# OSPRay – A CPU Ray Tracing Framework for Scientific Visualization

I Wald[1] GP Johnson[1] J Amstutz[1] C Brownlee[1,2] A Knoll[3,4] J Jeffers[1] J Günther [1] P Navratil[2]

[1]Intel Corp [2]Texas Advanced Computing Center [3]SCI Insitute, University of Utah [4]Argonne National Laboratory

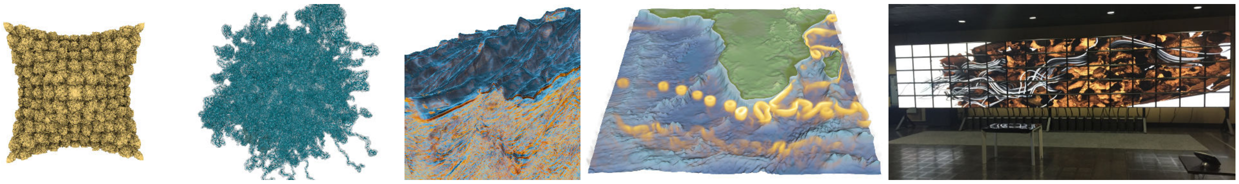

Fig. 1. OSPRay is a CPU-based ray tracing framework designed for visualization workloads including large polygonal, non-polygonal and volume data. It provides efficient rendering on Intel Xeon and Intel Xeon Phi based systems and supports "high-fidelity" ray traced illumination. These interactive examples show several use cases, rendered using OSPRay on a single dual-socket Xeon E5-2699 v3 processor workstation (unless otherwise noted). From left to right: An isosurface of 290 million polygons rendered in (OSPRay-enabled) *ParaView*; a complex *Organelle* rendered in OSPRay-enabled *VMD*; a 100 GB seismic volume data set; combined surface and volume rendering of the MPAS-Ocean model (ACME, U.S. Dept. of Energy) running data-parallel on the CPUs across eight nodes of TACC's *Maverick* cluster; OSPRay running on the 80-display *Stallion* display wall at TACC.

**Abstract**— Scientific data is continually increasing in complexity, variety and size, making efficient visualization and specifically rendering an ongoing challenge. Traditional rasterization-based visualization approaches encounter performance and quality limitations, particularly in HPC environments without dedicated rendering hardware. In this paper, we present OSPRay, a turn-key CPU ray tracing framework oriented towards production-use scientific visualization which can utilize varying SIMD widths and multiple device backends found across diverse HPC resources. This framework provides a high-quality, efficient CPU-based solution for typical visualization workloads, which has already been integrated into several prevalent visualization packages. We show that this system delivers the performance, high-level API simplicity, and modular device support needed to provide a compelling new rendering framework for implementing efficient scientific visualization workflows.

## 1 Introduction

As computing has become the scientific method's third pillar, visualization has become indispensable in interpreting generated data. In addition to guiding scientific discovery, visualization fosters novel analysis, validation of theoretical models, debugging of computational code, and communication of science to general audiences. Recent trends have shown visualization increasingly becoming integrated with large-scale simulation runs, making efficient visualization a critical component and potential bottleneck not just of post-processed analysis, but of simulation pipelines themselves.

Visualization rendering typically includes rendering data represented as explicit geometry or volumes, and traditionally utilizes standard graphics APIs (e.g., OpenGL) with hardware acceleration used to achieve interactive rendering. Increasing data sizes and associated data movement costs pose unique challenges to traditional rendering techniques. Increasingly, remote visualization occurs on high-performance computing (HPC) and cloud resources that lack dedicated rendering hardware; and even where GPUs are available, limited GPU memory and PCI bus bandwidth restrict the size of data that can be rendered in-core.

Traditional OpenGL rendering limits both the shading models and type of geometry that can be rendered. For example, advanced shading effects can greatly help in conveying the shape of complex spatial data [17] (also see Figures 2 and 8d), but is not trivially supported by traditional rasterization. Tessellating streamlines, isosurfaces or sphere glyphs can lead to an unnecessary explosion in triangle counts and the time and memory needed to compute and store their representations. For any of these issues—rendering on CPUs, large data sizes, advanced shading, etc—there are individual solutions such as software rasterization, level of detail, streaming, image-space ambient occlusion, etc; but combining all of them into existing rasterization-based visualization frameworks has proven challenging.

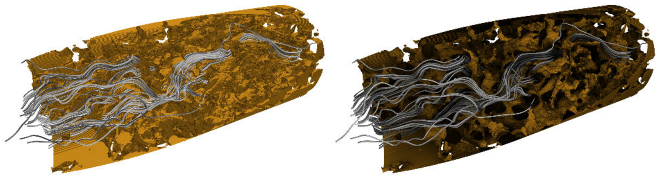

Fig. 2. *High-Fidelity Visualization* in the *FIU* data set, with OpenGL-like shading on the left, and OSPRay ambient occlusion on the right. Using advanced shading models in visualization is not about aesthetics, but about their ability to better convey the shape, depth and, ultimately, the meaning of complex data.

Ray tracing is appealing in that it potentially addresses *all* of these issues in a unified manner: it easily scales to large polygon counts and lends itself naturally to non-polygonal geometry, volume data, and advanced shading. Thanks to recent advances in compute performance and ray tracing algorithms (in particular, improved acceleration structure build performance and high-performance frameworks such as OptiX [37] and Embree [49]) it can already offer the performance required to drive visualization-style rendering at interactive rates. However, visualization pipelines require a higher level of abstraction than offered by OptiX or Embree: in particular, they need a *complete rendering framework* that offers things like an API, cameras, volumes, renderers, etc. Existing ray tracing systems have been utilized for scientific visualization, such as the open source ray tracer Manta [7], however these solutions have limited adaptability to modern architectures due to their reliance on hand-coded intrinsics and long acceleration structure build times. A full comparison of our system to existing implementations is provided in Section 8.1.

**Contributions.** In this paper we introduce OSPRay, a ray tracing framework for high-quality visualization rendering (see Figure 1). OSPRay's goal is to go beyond previous proof-of-concept ray tracing systems for visualization, and to offer a complete turn-key solution for existing production visualization software packages than can run efficiently on current hardware while offering modular device support for future hardware architectures. In particular, we propose a small and device-independent API for general but visualization-oriented ray tracing, as well as a specific, CPU-oriented implementation of this API that provides an efficient visualization rendering engine for general-purpose CPU workstations and HPC resources that can utilize varying SIMD widths. We demonstrate and evaluate the breadth, performance, and capabilities of this framework through integrations with widely used, off-the-shelf visualization packages such as VTK, ParaView, VisIt, and VMD.

## 2 Background

**Visualization** is the process of "providing new scientific insight through visual methods" [19]. It is most commonly described as a *pipeline* of operations—specifically: data analysis, filtering, mapping, and rendering—that transforms raw data into 2D images. Visualization's primary use is for interactively exploring simulation data, but also provides key debugging and tuning support for simulation codes [9]. Different simulation data types and sizes present unique visualization challenges. Consequently, visualization software, such as VTK, utilize an underlying data model to represent and process a wide variety of uniform, structured and unstructured grid data.

Visualization is highly interdisciplinary (see, e.g., [19]), resulting in a variety of specialized visualization solutions. However, a few packages have been widely adopted in the open science community, and are readily available on many HPC resources: For molecular visualization, the University of Illinois' *Visual Molecular Dynamics* package (VMD) is prevalent. For general-purpose scientific visualization, *ParaView* [3], *VisIt* [10], and *EnSight* [8] are analysis tools of choice. ParaView and VisIt are distributed-parallel analysis tools with a client-server architecture and user-friendly graphical interface. Both are open-source projects with large user bases and developer communities, and build upon the *Visualization Toolkit (VTK)* [29], a general-purpose analysis framework for visualization, and the de-facto standard for scientific visualization. Given the paramount need to handle very large data sets, all three tools offer powerful frameworks for parallel data processing (including data-parallel rendering) on multi-node HPC environments. Though ParaView, VisIt and EnSight have direct volume rendering capabilities, they primarily utilize indirect visualization (i.e., analysis that processes tessellated/triangle data) for rendering.

**Rendering.** Though much of visualization focuses on data analysis (i.e., pre-processing), the final pipeline stage is *rendering*, the process of transforming 3D geometric (and/or volumetric) primitives into 2D images. Rendering is also—and in fact, predominantly—used outside of visualization, and can be roughly organized into two categories: 1) real-time rendering and 2) production rendering. The gaming and entertainment industry predominantly drives real-time rendering, relying significantly on high-end graphics accelerator (GPU) triangle rasterization speed using the Z-buffer algorithm [44]. Given the ongoing demand for more realistic computer games, real-time rasterization has made tremendous advances, producing stunning real-time visuals when using properly tuned content and algorithms. Production rendering (e.g., movie rendering, photo-realistic rendering for design and virtual prototyping), on the other hand, aims primarily for highest image quality, driven increasingly (though not exclusively) by some variant of ray tracing [26,50]. Though originally an offline rendering technique, increasingly powerful parallel architectures have enabled interactive use cases. Most graphics and CPU manufacturers now offer platform-specific solutions for fast ray tracing: Intel released Embree [49], AMD released FireRays [1], and NVIDIA maintains a total of four separate products (*OptiX* a low-level library [37], *Iray* for photo-realistic rendering, *IndeX* for visualization rendering, and the NVIDIA *Visual Compute Appliance (VCA)* as a turn-key platform for Iray and IndeX). These solutions reflect growing adoption of ray tracing for both offline and interactive rendering.

