Full-Resolution Interactive CPU Volume Rendering with Coherent BVH Traversal

Aaron Knoll[*] Argonne National Laboratory
Sebastian Thelen[†] TU Kaiserslautern
Ingo Wald[‡] Intel Corporation
Charles D. Hansen[§] University of Utah
Hans Hagen[¶] TU Kaiserslautern
Michael E. Papka[‖] Argonne National Laboratory

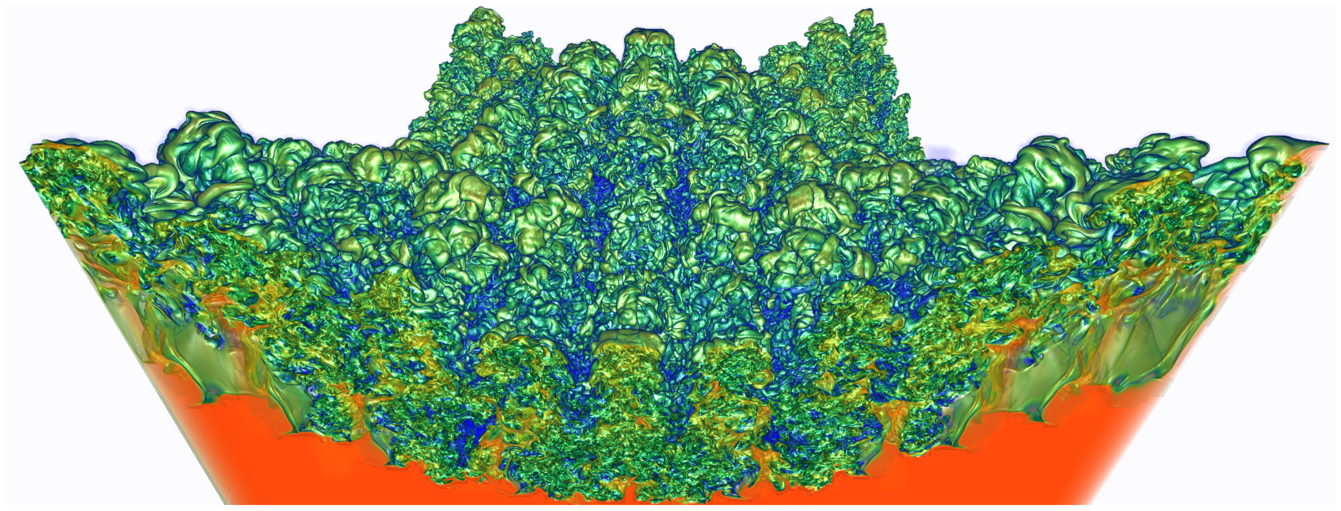


Figure 1: 2048x2048x1920 Richtmyer Meshkov instability CFD simulation, rendered at full data resolution (without LOD) into a 2048x768 frame buffer at 5.7 fps on a dual 4-core 2.67 GHz Intel Core i7 (X5550) workstation with 32 GB RAM, outperforming an out-of-core renderer on a NVIDIA 285GTX GPU by 80x.

Abstract

We present an efficient method for volume rendering by raycasting on the CPU. We employ coherent packet traversal of an implicit bounding volume hierarchy, heuristically pruned using preintegrated transfer functions, to exploit empty or homogeneous space. We also detail SIMD optimizations for volumetric integration, trilinear interpolation, and gradient lighting. The resulting system performs well on low-end and laptop hardware, and can outperform out-of-core GPU methods by orders of magnitude when rendering large volumes without level-of-detail (LOD) on a workstation. We show that, while slower than GPU methods for low-resolution volumes, an optimized CPU renderer does not require LOD to achieve interactive performance on large data sets.

1 Introduction

Direct volume rendering (DVR) is now a mature algorithm in computer graphics, employed in scientific and medical visualization of scalar field data, and increasingly in animated effects in games and production rendering. Because of its high computational cost, volume rendering has almost exclusively been implemented on graphics hardware. With dedicated memory and efficient built-in interpolation, GPU's have proven efficient at rendering moderate-size volume data interactively. Conversely, relatively few works have optimized volume rendering on the CPU, due to its comparatively low computational throughput.

Nonetheless, the CPU is potentially a desirable platform for volume rendering. In laptops and netbooks, GPU resources are frequently absent or are much less powerful than their desktop counterparts. In high performance computing, it is desirable to visualize large data directly on a CPU cluster, as opposed to downsampling or employing multiresolution rendering algorithms. To render large data, out-of-core GPU systems rely on level-of-detail (LOD) and progressive rendering to achieve interactive performance. While this approach is well suited for exploration, a GPU renderer can in fact underperform an optimized CPU system when rendering large data at full resolution, due to the CPU's direct access to main memory, multilevel cache, and efficiency in branch-intensive spatial data structure traversal.

This paper decribes a CPU volume rendering implementation that outperforms GPU approaches at both low and high ends of the hardware spectrum. This is accomplished partly by efficient instruction-level optimization and, more significantly, by heuristic traversal of a bounding volume hierarchy (BVH) acceleration structure. The main contributions of our system are a technique for traversing a min-max implicit BVH [20] using a preintegrated transfer function for heuristical pruning; a method for quickly integrating low-variance regions of the volume using ray packets and Streaming SIMD Extension (SSE) vector instructions; and faster methods for computing trilinear interpolation, gradient lighting and DVR integration for single rays in SSE. While technical, these enhancements are crucial to achieving interactive performance, and result in a scalable system that outperforms GPU DVR by over an order of magnitude when rendering large data.

2 Related Work

Volume rendering was first demonstrated in the software ray caster of Levoy [16]. With the introduction of fast rasterization hardware, texture slicing became the dominant method [2]. Preintegration [3] improved classification quality by separating integration of the scalar field and transfer function. Ray casting methods emerged on loop-capable programmable GPU's [13] and achieved performance parity with slicing methods on the NVIDIA G80 architecture. GPU DVR methods have also employed acceleration structures such as an octree [6] or kd-tree [9]. Interactive rendering of large data has proven a challenge for single-GPU renderers; due to GPU memory limits focus has shifted to using multi-GPU clusters to render data larger than 1 GB [4].

On the CPU, shear warp [14] remains a state-of-the art vol-

[*]e-mail: knoll@mcs.anl.gov
[†]e-mail:s_thelen@informatik.uni-kl.de
[‡]e-mail:ingo.wald@intel.com
[§]e-mail:hansen@cs.utah.edu
[¶]e-mail:hagen@informatik.uni-kl.de
[‖]e-mail:papka@anl.gov

ume rendering algorithm, which employs bilinear interpolation and affine transformations on axis-aligned slices, and delivered interactive performance for small volumes on mid-1990's hardware. Another efficient CPU DVR system was the Ultravis system [10], which achieved 10 fps on a dual-Pentium 3 500 MHz machine for upsampled 256^2 images. It used SSE assembly, a 3D distance map for space skipping, and aggressive cache management. As opposed to low-level optimization, later CPU volume rendering work has focused largely on distribution and scalability to multiple processors and larger data [7]. The work of Parker et al. [17] in interactive isosurface ray casting prompted numerous extensions including optimization with kd-trees and SSE [21], rendering from compressed octrees [**?**, 12], and out-of-core LOD [5]. We use a coherent packet-based CPU ray tracing framework to take advantage of efficient packet BVH traversal [19], similar to the tetrahedral volume isosurface ray tracing of Wald et al. [20]. Smelyanskiy et al. [18] show that for sufficiently large volumes, a multicore CPU implementation can outperform a GPU implementation. While this comparison handicaps the GPU by employing nearest-neighbor filtering, it nonetheless highlights the potential of optimized CPU approaches for rendering large data.

Splatting [23] is an alternative algorithm for adaptive direct volume rendering. Given its different characteristics in preprocess time, scalability and quality, we do not compare directly to splatting, but note that such approaches could prove competitive.

