BVH Traversal. In *Proceedings of the 2011 IEEE Pacific Visualization Symposium* (2011), pp. 3–10. 2

[LC87] LORENSEN W. E., CLINE H. E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques* (1987), SIGGRAPH '87, pp. 163–169. 1, 2

[MB90] MACDONALD D. J., BOOTH K. S.: Heuristics for Ray Tracing Using Space Subdivision. *Vis. Comput. 6*, 3 (May 1990), 153–166. 4

[MHDG11] MUIGG P., HADWIGER M., DOLEISCH H., GRÖLLER E.: Interactive Volume Visualization of General Polyhedral Grids. *Visualization and Computer Graphics, IEEE Transactions on 17*, 12 (2011), 2115–2124. 2

[MS06] MARMITT G., SLUSALLEK P.: Fast Ray Traversal of Tetrahedral and Hexahedral Meshes for Direct Volume Rendering. In *Eurographics/IEEE-VGTC Symposium on Visualization (EuroVIS)* (2006), pp. 235–242. 2

[NMHW02] NEUBAUER A., MROZ L., HAUSER H., WEGENKITTL R.: Cell-based First-hit Ray Casting. In *Proceedings of the Symposium on Data Visualisation 2002* (2002), VISSYM '02, pp. 77–ff. 6

[PM12] PHARR M., MARK B.: ISPC - A SPMD compiler for high-performance CPU programming. In *Proceedings of Innovative Parallel Computing (inPar)* (2012). 3

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization '98* (October 1998), pp. 233–238. 2

[SCCB05] SILVA C. T., COMBA J. L. D., CALLAHAN S. P., BERNARDON F. F.: A Survey of GPU-Based Volume Rendering of Unstructured Grids. *Brazilian Journal of Theoretic and Applied Computing (RITA 12* (2005), 9–29. 2

[Sch06] SCHROEDER W.: *The visualization toolkit : an object-oriented approach to 3D graphics*. Kitware, 2006. 2

[ST90] SHIRLEY P., TUCHMAN A.: A Polygonal Approximation to Direct Scalar Volume Rendering. In *Proceedings of the 1990 Workshop on Volume Visualization* (1990), VVS '90, pp. 63–70. 2

[WFKH07] WALD I., FRIEDRICH H., KNOLL A., HANSEN C. D.: *Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes*. Tech. Rep. UUSCI-2007-003, SCI Institute, University of Utah, 2007. (conditionally accepted at IEEE Visualization 2007). 2, 3

[WFM*05] WALD I., FRIEDRICH H., MARMITT G., SLUSALLEK P., SEIDEL H.-P.: Faster Isosurface Ray Tracing using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics 11*, 5 (2005), 562–573. 2, 4

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164. (Proceedings of Eurographics 2001). 3, 5

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree - A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) 33* (2014). 2, 4