**Rendering for Visualization.** Rendering, as the visualization pipeline's final stage, is responsible for transforming the output of the analysis stages into 2D images. Existing literature [15, 19] distinguishes between *direct* techniques (such as volume rendering) that map and render from input data with a shallow (if any) preprocess pipeline, and *indirect* methods, which typically convert continuous data into triangle geometry for rendering. The data types emitted from the final pipeline stage largely determine the rendering modality (i.e., volume rendering, triangle rasterization, glyph primitives). Though VTK, ParaView, VisIt and VMD provide comprehensive analysis tool-sets focused on "indirect" data processing for analysis and rendering, most visualization frameworks provide both direct (e.g., volume rendering) and indirect (e.g., mesh extraction) techniques.

To allow data exploration, rendering for visualization has to be interactive, and today is almost exclusively driven by rasterization using standard APIs (e.g., OpenGL). Ray tracing for visualization has had some selected use. For instance, VMD has had a ray tracing backend for two decades [43], and in biomolecular visualization the benefits of effects like soft shadows and ambient occlusion are well understood. Ray tracing has seen successful proof-of-concept uses even in interactive visualization rendering, typically focusing on a ray tracer's ability to handle large numbers of geometric primitives [7], non-polygonal primitives such as implicit isosurfaces [13,39], particle data [16,48]), and large volumes [30,36], often with good quality, and sometimes even in a single system [5,39,47]. Similarly, state-of-the-art direct volume rendering software on the GPU (see, e.g., [11,33]) largely leverage "ray-guided" techniques; a comprehensive survey can be found in [4]. While ray tracing naturally supports these application-specific "direct" approaches, we seek a ray-tracing solution that can readily integrate with general-purpose visualization packages employing deep analysis 'indirect' pipelines and rasterization focused solutions for a combination of polygonal, glyph and volume data. Most relevant to this paper's context, ray tracing backends have been previously integrated in off-the-shelf visualization packages through OpenGL interception [6], or direct integration [7,28]. The main limitation of those approaches was, in fact, not the ray tracing render performance, but rather, the time to build the acceleration data structures that ray tracers rely on, and the lack of readily available, optimized and well-maintained libraries for visualization oriented ray tracing. Despite significant technology and experience to build on, truly widespread adoption of ray tracing for visualization will require significant effort in making this technology more easily accessible for production visualization pipelines—which is the ultimate goal of our system.

## 3 Motivation and Design Goals

OSPRay is motivated by several recent and increasingly important trends in HPC and visualization.

### 3.1 Trends and Challenges

The first such trend is a growing need for effective visualization on HPC resources. Large-scale computing has been a cornerstone of science for at least the past four decades. Particularly this past decade, I/O and network transfer rates have failed to keep pace with the increase in computational throughput [9]. This I/O versus compute disparity has sparked considerable interest in *in situ* visualization (i.e., visualization on the compute resource itself, thereby minimizing data movement), as well as in *in transit* visualization (where data are immediately transfered to a dedicated visualization process, thus bypassing disk storage). In that context we note that the vast majority of HPC resources do not have GPUs: as of the November 2015 edition of the Top 500 list, only two in the top 10, and 15 out of the top 100, HPC systems have GPUs on all compute nodes [45]—and many of the upcoming large open-science HPC systems continue to be predominately CPU-based [20,24]. Efficient visualization on such machines will require either significantly faster software rasterization than currently available, or new approaches that rely on other paradigms (or both). Even dedicated visualization resources that do have GPUs often feature large memory and powerful CPUs; even when rendering is done on the GPU, most of the analysis pipeline typically runs on the host. Thus, software rendering can perform well even on resources equipped with GPUs.

The second trend is that increasingly large simulations are challenging traditional, rasterization-based visualization pipelines. Modern GPUs can rasterize billions of triangles per second, but achieve peak performance only under specific conditions (fixed polygon budget, pre-baked illumination, efficient acceleration techniques) that have proven difficult to meet in general visualization frameworks. Moreover, standard triangle rasterization cannot trivially render non-polygonal geometry used in common visualization modalities (implicit isosurfaces, particles, streamlines), nor advanced shading effects (soft shadows, ambient occlusion and global illumination). The latter are important for more effectively conveying the depth and shape of complex assemblies (see Figures 2 and 8d), which in turn is at its most useful for large and complex data. There are many techniques that solve both quality and data size/type related challenges using triangle rasterization (e.g., using rasterized impostors, screen-space ambient occlusion, level-of-detail methods, data-parallel GPU rendering, etc), but those have proven challenging to combine in a unified way that integrates well with existing visualization pipelines. Ray tracing promises to address all those challenges in a unified manner: it can handle both polygonal and non-polygonal surface data, can handle both surface and volume data, scales well to large data, is synonymous with advanced shading effects, and runs equally well on both GPUs and CPUs.

These advantages of ray tracing are generally well understood, and are increasingly widely accepted even in the visualization community: In the movie industry, products like RenderMan have already replaced REYES-style rendering (the equivalent of rasterization that it was originally built on) with ray tracing [46], and even in visualization the advantages of ray tracing have been amply demonstrated in a multitude of papers and systems (e.g., [5,7,39,47]). However, unlike in the movie industry ray tracing is not yet widely established in visualization: it has been shown to be a viable *technology* in academic papers and proof-of-concept systems, but the next step in making it a *reality*—the step from academic proof of concept to actual, widespread use in everyday production visualization tasks—needs significant effort in making the technology more accessible.

### 3.2 OSPRay: Goals

The main challenge with ray tracing as a visualization rendering backend is that it does not easily map to existing rasterization oriented APIs—it requires new APIs that more generally target visualization applications, and then integration work for those applications to utilize the new APIs. Frameworks such as OptiX and Embree are excellent starting places for building such solutions, but in themselves are on too low an abstraction level: A visualization application like, for example, VisIt or ParaView does not want to care about exactly which rays to trace or how exactly to shade each individual ray; rather, such applications wants to specify the underlying data and appearance, but leave the details of rendering to a dedicated renderer. This gap is exactly what OSPRay is designed to fill. More specifically, OSPRay is:

**A *library*, not a visualization tool.** Rather than designing a brand new visualization package, OSPRay is a library that many different visualization tools can then leverage.

**A *rendering* solution for visualization tools.** Visualization tools are complex, often relying on middleware libraries (such as VTK). OSPRay does not replace or compete with such middleware, and focuses exclusively on the visualization pipeline's rendering component. By broadening supported rendering primitives, shading models, data set sizes, etc OSPRay gives existing visualization tools' analysis stages additional choices in what they can ask the rendering stage to do.

**Focused on *visualization* rendering.** OSPRay emphasizes the rendering features needed by production scientific visualization—simple color-mapped geometry and palettes, and different renderers (primary, ambient occlusion and path tracing) that cater to a variety of needs. It does not aim for the photo-realism of professional graphics, nor for game performance.

**Focused on *HPC* visualization rendering.** Since "simple" problems are successfully handled by Mesa or GPU-based approaches, we explicitly focus on problems that remain challenging for visualization applications, such as large data, volume rendering and advanced shading. In particular, we strive to enable effective and performant visualization across all kinds of HPC resources, even those that lack GPUs. We do not discourage GPU use for all problems, but offer an efficient alternative for platforms that do not have any, and, more generally, wish to advance ray tracing solutions for those problems that can benefit from its characteristics.

**Focused on *performance*.** Though we do not have to achieve game-like frame rates, interactive data exploration requires performant rendering. Our implementation makes efficient use of threading, vectorization, and, if desired, node-parallelism; and leverages the most efficient ray tracing technologies available.

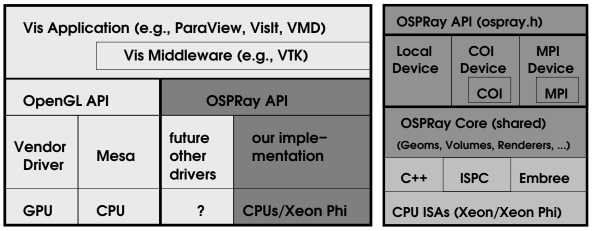

Fig. 3. (left) The OSPRay API in the context of the ubiquitous software stack found in visualization applications. (right) The components that comprise our CPU-based implementation. This paper primarily covers the dark shaded areas.

## 4 The OSPRay API

The OSPRay API exists as a layer between visualization applications and low level hardware resources. Figure 3 shows the OSPRay API in relation to other hardware and software components commonly found in visualization applications, as well as the components of our implementation discussed in Section 5. The API itself is designed to be platform independent—our implementation targets CPUs, but the API should equally map to GPUs, integrated graphics, etc. We deliberately chose a low level of abstraction similar to that of OpenGL, which is the abstraction level that current visualization tools already use for rendering. Akin to familiar solutions in OpenGL and GPGPU uses, our API focuses on the low-level data model, and on the syntax and semantics of creating—and communicating with—specific actors.