3 Background

Direct volume rendering commonly refers to a process in which samples from the volume are classified, lit, and blended in image-space order, irrespective of rendering algorithm and in contrast to isosurfacing, maximum intensity projection, and other modalities. For the underlying optical and mathematical models, we refer the reader to the original paper of Levoy [16] as well as that of Engel et al. [3] concerning preintegrated transfer functions. DVR integrates the radiative transport equation (Equation 1) on a ray segment along $[a,b]$. Given a transfer function ρ, where ρ_E is the emissive term or color, ρ_α is the opacity, and given a scalar field function $f(t) = f(\vec{O}+t\vec{D}) = f(\vec{R}(t))$, irradiance can be evaluated as:

$$I(a,b) = \int_a^b \rho_E(f(s))\rho_\alpha(f(s))e^{-\int_a^s \rho_\alpha(f(t))dt}ds \tag{1}$$

Evaluating $\rho(f)$ implies postclassification, where the transfer function is evaluated after the scalar field function. This integral is approximated discretely via a Riemann sum,

$$I \approx \sum_{i=0}^{n} \check{\rho}_E(i) \prod_{j=0}^{i-1}(1-\alpha_j). \tag{2}$$

where $\check{\rho}_E$ is approximated discretely along the ray as:

$$\check{\rho}_E(i) \approx \rho_\alpha(f(i\,\delta t))\rho_E(f(i\,\delta t)) \tag{3}$$

Preintegration employs a separate integral in transfer function space to estimate $\check{\rho}_E$ and ρ_a [3], specifically the Riemann sum of irradiance between two samples $f_a = f(a)$ and $f_b = f(b)$, assuming linear spacing of f values between these points. Typically, the colors $\check{\rho}_E(i)$ are associated (integrated alongside α_i).

$$\alpha_i \approx 1 - e^{-\int_0^1 \rho_\alpha((1-\omega)f_a+\omega f_b)d\ \mathrm{d}\omega} \tag{4}$$

Preintegration can improve the sampling behavior when the transfer function is sharp, and it is simple to implement as an optional classification. We also use the preintegrated table to optimize our implicit BVH traversal (Section 5).

Rather than sample uniformly along the ray, we use differential sampling [11]. This scheme increments the step between samples by a first-order differential, resulting in a quadratic pattern that samples more frequently closer to the eye. It ensures a constant sampling rate in image space and improves performance for equivalent visual quality.

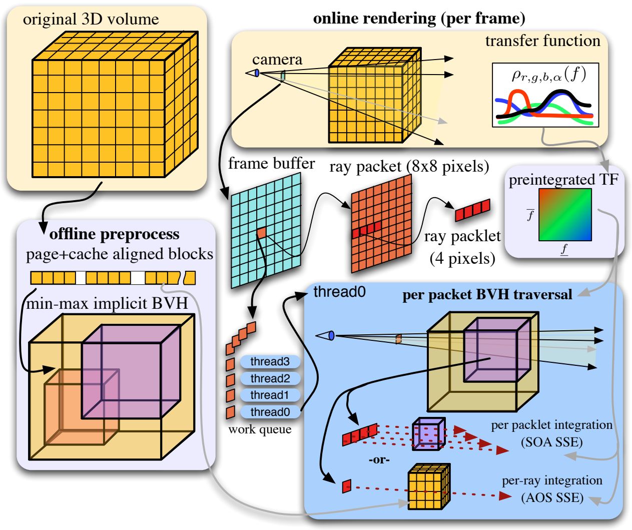


Figure 2: Overview of our system and algorithm pipeline.

4 Fast Direct Volume Ray Casting on the CPU

In general, direct volume rendering can be accelerated by reducing the total number of samples taken, and by lowering the cost of computing and integrating each sample. Our system performs both, employing a coherent BVH traversal method for exploiting empty and low-variance regions of the volume, and an optimized low-level SIMD routine for DVR integration with trilinear interpolation. Traversal is called from a multithreaded packet ray tracer distributed over image space. An overview of our system pipeline is shown Figure 2.

4.1 Domain Decomposition with the BVH

Minimizing the number of DVR samples entails space-skipping and adaptive methods. On the GPU, these are typically achieved by rasterization of a bounding proxy [8] and block-based multiresolution LOD. In our CPU system, we employ efficient traversal of a BVH acceleration structure and forgo LOD entirely. The efficiency of the acceleration structure depends on the amount of empty space in the scene. The cost of traversing the structure per-ray often outweighs gains from computing fewer DVR samples. Coherent traversal amortizes this cost over the rays in the packet, changing this dynamic significantly and making acceleration structures practical for denser volumes with less empty space. To further improve efficiency, we introduce novel heuristics for pruning the BVH based on the preintegrated lookup table of the user-chosen transfer function. To the best of our knowledge, this system represents the first pairing of coherent packet traversal with structured direct volume rendering. We describe our coherent BVH traversal approach in detail in Section 5.

4.2 Optimizing Integration with SSE

Optimizing brute-force DVR integration entails limiting cache and computational bottlenecks to maximize throughput. GPU hardware excels at this, with built-in 3D texture fetching and interpolation and numerous execution units. Ironically, implementing efficient DVR integration on the CPU is more challenging and less graceful, necessitating low-level SSE vectorization and efficient strategies for addressing the volume data in memory. We contribute a low-level yet flexible integration routine that can be employed in either a conventional single-ray tracer or a packet ray tracer such as our coherent BVH system. While efficient memory management is crucial for out-of-core GPU systems, it is less so for CPU approaches where the entire volume resides in main memory and a multilevel cache hierarchy is implemented natively in hardware. We improve performance chiefly by amortizing the cost of address translation, exploiting SSE swizzling behavior for trilinear interpolation, and performing blending and sampling operations directly

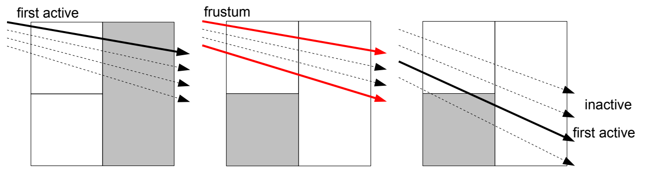


Figure 3: Coherent BVH traversal of interior nodes. **Left:** the first active ray (or SSE packlet) in a packet is speculatively tested against a child node bounding box. **Center:** if this test fails, an interval arithmetic frustum test tests whether we can discard the entire packet. **Right:** only then must we test all rays (packlets) against the node. By incrementing the “first active” ray for this level of the traversal stack, we can avoid redundant intersection tests.

on SSE vectors using carefully chosen masks. We have deliberately avoided precomputed gradients used in other CPU approaches [10], opting instead for more efficient computation of analytical gradients. In all, we achieve trilinear interpolation and Phong illumination at modest cost relative to a CPU naive implementation, with no associated storage overhead. Details are given in Section 6.

5 Implicit BVH for Structured Volume Rendering

The main algorithmic contribution of our system is the use of coherent BVH traversal to accelerate volume rendering, reducing the total number of DVR samples by exploiting empty and homogeneous space. Domain decomposition schemes are successful only when the gains justify the cost of traversing the structure; this limitation often discourages per-ray traversal. Coherent algorithms traverse the acceleration structure in groups, or *packets*, of rays, significantly lowering the per-ray traversal cost. We use the coherent BVH approach of Wald et al. [19], specifically the *implicit BVH* employing a min-max tree [20]. We chose the BVH expressly for this fast wide-packet traversal algorithm, which scales well to larger models compared to coherent grid [22] or octree [12] variants. Our general approach is to build an implicit BVH as an offline preprocess, and then to traverse it per-packet using the preintegrated transfer function for dynamic culling and pruning heuristics.