### 4.1 Categories of Actors

Ray tracing naturally supports an object-oriented programming (OOP) design in which different objects such as cameras, surfaces, etc, interact in specifying the frame to be rendered. We expose this concept of interacting objects through a small set of low-level, C-style functions to create, configure, and connect these actors. Conceptually, our API is heavily inspired by PBRT [40]: In PBRT, a scene file (the logical equivalent of a sequence of API calls to set up a frame) specifies a set of actors such as cameras, "shapes", etc. Each of these actors is a concrete type (e.g., a camera can be orthographic, perspective, etc) and has parameters that specify its configuration.

The OSPRay API exposes the following *categories* of actors:

**`OSPFrameBuffers`** hold the final result of a rendered frame. Information held can contain, but is not limited to, pixel colors, depth values, and accumulation information.

**`OSPData`** are 1D data arrays, similar to "buffers" in a GPGPU context. In addition to the typical scalar and 2-, 3-, or 4-dimensional vector data, data arrays can also contain references to other actors (including to other data arrays), in device-abstract fashion.

**`OSPGeometry`** contain geometric surface primitives such as triangles, spheres, cylinders, etc.

**`OSPVolumes`** represent 3D scalar fields that can produce, for any 3D position, a scalar value that a volume renderer can sample.

**`OSPTransferFunctions`** map scalars to RGBA colors.

**`OSPModels`** are collections of geometries and volumes – the parent objects of the hierarchy. Time-varying data are vectors of `OSPModels`.

**`OSPCameras`** generate primary rays for renderers to compute on.

**`OSPRenderers`** use cameras, models, etc, to render pixels.

**`OSPLights`, `OSPTextures`, and `OSPMaterials`** specify additional inputs for rendering, lighting, shading, etc.

**`OSPPixelOps`** are generic operations that can be used to post-process readily-computed pixels (for blending, tone mapping, etc).

### 4.2 Object Categories, Types, and Instances

In OOP parlance, the above categories are abstract objects that describe the role of objects, but not exactly how that role is filled. In practice, object categories have several discrete types—for example, a triangle mesh or set of spheres for geometry; perspective or orthographic camera, etc. For each such category the user can create instances of specific types through the use of factory methods.

Object interfaces are accessed through a generic parameter passing API. All objects contain a list of named parameters—the interface of specific object types is implicitly defined by the parameters the type understands and uses. This design strives to prevent compile-time coupling between specific object types, and allows maximal flexibility for implementations to be extended through type registration in the object factories. In particular, our implementation allows for compiling new actor types into demand-loadable shared libraries that, when loading the library, automatically register all actor types contained therein.

### 4.3 Commit Semantics

An important aspect of object parameters is that parameters do not get passed to objects individually; instead, parameters are not visible at all to objects until they get explicitly *committed* to an object via a call to `ospCommit(object)`, at which time all previously additions or changes to parameters are visible at the same time. If a user wants to change the state of an existing object (e.g., to change the origin of an already existing camera) it is perfectly valid to do so, as long as the changed parameters are recommitted.

The commit semantics allow for batching up multiple small changes, and specifies exactly when changes to objects will occur. This is important to ensure performance and consistency for devices crossing a PCI bus, or across a network. In our MPI implementation, for example, we can easily guarantee consistency among different nodes by MPI barrier'ing on every commit.

### 4.4 Models and Acceleration Structures

`OSPModels` are the objects that encapsulate multiple 3D objects that a renderer may query rays against. In addition to allowing a ray tracer access to all relevant volumes and geometries through a single handle, this offers implementations a natural place in which to build acceleration structures. Whereas ray tracing backends prefer all geometric primitives in one single data structure, hierarchical scene graphs (or middleware like VTK) generally prefer some higher level grouping of objects based on common properties (such as primitive types, materials, VTK Object they got generated from, etc). Having both models and geometries allows applications to use individual geometries for logical grouping of primitives, yet give the ray tracer a single entity to construct an acceleration structure over.

### 4.5 Frame Buffers and Rendering Semantics

Framebuffers contain pixel color information, as well as additional pixel related data such as accumulation and depth information. OSPRay supports OpenGL-like *accumulation buffers* to enable progressive refinement, allowing applications to trade image quality for interactivity. In a similar manner, OSPRay employs an OpenGL-like *depth buffer*. When combined with alpha (opacity), depth buffers enable alpha and depth compositing of OSPRay rendered imagery with other, OpenGL-rendered content (e.g., UI elements, font, wire-frame bounding boxes, etc). With these features, OSPRay runs out of the box with ParaView's and VisIt's sort-last parallel rendering modes.

OSPRay uses traditional frame semantics in which renderers render complete frames. To render a frame, the user calls `ospRenderFrame()`, and can then read the result using `ospMapFrameBuffer()`. Progressive refinement is handled by the renderer, which can use information from previously accumulated frames to adjust sample counts, random number sequences, etc. We note that progressive refinement is not uncommon in visualization tools; ParaView, for example, uses progressive refinement for volume rendering, and, by default, uses coarsified geometry during user interaction.

### 4.6 A Simple Example

We provide a simple API usage example in Figure 4. OSPRay provides the framework to create and parameterize all the respective actors required to render a frame in a device-independent manner. However, it is also important to understand what OSPRay will *not* do: In particular, the OSPRay API allows a user to select a renderer of a given type (say, a "scivis" renderer), but it does not by itself specify which renderers the implementation offers, nor what shading models are used, etc. Our current implementation (Section 5) comes with a set of clearly defined renderers, cameras, etc, that we expect to evolve into a de-facto standard of minimally supported actors; but the API specification itself intentionally does not prescribe the behavior of these renderers.

```
#include "ospray/ospray.h"

ospInit(&argc, argv);

// create and setup camera
OSPCamera camera = ospNewCamera("perspective");
ospSetf(camera, "aspect", width/(float)height);
ospSetVec3f(camera, "pos", cam_pos);
ospSetVec3f(camera, "dir", cam_view);
ospSetVec3f(camera, "up", cam_up);
ospCommit(camera);

// create and setup model and mesh
OSPData vtx = ospNewData(4, OSP_FLOAT3A, vertex);
OSPData col = ospNewData(4, OSP_FLOAT4, color);
OSPData idx = ospNewData(2, OSP_INT3, index);

OSPGeometry mesh = ospNewGeometry("triangles");
ospSetData(mesh, "vertex", vtx);
ospSetData(mesh, "vertex.color", col);
ospSetData(mesh, "index", idx);
ospCommit(mesh);

OSPModel world = ospNewModel();
ospAddGeometry(world, mesh);
ospCommit(world);

// create and setup renderer
OSPRenderer renderer = ospNewRenderer("scivis");
ospSetObject(renderer, "model", world);
ospSetObject(renderer, "camera", camera);
ospCommit(renderer);

// create and setup framebuffer
OSPFrameBuffer fb = ospNewFrameBuffer(size, OSP_RGBA_I8,
     OSP_FB_COLOR);

// render one frame and access framebuffer
ospRenderFrame(fb, renderer, OSP_FB_COLOR);
const uint *p = (uint*)ospMapFrameBuffer(fb, OSP_FB_COLOR);
```

Fig. 4. A simple example of using the OSPRay API.

## 5 Implementation

In this section we detail our specific implementation of OSPRay, which is freely available under an Apache 2.0 Open Source License [35]. While the OSPRay API has been designed in a device-abstract fashion, our particular implementation is specifically designed for CPU based platforms such as Intel Xeon and Xeon Phi based workstations and HPC resources. Figure 3 shows a depiction of our system's components, and of their relationship to each other.

### 5.1 Device Abstraction

Even in an otherwise homogeneous setup that only targets CPUs we note that the ideal implementation for directly rendering on a single compute node or workstation will differ from an offload-implementation for a PCI-card based *Knights Corner* Xeon Phi coprocessor, which in turn will differ from an implementation that targets MPI-parallelism. To support such different configurations we adopt a *device* abstraction in which the API calls are seen as a stream of commands that are internally routed to one of multiple possible backend devices, which execute the commands.

The main such device in our implementation is the `LocalDevice`, which executes all rendering right in the same process as the visualization application. We also include a `COIDevice` for offload to first-generation Xeon Phi *Knights Corner* coprocessor cards where available, as well as a `MPIDevice` that supports MPI-parallel rendering. MPI-parallel rendering is typically not required for most workloads, but can be used to drive large display walls (Figure 1e), for realizing data-parallel rendering (Section 6.5), or for fully path traced rendering without progressive refinement. Which device is used can be specified during startup, and is otherwise completely transparent to the application. All devices build on top of a shared rendering infrastructure that implements the actors (see Figure 3).

## 5.2 Software Infrastructure

Our implementation makes heavy use of two external tools: the *Intel SPMD Program Compiler (ISPC)* [41], and the *Embree* ray tracing kernels [49]. We use Embree as the basic foundation for building and traversing acceleration structures, and ISPC and C++ to build a general-purpose rendering and shading layer on top of that (also see Figure 3). Embree automatically selects the acceleration structure and traversal kernels that are best suited for a given CPU; however, where required it is equally possible to bypass Embree and implement new acceleration structures and traversals in C++ and ISPC. All volume-related code, for example, is implemented completely in ISPC.