5.1 Construction

The BVH construction consists of two stages. The first is a bottom-up enumeration of leaf nodes at some chosen base size L, corresponding to leaf bricks of L^3 voxels. Small values ($L = 1,2$) generate large trees and are beneficial only for sharp isosurface-like transfer functions. In most instances, particularly for large data, performance with $L = 4$ or $L = 8$ is equally good. In building the leaf nodes, we compute the minimum and maximum values not only for that brick but also for the forward-neighbors (up to $L+1$), accounting for the trilinear interpolation stencil. If the space is empty, we discard the leaf; otherwise we append it to a list. The min-max values of both the leaves and the BVH itself are purely data-dependent, and independent from the user’s choice of transfer function.

The subsequent step is a simple top-down median split BVH build based on the list of initial nonempty leaf blocks. This consists of computing centroids for each leaf, sorting these centroid separately along the X,Y and Z axes. Then, we recursively pivot in the center of each sorted list, splitting at the X, Y or Z position yielding the largest spatial diameter, and terminating when a leaf block has been reached. Requiring only an $O(N \log N)$ sort and an $O(N)$ split process for N primitives, the build is in practice fast. While structured volumes contain no overlapping primitives, object-space partitioning generates well-balanced trees compared to space-partitioning octree or kd-trees.

Having created our tree, we compute min-max values for each node using a top-down $O(\log N)$ routine. Although it is possible to precompute the pruning metrics within BVH nodes whenever the transfer function changes, this approach can reduce interactivity for large data. Computing heuristics dynamically during packet-BVH traversal is equally fast and incurs little penalty.

5.2 Coherent BVH Traversal

Our traversal is essentially that of the coherent implicit BVH [20] with heuristics for pruning the tree during descent based on the preintegrated transfer function. Although packets of 16x16 rays worked best in previous applications, we find 8x8 packets perform better in DVR, presumably due to the more costly primitive intersection. The algorithm is sketched in Listing 1.

Listing 1: Coherent BVH traversal pseudocode

```
void traverse(Node* nodes, RayPacket& packet){
  int id = 0; //BVH node index
  int first_active_packlet = 0; //first active SSE packlet in the packet
  int stack[32]; //BVH stack
  int fa_stack[32]; //stack for recalling first-active packlet
  int d=0;

  while(true){
    Node& node = nodes[id];
    //speculative min-max tree descent
    while(true){
      if (node.child == 0) break; //child is empty, i.e. leaf
      if (node_is_empty(nodes[node.child + 0])){ id = child + 1; continue; }
      if (node_is_empty(nodes[node.child + 1])){ id = child + 0; continue; }
      break;
    }
    //speculative first-active traversal
    int first_active_packlet = first_that_intersects(packet, nodes[id]);
    if (first_active_packlet < RayPacket::MAX_PACKLETS){ //if any packlet hit
      bool csv = constant_subvolume(node);
      if (node.child && !node_is_leaf(node) && !csv){ //interior
        int front_child = closest_child(node, packet);
        stack[d] = node.child + 1 - front_child;
        fa_stack[d] = first_active_packlet;
        id = node.child + front_child;
        d++;
        continue;
      }
      else if (node.child){ //leaf
        if (csv)
          dvr_constant(node, packet);
        else
          dvr(node, packet)
      }
    }
    if (d==0) return;
    id = stack[--d];
    first_active_packlet = fa_stack[d];
  }
}
```

As depicted in Figure 3, coherent BVH traversal [19] descends the tree, speculatively testing the first ray in a packet, and employing an interval arithmetic frustum test when it misses – in effect finding an interval of rays (when existing) that intersect each BVH node. For efficiency, intersection tests are performed 4-at-a-time in SIMD on a group of four rays referred to as a *packlet*. Redundant intersections are avoided by maintaining the index of the first-active packlet on the traversal stack and advancing this index to the next hit. The algorithm ascends the tree when both children have been examined. When a leaf is reached, all active packlets starting with the first-active are intersected against the leaf bounds. The ray-leaf bounding box test gives us the entry and exit distances for our DVR intersection algorithm, either the constant subvolume method or our horizontal (array-of-structs) SSE method in Section 6.

In the implicit BVH [20], we also speculatively descend based on the min-max values associated with BVH nodes, namely when one child but not the other has a range of scalar values overlapping the transfer function domain. We employ metrics based on the preintegrated transfer function to interpret the min-max interval, and designate BVH nodes as empty, interior, or leaves. Similarly, we can analyze a leaf to optionally employ a fast constant subvolume integration routine as opposed to the per-voxel SSE DVR integration routine. We note that all our heuristics are computed on-the-fly per-packet, with no precomputation necessary other than the statically built implicit BVH. These optimizations are illustrated in Figure 4, and detailed in the subsections below.

5.2.1 Empty Space Skipping

The choice of transfer function defines a subtree of the implicit BVH, which can be used to identify and prune empty regions outside the classification. Similarly to how an isosurface lies between

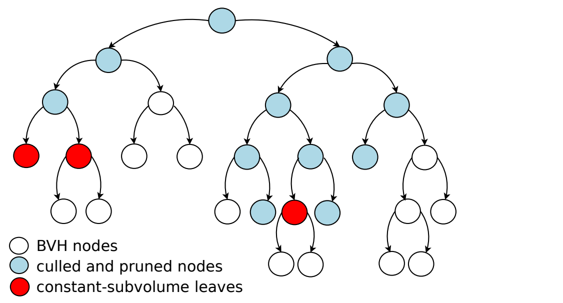


Figure 4: The implicit BVH can be heuristically pruned using the preintegrated transfer function, resulting in a smaller subtree. Similarly, it can detect constant subvolumes and perform less expensive DVR integration.

minimum and maximum values of each node of the subtree, in DVR we can check whether the transfer function contains nonzero opacity for any scalar field value in the min-max range. This is already encoded in the lookup table of the preintegrated transfer function, which estimates the integral over a min/max interval. To evaluate node_is_empty() in Listing 1, we check

$$\rho_\alpha(\underline{f},\overline{f}) > \delta_c, \tag{5}$$

where $\underline{f},\overline{f}$ denote the minimum and maximum, respectively; ρ_α is the opacity of the preintegrated transfer function (Equation 4) over $\underline{f},\overline{f}$; and δ_c is a culling threshold ($\delta_c <$ 1e-3 works well).

5.2.2 Pruning Heuristic

Always traversing to the deepest leaf nodes in the static BVH can be wasteful. A transfer function can convolve low-frequency transparent regions into high-frequency opaque ones, and vice-versa. In low-frequency and mostly transparent regions, it is desirable to use larger bounding boxes, as early termination is less likely and additional intersections are redundant. Conversely, in high-frequency regions we wish to fully traverse the BVH, subdividing as far as possible and exploiting early termination. To measure this frequency, we divide the average opacity ρ_α of a node by its relative size. To determine node_is_leaf() in Listing 1, we measure

$$\rho_\alpha(\underline{f},\overline{f})|\vec{D}_{volume}|/|\vec{D}_{box}| > \delta_p \tag{6}$$

where $|\vec{D}_{volume}|$ is the diagonal diameter of the whole volume, and $|\vec{D}_{box}|$ is the diameter of the node. In general, it is best to prune at one or two levels higher than the original leaf level of the tree. Aggressive pruning ($\delta_p = 1.5$) is best for noisy or entropic regions, while traversing further down ($\delta_p = 6$) is faster for scenes with smooth features and surfaces. Choosing multiples of 1.5 roughly corrects for the diagonal length. While we allow the user to adjust this value, $\delta_p = 1.5$ works well as a default.

5.2.3 Constant Subvolume Heuristic

We can also use preintegratation to determine regions of the volume that are sufficiently low-variance (convolved by the transfer function) to be treated as constant blocks. This subvolume can then be integrated by using a far less expensive routine, with neither per-voxel lookup nor interpolation, and using fast, vertical structure of array (SOA) SSE operations on 4 rays at a time (per packlet). Since constant regions have undefined gradient, one can forgo lighting. When used, this method delivers significant speedup.

Like the pruning metric, the metric for constant subvolume assumption is intrinsic to the transfer function and the min-max values of the node. We compute the variances in preintegrated opacity as follows, choosing a constant L_b conservatively to prevent loss of quality. We then evaluate the following heuristic, using the constant-block integration when it succeeds and the standard DVR routine when it fails, as shown with constant_subvolume() in Listing 1:

$$sup\{|\rho_\alpha(\underline{f},\overline{f}) - \rho_\alpha(\underline{f},\underline{f})|, |\rho_\alpha(\underline{f},\overline{f}) - \rho_\alpha(\overline{f},\overline{f})|\} < \delta_{sv} \tag{7}$$

Relatively small $\delta_{sv} <$ 1e-4 consistently produce good results without removing visible features. This metric can be precomputed and queried alongside the preintegrated table, though it is inexpensive to compute on the fly as well.

Constant subvolume detection is efficient at rendering scenes with homogeneous, non-empty space, such as the uniform red regions in Figure 1. When homogeneous regions are nonexistent or smaller than BVH leaves, one could still employ adaptive sampling, either per-node [12] or per-sample [15]. Such approaches are left outside the scope of this work, but we note that adaptive sampling with the BVH could be a promising avenue for performance gains.

6 SSE DVR Integration

Most SIMD-optimized ray tracers, including our coherent BVH system, store vectors as vertical structures of arrays (SOA), where direction vectors for a packlet (4 rays) are represented as three SSE registers, and computations are performed for that packlet in SIMD. This approach is efficient for most geometric primitives, including our constant subvolumes, in which numerous rays intersect the same object. However, DVR frequently projects multiple voxels to the same pixel, causing SIMD under-utilization with the SOA paradigm. Fortunately, DVR integration operates primarily on 4-vector positions ({x,y,z,t}) and colors ({r,g,b,a}). We thus employ *horizontal* SSE vector arithmetic operating on one ray at a time, using the array of structures (AOS) paradigm. From coherent BVH traversal, we simply convert from vertical SOA to individual rays using 4 SSE swizzle operations, computing a mask indicating which rays in the packlet are active. Then we iterate over the packlet, performing DVR for each active ray. Explicit C++ code is given in Listing 2 in the appendix.

6.1 Memory Layout and Interpolation

Reducing the computational and memory access costs of interpolation is the first target for optimization in DVR integration. Trilinear Lagrangian interpolation takes the form:

$$f(x,y,z) = \sum_{i,j,k=\{0,1\}} x_i y_j z_k \; v_{ijk}, \tag{8}$$

where (i,j,k) is the coordinate of the voxel vertex, v_{ijk} is the value at the vertex, $x_0 = i+1-x$, $x_1 = x-i$, and similarly for y and z with respect to j,k. Naive implementation requires over 32 muls, 34 adds, 3 casts, and 8 voxel address translations. Many of these computations are redundant or can be optimized with SIMD.

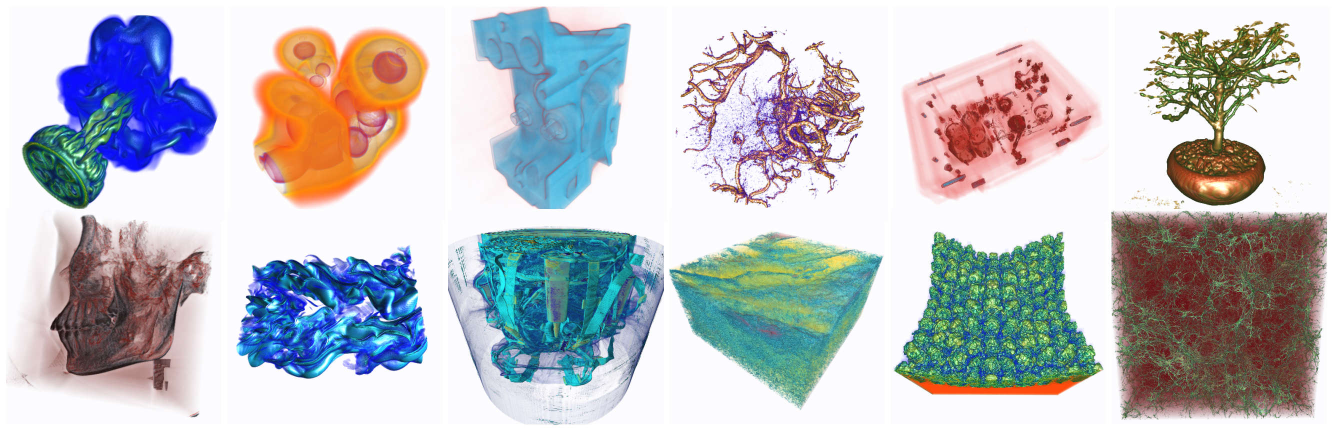
To mitigate cache thrashing and decouple performance from axis alignment, we employ a simple bricking scheme described in [17], which decomposes the volume into blocks aligned to match page (64 byte) and L1 cache (32K) sizes. This yields chunks of 4^3 voxels, which are convenient for multiples of $L = 4$. We store pointers to the X,Y and Z tables of this structure (*ls.* 53-55) and index into these tables given the 6 lower and upper voxel indices (*ls.* 83-89). We permutatively add these indices to retrieve the 8 voxel vertices, storing them in two integer SSE registers (*ls.* 91-92).

Rather than employ successive linear interpolations [10], we achieve 15% faster performance by exploiting SSE swizzling to generate the $x_i y_j z_k$ permutations with only 3 mul_ps operations and one add_ps. We combine common y,z terms to get a single SSE vector with the summed x_0 and x_1 components. With an SSE4.1 dot product instruction we can accomplish both multiplication and horizontal addition in a single instruction, followed by an SSE integer cast (*ls.*98-100). On older CPU's, we use an SSE multiplication, an SSE integer cast and 3 scalar int additions. Though an approximation, it is as fast as the dot product and yields no loss in quality.

6.2 Classification and Lighting

Classification (*ls.* 103-104) is a table lookup returning the (r,g,b,a) components at that sample. Though it makes little difference in performance, we use a 256^2 preintegrated table.

Per-sample lighting is expensive, requiring accurate gradients (i.e., derivatives of the trilinear interpolant) and computation of normalized vectors. Precomputing gradients and then interpolating them alongside the scalar field value is efficient [10]; however, it also increases storage requirements by a factor of 4, which is undesirable when rendering large volume data. To deliver efficient



		Scene			BVH			CPU – fps				GPU – fps				Ratio
Dataset	**Lit**	**Screen dims**	**Dimensions**	**Size**	**Size**	*L*	**Build**	**Core 2**	**Core i7**			**9400M**	**285GTX**			best
							time	2 core bvh	8 core gcc	sse	bvh	16 cores best †‡§	240 cores glsl †	avrc ‡	iv3d §	CPU/ GPU
heptane	p	512^2	302^3	28.5M	78M	2	.4s	**4.0**	3.7	16.0	**17.9**	2.6 ‡	55	118	39	.23x
neghip	u	512^2	64^3	256k	12M	1	.011s	**7.6**	5.5	20.8	**33.0**	6.0 †	130	184	125	.28x
engine	u	512^2	256x256x128	8M	8M	4	.67s	**3.0**	2.6	10.0	**13.1**	1.5 ‡	42	62	83	.25x
aneurism	p	512^2	256^3	16M	10M	2	.13s	**6.0**	3.1	8.3	**25.9**	1.9 ‡	50	87	77	.45x
fireset	u	512^2	512x256x512	65M	70M	4	.39s	**7.2**	1.2	12.8	**30.5**	0.9 ‡	67	89	26	.52x
bonsai	p	512^2	256^3	16M	64M	2	.13s	**4.8**	4.3	14.3	**21.2**	2.3 ‡	75	105	80	.30x
skull	d	512^2	256^3	16M	24M	4	.1s	**1.7**	1.4	5.7	**12.0**	1.4 †	79	63	36	.18x
jet	p	512^2	480x720x120	40M	124M	2	1.6s	**9.5**	6.0	24.6	**62.0**	1.9 ‡	20	80	43	.77x
backpack	d	512^2	512x512x373	65M	70M	4	.40s	**2.0**	1.2	4.5	**9.4**	.40 ‡	1.8	2.7	4.0	2.2x
zebrafish	p	512^2	900x500x910	390M	7.8M	8	2.9s	**1.0**	1.41	4.1	**11.1**	.18 §	.23	1.7	1.3	6.5x
RM	p	$1k^2$	2kx2kx1920	7.2G	1.7G	8	240s	-	.30	.99	**7.9**	-	-	-	.084	94x
enzo	p	$1k^2$	4kx3kx2k	24G	2.8G	16	403s	-	.11	.46	**1.25**	-	-	-	.028	45x