Though the API is expressed in low-level C, all the rest of OSPRay is implemented in either C++ or ISPC. Generally speaking, all throughput sensitive operations that require vectorization (e.g., rendering, shading, primitive intersection, etc) are implemented in ISPC, while everything else uses high-level C++11 constructs.

**Benefits and Challenges of Using ISPC** The main advantage of building on top of ISPC is that is allows us to transparently target multiple vector instruction set architectures (ISAs). In particular, we currently support Intel SSE4, AVX, AVX2, IMCI (Xeon Phi Knights Corner), and AVX-512 (Xeon Phi Knights Landing), all without a single line of low-level assembly or vector intrinsics code. ISPC can also generate multi-target binaries in which the compiler automatically chooses, during runtime, the best vector instructions available on the machine. ISA portability, and generally better performance than other approaches we experimented with, have made ISPC the method of choice for our particular implementation.

The biggest challenge of using ISPC is that it does not (yet) allow features such as templates, inheritance, or virtual functions. As the latter in particular are indispensable in a ray tracer we have to emulate this using "manual" inheritance and function pointers, which adds significant inconvenience. Furthermore, ISPC does not yet support templates, which requires significant amounts of code replication, or use of compiler macros. ISPC is not a performance panacea, either. It does handle code vectorization in a ISA-independent way, but does not fully relieve the programmer from writing code that allows the compiler to generate the kind of instructions that the underlying CPUs like. In particular, as with any SPMD compiler it is easy to write code that generates lots of performance-unfriendly scatter/gather instructions, and though ISPC offers better ways than other compilers we experimented with to avoid such performance pitfalls it still requires significant programmer effort (and experience) to avoid them.

Despite these limitations—and after considerable experimentation with alternatives such as OpenCL, OpenMP, `#pragma vector`, `#pragma simd`, and several others—we have concluded that for our purposes ISPC currently offers the best compromise between performance, portability, ease of use, and what the language can express. In particular, we found that ISPC offered a hardware-savvy coder the most options to tune the code to produce hardware-friendly output.

A complete discussion of ISPC performance considerations exceeds the scope of this paper, but two guidelines are important. First, any type, variable, or construct that is `uniform` should be explicitly expressed as such, as it enables the compiler to use scalar registers and control flow rather than vector registers and vectorized control flow. Second, we use ISPC in 32-bit mode, in which it maps all addresses for `varying` array accesses to vectors of 32-bit offsets relative to a shared 64-bit base pointer. This yields (much cheaper) 32-bit address computations, and makes better use of existing scatter/gather instructions—which together can lead to significant performance gains. However, in particular for large data visualization some arrays will undoubtedly be larger than what can meaningfully be addressed by 32 bit offsets. In those cases, we have to *manually* ensure correctness by treating each array as consisting of smaller segments, then using ISPC's `foreach_unique` statement to iterate over all unique segments addressed by a vector. Inside each such segment, array accesses are then guaranteed to be addressable by with 32-bit offsets, and thus safe.

# 6 Supported Features and Capabilities

Building on top of ISPC and Embree, our implementation offers a variety of geometries, renderers, and other actors that provide a powerful set of capabilities for visualization software to use.

## 6.1 High-Fidelity Shading

While initial implementation efforts focused on several specialized renderers, feedback from users since made us develop a single `scivis` renderer that combines many rendering techniques into a single renderer. In this renderer we focus on the needs of scientific visualization: we implement an OpenGL-like material model, with customizable contributions of transparency, shadows, ambient occlusion, and fully integrated volume rendering. In addition to the `scivis` renderer we also include a `dvr` volume renderer that allows for some specially designed, OSPRay-internal data-parallel rendering (which the `scivis` renderer does not yet support). Finally, we also support a fully photo-realistic `pathtracer` renderer that can be used for generating high-quality publication images, and that has since seen adoption even outside of scientific visualization (also see Section 7.3).

## 6.2 Surface Geometry

Most of today's visualization pipelines have been designed for GPUs. Consequently, triangles are still the most common geometry that any vis renderer must support. OSPRay implements a `trianglemesh` geometry that accepts a variety of data formats for specifying vertex, normal, color, and index buffers. We use C++ and ISPC code to maintain these data structures, but leave all BVH construction, traversal, and triangle intersection to Embree.

Thanks to Embree, we can efficiently handle even very large triangle meshes. Memory footprint per triangle (including acceleration structure) is in the order of 50 to 100 bytes per triangle, meaning that even on a single workstation, models with hundreds of millions of polygons are generally not a problem (see Figure 5). Build time, too, is not a major limitation; in some cases (e.g., an isosurface extraction filter), Embree's building of the acceleration structure is actually faster than generation of the input triangles. In perspective, we note that Embree achieves build performance of up to hundreds of millions of polygons per second [49], which is comparable to (and often better than) Mesa's performance for rasterizing these triangles, which would make it competitive with Mesa even if the entire acceleration had to be re-built every frame.

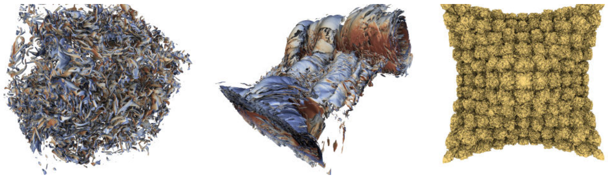

Fig. 5. Polygonal isosurfaces (using ParaView's contour filter) rendered with OSPRay's Ambient Occlusion renderer, from left to right: *isotropic turbulence*; *magnetic resonance*; *Richtmyer-Meshkov*. Despite high polygon counts of 21.5, 170, and 290 million triangles, using OSPRay a single dual-Xeon E5-2699 v3 workstation achieves roughly 48, 29, and 33 fps at 1k$^2$ pixels with direct shading, and at over 20 fps with ambient occlusion.

### 6.2.1 Non-polygonal geometry

In addition to polygons, our implementation can naturally handle any arbitrary geometry for which an intersection algorithm can be written. In our implementation, adding specific geometries is rather simple: In most cases, we can use Embree's *user-defined geometry* capability to handle BVH construction and traversal, and only have to implement three functions in ISPC: One to compute a given primitive's bounding box, one to perform a ray-primitive intersection, and one post intersect function to query shading data. Using this mechanism we currently support `spheres`, `cylinders`, `streamlines` (constant-radius tubes passing through specified control points), and implicit `isosurfaces` (defined through a volume and a set of iso-values). Several examples are shown in Figure 6. OSPRay also supports instancing through a special geometry object containing a reference to another model, plus an affine transformation matrix.

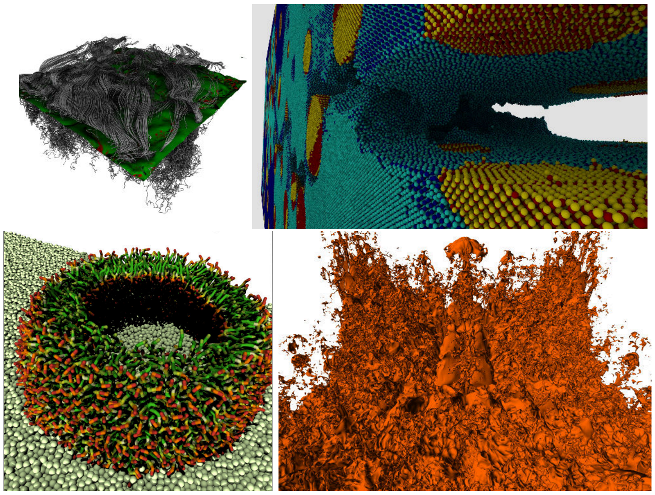

Fig. 6. Examples of non-polygonal geometry supported in OSPRay, clockwise from the top left: stream lines; particles using sphere glyphs; implicit isosurfaces in a volume data set; a mix of spheres, cylinders, and polygons.

### 6.3 Volumetric Data and Volume Rendering

Volumetric data is generally more widespread in scientific visualization than in games or professional ray tracing. Consequently, volumes are treated as first-class citizens in our implementation. However, volumes are also tricky in that most of the recent breakthroughs in ray tracing technology–such as better acceleration data structures and faster traversal kernels—do not easily apply to volume rendering (Embree, for example, does not even offer a data type for volumes). Thus, while the surface-related part of OSPRay could make heavy use of Embree, all volume related code had to be implemented from scratch, in ISPC, with significant effort in terms of both efficiency and generality.

#### 6.3.1 Volume Data Abstraction

In our implementation, `OSPVolume` objects are containers for various *types* of volume data. The most common volume data are structured volumes, for which we offer two different types: the `shared_structured` volume type allows for sharing voxel data with the application, avoiding the need to copy the data. To improve memory access patterns and thus performance, we also offer a `block_bricked` volume format that uses two levels of bricking [36] (one on $512^3$ blocks, one on $2^3$ bricks), and that renders up to 50% faster than naïve 3D arrays. To enable implementations to use such data layouts we introduced a special API call (`ospSetRegion()`) through which applications can populate an opaque structured volume type without having to understand which data layout the actual implementation will use. Internally, all volume types have a *accelerator* that skips fully-transparent regions of the volume. In our implementation, all structured grids use a grid of macro-cells as previously done by Parker et al. [36].