Figure 5: Small and moderate-size data benchmarked with various CPU and GPU volume renderers. Results with our CPU method using the BVH are in bold. Lighting (unlit (u), diffuse (d), or Phong (p)) is indicated next to the dataset name.

shading without major storage or computational requirements, we exploit the $x_i y_j z_k$ combinations already computed for trilinear interpolation to cheaply compute the analytical gradient of that filter. Specifically, this gradient reduces to a bilinear interpolant for each of the three partial derivatives,

$$\frac{\partial f}{\partial x} = \sum_{j,k=\{0,1\}} y_j z_k \left(v_{0jk} - v_{1jk}\right), \tag{9}$$

and similarly for $\frac{\partial f}{\partial y}$, $\frac{\partial f}{\partial z}$. We compute the four components of each bilinear interpolation in SIMD (*ls.* 114-124). By swizzling into four horizontal vectors and summing the result, we can simultaneously compute a single SSE register with the gradient and the dot product of the light vector. We can then efficiently unitize the n and l vectors, employing a single reciprocal square root for both (*l.* 137). Then, diffuse lighting can be computed with one additional dot product. Phong illumination requires computation of the normalized half-angle vector h, the dot product n · h, and four multiplications to compute the exponent (*ls.* 151-158).

6.3 Blending and Incrementing

Blending (Equation 3) and incrementing the sample along the ray are relatively inexpensive, but can nonetheless be optimized. By employing SSE multiplication with _0001f, we can perform alpha-blending without breaking an SSE register into component scalars. In incrementing the sample position, we use a single SSE addition for the x, y, z, t position along the ray. Finally, we employ SSE masks to check for both ray-box exit and early ray termination with a single _mm_movemask_ps() condition (*ls.* 184-190).

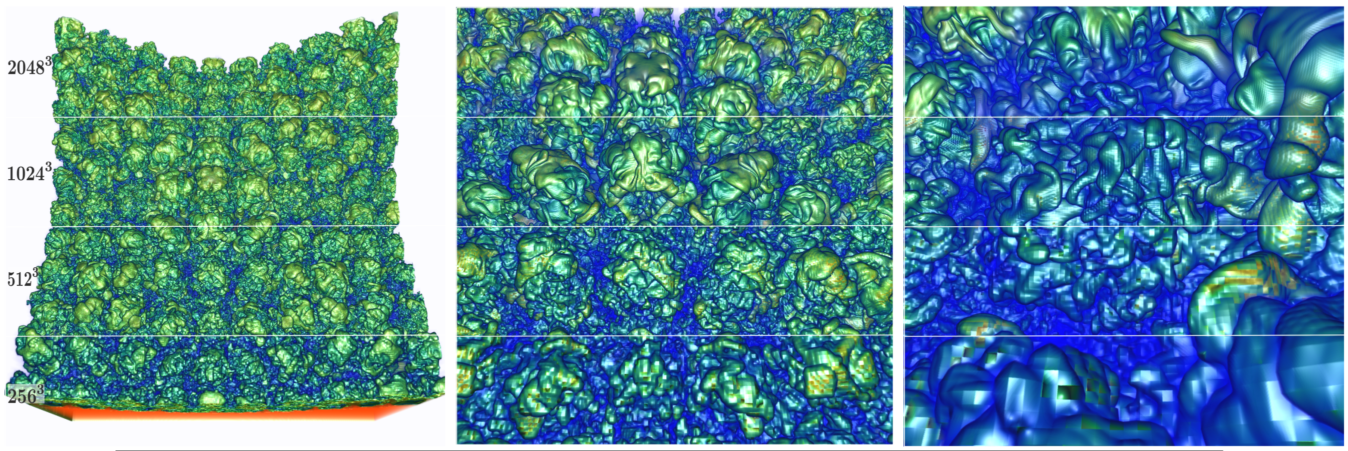
7 Results

Figure 5 and its table show benchmark results for a wide variety of volume data on both CPU and GPU hardware. Large data performance is examined more thoroughly in Section 7.1. Our hardware platforms are a Mac Mini Intel Core 2 Duo (Penryn) 2.0 GHz processor with 2 GB RAM and 2 cores, and a dual 2.67 GHz Core i7 (Nehalem X5550) desktop with 32 GB RAM (8 physical, 16 virtual cores). We list data size, BVH size, and BVH build time on one core of the i7 desktop. We compare performance with a naive floating-point implementation compiled with gcc (**gcc**), our SSE algorithm without an acceleration structure (**sse**), and our SSE method with coherent BVH traversal (**bvh**). We gauge performance with three GPU volume renderers: a brute-force GLSL raycaster (**glsl**); an optimized GLSL raycaster similar to [11] using a single-level uniform grid for acceleration, with both differential sampling and per-macrocell adaptive sampling (**avrc**); and ImageVis3D (**iv3d**), an efficient out-of-core LOD renderer designed for large data [1, 4]. We list the best-performing renderer on an integrated 9400M (128 MB RAM) in the Mac Mini and then benchmark all three GPU renderers on an NVIDIA 285 GTX GPU (1.5 GB RAM). All approaches except (**iv3d**) use differential sampling [11] with an initial differential step of 2^{-7} (rda in line 31 of Listing 2), which is comparable to uniform sampling at the Nyquist frequency (~2 samples per voxel). We employ 1D transfer functions that track the data histogram and are otherwise smooth. Since (**iv3d**) is a progressive renderer, we show the average time to load the finest LOD, approximating our transfer function as best as possible with their editor. Although (**iv3d**) supports raycasting, we used its slicing approach which is marginally faster. Unless noted, all benchmarks rendered into a 512^2 frame buffer. Lighting modalities are indicated next to the dataset.

In general, our method complements GPU approaches well. On laptop hardware, we exhibit 4x better performance than the integrated GPU (NVIDIA 9400M) on the zebrafish data, and even narrowly outperform it on the 64^3 neghip. On the desktop, for small data (less than 512^3), the 285 GTX GPU outperforms the 8-core CPU by up to 4x, particularly using (**avrc**). However, the CPU method scales better to 512^3 and larger volumes, outperforming (**avrc**) on the 285 GTX by 2.2x (backpack) and 6.5x (zebrafish) on the 8-core desktop. The backpack and zebrafish are both noisy, dense volumes that fit comfortably into GPU main memory and benefit only modestly from BVH traversal. We conclude that datasets need not be particularly large for our CPU BVH method to outperform GPU hardware. Larger data is examined below.

7.1 Scalability

Data Resolution. In Figure 6, we consider far and close views of the Richtmyer-Meshkov (**RM**) instability at original $2k^3$ and down-sampled resolutions. At full data resolution, our method on the 8-core Core i7 desktop is 20x-100x faster than the out-of-core GPU



LOD		BVH			Scene	CPU – fps				GPU – fps				Ratio
Dimensions	Size	Size	L	Build time (i7)		Core 2 2 core	Core i7 8 core			9400M 16 cores	285GTX 240 cores			best CPU/ GPU
						bvh	gcc	sse	bvh	best †‡§	glsl †	avrc ‡	iv3d §	
2048^3	8G	1.3G	8	240s	far	-	.124	.59	**6.6**	-	-	-	.083	79x
					medium	-	.091	.37	**3.5**	-	-	-	.071	51x
					close	-	.104	.29	**2.4**	-	-	-	.063	38x
1024^3	1G	171M	8	3.1s	far	**1.5**	.20	.92	**10.8**	.08 §	.21	.40	.98	11x
					medium	**.71**	.11	.62	**4.6**	.13 §	.18	.19	1.3	3.5x
					close	**.38**	.088	2.2	**2.4**	.13 §	.094	.34	1.4	1.7x
512^3	128M	171M	4	.87s	far	**1.9**	.64	2.2	**12.9**	1.5 ‡	1.2	12.6	2.0	1.0x
					medium	**1.0**	.18	1.0	**5.1**	.13 ‡	.71	6.3	1.2	.80x
					close	**.86**	.19	.82	**3.7**	.13 ‡	.28	3.2	1.0	1.2x
256^3	16M	171M	2	.36s	far	**2.0**	1.2	5.3	**14.0**	.82 ‡	5.5	18.0	6.6	.77x
					medium	**.98**	.44	1.8	**5.6**	.13 ‡	7.5	11.8	3.3	.47x
					close	**.90**	.36	1.5	**4.0**	.13 †	7.2	6.5	5.1	.55x

Figure 6: Benchmarks for the Richtmyer-Meshkov data at various resolutions, at 1024^2 screen resolution with Phong lighting, *rda*$= 2^{-7}$ (˜2 samples/voxel).

renderer on the NVIDIA 285 GTX GPU, and performs on par with $2k^3$ volumes on a 256-GPU cluster system [4]. This disparity can largely be attributed to the PCI bus. While GPU performance improves at lower LOD's, outperforming the CPU by over 3x at 256^3, CPU performance decreases only modestly when rendering roughly the same number of samples in a 256^3 or $2k^3$ volume. Not only is progressive rendering unnecessary with our renderer, but it would not be significantly faster than full-resolution rendering.

Though large and entropic, the RM data is clean simulation data that benefits greatly from BVH space optimizations. In contrast, in Figure 5 we consider a 4096x3072x2048 (24 GB) subset of an Enzo computational astrophysics dataset, which is both denser and noisier. We are still able to achieve a 45x performance increase (16x without the BVH) over the GPU, indicating there are advantages to in-core CPU rendering even for data such as this.

Sampling Rate. With either uniform or differential sampling, performance scales superlinearly with decreased sampling rate. This is due somewhat to better memory coherence, but in greater part to the BVH. Doubling the sampling rate typically incurs only 1.2x–1.8x decrease in performance. Figure 7 illustrates this trade-off. The ideal sampling rate is often less than the Nyquist rate. As seen in the Figure 6 (right), full-resolution data can in fact exhibit lower frequency than does downsampled data.

Screen Resolution. In scaling to image size, coherent ray tracers behave similarly to GPU renderers because of the cost amortization of multiple rays in packets. Scalability is superlinear; rendering at 1024^2 typically costs only 3x–3.5x more than at 512^2. This effect is stronger when the BVH incurs greater speedup.

Number of Cores. Thread scalability depends on the memory access behavior of a given scene. Rendering the $1k^3$ down-

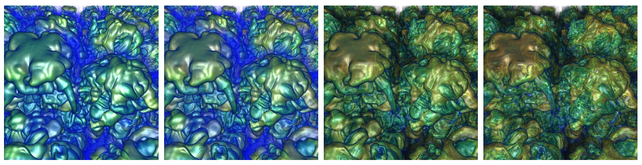


Figure 7: Sampling rate. Left to right, with differential sampling steps of *rda*=2^{-8},2^{-7}, 2^{-6}, and 2^{-5}, rendering at 6.2, 10.0, 11.1, and 15.8 fps, respectively ($2k^3$ Richtmyer-Meshkov at 512^2 on the 8-Core i7). 2^{-7} is qualitatively comparable to the Nyquist rate (>2 samples per voxel).

sampled Richtmyer-Meshkov data (Figure 6, upper left) at 1024^2, we achieve 100% scalability to 4 threads and 97% scalability to 8 threads on our dual 4-core i7 workstation. On a 4-CPU 2.93 GHz Core 2 (E7350) quad-core SMP workstation with 64 GB RAM, we see 100% scalability to 4 cores (3.15 fps), 98% scalability to 8 cores (6.3 fps), and 96% scalability to 16 cores (12 fps). These results are consistent with NUMA bottlenecks in similar systems [12].

7.2 Performance Analysis

In Table 1 we profile the percentage of CPU time spent in stages of the DVR algorithm. We compare the compiler-optimized naive implementation (**gcc**) and our SSE method (**sse**) with and without interpolation. For (**sse**), we compare the costs with BVH traversal, diffuse, and Phong lighting. While these costs vary, the heptane scene (Figure 5, upper left) is a representative average case.

With and without interpolation, voxel fetching dominates the compiler-vectorized routine (**gcc**). Amortizing address translation, our SSE code exhibits 3x better fetching performance. Trilinear interpolation (**tril**) is over 5x faster than the naive equivalent; integration with interpolation is only ~40% more costly than without (**NN**). Classification and blending cost relatively more in our SSE routine, but are difficult to optimize further. Overall, we remain bound by computation, not memory access.

The cost of lighting depends on the number of samples lit. Scenes with predominantely empty space are inexpensive to illuminate; denser scenes such as the heptane and backpack in Figure 5 can be up to 60% more costly to shade (30% for diffuse, 30% for Phong). By thresholding to omit shading of low-variance regions, one can reduce visual clutter and lower the lighting cost.

Stage	**gcc**		**sse**				
	NN	**tril**	**NN**	**tril**	**tril BVH**	**tril BVH diff.**	**tril BVH diff. phong**
vox fetch	51.7	24.8	39.6	22.3	10.7	9.7	8.9
interp		61.2		33.7	36.6	29.0	27.7
classif	4.5	4.0	18.1	8.5	10.2	9.4	8.6
blend	39.9	8.2	28.3	28.3	21.0	18.1	17.0
BVH trav					18.0	15.8	14.8
diffuse						15.2	13.9
phong							6.5
other	3.9	1.8	14.0	7.2	3.5	2.7	2.6
FPS	1.4	0.7	3.9	3.0	5.5	4.4	4.0

Table 1: CPU time profile for individual algorithmic stages of the naive (**gcc**) and hand-tuned (**sse**) methods, rendering the heptane scene from Figure 5.

7.3 BVH Performance, Size and Build Time

As seen in Figures 5 and 6, the BVH delivers from 1.5x to over 10x speedup. BVH traversal occupies 10%–35% of CPU time. This percentage and the BVH's impact on total performance depend on the static depth of the implicit BVH (L), the dynamic pruning metric δ_p, and the amount of homogenous space in the classified volume. Scenes with opaque features and transfer functions yielding surfaces induce early termination, further contributing to speedup. Choosing $L = 1$ or $L = 2$ can yield small (5%) improvements in frame rate for discrete isosurface classifications, but incurs a large memory footprint (48x and 4x for $L = 1$ and $L = 2$ on the neghip and bonsai, respectively), and thus is best avoided for larger data. $L = 4$ creates a BVH with roughly equal footprint as the original volume, and $L = 8$ is one eighth that size. For noisy and large data such as the zebrafish and full Richtmyer-Meshkov, we found no advantage to using $L = 4$ as opposed to $L = 8$. The time required to compute the BVH correlates strongly to the memory footprint. Small data compute in milliseconds, while medium-size data such as the heptane or zebrafish require several seconds. The 8 GB Richtmyer-Meshkov requires roughly 4 minutes on one core of the i7 workstation. This compares favorably to the time required to build multiresolution formats for large data. A sparse octree build [12] of the same data took 30 minutes, and a full LOD octree (**iv3d** UVF file) took roughly 55 minutes on our workstation. Moreover, the BVH can be computed once offline and stored.