An important aspect of OSPRay's volume data type is that it can abstract any volume type that can be *sampled* using a 3D sample location. In addition to the already supported structured data types this allows for eventually also supporting non-structured volume types such as tetrahedral grids, adaptive mesh refinement (AMR) data, etc, which offers promising avenues for future work.

#### 6.3.2 Implicit Isosurfaces

Volume data can also be used to define implicit isosurface geometry. Our implementation supports this using a dedicated surface type that uses a given volume's macrocell data for acceleration and its sampling function to find the closest isosurface intersection along the given ray, similar to the GPU approach of Hadwiger et al. [18]. The result is a `OSPGeometry` just like triangles or spheres, and fully compatible with all surface renderers. The general sample function can also be used for purposes other than volume integration; for example, to implement slice planes, or for otherwise mapping volume data to surfaces.

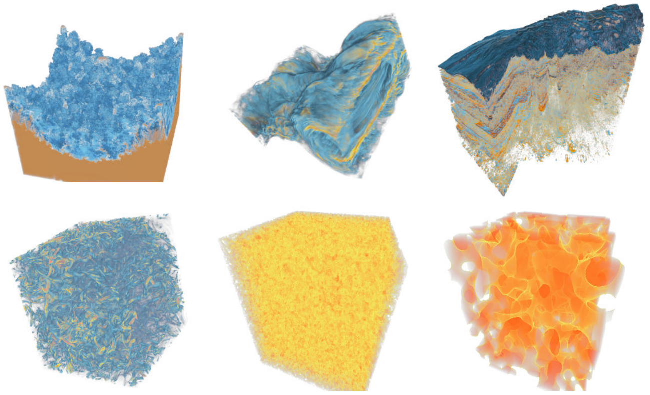

Fig. 7. Images demonstrating OSPRay based volume rendering. Top, from left to right: *RM* (2048×2048×1920); *magnetic resonance* ($512^3$ cells); *seismic* (100 GB). Bottom left: *isotropic turbulence* (512 MB). On a dual-Xeon E5-2699 v3 workstation these examples render at 35, 15, 10, and 41 fps, respectively ($1k^2$ pixels, with progressive refinement). Bottom center and bottom right: two time steps from the COSMOS *walls* dataset (40 time steps of $4k^3$, total 10 TB), rendered on a 16 TB SGI UV 300 (in cooperation with SGI and the Stephen Hawking Center for Theoretical Cosmology).

### 6.4 Large Volume Data.

Since many volumes are larger than 2 GB, we have to make use of 64/32-bit addressing mentioned in Section 5.2. For the block-bricked volume, for example, we first perform all index computations for bricking in vanilla SPMD, then do a `foreach_unique` over all unique blocks, then doing varying array accesses inside this block. ISPC maps all varying array addresses to integer vectors of offsets that are relative to the (scalar) base pointer of the array, so if we use the `foreach_unique` to iterate over the (less than 2 GB sized) blocks we know all offsets inside this block will be valid 32-bit values. The `foreach_unique` is expensive, but by introducing a 1-cell overlap between neighboring blocks we guarantee that all of a tri-linear interpolation's eight inputs will always be in the same block, so the costly `foreach_unique` has to be executed only once per interpolation.

Using these techniques the volume sizes our implementation can handle is limited only by the amount of available memory. Typical workstations today are in the range of 64–256 GB, but our system has also already been used on Stampede's 1 TB *largemem* vis nodes, on a NUMA workstation with 3 TB RAM, and on shared-memory SGI machines with up to 16 TB of RAM (see Figure 7).

Performance of the volume renderer is hard to quantify, as it depends significantly on sampling rate, chosen transfer function, amount of "empty" space in the volume, etc. We therefore intentionally refrain from even attempting any meaningful side-by-side comparisons, but refer to Figure 7 for what we believe gives a rough indication of what our system can do.

### 6.5 Data-Distributed Volume Rendering

With full access to host memory, our implementation can handle volumes that would quickly exceed single-GPU memory, even with 24 GB of on-board memory (the largest at the time of this writing). However, no matter how much memory is available, ultimately some data sets will require some form of data-parallel rendering. Data-parallel rendering can be done in either of three ways: sending data, sending rays, and sort-last alpha-compositing [34]. Of those three, the last is most problematic for a ray tracer in that it does not work for arbitrary secondary rays. It is, however, the only method that is proven by continued practical use to work for truly large volume data.

Even without any support in OSPRay, visualization tools can use OSPRay in existing data-parallel rendering frameworks by having each node render its part of the data (using OSPRay), followed by compositing the resulting partial images.

In addition, OSPRay also provides a specialized data distributed volume renderer of its own (see Figure 9). This renderer uses a extension of *segmented ray casting* [21] in which each ray is seen as a sequence

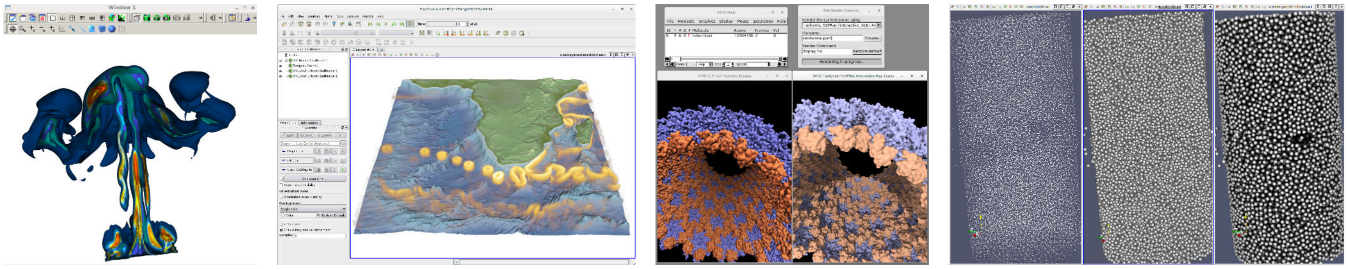

Fig. 8. Though still in Beta release, our OSPRay implementation is already prototypically integrated into three of the most widely used visualization tools, from left to right: VisIt; ParaView; VMD; a prototypical integration into VTK (done by Dave DeMarle at Kitware), showing a simple VTK application using three different VTK renderers—OpenGL points, GL Point Sprites, and OSPRay—side-by-side. Note the improvement in partical locality with ambient occlusion.

of segments that correspond to where that ray overlaps the distributed data blocks. In our approach each node generates all ray segments for its part of the data, then sends those shaded segments to whatever node owns the respective pixel. Once a pixel has received all segments it sorts and composites them, which it can do while other rays are still being traced. A full discussion of our approach is beyond the scope of this paper; however, a key advantage of our particular technique is is that it allows for mixing replicated surface geometry (which is generally small relative to volume data), with data-distributed volume rendering. I.e., it is perfectly valid to mix (replicated) surface geometry with data-distributed volume data (see Figure 1d).

## 7 OSPRay in Existing Visualization Tools

OSPRay comes with a set of sample model viewers that allow users to quickly experiment with OSPRay. However, to make the technology accessible to a wide range of actual end users it must ultimately get adopted by—and integrated into—existing, off-the shelf vis tools such as ParaView, VisIt, and VMD.

### 7.1 ParaView, VisIt, and VTK

*ParaView* [3] and *VisIt* [10] are both open source projects built on top of the Visualization Toolkit (VTK) [29]. Together, those two tools are interesting in the sheer sizes of their user communities, meaning that successful integration into those two tools alone would allow to reach significant numbers of actual end users and day-to-day visualization tasks. They are also interesting in that both projects face similar rendering challenges: growing data size, performance and quality issues with relatively old OpenGL implementations, and in particular, strong concerns regarding software rendering performance on upcoming GPU-less supercomputers—on which they will get widely deployed. Those challenges are, in fact, exactly what our project strives to address.

**ParaView.** We targeted ParaView by modifying a VTK-based ParaView module that Brownlee et al [7] had previously developed for the Manta ray tracer [5]. This VTK plugin extends `vtkPolyDataMapper` and `vtkVolumeMapper` to pass data and rendering meta-data to OSPRay, and is the foundation for both the ParaView and the VisIt integrations used in this paper.

ParaView's plugin architecture allows our plugin to insert OSPRay as a renderer without modifying existing ParaView code. We use progressive rendering to maximize interactivity during camera movements and quality when the camera is static—with the ability for end-users to customize quality and performance trade-offs. We also disable ParaView's geometry coarsification during camera movements: this coarsification aids interactivity for rasterization-based rendering, but does not help at all with a ray tracer, and only triggers acceleration structure rebuilds every time the application switches from full to reduced geometry setting.