8 Conclusions

We have presented a fast, scalable volume ray caster for multicore CPU's. Performance is achieved by heuristic traversal of a BVH acceleration structure and by SIMD optimization of the volume rendering integration. Although not as fast as desktop GPU approaches for smaller data, it is significantly faster at rendering large volumes and is strongly competitive with GPU's on laptop hardware.

Some limitations should be noted. Using preintegratation for BVH pruning would not extend to multifield data, though other metrics could be employed. Although superior at the low and high end of the hardware spectrum, our approach is clearly outperformed by GPU methods for small data on desktop machines. With GPU's continually improving, we do not claim the CPU will become the dominant platform for large-scale volume rendering. However, direct access to memory and multilevel cache clearly benefit CPU DVR performance, and coherent BVH traversal proves a powerful domain decomposition algorithm. Subjectively, we find interacting with large data without intermediate LOD to be a significant improvement over progressive rendering. However, LOD is an effective solution for antialiasing, and many users will prefer rendering at real-time rates with LOD to slower full-resolution rates without. Certainly, a full-resolution CPU renderer could be paired with a GPU LOD renderer for faster performance.

Future work could extend our system to clusters and tile displays for large-scale visualization. We would also like to explore compressed data and advanced illumination models.

9 Acknowledgments

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357; the Computational Postdoctoral Fellowship at Argonne National Laboratory under the American Reinvestment and Recovery Act, and the German Science Foundation (DFG) International Research Training Group (IRTG 1131). The authors thank Younis Hijazi, Carson Brownlee, Thiago Ize, and Jens Krueger for their help and insights.

References

[1] ImageVis3D: A Real-time Volume Rendering Tool for Large Data. Scientific Computing and Imaging Institute (SCI), 2009.

[2] T. J. Cullip and U. Neumann. Accelerating Volume Reconstruction With 3D Texture Hardware. Technical report, University of North Carolina at Chapel Hill, 1994.

[3] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-integrated Volume Rendering using Hardware-accelerated Pixel Shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16. ACM New York, NY, USA, 2001.

[4] T. Fogal, H. Childs, S. Shankar, J. Krueger, R. Bergeron, and P. Hatcher. Large Data Visualization on Distributed Memory Multi-GPU Clusters. *Proceedings of High Performance Computer Graphics (HPG10)*, 2010.

[5] H. Friedrich, I. Wald, and P. Slusallek. Interactive Iso-Surface Ray Tracing of Massive Volumetric Data Sets. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization*. Eurographics, 2007.

[6] E. Gobbetti, F. Marton, and J. Iglesias Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7):797–806, 2008.

[7] S. Grimm, S. Bruckner, A. Kanitsar, and E. Groller. Memory efficient acceleration structures and techniques for CPU-based volume raycasting of large data. In *2004 IEEE Symposium on Volume Visualization and Graphics*, pages 1–8, 2004.

[8] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum*, 24(3):303–312, 2005.

[9] D. Hughes and I. Lim. Kd-Jump: a Path-Preserving Stackless Traversal for Faster Isosurface Raytracing on GPUs. *IEEE Transactions on Visualization and Computer Graphics*, 15(6), 2009.

[10] G. Knittel. The ULTRAVIS System. In *Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 71–79. ACM Press, 2000.

[11] A. Knoll, Y. Hijazi, R. Westerteiger, M. Schott, C. Hansen, and H. Hagen. Volume Ray Casting with Peak Finding and Differential Sampling. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1571–1578, Nov-Dec 2009.

[12] A. Knoll, I. Wald, and C. Hansen. Coherent Multiresolution Isosurface Ray Tracing. *The Visual Computer*, 25(3):209–225, 2009.

[13] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization*, pages 287–292, 2003.

[14] P. Lacroute and M. Levoy. Fast Volume Rendering using a Shear-Warp Factorization of the Viewing Transformation. In *Proceedings of ACM SIGGRAPH*, pages 451–458. ACM Press, 1994.

[15] C. Ledergerber, G. Guennebaud, M. Meyer, M. Bächer, and H. Pfister. Volume MLS Ray Casting. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1372–1379, 2008.

[16] M. Levoy. Display of Surfaces from Volume Data. *IEEE Comput. Graph. Appl.*, 8(3):29–37, 1988.

[17] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization '98*, pages 233–238, October 1998.

[18] M. Smelyanskiy, D. Holmes, J. Chhugani, A. Larson, D. Carmean, D. Hanson, P. Dubey, K. Augustine, D. Kim, A. Kyker, et al. Mapping High-Fidelity Volume Rendering for Medical Imaging to CPU, GPU and Many-Core Architectures. *IEEE Transactions on Visualization and Computer Graphics*, pages 1563–1570, 2009.

[19] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1):6, 2007.

[20] I. Wald, H. Friedrich, A. Knoll, and C. Hansen. Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes. *IEEE Transations on Visualization and Computer Graphics*, pages 1727–1734, 2007.

[21] I. Wald, H. Friedrich, G. Marmitt, P. Slusallek, and H.-P. Seidel. Faster Isosurface Ray Tracing Using Implicit KD-Trees. *IEEE Transactions on Computer Graphics and Visualization*, 11(5):562–672, September 2005.

[22] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray Tracing Animated Scenes Using Coherent Grid Traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006. (Proceedings of ACM SIGGRAPH).

[23] L. Westover. Interactive Volume Rendering. In *Proceedings of the Chapel Hill Workshop on Volume visualization*, pages 9–16. ACM, 1989.