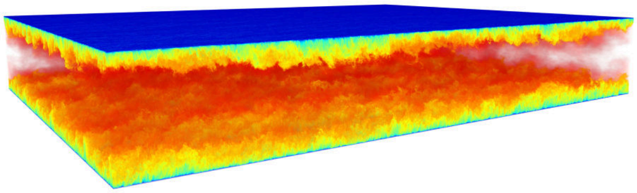

Fig. 9. Data-parallel volume rendering of the 450 GB *DNS* dataset (10240×7680×1536 `floats`). Using OSPRay's distributed volume rendering with the volume distributed across 8 dual-Xeon E5-2680 v2 nodes at the Texas Advanced Computing Center's *Maverick* machine this renders at 7.7 fps.

Though a full evaluation of the performance and capabilities of the OSPRay ParaView plugin are beyond the scope of this paper, in Tables 1 and 2 we briefly compare our OSPRay-based ParaView against off-the-shelf, OpenGL-based ParaView on two representative platforms—a high-end workstation (with two Intel Xeon E5-2699 v3 CPUs and a NVIDIA Titan X graphics card), and a visualization node in the Texas Advanced Computing Center (TACC)'s *Maverick* visualization cluster (two Intel Xeon E5-2680 v2 CPUs and an NVIDIA Tesla K40m GPU). For the GPU versions we include performance measurements for both off-the shelf ParaView (using OpenGL 1.3) and the most recent, much faster, OpenGL 2.0 rendering backend. For OSPRay we run on the same machines, but do not use the GPUs at all beyond display. We focus on data sets that are small enough to fit into GPU memory; however, for reference we also include one example (a 46 GB subset of the *DNS* data set, the full resolution reaches 900 GB of `doubles` per variable, per timestep) that does exceed GPU memory. For this data set, we did what any ParaView user would have done in this case: we run it data-parallel across 8 nodes of Maverick, which produces correct results, but obviously includes additional compositing overhead (OSPRay can render those data sizes on a single node). Timings are taken from ParaView's "Still Render" timing logs. For Mesa, we ran threaded *llvmpipe* for comparison; however, for larger datasets this gave an incorrect result with OpenGL2. For OpenGL1, immediate-mode was used when errors were encountered for larger data sets. In some cases of volume rendering the OpenGL2 backend was slower than OpenGL1 backend for volume rendering. This is using default parameters in ParaView and likely the result of different algorithms and parameters used by the two volume rendering backends.

Though any comparison of such wildly different algorithms on sim-

| | | OpenGL GPU | | OpenGL Mesa | | OSPRay | |
|---|---|---|---|---|---|---|---|
| model | #tris | v1.3 | v2.0 | v1.3 | v2.0 | simple | AO |
| High-End Workstation<br>2×Xeon 2699 v3 “Haswell”, 512 GB RAM, NVIDIA Titan X with 12 GB RAM | | | | | | | |
| isotropic | 21.5 M | 2.38 | 83.3 | < 1 | 1.49 | 47.6 | 25.6 |
| magnetic | 170 M | < 1 | 10.0 | < 1 | —* | 28.6 | 20.4 |
| RM | 316 M | < 1 | 4.95 | < 1 | —* | 38.1 | 20.7 |
| TACC Maverick Node<br>2×Xeon E5-2680 v2 “IvyBridge”, 256 GB RAM, NVIDIA K40m with 12 GB RAM | | | | | | | |
| isotropic | 21.5 M | < 1 | 25.64 | < 1 | < 1 | 19.6 | 10.4 |
| magnetic | 170 M | < 1 | 8.55 | < 1 | —* | 16.1 | 11.59 |
| RM | 316 M | < 1 | 3.92 | < 1 | —* | 18.2 | 8.77 |

Table 1. ParaView performance (fps) for surface rendering on two representative platforms. We compare OSPRay to both software and GPU rasterization, and include both ParaView's legacy 1.3 backend and its newest, much faster, OpenGL 2.0 backend. OSPRay “simple” produces similar shading as the OpenGL backends, ambient occlusion (“AO”) is included for reference. *Incorrect result.

ilarly different hardware platforms should be observed with care, the results in Tables 1 and 2 show that OSPRay is a compelling alternative to Mesa for CPU-only rendering. Furthermore, even where GPUs are available, OSPRay is at least *competitive* with GPU-based OpenGL rendering—while also offering a pathway to better, more integrated, shading models, and larger data sets. Based on initial performance results from this prototype Kitware has now included the OSPRay plugin as a pre-packaged option for their Version 5 release.

| | | OpenGL GPU | | OpenGL Mesa | | OSPRay | |
|---|---|---|---|---|---|---|---|
| model | size | v1.3 | v2.0 | v1.3 | v2.0 | simple | gradient |
| High-End Workstation<br>2×Xeon 2699 v3 “Haswell”, 512 GB RAM, NVIDIA Titan X with 12 GB RAM | | | | | | | |
| isotropic | 512 MB | 30.3 | 56.5 | $< 1$ | 2.05 | 40.98 | 29.5 |
| magnetic | 4 GB | 23.5 | 13.7 | $< 1$ | —$^\star$ | 15.2 | 8.47 |
| RM | 8 GB | 15.9 | 2.80 | $< 1$ | —$^\star$ | 34.6 | 25.1 |
| TACC Maverick Node<br>2×Xeon E5-2680 v2 “IvyBridge”, 256 GB RAM, NVIDIA K40m with 12 GB RAM | | | | | | | |
| isotropic | 512 MB | 23.3 | 42.9 | $< 1$ | 3.66 | 23.8 | 14.5 |
| magnetic | 4 GB | 16.8 | 6.99 | $< 1$ | —$^\star$ | 8.33 | 4.41 |
| RM | 8 GB | 7.63 | 1.73 | $< 1$ | —$^\star$ | 12.5 | 7.35 |
| DNS(sub) | 46 GB | 9.52$^\dagger$ | 2.66$^\dagger$ | $< 1$ | —$^\star$ | 3.07 | 1.48 |

Table 2. Volume rendering performance (fps) using ParaView's OpenGL volume renderer vs our OSPRay integration's volume rendering. We include Mesa for reference, but this often did not render correctly at all, and when it did was not interactive ($<$ 1fps). $^\star$Incorrect result (blank screen). $^\dagger$Due to data set exceeding available GPU memory, this was run data-parallel across 8 nodes.

**VisIt.** The OSPRay integration for VisIt builds on top of the same VTK plugin, with small modifications. VisIt lacks a plugin interface for rendering capabilities; instead, we extended VisIt's `avtPlot` and `avtVolumePlot` to incorporate OSPRay-enabled VTK calls. This provides OSPRay capability for surfaces and volumes, though not yet for other capabilities such as streamlines or spheres (e.g., large particle models). Visit is currently expected to support OSPRay in the next major release.

**VTK beyond ParaView and VisIt.** ParaView and VisIt have many users, but integration into VTK itself without relying on plugins reaches even more. KitWare is already pursuing direct OSPRay integration into the native rendering and analysis pathways within VTK by leveraging the newly-introduced concept of VTK *render passes*. Once completed, this would not only unify how ParaView and VisIt can use OSPRay, but would also make OSPRay available to a large class of VTK-based visualization applications that we do not yet address. A early prototype of a native VTK viewer that uses OSPRay (developed by Kitware) can be seen in Figure 8. In particular, this prototype is already able to map sphere glyphs to native OSPRay sphere geometries, or to use the ambient occlusion renderer in a standard VTK render view.

### 7.2 VMD

*Visual Molecular Dynamics* (VMD) is a widely used biomolecular visualization toolkit designed to visually inspect and understand molecular structures such as proteins and nucleic acids [23]. Representations of biomolecular complexes often contain large numbers of spheres and cylinders (that ray tracers can render without tessellation), and also often form complex structures, the details of which are most visible with complex lighting (soft shadows, indirect illumination, transparency, and/or ambient occlusion). As a result, the value proposition of ray tracing is already well understood in the biomolecular visualization community, and VMD itself has supported ray traced rendering (via the Tachyon Ray Tracer [43]) for the rendering of publication images for many years. More recently VMD also added a OptiX-based ray tracing backend, but this requires a high-end GPUs.

The existing backends for Tachyon and OptiX greatly eased the creation of a OSPRay backend for VMD (primarily developed by the author of VMD, John Stone): currently, it is realized through a specialized OSPRay render window and image generator. The OSPRay window contains a renderer in which the current state of the VMD scene is given to OSPRay to render. Although VMD itself does not support MPI parallel rendering of individual frames, OSPRay enables this without any additional VMD modifications. Full support for all VMD features will require adding some more features to our OSPRay implementation. In particular, VMD supports a minimum of six different camera types, and various ways of mapping material properties, volumetric effects, etc, that we do not fully support yet. Furthermore, our system's modular architecture permits these missing features to be added as an internal implementation detail of VMD.

### 7.3 Adoption beyond VMD, ParaView, and VisIt

In addition to OSPRay already spreading into VTK itself, we have seen early adoption in the oil-and-gas industry (INTViewer), the defense industry, in prototype in situ codes within the DOE labs, and in a variety of other projects (see Figure 10). These adoptions are increasingly implemented autonomously, without the involvement of OSPRay's core developers.

## 8 Discussion

In this section we discuss how OSPRay compares to existing systems, as well as current limitations and future directions.

### 8.1 Comparison to Existing Ray Tracing Systems

**Software Architecture.** The OSPRay API borrows heavily from OpenRT [47], OptiX [37], and PBRT [40]; the device concept and commit semantics was adapted from similar concepts in the Embree's path tracer [49]; all BVH construction and ray traversal for surface geometry use Embree outright; the ParaView, VitIt and VTK integrations heavily lean on work originally done for Manta [5, 7]; the VMD integration uses previous work from VMD's OptiX back-end; the device-independence comes naturally through the use of ISPC; the idea of mixing C++ and a ISPC-like SPMD language to architect a complete ray tracing system builds on previous experiments in the RIVL ray tracer [31]. In contrast to general ray tracing frameworks such as Mitsuba [25] and PBRT [40], OSPRay's emphasis on vectorization through ISPC orientation towards interactive visualization arguable make for a more hardware-tuned, though less fully-featured system. In addition to these similarities, there are also important differences. Compared to Embree and OptiX, OSPRay is located higher up in the software stack: Embree and OptiX provide frameworks to build ray tracing based renderes *with*, while OSPRay builds on top of embree in order to *be* a plug-and-play ray tracing renderer for production visualization tools, freeing developers from developing and maintaining efficient renderers. In the context of software architecture and features, OSPRay is closer to systems such as OpenRT [47], Manta [5], Razor [14], or RIVL [31], but even there important differences exist.

**Vectorization and Portability.** Like other high-performance CPU ray tracing predecessors (Manta, OpenRT, Razor), OSPRay makes extensive use of low-level intrinsics code across all parts of the system (e.g., for all of traversal, shading, rendering, etc). However, the effort required to write and maintaining such code made prior systems hard to extend, and nearly impossible to make portable. Both Manta and OpenRT realized this and eventually adopted domain-specific shading languages (DSLs) and compilers [38, 51], but this resulted in their own issues and limitations. Instead, OSPRay builds on lessons learned from RIVL [31] in adopting a general-purpose SPMD language. As discussed in Sec. 5.2, OSPRay builds on top of ISPC, which has the advantage of being a separately maintained, open-source software product, while still providing the user low-level control over SIMD control flow. Due to ISPC's generality and easy interfacing to external C++, we can build large parts of the system in it without a single line of intrinsics code. This will undoubtedly incur some performance penalty relative to low-level intrinsics code, but enables OSPRay to achieve a level of portability and extensibility—and ultimately, supported features—than neither Manta nor OpenRT could achieve.

**Features.** Particularly in comparison to Manta and Embree, OSPRay puts a much bigger focus on efficiently supporting volume data as a first-class citizen. Though not yet reflected in the API, OSPRay also

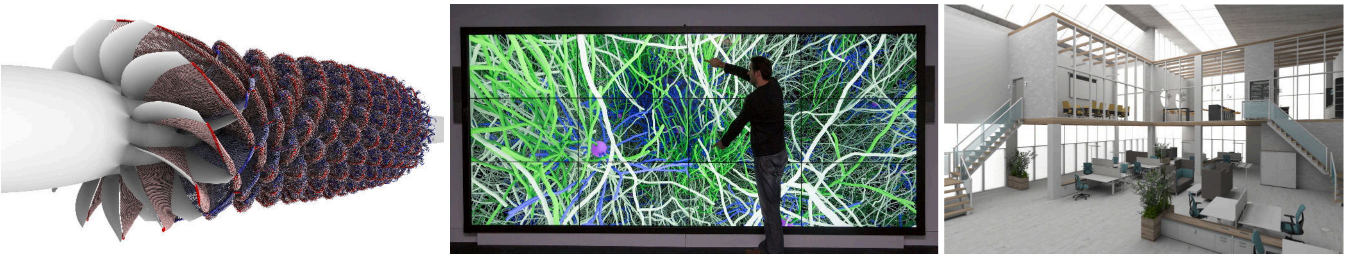

Fig. 10. Three examples of “external” adoptions of OSPRay that use our system for applications beyond the initially targeted VisIt, ParaView, and VMD, from left to right: a visualization tool (in this case, flow through a rotor) developed by Tim Sandstrom at NASA; visualization of complex neuron geometries (on a display wall), developed by Cyrille Favreau at EPFL as part of Blue Brain Project; a first adoption of OSPRay outside of visualization, showing EasternGraphics’ integration of the OSPRay path tracer into their *pCon.planner* interior space planning software, commonly run on consumer-grade Windows PCs.

supports MPI-parallel (and data-parallel) rendering. Compared to Mitsuba [25] which also supports distributed-parallel rendering, OSPRay is less feature-rich, but more oriented towards performance in HPC visualization settings. Another area where OSPRay differs from previous systems such as OpenRT is the focus on interactive usage scenarios with fast acceleration builds that do not impede interactive usage of dynamic data, offering build times of hundreds of millions of primitives per second – a common limitation of previous systems. Finally, OSPRay is different in its breadth of scope: Whereas Manta, OpenRT, and OSPRay were primarily academic (and often experimental) systems with specific use cases, OSPRay is intended to be a turn-key system for *general* end-user visualization rendering across a wide range of applications and hardware platforms – providing a fast CPU-based alternative to Mesa and OpenGL visualization. Whereas each individual one of OSPRay’s many features may previously have been supported by any one of the previous ray tracing frameworks, OSPRay aims at supporting all of them in a single framework.

### 8.2 Remaining Issues and Future Work

OSPRay addresses many rendering needs of scientific visualization software: support for large geometry, volume data, non-polygonal geometry, and direct implicit isosurface rendering, with both simple and advanced illumination. However, much remains to be done: our implementation does not yet support general unstructured and AMR data, though early work has ventured in that direction [42]. Our volume renderer currently defines volumes as *sampled* objects, which limits the renderer to a single volume integration algorithm (ray marching) that may not always be the best choice for tetrahedral meshes or AMR. Volume rendering optimizations such as adaptive sampling or pre-integration would further improve quality and performance. In addition, while OSPRay supports a wide variety of rendering modalities for polygonal data, its volume renderer is currently limited to direct illumination. Extending that to support global illumination, in the fashion of [32], would require new implementation though few (if any) changes to the API. Precomputed solutions for global illumination (i.e., [2]) or illustrative rendering techniques (i.e., [22]) fall outside the scope of the core OSPRay system and API, but could be explored by its users in implementations similar to those in OpenGL/GLSL.

The ability to run on compute resources is interesting for in situ and computational steering use cases. However, the API does not comprehensively support data-parallel use. Current distributed parallel implementations use either application-side sort-last rendering, or OSPRay’s data-parallel `MPIDevice` renderer invoked from a single process. Generalizing these capabilities to support data-distributed path tracing will require extensions to both the API and rendering internals.

ISPC has proven indispensable for our implementation, but a simpler and more extensible system could be built if ISPC supported C++ constructs, or if C++ compilers provided the capabilities of ISPC. OSPRay and ISPC are already motivating such language extensions. CPU-based visualization is also needed on supercomputers with neither GPUs nor Xeon-family CPUs, such as *Sequoia* and *Mira* (IBM BlueGene) and the *K Computer* (SPARC64). ISPC can be used on such platforms, but Embree currently cannot. Ultimately, we would like to map the OSPRay API to other platforms such as GPUs, integrated graphics, and dedicated ray tracing hardware.

Lastly, the OSPRay API offers a well-defined interface for visualization applications to interact with a ray tracer, and our implementation allows applications to experiment with this new paradigm. Wider adoption will ultimately require a process of standardizing the actors, shading models, etc, that different applications require. OpenGL has succeeded thanks to its widespread adoption beyond its original use cases, and both its API and applications using it have co-evolved. This process has only just started for OSPRay, and will require significant and ongoing future efforts. OSPRay is intentionally designed as an API for visualization, but ultimately raises the question whether and how similar features can be adopted into standard graphics APIs such as OpenGL or Vulkan [27].

## 9 Summary and Conclusion

We have presented OSPRay, a framework for ray traced visualization rendering that advances ray tracing as a solution to some of today’s key rendering challenges in scientific visualization. In particular, we demonstrate OSPRay’s suitability to drive real-world visualization through integration into ParaView, VisIt, and VMD.

We have also described an implementation of that API for multi-core and many-core CPU architectures that runs well on hardware ranging from commodity laptops to large-scale HPC resources. Our implementation supports both polygonal and non-polygonal data, volume rendering, advanced shading, and, in particular, can make full use of available memory. Integration into common visualization tools shows that our framework delivers performance that is at least competitive with existing GPU pipelines. In particular, OSPRay provides a viable alternative to Mesa rendering for current and upcoming supercomputers without GPUs. Even where GPUs are available, OSPRay can be a compelling alternative by providing access to larger CPU memory, performance for large data, and options for high-fidelity rendering.