Listing 2: SSE Volume Ray Casting

```
#include <xmmintrin.h>
#include <mmintrin.h>
#include <emmintrin.h>
#include <smmintrin.h>

#define abs4(x) _mm_and_ps(x, _signbit)
#define cset44(x,y,z,w) _mm_set44_ps(w,z,y,x)
#define swizzle4(ssea, sseb, x,y,z,w) \
   _mm_shuffle_ps(sse, sseb, _MM_SHUFFLE(w,z,y,x) ) \
#define swizzle4_vtoh(a, b, c, d, dim) \
   swizzle4(swizzle4(a,b,0,0,0,0), swizzle4(c,d,0,0,0,0),0,2,0,2) \
#define dot3(a,b) _mm_dp_ps(a,b, 0x7f)
#define dot4(a,b) _mm_dp_ps(a,b, 0xff)

typedef __m128 sse;
typedef __m128i ssei;
struct sse_u{ sse s; float f[4]; };
struct ssei_u{ ssei s; int i[4]; };

//constants and magic numbers
const sse _1f = _mm_set_ps1( 1.f );
const sse _0f = _mm_set_ps1( 0.f );
const ssei _1i = _mm_set1_epi32( 1 );
const sse _0001f = cset44(0.f, 0.f, 0.f, 1.f);
const sse _halff = _mm_set_ps1( .5f );
const sse _1110f = cset44(1.f, 1.f, 1.f, 0.f);
const int absmask = 0x7ffffff;
const sse _signbit = _mm_set_ps1((float&)absmask);
const sse _alpha_term = cset44(1e9999f, 1e9999f, 1e9999f, 0.95f);

template<bool DIFF_SAMPLE, int LIGHTING>
sse dvr(sse org, //{org.x, org.y, org.z, 0}
          sse dir, //{dir.x, dir.y, dir.z, 1}, normalized
          float tenter, float texit, //from AABB intersection
          float dt //step size, normalized on [0,1]
          float rda //for differential sampling (optional)
        )
{
   const sse _ray_texit = cset44(FLT_MAX, FLT_MAX, FLT_MAX, texit);
   sse rgba = _0f;
   sse_u p;
   ssei_u pi;
   p.s = _mm_add_ps(org, _mm_mul_ps(_mm_set_ps1(tenter), dir));
   pi.s = _mm_cvttps_epi32(p.s);

   sse sdt = _mm_mul_ps(dir, _mm_set_ps1(dt));
   if (DIFF_SAMPLE)
      srda = _mm_mul_ps(dir, _mm_set_ps1(rda));

   //the volume data is in a bricked 3D array, accessed via
   // volume(x,y,z) = volume−>data[off_x + off_y + off_z]
   const unsigned char∗ const restrict vdata = volume−>data;
   const int∗ const restrict voff_x = volume−>off_x;
   const int∗ const restrict voff_y = volume−>off_y;
   const int∗ const restrict voff_z = volume−>off_z;

   for(;;)
   {
      const int vx0 = voff_x[pi.i[0]];
      const int vy0 = voff_y[pi.i[1]];
      const int vz0 = voff_z[pi.i[2]];
      const int val = vdata[vx0 + vy0 + vz0];

      if (val)
      {
         //trilinear interpolation
         ssei_u pi1;
         pi1.s = _mm_add_epi32(pi.s, _1i);

         const sse pc = _mm_sub_ps(p.s, _mm_cvtepi32_ps(p.s));
         const sse _1mpc = _mm_sub_ps(_1f, pc);
         const sse ztmp = swizzle4(_1mpc, pc, 2,2,2,2);
         const sse z0101 = swizzle4(ztmp, z0tmp, 0,2,0,2);
         const sse y0011 = swizzle4(_1mpc, pc, 1,1,1,1);

         const int vx1 = voff_x[pi1.i[0]];
         const int vy1 = voff_y[pi1.i[1]];
         const int vz1 = voff_z[pi1.i[2]];

         //8 voxel vertices
         ssei_u icx0, icx1;
         icx0.i[0] = val;
         icx0.i[1] = vdata[vx0 + vy0 + vz1];
         icx0.i[2] = vdata[vx0 + vy1 + vz0];
         icx0.i[3] = vdata[vx0 + vy1 + vz1];
         icx1.i[0] = vdata[vx1 + vy0 + vz0];
         icx1.i[1] = vdata[vx1 + vy0 + vz1];
         icx1.i[2] = vdata[vx1 + vy1 + vz0];
         icx1.i[3] = vdata[vx1 + vy1 + vz1];

         const sse cx0 = cast4_if(icx0.s);
         const sse cx1 = cast4_if(icx1.s);

         const sse x0000 = swizzle4(_1mpc, _1mpc, 0,0,0,0);
         const sse x1111 = swizzle4(pc, pc, 0,0,0,0);
         const sse sw_yz = _mm_mul_ps(z0101, y0011);

         const ssei dpfv = _mm_cvtepi32_ps(dot4(sw_yz,
                                    _mm_add_ps(_mm_mul_ps(x0000, cx0),
                                    _mm_mul_ps(x1111, cx1))));
         const int fval = ∗((int∗)(&dpfv));

         //classification
         sse sample_rgba = transfunc−>preIntegrated[flast][fval];

         //put the alpha value only in the alpha channel
         const sse sample_alpha = _mm_max_ps(
            swizzle4(sample_rgba, sample_rgba, 3,3,3,3), _0001f);

         if (LIGHTING)
         {
            if (_mm_movemask_ps(cmp4_gt(sample_rgba, lightThreshold)))
            {
               sse_t dx, dy, dz; //analytical gradient
               dx = _mm_sub_ps(cx0, cx1);
               dy = _mm_sub_ps(swizzle4(cx0,cx1,0,1,0,1), swizzle4(cx0,cx1,2,3,2,3));
               dz = _mm_sub_ps(swizzle4(cx0,cx1,0,2,0,2), swizzle4(cx0,cx1,1,3,1,3));

               //compute 3 bilinear interpolants
               dx = _mm_mul_ps(_mm_mul_ps(y0011, z0101), dx);
               const sse_t x0011 = swizzle4(_1mpc, pc, 0,0,0,0);
               dy = _mm_mul_ps(_mm_mul_ps(x0011, z0101), dy);
               const sse_t y0101 = swizzle4(y0011, y0011, 0,2,0,2);
               dz = _mm_mul_ps(_mm_mul_ps(x0011, y0101), dz);

               sse_t l = _mm_mul_ps(_mm_sub_ps(lightPosition, p.s), _1110f);
               const sse_t ml = _mm_mul_ps(l,l);

               //sum the dx,dy,dz and ml at the same time
               sse_t dp = _mm_add_ps(_mm_add_ps(_mm_add_ps(_mm_add_ps(
                                    swizzle4_vtoh(dx, dy, dz, ml, 0),
                                    swizzle4_vtoh(dx, dy, dz, ml, 1)),
                                    swizzle4_vtoh(dx, dy, dz, ml, 2))),
                                    swizzle4_vtoh(dx, dy, dz, ml, 3))));

               sse_t n = _mm_mul_ps(dp, _1110f);
               const sse_t nl_rcp = _mm_rsqrt_ps(swizzle4(dot3(n, n), dp, 0,0,3,3));
               n = _mm_mul_ps(n, swizzle4(nl_rcp, nl_rcp, 0,0,0,0));
               l = _mm_mul_ps(l, swizzle4(nl_rcp, nl_rcp, 3,3,3,3));

               const sse_t n_dot_l = abs4(dot3(n, l));
               sse_t diffuse = _mm_add_ps(_mm_set_ps(.15f), n_dot_l);

               if (LIGHTING == DIFFUSE)
               {
                  sample_rgba = _mm_max_ps(_mm_mul_ps(sample_rgba, diffuse),
                     _mm_mul_ps(sample_rgba, _0001f));
               }
               if (LIGHTING == PHONG)
               {
                  sse_t h = _mm_mul_ps(_mm_sub_ps(l, dir), _1110f);
                  h = _mm_mul_ps(h, _mm_rsqrt_ps(dot3(h, h)));
                  const sse_t n_dot_h = dot3(n, h);

                  sse_t phong = _mm_mul_ps(n_dot_h, n_dot_h);
                  phong = _mm_mul_ps(phong, phong);
                  phong = _mm_mul_ps(phong, phong);
                  phong = _mm_mul_ps(phong, phong); //n.h^16

                  sample_rgba = _mm_max_ps(
                     _mm_add_ps(phong, _mm_mul_ps(sample_rgba, diffuse)),
                     _mm_mul_ps(sample_rgba, _mm_0001));
               }
            }
         }
         //blending
         const sse alpha_1msa = _mm_mul_ps(sample_alpha,
            _mm_sub_ps(_1f, swizzle4(rgba, rgba, 3,3,3,3)));

         rgba = _mm_add_ps(rgba, _mm_add_ps(sample_rgba, alpha_1msa));

         if (CLASSIFICATION == PREINTEGRATED)
            flast = fval;

      } //end if (val)

      //increment along the ray
      p.s = _mm_add_ps(p.s, sdt);
      pi.s = _mm_cvttps_epi32(p.s);

      if (DIFF_SAMPLE)
         sdt = _mm_add_ps(sdt, srda);

      //check for termination
      const sse alpha_term_mask = _mm_cmpgt_ps(rgba, _alpha_term);
      const sse sse_term_mask = _mm_or_ps(alpha_term_mask,
         _mm_cmpgt_ps(p.s, _ray_texit));

      if (_mm_movemask_ps(sse_term_mask))
         break;
   }

   return rgba;
}
```