We emphatically do *not* claim that ray tracing is always better than rasterization, nor that CPUs are always better than GPUs. GPUs and rasterization have many use cases for which they work very well, and neither CPU ray tracing nor OSPRay will be an exclusive solution for visualization any time soon. In perspective, we note that the film industry has taken decades to move from exclusively Reyes-based “shade-and-dice” [12] to now almost-exclusive use of ray tracing as in RenderMan RIS [46] (as well as others). However, our results indicate that ray tracing is already an interesting *addition* to software and hardware rasterization, and we believe that OSPRay is an important step towards more ubiquitous use of ray tracing in visualization rendering.

### Acknowledgements

This work was supported in part by the Intel Parallel Computing Centers program, the Argonne Leadership Computing Facility, DOE/SciDAC DESC0007446, and NSF award ACI-1339863. We thank: Greg Foss (TACC) for VisIt images; Mark Petersen (LANL) for MPAS-Ocean data; Dave DeMarle (Kitware) for VTK integration; John Stone (UIUC) for VMD data and integration; Paul Shellard and Juha Kaykka (COSMOS) and Karl Feind (SGI); Cyrille Favreau for HBP image; Tim Sandstrom for NASA data; Mike Papka and Joe Insley (ANL) and Priya Vashishta (USC) for silicon carbide data; Nick Malaya (ICES, UT-Austin) for DNS data; and the Moroccan Ministry of Energy (OHNYM) for seismic data.

## References

[1] AMD. The FireRays SDK. http://developer.amd.com/tools-and-sdks/graphics-development/firepro-sdk/firerays-sdk/.

[2] M. Ament and C. Dachsbacher. Anisotropic Ambient Volume Shading. *IEEE transactions on visualization and computer graphics*, 22(1):1015–1024, 2016.

[3] U. Ayachit. *The Paraview Guide: A Parallel Visualization Application*. Kitware, Inc., 2015.

[4] J. Beyer, M. Hadwiger, and H. Pfister. A Survey of GPU-Based Large-Scale Volume Visualization. In *Proceedings EuroVis*, 2014.

[5] J. Bigler, A. Stephens, and S. G. Parker. Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 187–196, 2006.

[6] C. Brownlee, T. Fogal, and C. D. Hansen. GLuRay: Enhanced Ray Tracing in Existing Scientific Visualization Applications using OpenGL Interception. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2012.

[7] C. Brownlee, J. Patchett, L.-T. Lo, D. DeMarle, C. Mitchell, J. Ahrens, and C. Hansen. A Study of Ray Tracing Large-scale Scientific Data in two widely used Parallel Visualization Applications. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 51–60, 2012.

[8] CEI. EnSight. https://www.ensight.com/.

[9] H. Childs. Architectural Challenges and Solutions for Petascale Postprocessing. In *Journal of Physics: Conference Series*, volume 78, 2007.

[10] H. Childs, E. Brugger, B. Whitlock, and 19 others. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *Proceedings of SciDAC 2011*, 2011.

[11] CIBC. ImageVis3D: An Interactive Visualization Software System for Large-scale Volume Data. http://www.imagevis3d.org, 2015.

[12] R. L. Cook, L. Carpenter, and E. Catmull. The Reyes Image Rendering Architecture. *Computer Graphics (Proceedings of ACM SIGGRAPH 1987)*, pages 95–102, 1987.

[13] D. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen. Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the IEEE PVG*, pages 87–94, 2003.

[14] P. Djeu, W. Hunt, R. Wang, I. Elhassan, G. Stoll, and W. R. Mark. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. *ACM Transactions on Graphics (TOG)*, 30(5):115, 2011.

[15] K. Engel, M. Hadwiger, J. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-Time Volume Graphics*. CRC Press, 2006.

[16] C. P. Gribble, T. Ize, A. Kensler, I. Wald, and S. G. Parker. A Coherent Grid Traversal Approach to Visualizing Particle-Based Simulation Data. *IEEE Transactions on Visualization and Computer Graphics*, 13(4), 2007.

[17] C. P. Gribble and S. G. Parker. Enhancing Interactive Particle Visualization with Advanced Shading Models. In *Proceedings of the 3rd symposium on Applied perception in graphics and visualization*, 2006.

[18] 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.

[19] C. D. Hansen and C. R. Johnson, editors. *The Visualization HandBook*. Academic Press, 2011.

[20] HPCWire. Last But Not Least: Argonne's $200 Million Supercomputing Award. http://www.hpcwire.com/2015/04/09/argonnes, 2015.

[21] W. M. Hsu. Segmented Ray Casting for Data Parallel Volume Rendering. In *Proceedings of the 1993 Symposium on Parallel Rendering*, 1993.

[22] M. Hummel, C. Garth, B. Hamann, H. Hagen, and K. I. Joy. Iris: Illustrative rendering for integral surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1319–1328, 2010.

[23] W. Humphrey, A. Dalke, and K. Schulten. VMD: Visual Molecular Dynamics. *Journal of molecular graphics*, 14(1):33–38, 1996.

[24] Inside HPC. Cray Wins $174 Million Contract For Trinity Supercomputer Based on Knights Landing. http://insidehpc.com/2014/07/cray-wins-174-million-contract-trinity, 2014.

[25] W. Jakob. Mitsuba renderer, 2010. *URL: http://www. mitsuba-renderer. org*, 3, 2010.

[26] J. T. Kajiya. The Rendering Equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

[27] Khronos Group. Vulkan API. https://www.khronos.org/vulkan, 2015.

[28] KitWare. Rendered Realism at (Nearly) Real Time Rates. http://www.kitware.com/media/html/RenderedRealismAtNearlyRealTimeRates.html, 2010.

[29] KitWare. VTK – The Visualization Toolkit. http://www.vtk.org, 2015.

[30] A. Knoll, S. Thelen, I. Wald, C. D. Hansen, H. Hagen, and M. E. Papka. Full-Resolution Interactive CPU Volume Rendering with Coherent BVH Traversal. In *Proceedings of the 2011 IEEE Pacific Visualization Symposium (PacificVis)*, pages 3–10, 2011.

[31] A. Knoll, I. Wald, P. A. Navrátil, M. E. Papka, and K. P. Gaither. Ray Tracing and Volume Rendering Large Molecular Data on Multi-Core and Many-Core Architectures. In *Proceedings of the 8th International Workshop on Ultrascale Visualization*, page 5, 2013.

[32] T. Kroes, F. H. Post, and C. P. Botha. Exposure Render: An Interactive Photo-Realistic Volume Rendering Framework. *PloS one*, 7(7):e38586, 2012.

[33] J. Meyer-Spradow, T. Ropinski, J. Mensmann, and K. Hinrichs. Voreen: A Rapid-Prototyping Environment for Ray-Casting-Based Volume Visualizations. *IEEE Computer Graphics and Applications*, 29(6), 2009.

[34] P. A. Navrátil. *Memory-Efficient, Scalable Ray Tracing*. PhD thesis, The University of Texas at Austin, Aug. 2010.

[35] The OSPRay Project on GitHub. http://ospray.github.io.

[36] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive Ray Tracing for Volume Visualization. *IEEE Computer Graphics and Applications*, 5(3):238–250, 1999.

[37] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, and A. Robison. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics (Proceedings ACM SIGGRAPH)*, 29(4), 2010.

[38] S. G. Parker, S. Boulos, J. Bigler, and A. Robison. RTSL: a Ray Tracing Shading Language. In *IEEE Symposium on Interactive Ray Tracing*, pages 149–160, 2007.

[39] S. G. Parker, W. Martin, P.-P. J. Sloan, P. Shirley, B. E. Smits, and C. D. Hansen. Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics*, pages 119–126, 1999.

[40] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufman, 2nd edition, 2010.

[41] M. Pharr and B. Mark. ISPC: A SPMD Compiler for High-Performance CPU Programming. In *Proceedings of Innovative Parallel Computing (inPar)*, pages 184–196, 2012.

[42] B. Rathke, I. Wald, K. Chiu, and C. Brownlee. SIMD Parallel Ray Tracing of Homogeneous Polyhedral Grids. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 33–41, 2015.

[43] J. E. Stone. An Efficient Library for Parallel Ray Tracing and Animation. Master's thesis, University of Missouri, 1998.

[44] W. Straßer. *Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten*. PhD thesis, TU Berlin, 1974.

[45] TOP500 List – November 2015. http://www.top500.org/list/2015/11/.

[46] R. Villemin, C. Hery, S. Konishi, T. Tejima, R. Villemin, and D. G. Yu. Art and Technology at Pixar, from Toy Story to Today. In *SIGGRAPH Asia 2015 Courses*, 2015.

[47] I. Wald, C. Benthin, and P. Slusallek. OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical report, Saarland University, 2002. Available at http://graphics.cs.uni-sb.de/Publications.

[48] I. Wald, A. Knoll, G. P. Johnson, W. Usher, V. Pascucci, and M. E. Papka. CPU Ray Tracing Large Particle Data with Balanced P-k-d Trees. In *Proceedings of IEEE Visweek*, 2015.

[49] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 33, 2014.

[50] T. Whitted. An Improved Illumination Model for Shaded Display. In *ACM SIGGRAPH Computer Graphics*, volume 13, page 14. ACM, 1979.

[51] S. Woop. *DRPU: A Programmable Hardware Architecture for Real-time Ray Tracing of Coherent Dynamic Scenes*. PhD thesis, 2006.